HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Fuzzing Windows Applications and Network Protocols

Bachelor Thesis

Department of Computer Science
University of Applied Science Rapperswil

Spring Term 2012

Authors: Kevin Lynn, Michael Fisler
Advisor: Prof. Dr. Andreas Steffen
Project Partner: Compass Security AG
External Co-Examiner: Dr. Ralf Hauser
Internal Co-Examiner: Prof. Dr. Luc Bläser

Abstract

Fuzzing is a technique for detecting software flaws by intentionally sending invalid input to a target of evaluation, generally involving a high degree of automation. For software engineers it is crucial to identify and eliminate such flaws since they might be exploitable by a remote attacker. As a consequence an attacker could compromise the application as well as the operating system the software is running on, gain unauthorized access and steal or modify confidential data. Fuzzing provides an instrument to find bugs fast with relatively low costs. While the basic idea behind fuzzing is simple, creating thorough, precise and performant fuzzers is a real challenge. With time and computing power being limiting factors, the success of fuzzing depends on the level of detail when modeling the protocol as well as the effectiveness of the data mutations performed. The goal of this paper is to present methods to effectively fuzz a network (i.e. server) application using an example protocol and a custom fuzzer.

By choosing the standards based Extensible Messaging and Presence Protocol (XMPP) we not only selected a contemporary protocol that has a gaining popularity in real-time communication applications but that also incorporates many advanced concepts found in other common networking protocols. The scope of the XMPP protocol including all its extensions is vast and therefore presents a major challenge in terms of protocol modeling and target coverage.

As our tool of choice we selected the Peach Fuzzing Platform for fuzzer modeling. Peach is a free, powerful framework including many automation features. Partly due to the documentation, which in some areas is very scarce, Peach has quite a steep learning curve. By documenting our experience, successes and mistakes we aim to reduce the initial effort required to become acquainted with Peach and its varied features. Furthermore, by following an incremental approach to build-up expertise moving from simple to more challenging problems and techniques proved a viable project guideline.

In addition to general fuzzing considerations such as the circumstances under which fuzzing makes sense and the limitations of fuzzing we also present methods to approach fuzzing extensive XML-based protocols and tuning Peach to achieve higher performance and point out strategies to a high level of coverage and precision.

Contents

Part I.

Technical Report

1. Introduction

Fuzzing is a dynamic testing technique for detecting potential vulnerabilities in software by intentionally sending invalid data to a target system. Fuzzing can range from randomly hitting keys on a keyboard to the transmission of specifically tailored data in an attempt to discover code flaws that were not detected with conventional testing. The key to fuzzing is to combine randomness, protocol knowledge or protocol sequence samples and attack heuristics, generally coupled with a high degree of automation. Such attempts promise exposing more flaws at lower costs compared to manual testing.

The term 'fuzzing' was first used in 1989 by Barton Miller who developed a primitive fuzzer to test the robustness of UNIX utilities [1]. The work was inspired by an incident that occurred to one of the paper's authors when connected on a dial-up modem during a storm, with rain causing lots of line noise. The noise introduced random characters that led to program crashes. This observed behavior was coined with the term 'fuzz'.

Nowadays, fuzzing is used by security analysts to uncover and report bugs, by QA personnel to improve the quality of their software, and by hackers to discover and secretly exploit software. Established software companies include fuzz testing in their software development lifecycle [2] to produce better and more robust software products.

1.1. Types of Fuzzing

With respect to fuzzing, three basic types can be differentiated.

- White-box Fuzzing: In white-box fuzzing source code is known to the analyst. This is often the case for in-house developed applications and open source applications.
- Grey-box Fuzzing: In grey-box fuzzing source code is not available. Testing exercises a target program or process by instrumenting binary/assembly code using a debugger and monitoring various statistics. As most vendor developed software is delivered closed source, this is the main focus of fuzzing.
- Black-box Fuzzing: In black-box fuzzing, source code is not available and testing exercises a target program or process without examining any code, whether source code nor binary/assembly code.

1.2. Attack Surface

Prior to fuzzing a system it is important to determine the potential attack surface, i.e. all external interfaces where a (remote) attacker might pass potentially malicious data to the system. The attack surface includes all kinds of parsers for protocol and file handling and processing as well as application programming interfaces (API), if present, as well as the dynamic injection of external libraries. The scope of our project is limited to network (i.e. remote) based attacks.

1.3. Fuzzer Types

There are two main types of fuzzers:

- Mutation-based: Test cases are obtained by mutation of valid, known good samples, e.g. using valid files or capturing network traffic to get protocol sequence samples.
- Generation-based: Test cases are obtained by modeling files or protocol specifications.

Research suggests that generation-based fuzzers possess a higher effectiveness in finding vulnerabilities. [3]

1.4. Failure Analysis

Although there are automated tools for error classification, every exception and failure triggered by fuzzing has to be analyzed manually in oder to gauge the potential severity. To avoid intensive, non-productive labor, fuzzing test cases have to be geared towards efficiency to avoid finding duplicate bugs over and over again. Bucketing attack input, i.e. identification of inputs targeting identical bugs, further decreases post fuzzing analysis and allows for finding better strategies and/or initial attack data.

1.5. Fuzzing Approaches Of Major Companies

In order to successfully fuzz applications it helps to know how fuzzing is approached by software companies to avoid wasting time on areas potentially already covered. Any fuzzing attempts against such software probably needs a certain degree of sophistication for satisfactory results. Microsoft is fuzzing at large scale and even employs clusters of idle computers to find vulnerabilities [4]. Google too is using raw computing power to test their own as well as embedded software [5]. In addition to basic generation-based tests, more advanced approaches are employed by the aforementioned companies but the exact nature of those tests are not disclosed to the public.

1.6. State of the Art Fuzzing Tools

A variety of fuzzing tools have been developed for a broad range of protocols and scopes. All in all, more than 300 tools are available today [6]. The fuzzing tools can be divided into two categories: fuzzers for one or more specific protocols and fuzzing frameworks. Fuzzing frameworks support fuzzer development by providing common heuristics for data mutation and may also include debugger integration, network communication modules and many other features.

1.7. Project Goals & Vision

The goal of this project is to demonstrate the chances and limitations of fuzzing by evaluating and targeting an adequate server. In addition, general fuzzing considerations such as the circumstances under which fuzzing makes sense and the limitations of fuzzing are treated. Methods are presented on how to tackle a new protocol and tuning a fuzzer to achieve a better performance. We also point out strategies to improve the level of coverage and precision.

By documenting our experience, successes and mistakes we aim to reduce the initial effort required to become acquainted with a fuzzer and its varied features.

1.8. General Conditions, Environment, Definitions, Limitations

The following list of conditions sets the limit of our project scope:

- The application to be fuzzed has to be a server application
- The application has to be reachable over the internet
- Target has to run on Windows
- Extend fuzzer to support new protocol or write new fuzzer from scratch
- The fuzzer has to be customizable

1.9. Procedure & Setup

Since the project team is new to fuzzing, a incremental knowledge build-up approach was chosen, adding complexity at each step.

The course of our project progresses from white-box to grey-box fuzzing since fuzzing is easier if source code is available. First, the team should get an overview of all things related to fuzzing, in a second step chose a suitable protocol, in a third step chose a server that handles the protocol and exists in an open source and closed source version. Last but not least a fuzzer should be chosen and customized to examine the chosen server.

The functionality and adequacy of the custom fuzzer needs to be assessed with a proof of concept.

After the elaboration phase, the project team should start implementing a fuzzer for the open source server and evaluate optimizations, for which a code coverage tools should be found.

Near the end of the project, the implemented fuzzer should be run against a closed source server using grey-box fuzzing.

The detailed process and milestones can be found in the project management chapters (part IV).

1.10. Chapter Overview

The purpose of this chapter was to give an introduction to fuzzing as well a an overview of the project setup. In chapter 2 we will look at our evaluation process which details the criteria we made for choosing a protocol for fuzzing, evaluating the fuzzer and also the target server. Later in chapter 3 we will dig deeper into the protocol specification, and in chapter 4 we will showcase the fuzzer we selected. Chapter 5 will enumerate all work items and milestones in our project realization, with chapter 6 concluding in our results and providing an outlook for future development. This finalizes Part 1.

Part 2 will present our software documentation, in which we highlight developed tools and software engineering decisions.

Part 3 will point out instructions for all software installations.

Part 4 will outline all items related to project management.

2. Evaluation

2.1. Protocol

The protocol to examine has to be chosen with regard to the project conditions defined in chapter 1.8. The selection of the protocol affects all other aspects of our project and as a consequence is the first evaluation conducted.

The following sections define our criteria, their application on evaluated protocols and conclude with the chosen protocol.

2.1.1. Criteria

The following list illustrates our criteria.

- Availability of Implementations: Exotic protocols with custom, potentially flawed implementations might exhibit a higher possibility of finding vulnerabilities, but their general availability is restricted. In contrast, broadly used protocols are often implemented by a large numbers of open- and closed-source software products.
- Complexity: The broader the scope of a protocol, the more features a software product must implement and as a consequence, the higher the possibility of hidden vulnerabilities. The evaluated protocol should preferably be stateful. The measure of complexity is a rough estimate based on the number of commands, states and options available.
- Fuzzer Availability: Protocols that are not yet covered by other open-source fuzzers are preferred.
- Documentation: Open, standardized protocols are preferred due to the fact that proprietary protocols involve a higher investment in time for analysis.
- Vulnerabilities: For a proof-of-concept, known vulnerabilities that can be exposed by fuzzing should exist. The likelihood of a successful proof-of-concept fuzzing increases with the number of known vulnerabilities present in a specific target of evaluation.

2.1.2. Assessment

Table 2.1 presents a comparison of all applied criteria.

Protocol	Impl.	Complexity	Fuzzer	Documentation	Known Vulnerabilities
XMPP	open- and closed-source	stateful, high	none known	RFC 3920 - 3923, 6120 - 6122, additional documentation	various vulnerabilities
SIP	open- and closed-source	stateful	KiF, sip-fuzzer, voiper, INTER-STATE, PROTOS	RFC 3261, 2543, extension RFCs	very high number of vulnerabilities
RTSP	open- and closed-source	medium	rtspFUZZ	RFC 2326	various vulnerabilities
SMB	open- and closed-source	unknown	unknown	MS proprietary	various vulnerabilities
NFS	open- and closed-source	unknown	unknown	RFC 3530	very high number of vulnerabilities, but mostly on Unix systems
Active Sync	closed-source	possibly stateless, unknown	none known	No public documentation	very few known

Table 2.1.: Set-up of network protocols most suitable for our project, ordered by preference.

2.1.3. Conclusion

The assessment substantiates that XMPP is the most suitable protocol. The availability of open source implementations allows us to track our fuzzer more easily and supports our approach of adding complexity with each step, moving gradually from white-box fuzzing to grey-box fuzzing. In terms of complexity XMPP beats all considered protocols. Not only is it stateful but also entails a large number of features and thus significantly increases the chances of finding vulnerabilities. Further, a higher protocol complexity increases the scalability of our project. At the time of evaluation we were not able find an open-source XMPP-specific fuzzer. Given our timeframe and the project's goals, closed protocols are not a valid option. In addition to the reverse engineering effort required, non-public protocols possibly involve a great deal of fuzzing overhead due to the unknown protocol structure, thus resulting in longer fuzzing runs. The level of XMPP protocol documentation is very promising: XMPP has a dedicated webpage featuring all RFCs and extension RFCs (XEP) as well as references to books covering the protocol. The protocol seems to be gaining popularity which makes it a promising target to examine.

2.2. Fuzzer

After applying our conditions from chapter 1.8, only a few fuzzers remain an option from the >300 fuzzers as stated in chapter 1.6. Since the protocol evaluation concluded in no known fuzzer for the XMPP protocol, we are left with two options: implementing a new fuzzer from scratch or making use of a fuzzing framework. The projects limited time frame constrained us to focus on fuzzing frameworks, where a common goal is to provide a quick, flexible, reusable, and homogenous development environment for fuzzer developers. In addition, the frameworks' authors bundle a high level of experience.

Another option considered is the use of mutation-based fuzzers that capture good samples and mutate these to form semi-valid input. Although this approach has a faster fuzzer implementation time and is less expensive, it is considered not very efficient and involves a large overhead [3]. Only fuzzers following the generation based, modeling approach are considered.

The following sections provide a high level overview of suitable fuzzing frameworks [7][8], define our criteria, the application thereof on a set of selected fuzzers and conclude with the chosen fuzzer.

2.2.1. Fuzzing Frameworks

antiparser

Antiparser [9] is an API designed to assist in the construction of fuzzers.

Autodafé

Autodafé [10] is a fuzzer developed as a proof of concept at EPFL. It uses a block based fuzzer definition just like SPIKE (see below) and adds a technique called weighting attacks with markers. Autodafé's goal is to reduce the size and complexity of the input space by focusing on areas of the protocol that are likely to contain vulnerabilities. The Markers Technique provides a weight to each fuzzing variable, which then determines the order in which to process the fuzz variables with a focus on higher weight. The included debugger sets breakpoints on common dangerous APIs. When transmitting test cases to both the debugger and target, the debugger monitors the API calls looking for strings originating from the fuzzer, which then increases the weight assigned to the variable. This prioritization can drastically reduce the input space.

Fuzzled

Fuzzled [11] is a fuzzing framework written in Perl. Fuzzled includes basic fuzzer components but further information is very scarce.

General Purpose Fuzzer / Evolutionary Fuzzing System

GPF [12] is a mutation-based fuzzer and exposes functionality as a number of modes. PureFuzz is a random fuzzer that uses a seed for stream replay, Convert allow the import of network capture files into GPF files, GPF main mode mutates GPF files and replays them, Pattern Fuzz automatically tokenizes and fuzzes detected plain text portions of protocols and SuperGPF is a Perl script GPF wrapper that is a fuzzing starting point for unknown ASCII protocols. A new mode is Evolutionary Fuzzing System which is evolutionary based. EFS uses genetic algorithms to generate inputs. A modified version of PaiMei that stores hits is used as the basis for calculating fitness for each session, and the sessions with the best fitness are allowed to breed. Although stated in the introduction to this chapter that only generation based fuzzers are considered, the approach with evolutionary fuzzing is quite promising and therefore added.

Peach

Peach [13] is a fuzzing framework featuring a flexible architecture and promoting the most code reuse. The framework exposes a number of features for modeling new fuzzers including generators, transformers, data models, state models, publishers, agents and monitors. Fuzzers are modeled in XML. Peach includes the capability to capture network traffic in case of target crash, error bucketing (identification of inputs targeting identical bugs) and automatic bug classification, target restart after crashes and virtual machine control and generally a high degree of automation.

SPIKE

SPIKE [14] is claimed to be the most widely used and recognized fuzzing framework and pioneered the block-based protocol representation. Blocks may be embedded in other blocks and as a consequence arbitrarily complex protocols can be broken down into smallest atoms. The power of this approach is that the values for each of the size fields are automatically updated as various mutations are preformed. The block-based approach has gained popularity and a number of other fuzzing frameworks have adopted the technique.

Sulley

Sulley [15] is a fuzzer development and fuzz testing framework consisting of multiple extensible components. The framework aims not only to simplify data representation, session modeling, but also data transmission and target monitoring. Sulley has a number of interesting features including maintaining network captures for each test case, monitoring the health of the target with an agent, communication with both network capture agent and target monitoring agent with a custom RPC protocol, automatic fault categorization, virtual machine control, internal web server, parallelization of fuzzing and generally a high degree of automation. Fuzzers are modeled using the block-based approach like with SPIKE.

2.2.2. Criteria

The following list defines our criteria.

- Programming Language: The team member's background in Java and C/C++ favor fuzzers in those languages.
- Fuzzer Platform: The platform(s) the fuzzer supports.
- Monitoring Component/Tracer: A component running on the target platform communicating back to the fuzzer.
- Documentation and Community: Since we do not have experience in the area of fuzzing, we strongly consider the documentation available.
- Development Activity: Tools that are actively developed and are highly advertised in the fuzzing community gain a higher priority per se. In addition, future development plans are considered.
- License: The fuzzer license.

2.2.3. Assessment

Table 2.2 presents the comparison of all applied criteria for all considered fuzzing frameworks.

Fuzzer	Prog. Language	Platform	Monitor	Docs	Development
Peach	Python, XML	Win/Unix	Win/Unix Agent	mediocre docs, good fuzzer examples, active message board [16], tutorial available	ongoing maintenance by author, widely used, planned switch from Python to C#
Sulley	Python	Win/Unix	PaiMei on Win/Unix	poor docs, good fuzzer examples, inactive message board [17]	no dev since 2007
Autodafé	C	Unix	adbg tracer Win/Unix	Limited	Proof of concept, no active development
Fuzzled	Perl, XML	Unix	None	very scarce	not very active
GPF / EFS	C	Unix	PaiMei Win/Unix in EFS	poor docs	active
Spike	C	Unix	None	very scarce	not very active
antiparser	Python	Win/Unix	None	API only	active

Table 2.2.: Set-up of fuzzers most suitable for our project, ordered by preference.

2.2.4. Conclusion

The assessment does not conclude with a distinct choice. A high emphasis is given to the integration of monitor and fuzzer, which combined with the platform constraint to Windows, restricts us to only three fuzzers of which Autodafé is only a proof of concept, although an interesting one. In terms of features, Peach and Sulley seem to have gained an edge over all other fuzzing frameworks. None of the fuzzers account for our preferences of programming language and as a consequence we have the additional task to study a new programming language: Python. Both Peach and Sulley seem to be mature projects and in a head to head race, the proof of concept in chapter 5.1 was realized with both in order to define a clear preference. In the end the preferable overall documentation and the fact that Peach is still actively developed suggests that Peach is the most suitable framework for our purposes.

2.3. Open Source XMPP Server

As a target for white-box fuzzing we evaluated the following open source XMPP servers [18].

2.3.1. Criteria

The following list defines our criteria.

- Platform: The server has to support Windows.
- Programming Language: The team member's background in Java and C/C++ favors servers in those languages.

2.3.2. Assessment

Table 2.3 presents a comparison of all applied criteria.

Server	Platform	Prog. Language
jabberd2	Windows / Linux / Solaris	C
Openfire	Windows / Linux / Mac OS X / Solaris	Java
Apache Vysper	Windows / Linux	Java
Tigase	Windows / Linux / Mac OS X / Solaris	Java
ejabberd	Windows / Linux / Mac OS X / Solaris	Erlang
psyced	Windows / Linux / Mac OS X	LPC
Prosody	Windows / Linux / Mac OS X	Lua
Kwickserver	Windows	unknown

Table 2.3.: Set-up of open source XMPP servers deemed most suitable for our project, ordered by preference.

2.3.3. Conclusion

The programming language the server is implemented in is the decisive factor for there often exists a different debugger for each language, e.g. a C-debugger cannot debug a Java program. According to our fuzzer evaluation, high emphasis was given to the monitoring integration with the fuzzer, and Peach only has native debugger support for C programs. Jabberd2 being the only server implemented in native C is our simple choice. Additionally, uncovering exploits in native C/C++ is easier than in Java or .Net, the latter exposing added security features out of scope for this project.

2.4. Closed Source XMPP Server

As a target for grey-box fuzzing we evaluated a closed source XMPP server [18].

2.4.1. Criteria

The following list defines our criteria.

- Platform: The server has to support Windows.
- Free or trial version: The vendor has to provide a trial or free version.

2.4.2. Assessment

Table 2.4 presents a comparison of all applied criteria.

Server	Platform	Trial version
Coversant SoapBox Server 2010 Express	Windows	yes
CommuniGate Pro	Windows / Linux / Mac OS X	no
IceWarp	Windows / Linux	no
Isode M-Link	Windows / Linux / Solaris	no
Cisco Jabber XCP	Windows / Linux / Solaris	no
Jerry Messenger	Windows / Linux	no
Oracle Communications Instant Messaging Server	Windows / Linux / Solaris	no

Table 2.4.: Set-up of closed source XMPP servers deemed most suitable for our project, ordered by preference.

2.4.3. Conclusion

The majority of vendors do not provide a free or trial version for download, and as a consequence we chose Coversant SoapBox 2010 Express [19].

3. The XMPP Protocol

XMPP is an acronym for eXtensible Messaging and Presence Protocol. It is an XML-based protocol formerly known as Jabber. Due to the protocol's extensibility and open standard status, the description is contained in several RFCs and extension RFCs.

3.1. XMPP Core Technology

The Core RFC [20] of XMPP defines the basic protocol concepts, i.e. the XML stream concept, session security negotiation by means of SASL, use of TLS and stanza semantics. Some server implementations may be based on the deprecated Core RFC [21].

3.2. XML Stream

Every XMPP connection is defined by a so-called stream which is set up using a <stream> XML element. A stream can be thought of as a container for data being sent from one node to another during a session. The closing </stream> tag closes the XMPP stream but not necessarily the underlying connection. Every XMPP interaction is encapsulated by a stream.

3.3. Roster

In the XMPP terminology a roster is the client's buddy list, i.e. the list of all known clients.

3.4. Stanza

Within streams, stanzas are used to encapsulate protocol features. The following stanzas are defined for client- and server-namespaces:

- <message>: Message exchange. Push mechanism from one XMPP node to another.
- <presence>: Presence status updates. Publish-subscribe mechanism for delivery to every subscripted node (no "to=" attribute set) or push mechanism to a single node defined by the "to=" attribute
- <iq>: Info/query exchange. The iq stanza is the most versatile of all. It allows to query data from the server and depending on the qualified namespace, perform various actions such as roster management and many more. Iq is a request/response

mechanism, therefore all iq requests should be acknowledged with a success or error response.

3.5. Simple Authentication and Security Layer (SASL)

Before being able to interact with other clients or querying/updating the server, a client has to authenticate the stream with the server by means of traditional authentication or SASL [22][20]. Since Non-SASL authentication has been obsoleted in favor of SASL [23] we will focus on SASL authentication even though the servers we have evaluated could be configured to allow traditional authentication. During SASL authentication the server offers a list of authentication mechanisms ordered by preference. Subsequently the client chooses the preferred mechanism and sends its credentials. Depending on the mechanism chosen and the server implementation, SASL authentication may include several requests and responses until successful authentication or failure. Details of the authentication process are defined in the Core RFC (section 6). After successful SASL negotiation client and server must restart the stream.

3.6. Client Registration

To authenticate using SASL or other methods, a client has to register with the XMPP server first. Figure 3.1 shows the stream setup and registration procedure.

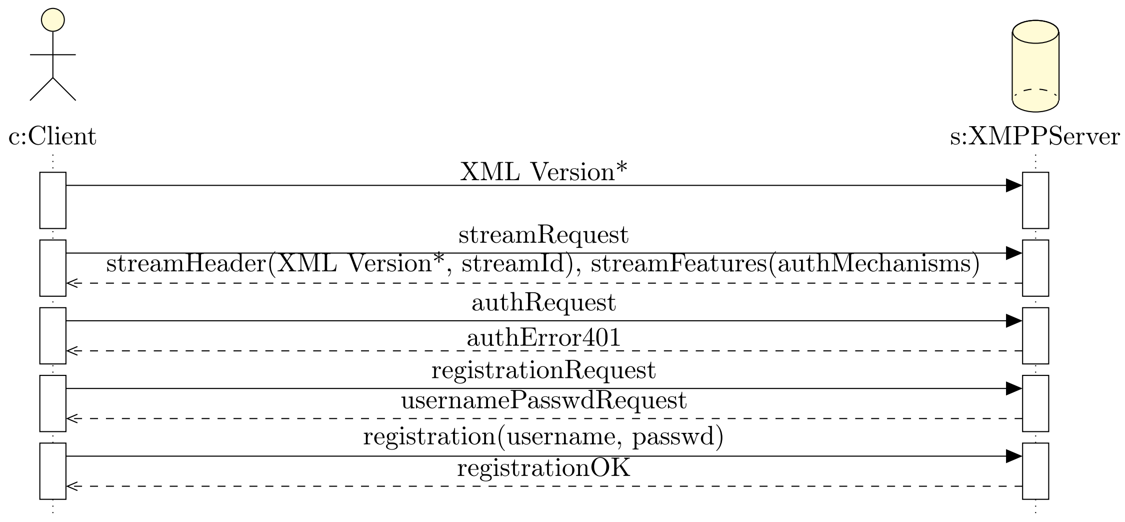


Figure 3.1.: Successful registration handshake procedure between the jabberd2 server and a previously unregistered IM-client. The XML Version is declared optional as far as the RFC is concerned.

4. The Peach Fuzzing Framework

To get an idea of how Peach works, the following sections explain the basic concepts behind Peach and introduce parts of the Peach terminology. In general we will only briefly illustrate the different features of Peach since certain parts are well documented in the official documentation [13]. In our opinion, other parts are however lacking important information. In order to facilitate an entry into fuzzing with Peach we will advise on how to avoid pitfalls and how to accomplish certain tasks.

4.1. General Peach Concepts

While Peach can be used for mutation-based fuzzing, it will only unleash its full potential when used for generation-based fuzzing. In Peach, modeling is done with two separate kinds of models, the *DataModel* and the *StateModel*, in order to separate structural models from protocol sequence models. *DataModel*s offer a wide range of predefined building blocks, such as *String* elements for modeling common strings, *Blob* for modeling binary data and many more. Peach also enables defining custom types. Elements defined in a *DataModel* are associated with values that are then transferred to the fuzzing target. By providing a mutable attribute, each element can be declared to be mutable, i.e. fuzzable, or immutable as depicted in figure 4.1.

```
<DataModel>
    <String value="My number: ">
    <Number size="8" value="42" mutable="false"/>
</DataModel>
```

Figure 4.1.: A simple *DataModel* containing a mutable string, “My number: ”, and an immutable number, 42.

4.2. Peach Fuzzer

Peach Fuzzer is the component sending fuzzed data to the target system based on a configuration file, the so-called Peach Pit file. Peach fuzzers can be run in parallel using a network of computers for fuzzing. The command to start fuzzing is: python peach.py pit_file.xml and can optionally be run with the –debug argument for better traceability. With the debug option, every packet sent between fuzzer and target server is printed to standard output, displaying every byte of the packet’s payload. When redirecting the output to a file, this can serve as an additional logging mechanism.

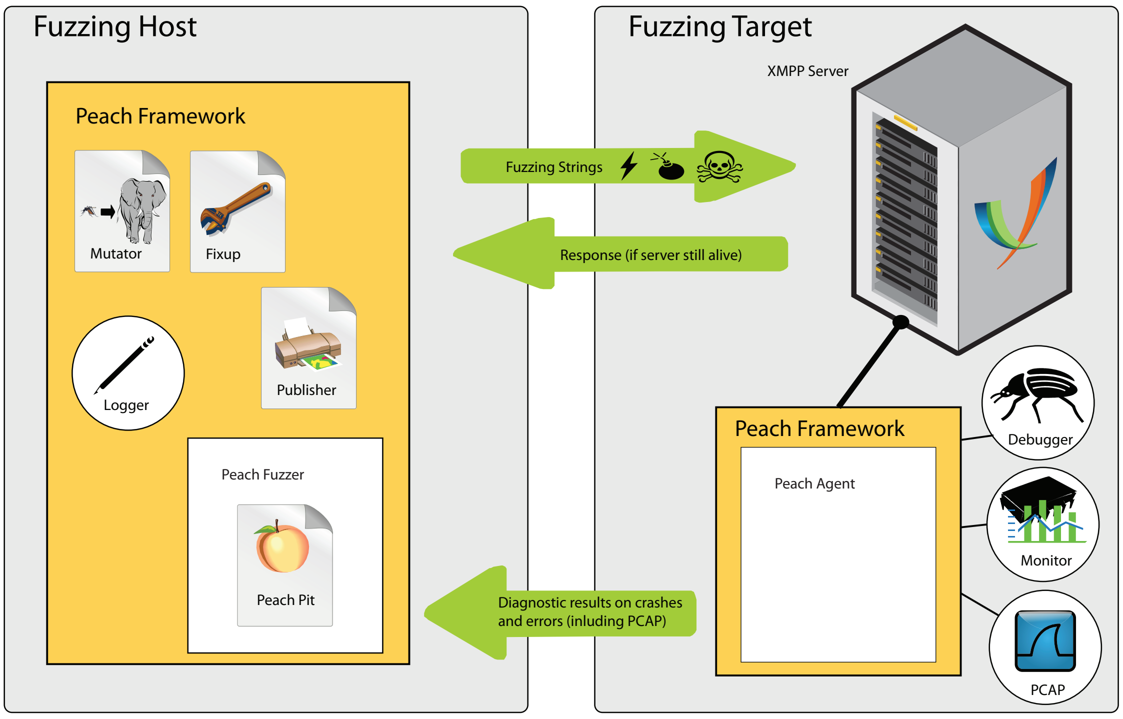


Figure 4.2.: The Peach framework and its components.

4.3. Peach Agent

In order to diagnose a system under test, an instance of Peach, a Peach agent, can be run on the target system. The agent's main responsibility is attaching a debugger or other monitor to the target application or process and informing the fuzzing source (i.e. the Peach Fuzzer) when a fault or an exception occurred during a fuzzing run. While it is possible to use Peach without agents, but agents significantly reduce the effort needed to fully automate the fuzzing process by providing features like automatic restart of crashed programs and virtual machines as well as automated bug analysis. An Agents accepts only a single connection from Peach fuzzers.
During our fuzzing runs the Peach Agent has proven fairly unreliable when running low on memory. When executing an Agent and a memory hungry application simultaneously, the Peach Agent filed an “Agent unreachable” failure in the logs and the Peach fuzzer aborted the fuzzing process.

4.4. Peach Fuzzer Configuration File (Pit File)

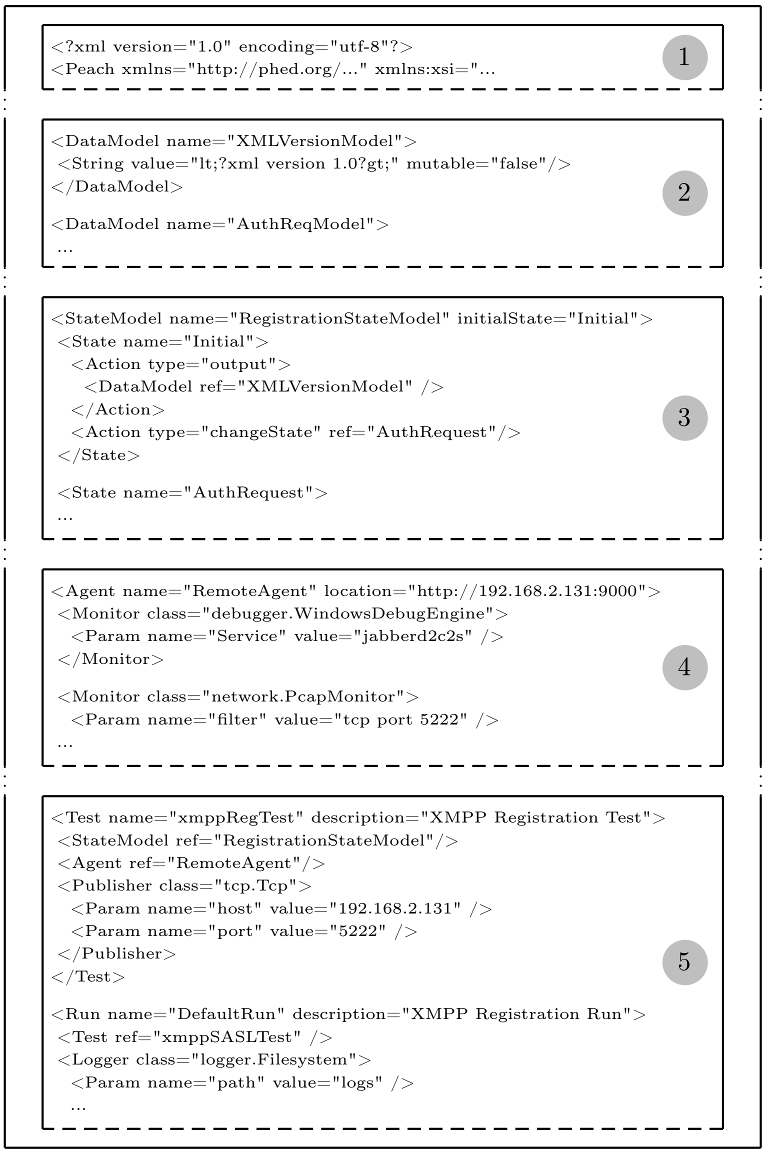
A Peach Pit file is essentially an XML file instructing Peach how to fuzz an application. It contains the logic and/or format of a protocol as well as parameters required for testing. Writing a fuzzer basically means implementing a Pit file which can then be analyzed and executed by the Peach framework. As figure 4.3 shows, a Pit file can be broken down into five sections. These sections are self-contained but do not, except for the first section, need to be in any specific order.

General Information

In the General Information section (figure 4.3, (1)) default namespaces are declared as well as the location of the schema to be used. In addition, external Python search paths can be added allowing to place custom Python scripts in other directories then the Peach installation folder. In order to use custom Python code, Import XML entities have to be specified: <Import import="own_script">

Data Models

The Data Model section (figure 4.3, (2)) contains any data model required to instill the structure of a protocol data unit (PDU) or the file format if used in the context of file fuzzing into Peach. Fuzzing success, among other factors, depends on the granularity of the Data Models available. The better a protocol is modeled, the better the chances to find bugs. On the other hand the complexity of a data model significantly influences the run-time of a fuzzing process. In some cases, especially with huge protocols, it may be necessary to avoid over-engineering data models, e.g by avoiding modeling every detail such as the IP address format or the like. For a list of every element available to model data models refer to the official documentation [24].



```
<?xml version="1.0" encoding="utf-8"?>
<Peach xmlns="http://phed.org/..." xmlns:xsi="...
```

```
<DataModel name="XMLVersionModel">
 <String value="lt;?xml version 1.0?gt;" mutable="false"/>
</DataModel>

<DataModel name="AuthReqModel">
 ...
```

```
<StateModel name="RegistrationStateModel" initialState="Initial">
 <State name="Initial">
   <Action type="output">
     <DataModel ref="XMLVersionModel" />
   </Action>
   <Action type="changeState" ref="AuthRequest"/>
 </State>

 <State name="AuthRequest">
 ...
```

```
<Agent name="RemoteAgent" location="http://192.168.2.131:9000">
 <Monitor class="debugger.WindowsDebugEngine">
   <Param name="Service" value="jabberd2c2s" />
 </Monitor>

 <Monitor class="network.PcapMonitor">
   <Param name="filter" value="tcp port 5222" />
 ...
```

```
<Test name="xmppRegTest" description="XMPP Registration Test">
 <StateModel ref="RegistrationStateModel"/>
 <Agent ref="RemoteAgent"/>
 <Publisher class="tcp.Tcp">
   <Param name="host" value="192.168.2.131" />
   <Param name="port" value="5222" />
 </Publisher>
</Test>

<Run name="DefaultRun" description="XMPP Registration Run">
 <Test ref="xmppSASLTest" />
 <Logger class="logger.Filesystem">
   <Param name="path" value="logs" />
   ...
```

Figure 4.3.: The structure of Peach Pit files used to configure custom fuzzers.
(1) General configuration data
(2) Data models describing the structure of the data to be sent to a fuzzing target
(3) State models
(4) Agent and monitor configurations to survey the fuzzing target
(5) Run and test settings

State Models

The State Model section contains any states the fuzzer should traverse when communicating with the target of evaluation. While it may make sense to model states on PDUs defined in the protocol's specification, in some cases a different approach is necessary. If we were to model every request of a multi-request authentication procedure in a separate state, depending on the mutation strategy, chances of achieving a successful login are minimal. Using Peach's default settings and mutations, the path passing through a state model will not be sequential one. As an example, if we want to request a stream first, then select the auhentication method and last authorize, chances are that Peach will try to send the auhentication credentials before the auhentication method request, thus failing proper authentication and at worst dropping the connection to the server altogether. For details on state modeling syntax efer to the official documentation [25].

Agents and Monitors

The Agent and Monitor section in a Pit file enables a user to define the CPU and memory monitoring and debugger features to use. Peach has built-in support for different debuggers on different operating systems. WinDbg integration is by far the most advanced and includes features such as automatic exploitability checks using the WinDbg plugin !exploitable and allows to bucket fuzzing strings targeting identical bugs to reduce post-fuzzing analysis time.
Unfortunately, when attaching WinDbg to a service or running process instead of directly launching an executable Peach does not properly detach from the process after completing the test run. The outcome of this behavior is a crashing the process. Therefore, if a service is not executing anymore after a fuzzing run, chances are the debugger killed it while shutting down.

Test and Run Configuration

The Test and Run Configuration contains one or more tests to be executed and allows to define loggers to document the fuzzing results.

4.5. Peach Mutators

Peach's mutators are responsible for altering values of fuzzable elements or modifying the structure of the data tree Peach constructs while parsing the Pit file. While the term "mutator" suggests that mutation is performed, most mutators really just substitute values in *DataModel* with predefined values from the mutator's internal value list.
When using Peach's default mutators it may seem as if mutators ignore the fact that certain elements have been set to immutable using the attribute mutable="false", see figure 4.1 . The reason for this odd behavior is that mutators are applicable not only to elements like *String*, *Blob*, etc. but also to the top-level elements like *DataModel*. In some cases it may be necessary to disable these mutations. A login procedure, for example, is most likely to fail if data is altered during authentication. Excluding certain mutators is fairly easy: just declare all mutators you want to use in the "Test" section of the fuzzer configuration file and all other mutators are disabled. Figure 4.4 shows our NCNameMutator to illustrate how substituting Mutators are implemented. In order to use this specific mutator, a *Hint* is needed. Figure 4.5 illustrates how to define a *Hint* in a *DataModel*.

4.5.1. Mutation Strategies

The mutation strategy defines the order in which substitutions are performed. Table 4.1 below lists the different strategies defined in Peach.

Strategy	Description
SequentialMutationStrategy	Follows all elements in sequential order
RandomDeterministicMutationStrategy	Randomly select an element to fuzz until every element has been completed. Uses a seed to spread fuzzing over all elements, so the course of action is reproducible
RandomMutationStrategy	Selects N fields at random from Data-Model to fuzz on each test case. Can be seeded.
SingleRandomMutationStrategy	Selects one field at random from Data-Model to fuzz on each test case.
DoubleRandomMutationStrategy	Selects two fields at random from Data-Model to fuzz on each test case.

Table 4.1.: Peach mutation strategies

4.5.2. Peach's Default Mutators

Table 4.2 lists Peach's included default string mutators.

```
class NCNameMutator(Mutator):

    def __init__(self, peach, node):
        Mutator.__init__(self)

        # Do we have a finite number of values to produce?
        self.isFinite = True

        # Name of mutator (typically same as class)
        self.name = "NCNameMutator"

        self._peach = peach

        # Here is the list of values the mutator produces
        self._values = [':eatthat!', 'eat:that!', 'eatthat!:']

        self._count = len(self._values)

        # Our position in _values
        self._index = 0

    def next(self):

        self._index += 1
        if self._index >= self._count:
            raise MutatorCompleted()

    def getCount(self):

        return self._count

    def supportedDataElement(node):

        if isinstance(node, String) and node.isMutable:
            #restrict use of the Mutator to elements with NCName hints
            #for better performance
            for hint in e.hints:
            if hint.name == "NCName":
                return True

        return False
    supportedDataElement = staticmethod(supportedDataElement)

    def sequencialMutation(self, node):
        ...
```

Figure 4.4.: Source code of the NCNameMutator. Its purpose is to produce invalid xs:NCName values. Since the colon (:) must not be used in NCNames, the Mutator provides three strings: using a colon at the beginning, in the middle and at the end. Mutation strategy parts have been left out.

```
<String name="MyNCName" value="SomethingWithoutColon">
  <Hint name="NCName" value="true" />
</String>
```

Figure 4.5.: A *String* containing the NCName *Hint* to enable the specific NCNameMutator.

Name	Description	Number of iterations
StringCaseMutator	Changes the case of a *String* value in three ways: first to lower-case each character, then upper-case each character and then randomly decide to upper- or lower-case each character	Dependent on string length
StringMutator	Performs common string mutations	2372
UnicodeBadUtf8Mutator	Performs bad UTF-8 string mutations	152
UnicodeStringsMutator	Performs common unicode string mutations	201
UnicodeBomMutator	Performs BOM string mutations	1414
UnicodeUtf8ThreeChar Mutator	Performs long UTF-8 three byte string mutations	308
ValidValuesMutator	Allows to specify additional data	Dependent on values
FileNameMutator	Performs file name mutations	14590
HostnameMutator	Performs host name mutations	3891
PathMutator	Performs path mutations	18462
XMLW3CMutator	Perfroms the W3C parser tests	1419

Table 4.2.: Peach string mutators

4.6. Peach Fixup

A fixup is a Python script that changes or substitutes a value from a *DataModel*. Fixups can for example be used to dynamically calculate checksums or to create a response to a server challenge. We use fixups to create the SASL PLAIN and DIGEST-MD5

```
import string
from Peach.fixup import Fixup

class ExampleFixup(Fixup):

    def __init__(self, ref):
        Fixup.__init__(self)
        self.ref = ref

    def fixup(self):
        self.context.defaultValue = "42" #not recommended!
        att_value = str(self.context.getValue())

        if att_value == None:
            raise Exception("Error: ExampleFixup was unable to locate [\%s]" \
                            % self.ref)

        #if att_value does not contain an equal sign, return an empty string
        if len(string.split(att_value, '=')) == 1:
            return ""

        return string.split(att_value, '=')[1]
```

Figure 4.6.: An example Fixup to demonstrate Fixup implementation. It extracts the value from an attribute - value pair (e.g. color=blue yields blue)

authentication strings.

4.6.1. Creating Custom Fixups

Figure 4.6 shows an example Fixup that contains the basic concepts. Although Fixup programming is pretty straight-forward, there are some pitfalls.
A default value can be set for a Fixup but we strongly recommend not to do that. The problem is that, especially if the Fixup result is a hash or another very long encoded string, it is difficult to see if what the Fixup delivered was the default value. Besides, if any, there are probably very few reasons to use a default value.
To apply a Fixup to a value from the Peach *DataModel*, 'str(self.context.getValue())' can be used. This might not be necessary if the goal is to return a static value (in our case a base64 encoded SASL PLAIN authentication string with data from a config file). If the implementation relies on the existence of a certain character (in example 4.6 a string containing an equal sign) within the value to fix up, you need to include a guard clause or else the Fixup will crash Peach since the fixup() function seems to be called not only when a value is requested but also at instantiation of the Fixup. In any case we advise to include debug messages printed to the standard output to verify that the Fixup works as intended, at least until it functions flawlessly.

4.6.2. Including The Fixup in the DataModel

The official Peach documentation states that Fixups can be applied to *DataModel* and *Block*, but they are applicable to *String* elements, too. Figure 4.7 shows how to correctly use a Fixup with a target string.

```
<DataModel name="ChallengeModel">
    <String name="Challenge">
        <Fixup class="fixups.MD5DigestSASLFixup">
            <Param name="ref" value="Challenge"/>
        </Fixup>
    </String>
</DataModel>
```

Figure 4.7.: Placement of the Fixup within the *DataModel* with Challenge being the string to be fixed up.

4.7. Slurp Action

A Slurp Action is a feature for transferring data from one *DataModel* to another. If slurping is used in conjunction with a Fixup, be sure to first apply the Fixup and then transfer the fixed-up value to the target *DataModel* as illustrated in figure 4.8.

```
<StateModel name="md5auth" initialState="Initial">
    ...
    <Action type="input">
        <DataModel ref="ChallengeModel"/>
    </Action>
    <Action type="slurp" valueXpath="//ChallengeModel//Challenge"
setXpath="//ResponseModel//Response" />
    <Action type="output">
        <DataModel ref="ResponseModel"/>
    </Action>
    ...
</StateModel>
```

Figure 4.8.: Generating a response based on a challenge using slurping.

4.8. Peach Publisher

The Publisher component allows sending data from a *DataModel* to the fuzzing target. For network based fuzzers Peach already has a TCP Publisher built-in.
Since a Publisher is an ordinary Python script, the use of Publishers is not only restricted

to sending data, but can also be extended to realize fuzzing environment set-up, e.g. cleaning-up databases and registering users with a server.

5. Realization

5.1. Proof of Concept: Savant Web Server

As a first introduction to fuzzing, we searched for a simple demonstration of the usage of a fuzzer. Ideally, this would coincide with our prerequisites in chapter 1.8 but must not necessarily include an XMPP server. We found a tutorial on youtube for Sulley [26]. The demo involves a basic Web Server called Savant, and demonstrates how Sulley finds an exploitable known bug [27]. In an effort to reproduce the demo, we installed Sulley and Savant and could successfully find the known bug. Also, as a comparison, we installed Peach and were able to reproduce the same result. This proved that both open source products are in an operational state and are able to find known bugs. In addition, this gave us a first overview of the necessary configuration work for the two fuzzers, which was incorporated into the fuzzer evaluation (see chapter 2.2).

5.2. Proof of Concept: Known-Exploit Fuzzing

In order to verify the effectiveness of our fuzzer, a proof of concept involving an XMPP server with a known vulnerability has been planned. To date several vulnerabilities have been published for jabberd2:

- Nested XML Entities Denial of Service Vulnerability [28]
- SASL Negotiation Denial of Service Vulnerability [29]
- Client to Server Component Buffer Overflow Vulnerability [30]

Since Peach is much more advanced on Windows in terms of debugger integration and post crash analysis we first planned to set up the jabberd2 server on a Windows platform. After hours worth trying to compile jabberd2 on Windows we decided not to waste additional time and switched to Unix, where compiling jabberd2 is an easy task. For our proof of concept the very basic Apport debugger integration did the job.
We were able to crash the server with our fuzzer, but did not hit the bug we intended. The server shut down while trying to process a malformed namespace.
Nonetheless we gained assurance that Peach is suitable for fuzzing XML protocols and fully functional.

5.3. General Fuzzing Considerations

A very important issue of fuzzing is reproducibility. Running the same tests multiple times with the same preconditions, the result should always be the same. If reproducibil-

ity is not a given, the fuzzer can not be used for regression tests and analysis of the test data requires much more time. To ensure same preconditions, database initialization or truncation of database tables may be necessary. Knowing the database layout the target server uses, a custom database set-up and tear-down Publisher can be written.
In the past, the jabberd2 server exhibited a known vulnerability caused by the XML parser. Our own tests confirmed that older versions of the jabberd server seemed to have issues resolving invalid (i.e. fuzzed) namespaces. Targeting the parser may therefore, to a certain extent, make sense. However, the XMPP protocol requires stanzas to be well-formed and the RFC requests the servers to discard any non well-formed stanzas. Barring any major XML parser flaws, we expect the majority of parsers to correctly determine well-formedness and rejecting non-compliant stanzas without passing them to the server for further processing. We therefore restrict fuzzing the parser to namespace fuzzing.

5.4. Choice Of Database

XMPP servers store their information in databases. For example if a user registers with the server, the user's credentials are stored in a database record. While fuzzing we strive for repeatability, therefore we may need to be able to "reset" all persisted data before running tests. In order to be able to modify and read the database remotely, preferably from within Peach, we chose MySQL over the default SQLite.

5.5. Authentication

Our fuzzing approach contains two phases. In the first phase we create fuzzers with the stanzas intended for registration and authentication, in the second fuzzers which authenticate with the target server, leaving the authentication portion untouched to ensure successful authentication. The first attempt at including the server login was to write a custom Publisher, using a Python XMPP library, to register and login with a predefined user and fuzzing the remaining stanzas after authentication and session setup. The advantage of using an external library would have been to reduce the time to create a fuzzer for a new server. However, we could not get our own Publisher to work in conjunction with Peach's TCP Publisher. The Publisher successfully authenticated with the server, but we did not find a way to sustain the TCP connection, in spite of Peach's documentation suggesting that the connection by default lasts until the end of the test run.
As an alternative we modeled the whole authentication process with Peach's *StateModels* and *DataModels*. The downside of this approach is the limited portability to other servers. Our tests have shown that the servers do not always implement authentication the same way and that the RFC leaves room for interpretation by stating that multiple requests and responses may be used for authentication. It also depends which RFC the implementation is based on. We disabled TLS on the server and restricted the SASL authentication to PLAIN and DIGEST-MD5. This allows us to use instant messaging

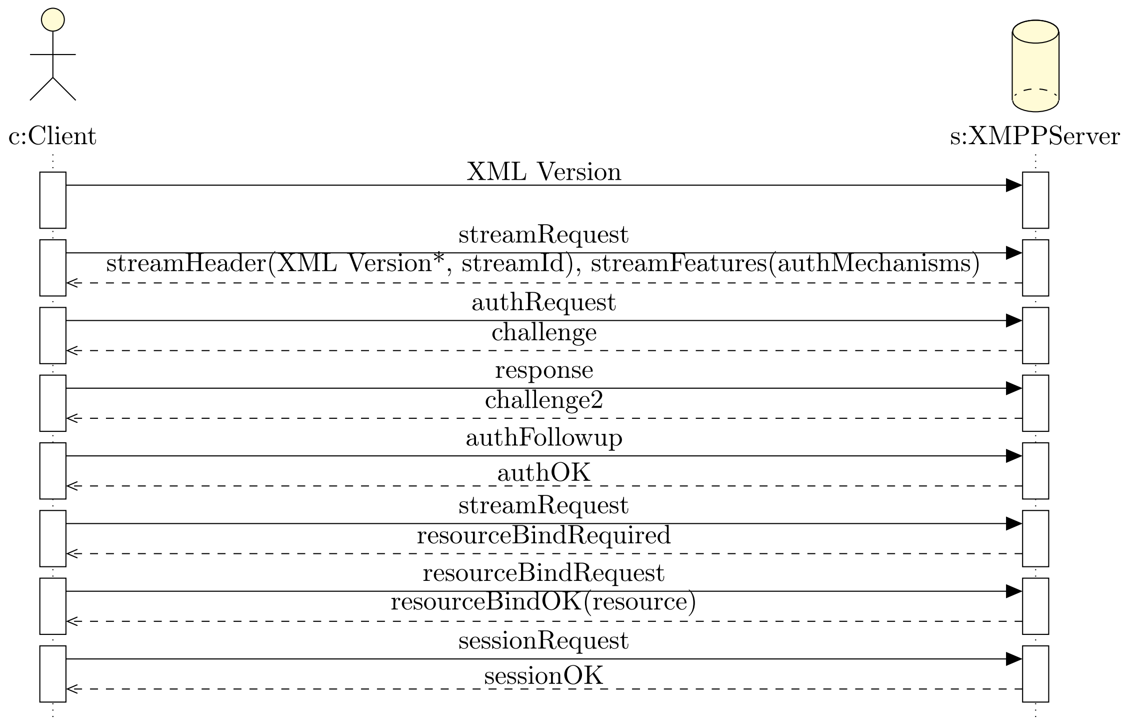


Figure 5.1.: A successful XMPP DIGEST-MD5 authentication, resource bind, stream request and session setup procedure as implemented in our fuzzers.

clients to verify the server's functionality as well as getting samples of the client-server communication with Wireshark. Figure 5.1 shows how authentication is implemented in our fuzzer generating scripts.

5.6. XMPP-specific Peach DataModels

The following subsections describe the *DataModel*s we have implemented to fuzz XMPP-specific datatypes. All data contained therein could in principle be modeled with a simple *String*, but in order to dig deeper and check if a server can handle ill-structured types, a specific *DataModel* is needed.

5.6.1. JID

The Jabber-ID (JID) [20] is an identifier used to address an entity (i.e. users, servers, etc.) within an XMPP-network. The JID's format in the Augmented Backus-Naur Form [31] is defined as follows:

jid = [node "@"] domain ["/" resource]
domain = fqdn / address-literal

fqdn = (sub-domain 1*("." sub-domain))
sub-domain = (internationalized domain label)
address-literal = IPv4address / IPv6address

In addition to these definitions the following restrictions exist:

- The node, domain and resource identifiers must not exceed 1023 bytes in length
- The JID's total size therefore must not exceed 3071 bytes
- A domain identifier must be an "internationalized domain name" [32]

JID string fuzzing may therefore aim at invalid identifier lengths, invalid identifier formats and all kinds of invalid characters.

5.6.2. DateTime

DateTime is a primitive data type defined in the W3C recommendations [33]. In the context of XMPP it is used for all time stamps, e.g. the time and date a note was created. A xs:dateTime time stamp follows the rule:

'-'? yyyy '-' mm '-' dd 'T' hh ':' mm ':' ss ('.' s+)? (zzzzzz)?

Example: 2012-03-09T23:59:59.1234+01:00 = March 9, 2012, 23:59:59.1234, UTC/GMT
The details and restrictions to this rule are listed in the W3C recommendations.

5.7. Peach Mutator Optimization

A fuzzing run with a stream request stanza as shown in figure 5.2 where only the "to" attribute was fuzzed took two hours to complete on a computer equipped with a 2.4GHz Intel Core 2 Duo processor. Investigating the cause revealed that in most cases a lot of mutations performed do probably not yield new results. For example a string was mutated into a number which then was subsequently incremented by one.

```
<stream:stream to="value2fuzz" xml:lang="de" xmlns="jabber:client"
  xmlns:stream="http://etherx.jabber.org/streams" version="1.0">
```

Figure 5.2.: A stream request stanza as used to determine fuzzing run time. Only the "to" attribute has been set to fuzzable

Our strategy to optimize the mutators is based on an attempt to reduce the number of mutations present in the original mutators that came with Peach, while trying to still cover a majority of potential bugs. The reason being that the time needed for a fuzzing run is proportional to the number of fuzzing strings generated by the mutator.

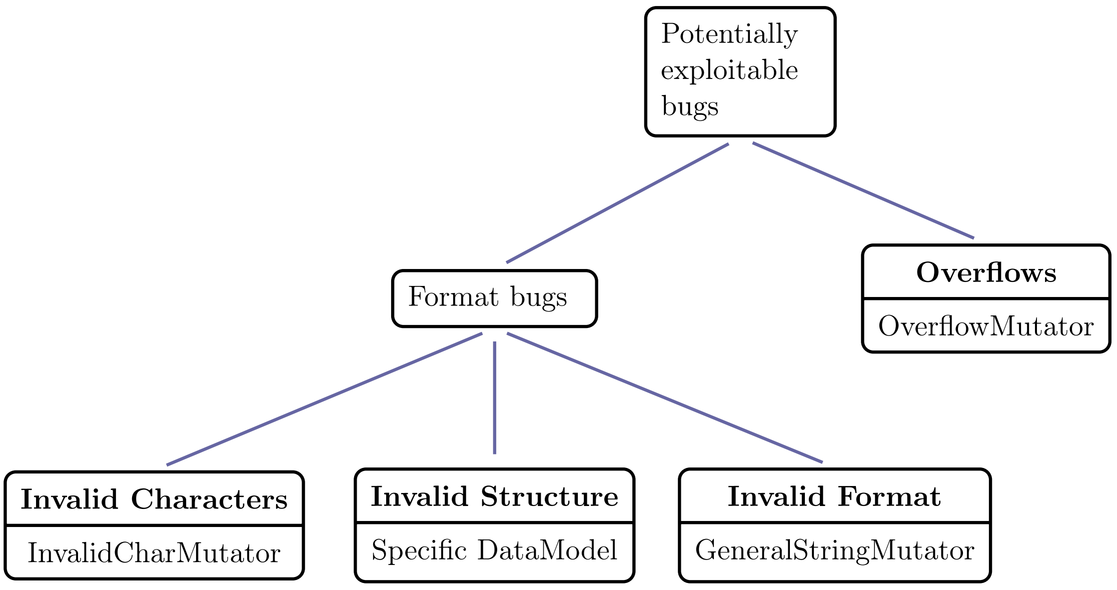


Figure 5.3.: The exploitable bug classes used to define more specific and therefore more efficient mutations

In order to maximize the efficiency and effectiveness of our mutators we consider the different classes of bugs as depicted in figure 5.3. For each class we define separate mutators or *DataModels* which should cover most exploitation scenarios while needing a minimal set of mutations.

5.7.1. General String Mutators

The "General String Mutator" class uses a subset of the strings in Peach's *String* mutator class. It contains mutations which should be applicable to all fuzzing strings. The strings in the "General String Mutator" category can be broken down into the following classes:

- empty string
  Mutator string: ''

- a string containing random upper-case, lower-case and mixed-case letters
  Example Mutator strings: 'vaskdjfnbf', 'BLKJKEI', 'GejbJE'

- strings interpretable as booleans: 0, 1, true, false, yes, no, ok, nok
  Example Mutator strings: '1', '0', 'true', 'false', 'True', 'False', 'TRUE', 'FALSE'

- control characters: NUL, DEL, CR, etc.
  Example Mutator strings: 'm\nm', 'm\rm', u'bl\x00ub', u'bl\x0Eub'

- numbers: integer type, float type, positive, negative
  Example Mutator strings: '1337', '-1337', '1.337', '-1.337', '1E-337', '-1e-337'
  Overflow strings: '259', '65539', '4294967299', '18446744073709551619'

- symbols, special characters
  Example Mutator strings: u'm\U0001F4A9', u'\u1337'

The goal of the General String Mutators is to deliver strings which could lead to errors due to conversion when processed by the server. E.g. if a server uses a non signed integer but fails to check the integrity, a very long integer value might lead to an integer overflow and cause havoc. Due to efficiency considerations we just pick a few examples for each string category mentioned above. If a server cannot parse or process characters belonging to a certain unicode class, it is sufficient to only send one single example.

5.7.2. Overflow Mutator

The purpose of the "Overflow Mutator" is to cause an overflow without side-effects caused by special or invalid characters. For performance reasons we decided to just use a string and a number with a length of 10000 characters. The reason why we use a number as well as a string is that the targeted field could exclusively require either format. We think that an upper bound of 10000 characters is sufficient to trigger most overflow issues. Strings with a lower number of characters are not necessary because they are included implicitly. Using more than 10000 characters may lead to the server rejecting the stanza. The default value for the maximum stanza size is 65535 bytes with jabberd2. Setting the maximum size lower than 16384 bytes is not recommended by the jabberd2 developers.
A special case of the overflow mutator is the "off-by-one" mutator. By parsing the maxLength restrictions of all XMPP schemas, we discovered that three length constraints are explicitly defined: 1023 bytes, 1024 bytes and 3071 bytes. We therefore added strings with a length of 1024, 1025 and 3072 bytes to our mutator. Although there was no restriction of 2046 bytes, we added a 2047 bytes string to the mix, since the maximum length of the bare JID (i.e. a JID without the resource part, e.g. user@domain.com [34]) is two times 1023 characters and the '@' sign.

5.7.3. Invalid Char Mutators

While the "General String Mutators" are applicable to all *String* elements, the "Invalid Char Mutators" are specific to a certain datatype. In order to cover all existing XMPP specific datatypes we analyzed the XMPP schemas, collecting all element types. Table 5.1 lists types defined in the schemas and offers a short description. Types without specific limitations other than the common string restrictions are covered by our General String Mutator and therefore excluded from the list.

5.8. XMPP Schema Coverage: Input Space Determination

XMPP together with its numerous extensions form a huge protocol with hundreds of different elements. It therefore is not practical to manually determine the protocol coverage. Since XMPP is XML based, schemas are available which describe the structure of valid elements. Our goal was therefore to process all XMPP schemas, extracting all

Type	Description	Invalid Chars	
xs:base64Binary	Base64 encodable characters	all except #x20, A-Z, a-z, 0-9, =, +, /	m
xs:anyURI	Uniform Resource Identifier Reference (URI)	see RFC 2396 Appendix A[35][36]	m
color	Contains color strings or RGB color values	all except #, 0-9, A-F, a-f; hexadecimal part must be 6 characters long	m
xs:NCName	non-colonized name	':'	m
fullJIDType	see section 5.6		d
xs:dateTime	see section 5.6		d
ArrayType			(d)
candidateElementType			(d)
contentElementType			(d)
cryptoElementType			(d)
linkType			(d)
MemberType			(d)
messageType			(d)
mutingElementType			(d)
remoteCandidateElementType			(d)
StructType			(d)
ValueType			(d)

Table 5.1.: Types defined in the XMPP schemas. Types denoted by 'm' are covered by specific Mutators, types denoted by 'd' are represented in a *DataModel*. Types denoted by (d) could be described in a *DataModel* if a schema-based stanza creating approach was used.

possible XML constructs defined therein.
In order to get a read of the complexity of the XMPP protocol and thus being able to calculate the protocol coverage, we have to determine how many semantically different stanzas can be generated from the schemas. Figure 5.4 shows an example of two different, yet semantically possibly equal stanzas. As a first step to determine the number of possible stanzas, all root XML elements, i.e. elements which are not referenced by others, have to be identified. Figure 5.5 shows an example schema with 2 root elements. However, the number of root elements is not a true indicator of the overall protocol coverage for multiple reasons.
Problem 1: Some root XML elements discovered may not be client requests and therefore not add up to the total coverage, if the scope is client to server communication only.
Problem 2: The root elements may exist in multiple flavors which do not necessarily need to be equal in terms of protocol semantics. E.g. a stream containing an iq stanza, which

```
stanza 1:
<presence>
<x xmlns="vcard-temp:x:update">
<photo />
</x>
<show>xa</show>
<status>Give it up, I'm not in!</status>
</presence>

stanza 2:
<presence>
<show>xa</show>
<status>Give it up, I'm not in!</status>
</presence>
```

Figure 5.4.: Two different stanzas that are probably semantically equal, since the photo tag in stanza1 is empty.

```
<xs:schema>

  <xs:annotation>
    This is just an annotation, it therefore cannot be a root element.
  </xs:annotation>

  <xs:element name="A" type="xs:string" />

  <xs:element name="B">
    <xs:sequence>
      <xs:element name="B1" type="xs:string"/>
      <xs:element name="B2" ref="C"/>
    </xs:sequence>
  </xs:element>

  <xs:element name="C" type="xs:dateTime">

</xs:schema>
```

Figure 5.5.: A simplified example schema containing 3 potential root elements: 2 definitive root elements (A, B) and one referenced element (C).

```
<xs:schema
...
  <xs:element name='stream'>
    <xs:complexType>
      <xs:sequence xmlns:client='jabber:client'
                   xmlns:server='jabber:server'>
      ...
        <xs:choice minOccurs='0' maxOccurs='1'>
          <xs:choice minOccurs='0' maxOccurs='unbounded'>
            <xs:element ref='client:message'/>
            <xs:element ref='client:presence'/>
            <xs:element ref='client:iq'/>
          </xs:choice>
          ...
```

Figure 5.6.: Excerpt from the stream.xsd schema.

```
<xs:schema
...
  <xs:element name='activity'>
    <xs:complexType>
      <xs:sequence>
        <xs:choice>
          <xs:element name='doing_chores' type='general'/>
          <xs:element name='drinking' type='general'/>
          <xs:element name='eating' type='general'/>
          <xs:element name='exercising' type='general'/>
          ...
```

Figure 5.7.: Excerpt from the activity.xsd schema. Choice elements do not deserve an own root element because they are the same in terms of protocol semantics

can be used to register with a server, is completely different from a stream containing a message stanza used to send messages to other XMPP entities, as shown in figure 5.6. All root elements containing multiple choices may therefore represent multiple semantically distinct root elements.

However, if we were to treat every choice as part of an own root element, the number would be too high as figure 5.7 illustrates. We therefore determine whether choice elements are trivial or complex depending on whether the choice elements have references to other elements or whether they have children. Complex choice elements are treated as part of an own root element, while trivial elements do not increase the number of root elements. E.g. in figure 5.6 the stream root element would be counted three times (i.e. as three semantically different root elements) since it contains three non-trivial choices. Problem 3: Another factor contributing to the number of root elements are the quantifiers "minOccurs" and "maxOccurs" which describe if an element needs to occur and how many times it may occur. We discovered that minOccurs is either 0 or 1 and maxOccurs

Root Elements (w/o consideration of quantifiers, choices)	209
Root Elements (w/ non-trivial choices, w/o quantifiers)	263
Root Elements	336

Table 5.2.: The number of root elements, i.e. semantically different stanzas, as determined by xmpp_schema_analyzer.py using all standard XMPP schemas from xmpp.org. The protocol coverage for a fuzzer containing one stanza is 1/<number of root elements>.

is either 1 or unbounded. If an element is repeated it will probably not change the protocol semantics. We therefore do not evaluate if the maximal occurrence is infinite but only if an element is optional, i.e. if it contains "minOccurs=0". The presence or absence of optional elements possibly changes the protocol semantics. Unfortunately XML schemas are unable to define if a certain element needs to be present depending on the presence of other elements, i.e. the restriction that an optional element A is present if and only if the optional element B is present, can not be modeled in an XML schema. We therefore have to assume that all combinations of optional elements are possible which yields 2^n possibly different root elements, with n being the number of times "minOccurs=0" appears in the respective root element. If all quantifiers are considered, the number of semantically distinct stanzas is way too high, i.e. in the neighborhood of 25 millions, mainly because of a handful of schemas containing large numbers of quantifiers. Closer inspection revealed, that those schemas contained lots of elements with basic types, i.e. xs:string, xs:decimal, etc. Those kinds of types can be ignored when determining semantically different stanzas, since their presence or absence will not change semantics.
Using schemas to determine the protocol coverage will yield an upper boundary of potentially different elements. We therefore expect the real coverage to be higher than the one determined using the above mentioned automatic schema analyzing approach, regardless of which criterion of root elements listed in table 5.2 is applied.

5.9. Server Feature Coverage

We base the jabberd2 server feature coverage, i.e. all stanzas, RFCs or XEPs the server supports, on the information available on the developers website [37]. We discovered that even if we covered all stanza described in the public schemas, we would nonetheless miss a significant number of features present in the server. Many XEPs are not part of the standards-track specification within the XMPP Standards Foundation's standards process and are therefore omitted from the schemas. Table 5.3 lists all XEPs supported by jabberd2 and if schema or DTD descriptions are available. Approximately a third of all XEPs supported by jabberd2 do neither contain a schema nor a DTD. Further examination of those XEPs has shown that they mostly nonetheless contain stanzas. Those XEP do therefore not provide field format information, but this information might in most cases already be contained in one of the schemas contained in another XEP or in the XMPP Core RFC. In order to increase the server feature coverage we implemented the xep_ripper.py which allows us to extract stanzas and schemas contained in the XEPs and therefore increase the server feature coverage.

XEP	schema
XEP-0012	*
XEP-0016	*
XEP-0022	*
XEP-0023	*
XEP-0030	*
XEP-0048	*
XEP-0049	*
XEP-0054	dtd
XEP-0073	
XEP-0077	*
XEP-0078	(*)
XEP-0079	*
XEP-0083	*
XEP-0086	
XEP-0090	(*)
XEP-0091	(*)
XEP-0092	*
XEP-0093	(*)
XEP-0114	*
XEP-0128	
XEP-0138	*
XEP-0145	*
XEP-0153	*
XEP-0157	
XEP-0160	
XEP-0170	
XEP-0175	
XEP-0178	
XEP-0185	
XEP-0190	
XEP-0191	*
XEP-0192	
XEP-0193	
XEP-0198	*
XEP-0199	*
XEP-0202	*
XEP-0203	*
XEP-0205	
XEP-0209	(*)
XEP-0212	
XEP-0216	
XEP-0220	(*)
XEP-0225	(*)
XEP-0232	
XEP-0237	*
XEP-0238	
XEP-0243	

Table 5.3.: All XEPs supported by jabberd2.
XEPs which are described in a standard XML schema are denoted by '*'.
XEPs which are described in a non-standard XML schema are denoted by '(*)'
XEPs which are described in a DTD are denoted by 'dtd'

5.10. Code Coverage, Fuzzer Tracking

Code coverage is a measure in software testing for quantifying the degree to which the source code of a program has been tested. The granularity level of code coverage, also known as the basic coverage criteria, can be categorized into function coverage, decision/condition coverage and statement coverage.

Since fuzzing is negative testing, we are looking for a related measure quantifying the degree to which the source code of a program has been fuzzed. Instead of code coverage the term fuzzer tracking may be better suited [7]. Fuzzer tracking is directly dependent on input space and can be used to cross reference with test cases.

Fuzzer tracking has two overall goals: firstly to improve the fuzzer through a feedback effect, since often flaws in software occur in seldom used features, and secondly to know when to stop fuzzing, since in the optimal case we can stop as soon as all the code has been covered.

In white-box fuzzing, the approach is simple. The target under evaluation is monitored with a coverage tool integrated in common IDEs. In grey-box fuzzing, no source code is available, only assembly code. The typical approach is to work backwards and examine the log and attach a debugger to pinpoint the location of the fault. An alternative approach would be to work forward, actively monitoring the set of instructions in response to a fuzz request. But there exits a trade-off between inspection depth and speed. Attaching a regular debugger that covers individual statements to the target in the forward approach is resource intensive and consequently has a negative impact on fuzzing performance, especially when the target or input space are large.

The general idea in grey-box fuzzing could be described as flagging: flags inserted throughout the code are hit during program execution in response to a fuzzing request which yields the location execution takes place and in what order. Disassembled binaries can be visualized as control flow graphs (CFG), where each function is a node and each call is an edge between nodes. Setting flags on each node yields function coverage. If you do not want such a coarse grained measure, we could visualize each basic block as a node, and branches between basic blocks as edges. So what is a basic block? In assembly language, a basic block is the sequence of instructions run in order. The following example illustrates the issue [7].

```
00000010 sub_00000010
00000012 push ebp
00000014 mov ebp, esp
...
00000020 jz 00000050
00000022 mov eax, 0Ah
...
00000050 xor eax, eax
00000052 xor ebx, ebx
```

This example subroutine has 3 basic blocks. The first one is until the jump (jz) instruction, the second until the join of the branch and from there, the 3 block begins. A code coverage tool now marks the start address of each block and registers hits as soon as the code is executed. The judgement of this approach is balancing inspection depth versus performance.

Our approach was to find a coverage tool for grey-box fuzzing. Assuming that a closed source XMPP server has similar functionality as an open source server, to tune our fuzzer using the feedback effect on that server. In addition, comparing the white-box code coverage tool to the grey-box coverage tool on jabberd2 should demonstrate the effectiveness of the coverage tool.

5.10.1. PaiMei

PaiMei is a tool promoted in [7]. It combines a python debugger, called pydbg, with a fuzzer tracker, called pstalker. PaiMei relies on a disassembler to generate the CFG graph. The binary target is therefore loaded into IDA Pro, and a Python plug-in, pida_dump.py, generates the graph known as a PIDA file. PaiMei then loads the PIDA file and attaches itself to the target process. Subsequent hits are saved into a backend storage, in this case a MySQL database, see chapter 12.2. PaiMei enables filtering hits in order to avoid double counting.

PaiMei is very much dependent on the static analysis of IDA Pro, and misinterpretations, e.g. mistaking data for code, have a significant impact on the value of code coverage as a measure.

5.10.2. White-box code coverage

In order to fuzz jabberd2, for which the source code is available as a Visual Studio solution [38], we found a coverage tool built directly into the IDE, see chapter 12.1. Unfortunately, being an open source server targeted primarily for Unix platforms, the Windows build proved to be a real obstacle. After a set limit and an unsuccessful search for help, we decided to cancel any further attempts to build jabberd2. Therefore, coverage results originating from PaiMei can not be benchmarked against white-box code coverage results. Nevertheless, jabberd2 also provides debug symbols, so using PaiMei as a coverage tool at least yield the covered function names during a fuzz request. Using these and inspecting the source code inside a function, the feedback effect of code coverage is possible, although manual and labor intensive.

5.10.3. Grey-box code coverage

Using PaiMei as the coverage tool, a first attempt was to review the plausibility of results by using simple C programs. We implemented a few programs with conditionals and loops to check that PaiMei shows the matching number of blocks build into the program. However, even a simple C program with no statements and including no header files results in a block count of around 140. The probable reason being the Microsoft C compiler and linker. A search for possible mediation proved unsuccessful. As a consequence, the code coverage measure as an absolute asks for caution and servers only as a indication.

5.10.4. Results

Code coverage is an important part of fuzzing, and the feedback effect helps tune fuzzer precision. Unfortunately, the missed build of jabberd2 makes comparing the effectiveness of PaiMai against white-box code coverage impossible.

As for the block count issue, a possible solution could mark or remove blocks originating from compiler or linker inclusion, assuming that the Microsoft C compiler always includes the same system functions or places the code at the same base address. This could bring the block count near or even matching the effective block number.

5.11. Logger Considerations

Peach enables creating custom loggers in order to log more specific aspects. During analysis of our fuzzing logs we missed accounting for the actual database content. This may be helpful in the case of rapidly reproducting an application error without having to run the complete fuzzing test case again. If the database content is known right before the crash, it might be possible to reproduce errors by sending only a single packet if the database's state is exactly reproduced.

6. Results

6.1. Benefits of Fuzzing

- Unconventional Thinking: Fuzzing might exhibit flaws that would not be otherwise found by tailoring and checking abuse cases. With the use of artificial intelligence, mutators might be more powerful than their manually engineered counterparts.
- Complexity: A fuzzer is not plagued by very complex software, poorly documented or cryptically written code
- Dynamics: Certain flaws can only be detected dynamically, e.g. in order to trigger race conditions a fuzzer may prove valuable.
- Time restraints: If the time frame to test a piece of software is too short, fuzzing can be used in addition to manual examination. Especially neuralgic spots can be investigated in depth with the addition of a fuzzer.
- Speed of Execution: A fuzzer is an order of magnitude faster than a manual tester.
- Hacker's perspective: A hacker aiming to exploit a certain application, without having access to source code, either takes the hard route using a disassembler or the more convenient way with a fuzzer to cover more code in a smaller amount of time. For tiger teams a fuzzer might therefore prove valuable.

6.2. Limitations of Fuzzing

6.2.1. Time-Detail-Precision Tradeoff

While fuzzing we constantly had to weigh precision against performance and thus time we wanted to invest for a fuzzing run. More precise mutators means introducing more mutations which in turn will lead to longer fuzzing runs. The same holds true for the level of detail. The more detail a fuzzer has to explore, the longer a fuzzing run takes.

6.2.2. How Secure Is A Fuzzed Software?

In general, even thorough fuzzing will not guarantee flawless software. At some point the decision has to be made where fuzzing makes the most sense and omit the remaining parts when fuzzing larger software projects. The degree to which a software has been examined depends on the level of detail the protocol model exhibits as well as the quality of the mutations performed. While substitution is very fast the results will probably be inferior to those gained through random, yet intelligent mutation using genetical algorithms. By being able to gain knowledge over time on which mutations perform best, the mutators turn increasingly efficient. Unfortunately we did not have time to compare our fuzzing results against those of a fuzzer using artificial intelligence.

6.2.3. Misconfiguration

There is no practical way to determine if certain misconfigurations will render an otherwise stable server vulnerable to denial of service or other attacks. A pair of identical requests could end in a different outcome on a pair of identical but divergently configured servers.

6.2.4. Automated Fuzzer Creation

When dealing with extensive protocols and time is a limiting factor, an all-in-one test run will induce too great of an overhead for multiple reason:

- Relaying: A server might not interpret or process all data being sent. An attempt to trigger an overflow using a message body may cause harm to another client but not affect the server at all.
- Bug Analysis: The larger a fuzzer gets, the more post fuzzing analysis will be required in case of found bugs. The logs substantially increase in size and the prerequisites might change after each PDU sent to the server. It might be more convenient to have fewer server state changes.
- Unnecessary Requests: If the semantics of a request is not evaluated and if the server configuration is not adapted accordingly, the fuzzer might end up with lots of unnecessary requests. When attempting to request the last time a user X logged out from a server, results will vary depending on whether the user exists or not. Depending on the server policy it will most likely be impossible to request data from users that did have not authorized the fuzzer as a buddy.
- Causality: When too many states are involved, tracking and managing dependencies is very time consuming or even impossible. If no dependencies are respected, the time to thoroughly test all possible paths is in $O(n!)$ with n being the number of states.

6.3. Strategies for Efficient and Effective Fuzzing

6.3.1. Selection of Features

If time is limited, the best bet for finding bugs is to determine the least used features, custom server features or non-standard compliant features. For every test cases must be geared towards efficiency by focusing on potentially vulnerable spots and any protocol restrictions need to be considered in order not to loose any time.

```
<DataModel name="StreamReqStanza">
  <String value="&lt;stream:stream to=&quot;" mutable="false" />
  <String name="IPAddress" value="127.0.0.1" />
  <String value="&quot; xml:lang=&quot;de&quot;
    xmlns=&quot;jabber:client&quot;
    xmlns:stream=&quot;http://etherx.jabber.org/streams&quot;
    version=&quot;1.0&quot;&gt;" mutable="false" />
</DataModel>
```

Figure 6.1.: A simple stream request stanza containing an IP address string to be fuzzed

6.3.2. Mutator Quality

While code coverage is an indicator of how much server code has been executed, there is no information about the accuracy of the mutations performed. It might be the case that even though coverage is at 100% not all bugs were found. Whether or not bugs are detected essentially depends on the mutator's precision. With substitution-based mutations the values have to be chosen carefully, preferably based on a multitude of server implementations. For best results, each protocol will require a custom set of mutators for effective fuzzing.

6.3.3. Modeling Detail

The level of detail of a *DataModel* determines the fuzzer's effectiveness. The more detail a model covers, the deeper a fuzzer can reach into the target's code. However, the depth of modeling protocol detail heavily affects the time a fuzzing test will be running. Given the stanza in figure 6.1 and assuming a fuzzer that performs a thousand mutations per value, in total a thousand mutations are performed. In comparison, the same stanza with an IP address fully modeled as depicted in figure 6.2, would cause an additional 3000 mutations, since every octet is mutated separately. With the IP address possibly being part of multiple stanzas, modeling details in its entirety, such as the IP address, will introduce significant overhead.

6.3.4. Features of an Optimal Fuzzing Framework

Besides qualities desirable in every application, such as good memory management and good performance, there are a number of features a fuzzing framework should offer to render fuzzing as effective as possible and to reduce the time required to create a fuzzer. In this section we will list the criteria and features a fuzzing framework should possess based on the experience we had while fuzzing. We will evaluate whether Peach meets these criteria and in order to determine strengths and weaknesses.

- High Automation: The fuzzing framework should offer as many automation features as possible, a user should not have to worry about manually integrating external analysis tools.

```
<DataModel name="StreamReqStanza">
  <String value="&lt;stream:stream to=&quot;" mutable="false" />
  <String ref="IPAddress" />
  <String value="&quot; xml:lang=&quot;de&quot;
    xmlns=&quot;jabber:client&quot;
    xmlns:stream=&quot;http://etherx.jabber.org/streams&quot;
    version=&quot;1.0&quot;&gt;" mutable="false" />
</DataModel>

<DataModel name="IPAddress">
  <String name="FirstOctet" value="127" />
  <String name="IPOctetDelimiter" value="." mutable="false" />
  <String name="SecondOctet" value="0" />
  <String ref="IPOctetDelimiter" />
  <String name="ThirdOctet" value="0" />
  <String ref="IPOctetDelimiter" />
  <String name="FourthOctet" value="1" />
</DataModel>
```

Figure 6.2.: A stream request stanza containing an IP address defined in a separate *DataModel*

Peach offers a variety of modules to include external debuggers and diagnostic tools. However, the greatest issue with Peach might be the missing code coverage and tracing functionality. In order to get a reading of our fuzzer's coverage we always had to perform a pair of runs: the first for fuzzing and a second to determine the coverage.

- Separation of Concerns: Different concepts should be separated for simplified fuzzer management, maintenance and fuzzer reuse.
  In this regard Peach is very well designed. It separates state modeling from data modeling on a conceptional level and is architected to be very modular by providing an interface to include custom components without having to understand how Peach works internally.

- Cross-Platform Capability: The framework should be deployable on different operating systems and platforms to be suitable for as many target applications as possible.
  Due to the fact that Peach is written in Python, it supports Windows, Macintosh and Unix systems alike. An interesting feature might be an agent that is deployable on a mobile phone to fuzz mobile applications.

- Effective Mutations/Heuristics: For efficient and effective fuzzing, the fuzzing framework should already provide proven heuristics and mutations. Ideally it offers customized mutators for a wide range of different protocols to be able to cover specific protocol characteristics.

This is another perceived weakness of Peach. While general mutators are available that probably are suitable for most applications, specifically tuned mutators perform significantly better due to less overhead. The implementation of customized mutators falls to user.

- Interface for Easy Fuzzer Creation: The fuzzer should provide a thorough documentation and an easy to use interface to create fuzzers. Ideally, a user should not need programming language skills to assemble a fuzzer.
  The basic design of Peach provides an interface to model fuzzers using XML data. However, to fully take advantage of some features, Python knowledge is indispensable.

6.3.5. Combining mutation-based and generation-based fuzzing

Modeling a large protocol takes a fair amount of time in which no vulnerabilities are found. A possible solution could be combining two different fuzzers, or employing a fuzzer capable of both, to perform mutation-based fuzzing while modeling the protocol for generation-based fuzzing, provided the protocol allows for mutation-based fuzzing, i.e. ASCII-based protocol. Chances are that a vulnerability is found even before protocol modeling ends.

6.4. Future Work

6.4.1. Mutators

- Mutator improvement through genetic algorithm generation and selection may yield better and more random mutators. Generate a lot of mutators and determine which mutators perform best on a wide range of servers.
- Determine and improve effectiveness of our mutators.
- Implement mutators that modify (i.e. mutate) given data intelligently (i.e. without producing a lot of overhead) instead of just substituting.

6.4.2. Fuzzer Tracking / Code Coverage

- Build jabberd2 from source on Windows in order to trace the fuzzer in code.
- Determine how IDA Pro identifies functions and blocks to fully understand coverage metrics.
- Extend PaiMei to create reports based on instructions instead of blocks or functions to get a better estimate of the code coverage.
- Evaluate if inspecting instruction execution is practical. If that's the case, are there any restrictions?
- Find a solution to integrate PaiMei into Peach to be run automatically.

6.5. Personal Reports

6.5.1. Kevin Lynn

Fuzzing is an exciting topic in IT security. The fact that while testing a program has potentially exponential runtime, which would suggest not even following this route from the beginning, in reality numerous vulnerabilities are found in software which demonstrates the potential fuzzing has. On the other hand, the fact that finding so many bugs result from fuzzing leads to the conclusion that a lot of software is of really bad quality. This suggests that fuzzing, which is also known as negative testing, must be included into every software development lifecycle.

As a beginner in fuzzing the project showed just how much work a security expert tucks into actually finding a flaw and how seemingly easy it is in retrospective. Using fuzzing as a judge, assessing the robustness of a software product and being proven right by the fuzzer is a satisfactory experience. Also, exploring software development at a deeper, basic level than normally taught by introduction programming classes fostered my interest.

Unfortunately, the brief project time frame made it impossible to go into any depth and proved me that so much more knowledge is necessary to produce any real facts. Occasionally, acquiring knowledge in an alien subject and not being able to ask anybody for guidance felt quite lonely.

I would like to personally thank Mike for his continued commitment, especially since I was working on another project in parallel. His ability to solve problems and produce results so quickly stunned my on many occasions and deserve recognition.

6.5.2. Michael Fisler

Overall, this was a very interesting project. The topic covered an interest of mine, IT security and the project description allowed for a lot of freedom to experiment with different previously unknown concepts and technologies. During the project I learned Python, a programming language that has a lot of interesting concepts and features I have never seen in other languages. The project was a nice addition to the module “Internet Security 2” that I was enrolled in, allowing me to explore many subjects hands-on. After the first few weeks of research I was overwhelmed with the sheer number of risks that we faced and had serious doubts that we would be able to conquer everything. All in all I am pleased with what we accomplished, except for the fuzzing part, where I would have liked to invest some more time at the expense of writing utility scripts. Unfortunately we did not find any bugs and therefore could not examine them.

Lessons learned

In some instances much time was lost debugging the scripts I wrote. I should have refactored certain scripts much earlier.

Another problem was that I consulted the Python style guide relatively late. By then I had already produced a huge amount of code which needed to be changed. Some conventions of the PEP 8 style guide resulted in code with a much better readability and maintainability and would therefore have aided faster development.
Python is a valuable tool to quickly write code but not suitable for bigger software projects. Python's memory management is hideous.
Although I have used LaTeX for two previous projects, I was again confronted with numerous time consuming peculiarities. But on the whole, the result was worth the effort.

6.6. Acknowledgments

We wish to thank Prof. Dr. Steffen for the support and advice we received during the course of this thesis. We appreciate the time he took to attend our some times prolonged weekly meetings. He contributed to our decision making process and gave us pointers where to improve our work and what to focus on.

We also like to thank Cyrill Brunschwiler from Compass Security AG who gave us a lot of valuable feedback about our work and especially our documentation.

Part II.

Software Documentation

7. Overview of Tools

In order to facilitate the fuzzer creation process and to automate certain analysis tasks we created several Python scripts.
Additionally, we created several Peach extensions to be able to fully harness the potential and evaluate the limits of the Peach framework. The following sections contain a brief description of every tool we created.
In order to run our scripts, Python 2 has to be installed.
The scripts have only been tested on Unix-based systems, there is no guarantee they will run correctly on Windows. For usage and additional information every script has a help option: 'python <scriptname> -h'

7.1. Fuzzer Assisting and Analysis Tools

7.1.1. config.py

Purpose

Configuration file parser and writer module.

Description

The config module allows to store information needed for fuzzer creation, as well as for fuzzing tests, in a configuration file. The configuration file ensures that global parameters, such as the fuzzing target's IP address, the username and password the fuzzer uses and the like, are in sync throughout all fuzzer assisting scripts and therefore in all fuzzers.
The config writer is currently unused and therefore not tested.

fuzzer.cfg

The fuzzer configuration script used by config.py is by default named fuzzer.cfg and has to reside in the same directory as config.py, preferably in <peachdirectory>/CustomModules. Make sure not to delete existing entries since the scripts relying on config.py will not check whether an entry exists, in order to avoid additional code. Missing entries will therefore lead to a crash if accessed. It is possible to add custom comments by using a new line, beginning with a '#'.

7.1.2. xmpp_schema_analyzer.py

Purpose

Schema analysis and stats.

Description

The schema analyzer tool has two operation modes. In default mode, it allows to gather stats to gauge the number of XML instances that can be generated from a schema. In alternative mode all types, attributes and the maximum length restrictions contained in the schemas can be extracted. The types are a means to improve fuzzing depth. By analyzing the types that could be defined in a *DataModel* or the restrictions that apply to certain types, more thorough and precise mutators can be created. Attributes can be used to determine kinds of restraints used in the schemas. These restraints may affect the number of potential XML instances resulting from a schema.

7.1.3. jabberd_mysql_reset.py

Purpose

Resetting the database jabberd2 uses.

Description

The jabberd MySQL reset script will attempt to clear, i.e. truncate, all existing tables. All persisted data on the server will be deleted. It can be used to achieve reproducibility if run at the beginning of each fuzzing run. Different servers probably use different database layouts, therefore this script will probably only work with databases created for the jabberd server. If the script is used remotely, the target MySQL database has to be configured accordingly, granting permissions to the remote jabberd user.

7.2. Pit Generation Scripts

7.2.1. pit_creator.py

Purpose

Automatic generation of fuzzers based on XEP examples.

Description

The pit creator creates entire Peach Pit files, using all scripts from figure 7.4 but schema2xml.py. By default all XML attributes will be set to mutable and all available tags will be applied (see xep_ripper.py in subsection 7.2.2 for details about tags), to reduce the number of resulting stanzas. The SASL authentication method used by

the fuzzer will be set to “PLAIN” unless defined otherwise. If certain servers or configurations require an alternative method, “DIGEST-MD5” or “PLAIN” may be passed as an argument. Pit creator saves all temporary data to a temporary folder named ./pit-tempdir-<creation date/time in epoch>, a name which should almost certainly be unique on the host system, allowing removal upon completion. The directory and its entire contents are deleted from the hard-disk regardless if the were already present on the filesystem previously to starting the script.
In order to keep the temporary folder and its contents for debugging or further analysis the ’-x’ argument may be passed.
By default all custom mutators defined in the configuration file (fuzzer.cfg in subsection 7.1.1) will be enabled. Modifying the config file or passing mutators with the ’-m’ argument allows to use a subset of all defined mutators. If ’-m default’ is passed, Peach’s default mutators will be used.

7.2.2. xep_ripper.py

Purpose

Rip stanza content from XEP files, tag useless or destructive stanzas.

Description

A lot of examples are included in the XEPs. In order to facilitate managing the input space coverage, the xep ripper will extract all well-formed examples contained in the XEPs. In order to reduce the number of stanzas to process, the following tags can be prepended to the stanza’s filename:

- [*error*]: For all stanzas containing the keyword “error”. Errors are in general only sent by servers.
- [*result*]: For all stanzas containing “type=’result”’. Results are in general a response to a client request and therefore a stanza sent by servers.
- [*SASL*]: For all stanzas containing the keyword “sasl”. SASL stanzas should be used for authentication negotiation and may disrupt the fuzzing process if sent after successful SASL authentication.
- [*stream*]: For all stanzas containing the closing stream tag (</stream:stream>), causing the fuzzing run to interrupt.
- [*TLS*]: For all stanzas containing the xmpp-tls namespace. TLS negotiation is supposed to be done before authentication and might, depending on the server implementation, disrupt fuzzing.

The following tags will always be applied:

- [*fuzzdelim*]: For stanzas which contain the fuzzing delimiter defined in the config file. Stanzas already containing fuzzing delimiters would not be handled correctly due to the implementation in xml2pdm.py 7.2.6.
- [*unregister*]: For stanzas which contain the namespace "jabber:iq:register" and the "remove" keyword. Such stanzas would unregister the fuzzer and disrupt the fuzzing process.

Use caution when processing the ripped schemas, at least one instance of the ripped schemas is not well-formed due to errors in the respective source!
In xmpp.org/extensions/xep-0309.html both schemas are invalid since <xs:sequence> tags are closed with </xs:choice> .

7.2.3. configure.py

Purpose

Define fuzzing areas, assign configured values.

Description

If default settings are used, the configure script will set the values of all XML attributes to mutable, surrounding them by the fuzzing delimiters stored in the config file (using config.py from subsection 7.1.1). If attributes should be excluded from fuzzing, they can be passed with the '-n' argument. E.g. running 'configure.py -d /Users/fuzzer/stanzas/ -n from,to' will set all values except the "to" and "from" values to mutable in all stanza files contained in /Users/fuzzer/stanzas/. Figure 7.1 shows an example of how such a stanza may look like, with the premise that the fuzzing delimiter is set to '*'.

```
<stream:stream to="jabber@srv" xml:lang="*de*" xmlns="*jabber:client*"
  xmlns:stream="*http://etherx.jabber.org/streams*" version="*1.0*">
```

Figure 7.1.: A stanza where values have been surrounded by the fuzzing delimiter '*'. Values that are not surrounded by '*', in this case the to-value, will be set to immutable in the *DataModel*.

In the config file, default values can be set. In order to fuzz successfully certain values like the "from" value need to be set to a specific string. Using the config.py module, certain values can be replaced by the default value stored in the config file.

7.2.4. tag_dupl.py

Purpose

Tagging duplicate stanzas.

```
<StateModel>
  ... Authentication State...
  <State name="xep-0012Ex1">
    <Action type="output">
      <DataModel ref="xep-0012Ex1" />
    </Action>
    <Action type="input">
      <DataModel ref="ServerResponse" />
    </Action>
    <Action ref="xep-0012Ex10" type="changeState" />
  </State>
  <State name="xep-0012Ex10">
  ...
</StateModel>
```

Figure 7.2.: An excerpt from a (simplified) generated *StateModel* created by xml2sm.py. It contains the output *DataModel* and an input *DataModel* to be able to parse the server response.

Description

Analysis of the extracted stanza files has shown that files with identical content existed. After applying default values to the stanza files by using the configure.py script from subsection 7.2.3, additional duplicates might be generated. Sending the same stanza twice does not make much sense from an efficiency standpoint, therefore duplicates can be tagged with [*dupl*] to avoid unnecessary redundancy.

7.2.5. xml2sm.py

Purpose

Create StateModel based on the stanzas provided.

Description

The xml to statemodel tool will create an entire *StateModel* including the authentication part. It uses the stanza file's names as *DataModel* reference names, since they are unique by default. Figure 7.2 shows an excerpt of a generated *StateModel*, assuming that the XEP stanza files xep-0012Ex1.xml and xep-0012Ex10.xml were present in the input directory. Stanza files have to comply to the filename convention xep-<xep nr>Ex<example nr>.xml or else they will be ignored. Make sure there are no junk (e.g. empty) files in the input directory or else states are created for non-existing *DataModel*s.

```
stanzafile.xml:
<stream:stream to="jabber@srv" xml:lang="*de*" xmlns="*jabber:client*"
  xmlns:stream="*http://etherx.jabber.org/streams*" version="*1.0*">

stanzafile.pdm:
<DataModel>
<String value="&lt;stream:stream to=&quot;jabber@srv&quot; xml:lang=&quot;"
  mutable="false"/>
<String value="de"/>
<String value="&quot; xmlns=&quot;" mutable="false"/>
<String value="jabber:client"/>
<String value="&quot; " mutable="false"/>
<String value=" xmlns:stream=&quot;" mutable="false"/>
<String value="http://etherx.jabber.org/streams"/>
<String value="&quot; version=&quot;" mutable="false"/>
<String value="1.0"/>
<String value="&quot;&gt;" mutable="false"/>
</DataModel>
```

Figure 7.3.: A stanza file (stanzafile.xml) where values have been surrounded by the fuzzing delimiter '*' and the resulting *DataModel* (stanzafile.pdm).

7.2.6. xml2pdm.py

Purpose

Create DataModels based on the stanza files provided.

Description

The *DataModel* creation script translates XML files into Peach *DataModel*s using *String* elements. In inverse mode *DataModel*s displays or writes the XML content of the difficult to read *String* elements to the console or to a file. If supplying stanza files manually make sure they are well formed and contain valid XML data. The script will not validate the input files. All values surrounded by fuzzing delimiters will be set to mutable as shown in figure 7.3. Avoid using fuzzing delimiters that are contained in the stanza files or else the result might be unusable.

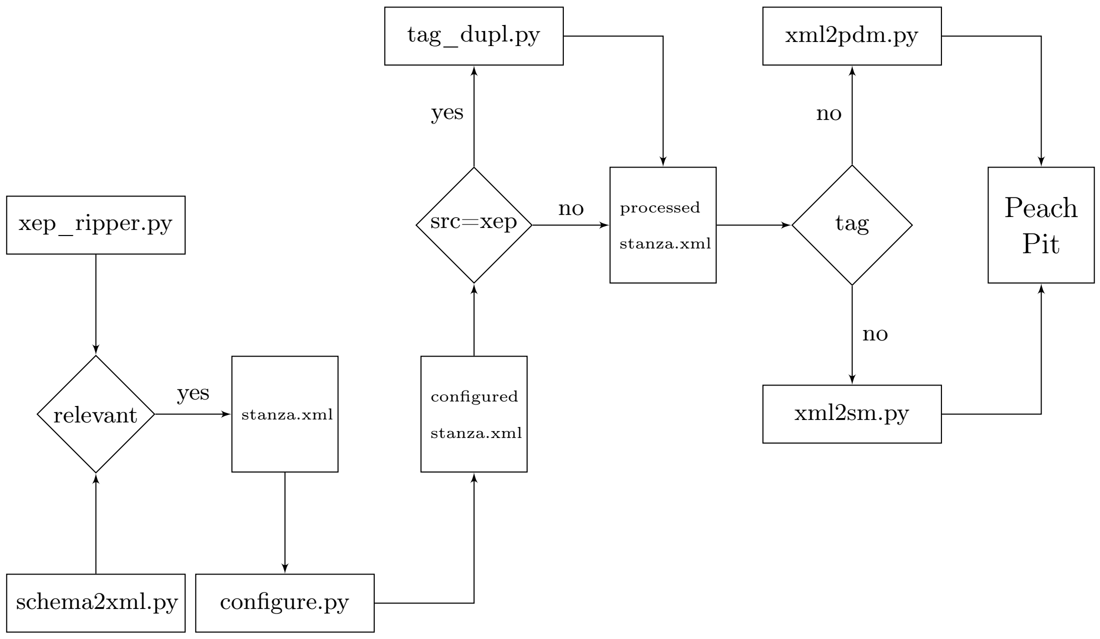


Figure 7.4.: XML Processing: Generation of the Peach Pit file's *DataModel* and *State-Model* sections based on the XML content extracted from the XMPP schemas or XEP files.

8. Software Decisions

8.1. Software Engineering Decisions

8.1.1. Script Robustness

To reduce the development time of our scripts as well as their complexity we do not strive for robustness. We are well aware that in certain cases, our scripts may crash using maliciously engineered source files. For example we assume that every schema xs:element must have a name attribute. Since the xs:name attribute defines the name of the respective XML element, this assumption may hold true with all usual schemas. However, feeding our schema processing script an xs:element without a name attribute will lead to a crash.

8.1.2. Script Scope

While we attempt to make the scope of our scripts as extensive as possible, enabling the scripts to be applicable to other XML based protocols too, we do not check every XEP if our resulting XML files are semantically distinct and correct. Reading all of the hundreds XEPs would consume far too much time.

8.1.3. Coding Style

In terms of coding style we decided to adhere to the common PEP 8 style guide for python code [39]. One reason is the fact that the majority of Python programmers adopt these conventions and are used to read code written using PEP 8. Other reasons are the superior readability and maintainability. For example using a 4 space indentation seems tedious at first, compared to using tabs, but with larger scripts the readability suffers from the excessive line wrapping. Mixing tabs and spaces for indentation is a bad idea. During refactoring and especially if using different editors, indentations might get changed automatically, which due to the nature of the Python language, induces bugs which may not be easily detected if the whole script is not tested again.

8.1.4. Code Smells

Our scripts exhibit a number of code smells coined by Kent Beck and later used by Joshua Kerievsky in his book Refactoring to Patterns [40]. The main reason being, that we just use our scripts as expedients to assist fuzzing and for analysis purposes. Because of their very specific nature, chances of further development are low.

Primitive Obsession

In our scripts there are multiple instances of the “Primitive Obsession Smell” [40]. This is either because dedicated XML classes would have introduced an unnecessary overhead or because in some instances Python’s XML libraries do not satisfy our needs. With ElementTree for example, there is no pretty way to parse an XML string which does not contain namespaces, modify its elements or attributes and write the resulting XML back to a file without ElementTree adding namespace declarations.

Use of Conditionals, Long Method

Instead of using “Strategy” [40] and similar patterns we heavily rely on if/else-statements in our code. Refactoring in Python is a rather delicate issue where one has to be careful not to mess up the indentation levels. We decided to focus rather on the process of fuzzing and improving coverage and precision than spend our time writing beautiful code. The same holds true for the “Long Method Smell” [40].

8.1.5. Scripts vs Modules

All scripts are designed to run in standalone-mode, i.e. they have a main() method and receive command-line arguments. Although the primary goal of the scripts is to create an entire Peach Pit file one may want to just generate certain sections or if encountering XML content containing *String* elements, to see what the contained XML looks like without having to run the entire script. The only modules we have written are shared components such as the configuration parser or components which do not need to be accessible to the user such as the SASL message generation module.

8.1.6. Testing

Creating Unit-Tests for the scripts is very time consuming. In order to thoroughly test every detail we would have needed to write test cases for every stanza characteristic possible, which would have resulted in hundreds of test cases. Given our schedule, a test driven design is therefore not practical. The lack of Unit-Tests makes refactoring much harder. Every time changes are made to the code, manual tests have to be performed to ensure correct operation. We therefore decided not to refactor at large scale.

8.2. Implementation Decisions

8.2.1. Abortion of schema2xml.py

The script to extract all possible XML stanzas from the schemas has been abandoned for two reasons.

- Excessive Output: While the generation of all possible XML stanza from the schemas would yield an input space coverage of 100%, our test fuzzing runs indicated that such an approach would not be practical due to the massive amount of stanzas generated. Furthermore, many generated stanzas would probably be valid, but not usable for XMPP communication, thus creating a substantial overhead.

- Time Constraints: The implementation, which seemed straight-forward at first, proved to be much more complex and time consuming. Given the tight timeframe, we decided to base input space coverage improvement on the much less complex XEP extraction.

By basing our fuzzing stanza samples on the XEPs, using xep_ripper.py, we lose the type declarations contained in the schemas and therefore can not automatically include our *DataModel*s defined in section 5.6. However, since those *DataModel*s determine whether a server fails to parse ill-structured content, it may be sufficient to verify correct handling of each *DataModel* once instead of using the *DataModel*s everywhere. By creating separate fuzzers to find structure parsing flaws we are able to further decrease overall fuzzing time.

8.2.2. Script vs Peach XML extension

Due to the fact that Peach fuzzer configuration files are written in XML and since XMPP is using XML messages as well, XML data has to be inserted into XML files. In order to accomplish that, the Peach tags *String* and *XmlElement* could theoretically be used. While *String* elements would work, they are quite hard to read because all quotes, "less-than" and "greater-than" characters have to be escaped using XML entities. A '"' would therefore have to be written as '&quot;', '<' as '&lt;' and '>' as '&gt;'. On top of that, the XML tags get sprawled over multiple lines because areas which are supposed to be fuzzed have to be contained in an own *String* element that is mutable. Figure 8.1 compares XML to the Peach *String* representation.

The Peach *XmlElement* would yield lower overhead than *String* elements as figure 8.2 shows but has a major drawback which makes it unusable for XMPP: The *XmlElement* representation generates well-formed XML files. The problems are twofold: We have discovered that the jabberd2 server does not like the <?xml version="1.0"?> tag which by default is added to every XMPP-stanza sent over the network. The server claims that the XML is not well-formed and therefore terminates the underlying TCP connection when *XmlElement* is used to represent protocol data. The second problem is XMPP's <stream> tag which must not be closed by </stream> unless one wants to terminate the stream. Since there is no way to tell *XmlElement* to omit the closing tag and insert it later, thus preventing the creation of a stream spanning multiple stanzas distributed over time. The fuzzing possibilities would be drastically reduced and certain areas would not be fuzzable at all, if multiple subsequent stanzas are needed.
In order to obtain readable and easily maintainable XML fuzzer definitions there are two options: Either writing a script which converts from XML to Peach-XML or extending Peach in a way that custom non-well-formed XML data can be written similar to Peach's *XmlElement*.

We have chosen to take the *String* route, implementing a script, for the following reasons:

```
XML:
<Element value="value1" >
     <Element2 value="value2">
          <Element3 acronym="E3" value="value3"/>
     </Element2>
</Element>

Peach Strings:
<String value="&lt;XMLElement value=&quot;" mutable="false"/>
<String value="value1"/>
<String value="&quot; &gt;" mutable="false"/>
<String value="&lt;XMLElement2 value=&quot;" mutable="false"/>
<String value="value2"/>
<String value="&quot;&gt;" mutable="false"/>
<String value="&lt;XMLElement3 acronym=&quot;" mutable="false"/>
<String value="E3"/>
<String value="&quot; value=&quot;" mutable="false"/>
<String value="value3"/>
<String value="&quot;/&gt;" mutable="false"/>
<String value="&lt;/XMLElement2&gt;" mutable="false"/>
<String value="&lt;/XMLElement&gt;" mutable="false"/>
```

Figure 8.1.: Comparison between XML and the *String* representation in Peach, assuming that all values need to be fuzzed.

- Extending Peach is far more complex and time consuming. We better invest time in fuzzing and bug analysis than extending Peach.
- Due to the fact that Peach is currently rewritten in C# a custom *XmlElement* implementation in Python is obsolete once Peach 3 is released. An external script might still be useful if Peach 3 does still not feature custom XML elements.
- Writing a custom *XmlElement* might require the implementation of custom Mutators. Since we have not yet fully evaluated the Mutators, the time and effort needed would probably exceed our timeframe. On top of that we would have to make sure to transfer all heuristics contained in the Peach default mutators to our own. Depending on the implementation this might prove difficult.
- The lack of readability can be mitigated by implementing a Peach *String* representation to XML conversion feature.
- The *String* representation generally is much shorter due to the lack of closing XML tags

To practice Python and therefore become more adept and efficient in analyzing Peach's implementation we have chosen to write the script in Python.

```
 XML:
<Element value="value1" >
     <Element2 value="value2">
          <Element3 acronym="E3" value="value3"/>
     </Element2>
</Element>

XmlElement:
<XmlElement elementName="Element" mutable="false">
     <XmlAttribute attributeName="value" mutable="false">
          <String value="value1"/>
     </XmlAttribute>
     <XmlElement elementName="Element2" mutable="false">
          <XmlAttribute attributeName="value" mutable="false">
               <String value="value2"/>
          </XmlAttribute>
          <XmlElement elementName="Element3" mutable="false">
               <XmlAttribute attributeName="acronym" mutable="false">
                    <String value="E3"/>
               </XmlAttribute>
               <XmlAttribute attributeName="value" mutable="false">
                    <String value="value3"/>
               </XmlAttribute>
          </XmlElement>
     </XmlElement>
</XmlElement>
```

Figure 8.2.: Comparison between XML and the *XmlElement* representation in Peach, assuming that all values need to be fuzzed.

Part III.

Software Installation & Configuration

9. General Setup

This chapter provides an overview of the lab setup we used in order to perform fuzzing.

9.1. Hardware

Team members used their personal MacBooks for implementation and configuration tasks. After the proof of concept phase, we moved to a dedicated server hardware in the HSR DMZ to run our fuzzers and XMPP servers.

9.2. Virtualization

The XMPP servers and Peach agents run inside a virtual machine. Peach has a virtual machine control feature. We used VMware Workstation 8 and VMware Fusion 4.

9.3. Operating System

As an OS, we used Windows XP 32bit for the server part, Ubuntu 11.10 for the proof of concept, and Mac OS X 10.7 Lion for implementation and documentation.

9.4. Development Environment

Peach extensions are developed in Python. We used a mix of Python 2.6 and 2.7, some installations went smoother with the older version. See the installation chapter. jabberd2 build is done with Microsoft Visual Studio 2008 Team System SP1. For debugging on Windows we used Microsoft WinDbg, on Linux we used Ubuntu Apport. The disassembler is IDA Pro 6.2. And for wire captures we used Wireshark.

9.5. Database

For the XMPP server database backend, we chose MySQL 5.1 for jabberd2 and Microsoft SQL Server 2008 Express for SoapBox.

9.6. Clients

Chat clients were used for testing server functionality and for capturing sessions, namely Pidgin, Pandion and Miranda, which has a useful XML console to send raw XML.

10. Fuzzers

10.1. Peach on Windows

10.1.1. Preliminary

Table 10.1 lists the required software to install Peach.

Software	Version	Source
Peach 32bit or 64bit executable	2.3.8	`http://www.peachfuzzer.com`
WinPcap	4.1.2	`http://www.winpcap.org`
Debugging Tools for Windows	6.10.3.233	`http://www.microsoft.com`

Table 10.1.: Prerequisites for Peach on Windows

10.1.2. Installation

The following steps are required to install Peach.

1. Install WinPcap
2. Install Debugging Tools for Windows, use program folder name without (x86) or Peach will not find WinDbg: e.g. C.\Program Files\Debugging Tools for Windows\
3. Install Peach and follow the installation wizard

10.1.3. Configuration

No special configuration is necessary for Peach.

10.2. Peach on Linux

10.2.1. Preliminary

Table 10.2 lists the required software to install Peach.

Software	Version	Source
Python	2.7	`http://www.python.org`
Peach	2.3.8	`http://sourceforge.net/projects/peachfuzz`
4Suite XML Library	1.0.2	Peach dependencies folder
cDeepCopy	unknown	Peach dependencies folder
cPeach	unknown	Peach dependencies folder
Zope	3.6.1	Peach dependencies folder
Twisted	10.2.0	Peach dependencies folder
PSUtil	0.2.0	Peach dependencies folder

Table 10.2.: Prerequisites for Peach on Linux

10.2.2. Installation

The following steps are required to install Peach:

- Install Python 2.7 from source.
- Check out Peach from SVN
- All the dependencies for Peach are in a subdirectory of the Peach main directory
- Install 4SuiteXML from source with sudo python setup.py install
- Install cDeepCopy from source with sudo python setup.py install
- Install cPeach
- Unzip Zope and install
- Unzip Twisted and install
- Unzip PSUtil and install
- Enable Ubuntu Apport with sudo nano /etc/default/apport and change enabled from 0 to 1

10.2.3. Configuration

No special configuration necessary, Peach is preconfigured for use with Apport. Important: Start the agent on the target machine with sudo.

10.3. Sulley on Windows

10.3.1. Preliminary

Table 10.3 lists required software to install Sulley.

Software	Version	Source
Python	2.7.2	`http://www.python.org`
MinGW Minimalist GNU for Windows	v20101030	`http://sourceforge.net/projects/mingw/`
WinPcap	4.1.2	`http://www.winpcap.org`
WinPcap Developer Pack	4.1.2	`http://www.winpcap.org/devel.htm`
libdasm	1.5	`http://code.google.com/p/libdasm/`
Pcapy	0.10.5	`http://oss.coresecurity.com/projects/pcapy.html`
Impacket	0.9.6.0	`http://oss.coresecurity.com/projects/impacket.html`
Pydbg	Latest	`https://github.com/OpenRCE/pydbg`
Sulley	Latest	`https://github.com/OpenRCE/sulley`

Table 10.3.: Prerequisites for Sulley

10.3.2. Installation

The following steps are required to install Sulley. [41] [42] [43]

1. Set Windows Environment Variables in Path to C:\Python27\; C:\Python27\scripts\; C:\MinGW\bin\;
2. Install MinGW, choose the C++ compiler option
3. Install WinPcap and extract the WinPcap Developer Pack to C:\WpdPack_4_1_2\
4. Install Python
5. For libdasm, cd into libdasm\pydasm, type python setup.py build_ext -c mingw32, then python setup.py install
6. For Pcapy, extract and type python setup.py build_ext -c mingw32 -I "C:\WpdPack_4_1_2\WpdPack\Include" -L "C:\WpdPack_4_1_2\WpdPack\Lib", then python setup.py install
7. For Impacket, extract and type python setup.py install
8. Install Pydbg, type python setup.py install
9. Run Sulley

10.3.3. Configuration

No special configuration is necessary for Sulley.

11. Servers

11.1. Savant Web Server

11.1.1. Preliminary

Table 11.1 lists required software to install Savant.

Software	Version	Source
Savant	3.1	`http://savant.sourceforge.net/`

Table 11.1.: Prerequisites for Savant

11.1.2. Installation

The following steps are required to install Savant.

1. Run the executable and follow the wizard.

11.1.3. Configuration

No special configuration is necessary for Savant.

11.2. jabberd2 on Windows XP 32bit with Debug Symbols

11.2.1. Preliminary

Table 11.2 lists required software to install jabberd2.

Software	Version	Source
jabberd2	2.2	`http://www.nanoant.com/portfolio/jabberd2-win32`
jabberd2 with Debug Symbols	2.2	`http://www.nanoant.com/portfolio/jabberd2-win32`
Microsoft Visual Studio 2008 Team System	2008 SP1	`http://www.microsoft.com`
MySQL Server	5.1	`http://www.mysql.com`

Table 11.2.: Prerequisites for jabberd2 on Windows

11.2.2. Installation

The following steps are necessary to install jabberd2:

1. Install Visual Studio with SP1
2. Install MySQL Server
3. Install jabberd2 following the wizard
4. Stop all jabberd2 services and uninstall them with sc delete <servicename>
5. Extract jabberd2 with Debug Symbols
6. Copy every file from the jabberd2 programs folder to the extracted folder, so that the two match, e.g. configuration files, dlls, SQLite database and so on
7. Install jabberd2 executables as service: run router.exe -I, c2s.exe -I, sm.exe -I, s2s.exe -I
8. Start all services in the Windows control panel

11.2.3. Configuration

Jabberd2 is configured by default to use the SQLite database. In order to use MySQL as a backend database, check out jabberd2 from SVN and run the same steps for the database configuration as in the Ubuntu installation below. Now in c2s.xml modify

```
<authreg>
        <module>sqlite </module>
</authreg>
```

to use MySQL as well as in sm.xml

```
<storage>
        <driver>sqlite </driver>
</storage>
```

The detailed jabberd2 Installation and Administration Guide is at `https://github.com/Jabberd2/jabberd2/wiki/InstallGuide`

11.3. jabberd2 on Ubuntu 11.10 32bit

11.3.1. Preliminary

Table 11.3 lists required software to install jabberd2.

Software	Version	Source
libssl-dev	latest	Ubuntu Repository
libidn	1.24	`http://ftp.gnu.org/gnu/libidn`
MySQL	Latest	Ubuntu Repository
libmysqlclient-dev	Latest	Ubuntu Repository
jabberd2	2.0s6	`http://pkgs.fedoraproject.org/repo/pkgs/jabberd`

Table 11.3.: Prerequisites for jabberd2 on Ubuntu

11.3.2. Installation

The following steps are necessary to install jabberd2:

1. Add directories for PID's and Logs:

   ```
   sudo mkdir -p /usr/local/var/jabberd/pid
   sudo mkdir -p /usr/local/var/jabberd/log/
   ```

   and make jabber the owner

   ```
   sudo chown -R jabber:jabber /usr/local/var/jabberd/pid/
   sudo chown -R jabber:jabber /usr/local/var/jabberd/log/
   ```

2. Install libssl-dev

```
sudo apt-get install libssl-dev
```

3. Install libidn:

   ```
   tar -zxvf libidn-1.24.tar.gz
   cd libidn-1.24
   ./configure --prefix=/usr
   make
   sudo make install
   ```

4. Install MySQL:

   ```
   sudo apt-get install tasksel
   sudo tasksel
   ```

   and select OpenSSH Server & LAMP

5. Install libmysqlclient-dev:

   ```
   sudo apt-get install libmysqlclient-dev
   ```

6. Install jabberd2:

   ```
   tar -xvf jabberd2.0s6.tar.gz
   cd jabberd-2.0s6
   ./configure --enable-mysql --enable-ssl --enable-idn
   make
   sudo make install
   ```

11.3.3. Configuration

Operating System

Create a user jabber and add the user to the jabber group. Disable the firewall with sudo ufw disable.

Database

To create the database backend for jabberd2, in the jabberd2.0s6/tools directory there is a script. Login to MySQL with

```
mysql -u root -p
```

and run the script with

```
\.. db-setup.mysql
```

After that, run

```
GRANT select,insert,delete,update ON jabbered.*
TO jabbered2@localhost IDENTIFIED BY 'secret';
```

Then, add a symlink to

```
ln -s /var/run/mysqld/mysql.sock /tmp/mysql.sock
```

jabberd2

Edit the server identification in /usr/local/etc/jabberd/sm.xml from

```
<sm>
        <id>localhost</id>
</sm>
```

to

```
<sm>
        <id>ubuntu</id>
</sm>
```

as well as in /usr/local/etc/c2s.xml change it from

```
<local>
        <id>localhost</id>
</local>
```

to

```
<local>
        <id>ubuntu</id>
</local>
```

Start the server in /usr/local/bin with

```
./jabberd
```

The detailed jabberd2 Installation and Administration Guide is at `https://github.com/Jabberd2/jabberd2/wiki/InstallGuide`

11.4. Coversant SoapBox Server 2010 Express

Coversant states on their website that MySQL can be used as a backend. Unfortunately, customer support explained that the current release does not support MySQL, but only a Microsoft database server.

11.4.1. Preliminary

Table 11.4 lists required software to install SoapBox.

Software	Version	Source
Microsoft SQL Server Express	2008	`http://www.microsoft.com`
Coversant SoapBox Server Express	2010	`http://www.coversant.com/products/sbs`

Table 11.4.: Prerequisites for Soapbox on Windows

11.4.2. Installation

The following steps are necessary to install SoapBox:

1. Install Microsoft SQL Server Express and all its prerequisites

2. Install Coversant SoapBox Server 2010 Express and follow the installation wizard

11.4.3. Configuration

Coversant provides a database configuration wizard. Follow the steps to configure SQL Server. Detailed configuration can be found in the SoapBox program folder. The authentication is set in Configuration.xml in the SoapBox program folder.

Coversant provides detailed Installation, Configuration and Administration guides at `http://www.coversant.com/products/sbs`

12. Code Coverage Tools

12.1. Visual Studio 2008 Profiling Tools

Microsoft has a code coverage tool in Visual Studio 2008 Team System.[44] Usage steps:

1. Compile project with /PROFILE link option.
2. Add instrumentation code to .exe with vsinstr.exe <your.exe> /COVERAGE
3. Start listener with VSPerfMon.exe /COVERAGE /OUTPUT:<Report-File-Name>
4. Start your application and perform fuzzing.
5. Stop coverage tool with VSPerfCmd.exe /SHUTDOWN from a second terminal.
6. Drag .coverage file into Visual Studio to view results.

12.2. PaiMei

12.2.1. Preliminary

Table 12.1 lists required software to install PaiMei.

Software	Version	Source
Python	2.6.6	`http://www.python.org`
MySQL	5.1	`http://www.mysql.com`
MySQL for Python	1.2.3 for Python 2.6 and MySQL 5.1	Source: `http://www.sourceforge.com/projects/mysql-python` Precompiled: `http://www.lfd.uci.edu/~gohlke/pythonlibs/#mysql-python`
IDA Pro	6.2	`http://www.hex-rays.com/products/ida/index.html`
IDA Python	1.5.3 for IDA Pro 6.2 and Python 2.6	`http://code.google.com/p/idapython/`
wxPython	2.8	`http://www.wxpython.org`
uDraw	3.1.1	`http://www.informatik.uni-bremen.de/uDrawGraph/en/home.html`
GraphViz	2.28.0	`http://www.graphviz.org`
PyDot	1.0.28	`http://code.google.com/p/pydot/`
Oreas GDE	1.3.1	`www.oreas.com/gde_en.php`
Pydbg	Latest	`https://github.com/OpenRCE/pydbg`
PaiMei	Latest	`https://github.com/OpenRCE/paimei`

Table 12.1.: Prerequisites for PaiMei

12.2.2. Installation

- Install Python 2.6
- Install MySQL 5.1, root login, root password
- Install MySQL Python
- Install IDA Pro 6.2 Trial
- Install IDAPython: copy python directory to IDA Pro directory, copy contents of plugins directory to IDA Pro\plugins and copy python.cfg to IDA Pro\cfg
- Install wxPython
- Install uDraw
- Install GraphViz, edit installation path to C\Program Files\GraphViz (no version number)
- Install PyDot with python setup.py install
- Install Oreas GDE
- Copy PyDbg to PaiMei directory

12.2.3. Configuration

To check all required components, PaiMei includes a checking tool in the PaiMei directory.

```
python __install_requirements.py
```

To configure the backend database to store hits, PaiMei has a setup script in the PaiMei directory.

```
python __setup_mysql.py localhost root root
```

To start PaiMei, navigate to the console directory and run

```
python PAIMEIconsole.pyw
```

12.2.4. Usage

In IDA Pro, load a binary and dump the graph with pida_dump.py in the PaiMei directory. Load the PIDA file with PaiMei. The PaiMei directory contains a docs directory with a tutorial video.

Part IV.

Project Management

13. Project Planning

13.1. Milestones

The project was defined by the following milestones:

- MS0 (22.02.2012, 15:00): Project start, Kick-off meeting
- MS1 (11.03.2012): Fuzzing framework evaluation deadline, proof-of-concept
- MS2 (30.03.2012): Proof-of-concept with jabberd2.0
- MS3 (27.05.2012): Feature freeze
- MS4 (03.06.2012): Code freeze, moved to end of project
- MS5.1 (08.06.2012): Abstract and A0 poster hand-in
- MS5.2 (15.06.2012, 12:00): Documentation hand-in and A0 poster revision deadline
- MS6 (15.06.2012, 16:00 - 20:00): Presentation, Project end

13.2. Week By Week Breakdown

Week 1: 20.02. - 26.02.

Gather Information about fuzzing, evaluate books and papers.
Reading

Week 2: 27.02. - 04.03.

Evaluation of fuzzers.
Choose protocols.
Begin risk analysis (define criteria for choice of fuzzer and protocol)

Week 3: 05.03. - 11.03.

Definitive choice of framework.
Extend risk analysis.

Week 4: 12.03. - 18.03.

Begin implementation of XMPP fuzzer.
Server installation

Week 5: 19.03. - 25.03.

Installation of Jabber2.0 server containing a vulnerability for PoC.
Trigger known vulnerability with exploit.

Week 6: 26.03. - 01.04.

Trigger known bug with fuzzer.

Week 7: 02.04. - 08.04.

Pre-processor script for automatic DataModel generation.
Database reset script.

Week 8: 09.04. - 15.04.

Peach Mutator analysis.
Write own mutators: Buffer overflow mutator, invalid format mutators, DataModels for specific XMPP data types.
Gauge XMPP feature coverage (i.e. stanza/attributes coverage), implement stanza counting feature.

Week 9: 16.04. - 22.04

Automatic schema processing, introduce hint tags to optimize fuzzing.
Measure and document performance gains of own mutators.
Implement Pit configuration file writer (.cfg file to store preferences)
Create Peach DataModels for XMPP types.

Week 10: 23.04. - 29.04.

Evaluate/implement automatic state modeling scripts.

Week 11: 30.04. - 06.05.

Determine server feature coverage.
Determine total code coverage.

Week 12: 07.05. - 13.05.

Write custom loggers (database content snapshot, adapt logging to better support resuming fuzzing sessions and bug analysis)
Elaborate concepts for fuzzing resuming on crashes or other interruptions.
Implement fuzzing resuming concepts.
Improve fuzzer coverage.

Week 13: 14.05. - 20.05.

Fuzz other XMPP servers (grey-box).
Adapt fuzzers/scripts if necessary.

Week 14: 21.05. - 27.05.

Bug analysis: Exploitation potential analysis, try to write exploits for found bugs, if any. Evaluate potential and limitations of fuzzing in general and with peach and our approach.

Week 15: 28.05. - 03.06.

Evaluate and optimize fuzzing.
Additional fuzzing.
Documentation. Presentation prep.

Week 16+: 04.06. - 15.06.

Evaluate fuzzing.
Additional fuzzing.
Documentation. Presentation prep.

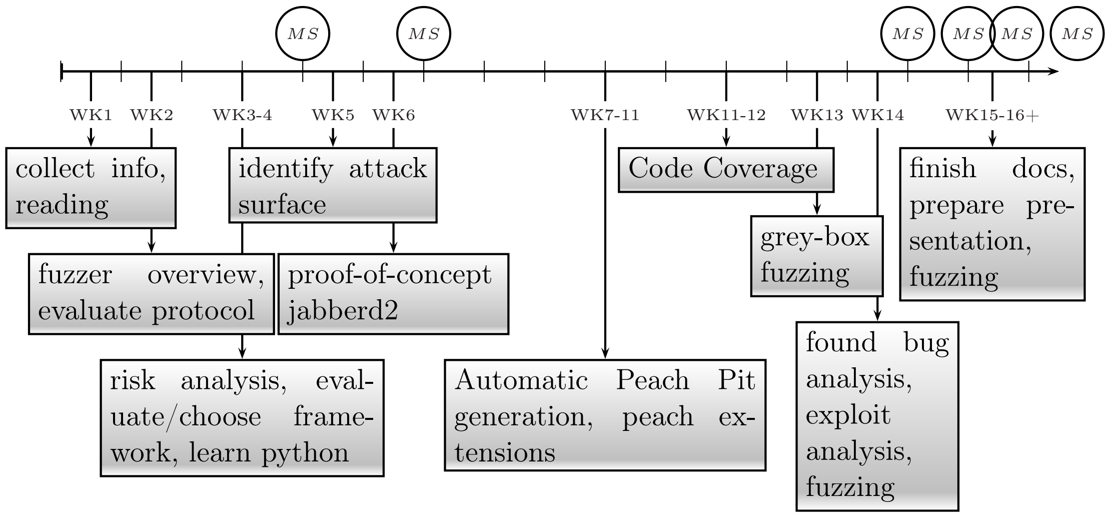

MS
MS
MS
MS
MS
MS
WK1
WK2
WK3-4
WK5
WK6
WK7-11
WK11-12
WK13
WK14
WK15-16+
collect info, reading
identify attack surface
fuzzer overview, evaluate protocol
proof-of-concept jabberd2
risk analysis, evaluate/choose framework, learn python
Automatic Peach Pit generation, peach extensions
Code Coverage
grey-box fuzzing
found bug analysis, exploit analysis, fuzzing
finish docs, prepare presentation, fuzzing

Figure 13.1.: Project task timeline

13.3. Uncertainties

The project was characterized by numerous uncertainties. We initially had little to no fuzzing-knowledge, assembly level debugging skills and vulnerability exploitation experience. As a consequence the inception and elaboration phases were longer than usual to identify and eliminate as many risks as possible.

13.4. Segregation Of Work

The responsibilities for this project were distributed as follows:

13.4.1. Both Members

- Evaluation
- Documentation
- Strategic planning

13.4.2. Kevin Lynn

- Setup infrastructure
- Proceedings
- Code coverage

13.4.3. Michael Fisler

- Programming tools
- Testing
- Peach extensions

14. Planned Features

The following tasks were used as a guideline for planning. Since it was uncertain if we would even find bugs, it was very difficult to estimate the progress and therefore set definitive milestones between proof-of-concept and code-freeze. Changes have therefore been made to the time plan in consideration of our actual progress. The task priorities are defined as follows: (1): must have, (2): should have, (3): nice to have, (4): optional

14.1. Fuzzing

State-modeling (1)
Intelligent fuzzing, e.g. handling server messages (2)
Optimize server feature coverage (2)
Optimize XMPP attack surface coverage (2)
Optimize total code coverage (3) Optimize fuzzing precision (3)

14.2. Peach extensions

Custom XML implementation / XML to Peach DataModel conversion script (1)
Implement new mutators (2)
Implement custom loggers (4)
Implement automatic Pit generation (3)

14.3. Code coverage

Determine total code coverage (3)
Determine XMPP attack surface coverage (1)
Determine server feature coverage (2)

14.4. Bug analysis

Determine kinds of bugs and their potential exploitability (2)
Determine accuracy of the peach bucketing and bug-classification (3)

14.5. Exploit

Tailor exploit for found bugs (4)
Study concepts to circumvent DEP, etc. (4)

15. Project Monitoring

The gantt chart in figure 15.1 shows the distribution of the work packages over time and if there were deviations from the projected work package durations. Due to the fact that some work packages took longer to complete, some less important work packages, which were not essential for fuzzing, were delayed. The work packages that were identified as least important or that could be replaced by an adequate but easier to implement solution were canceled.

15.1. Work Packages finished in time

Initially we planned a target time of 20.25 hours, based on the number of ECTS credits attainable and the risk management requirements for time reserves. Although a lot of work packages could be finished within the projected time frame, the weekly workload was higher than anticipated. The reasons are the following:

- Evaluating and setting up the testing environment consumed an enormous amount of time
- Dependencies of older software can be a nightmare to deal with
- Initial effort needed to get acquainted with all of Peach's concepts is big. The documentation is average at best. A lot of concepts had to be reverse engineered from examples.

15.2. Delayed or canceled Work Packages

In some cases work packages could not be finished in the projected time frame. The reasons are diverse:

- Relying too much on server answers when implementing DIGEST-MD5. Jabberd2 insisted that the structure of the DIGEST-MD5 response was wrong, even though the problem was an authentication failure.
- The input space coverage determination using schemas proved to be more complex than anticipated. If the criterion "semantically distinct" is not chosen very carefully, the resulting coverage metric is useless.
- Python's XML libraries proved to be rather annoying and involved longer than planned evaluation.

- By gradually adding new features or adding restrictions, the scripts got rather complex in some cases.
- Testing was very time consuming. Since thorough testing is impractical with such a large protocol, different bugs or poor implementations were discovered over time, e.g. during fuzzing.
- Changing the fuzzing approach or strategy had ramifications for Pit creation tools resulting in some Pit-creation-related work packages to last until the end of the project.
- We lacked the knowledge to compile the entire jabberd2 server using Microsoft Visual Studio and no one could help us. In order not to lose too much time we decided to abandon compiling jabberd2 on Windows.

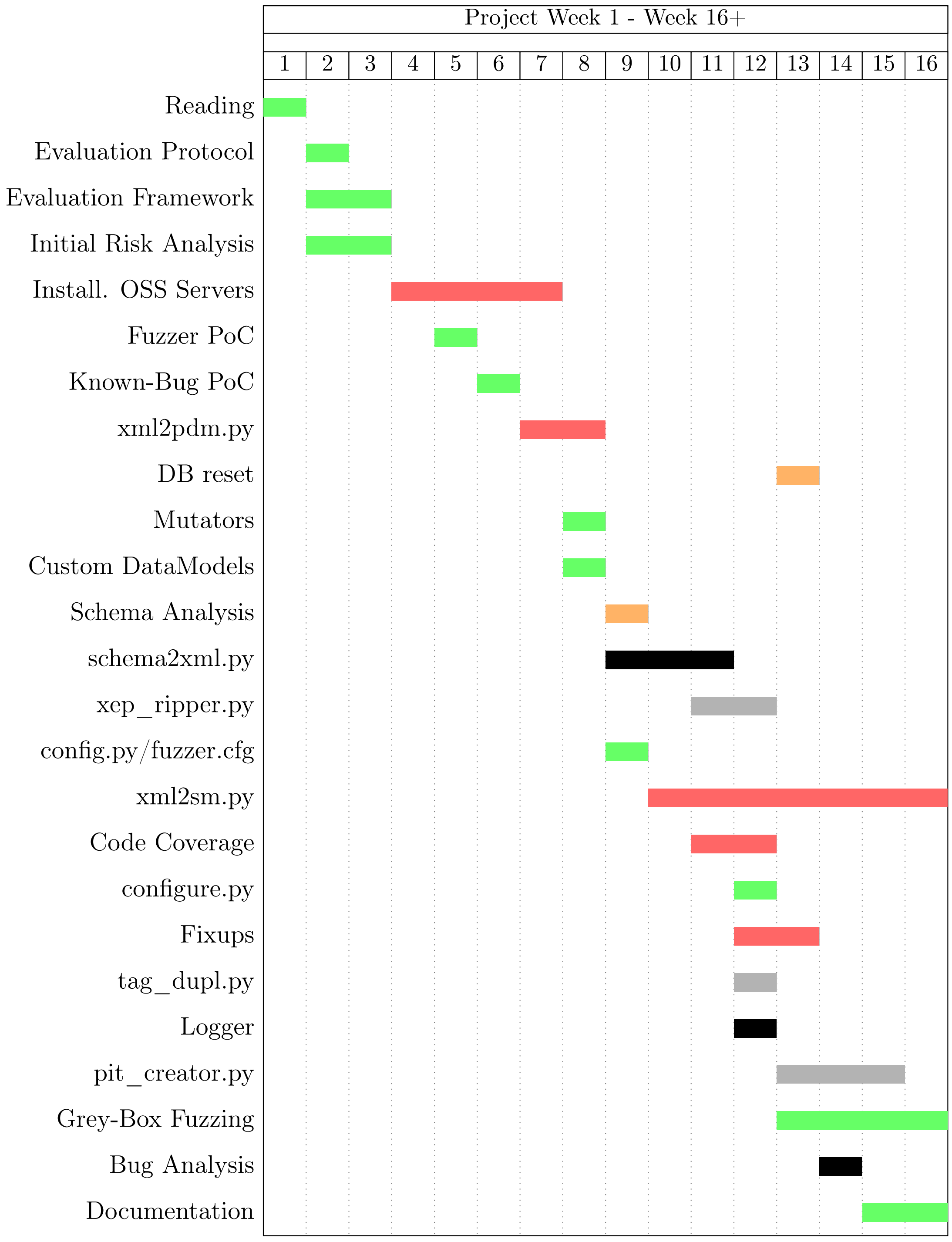


Figure 15.1.: Work packages (WP) distributed over time. Green bars: WP were finished in targeted time and on schedule, orange bars: WP were finished in targeted time but with delay, black bars: WP have been canceled, gray bars: WP have not been planned originally.

16. Risk Management

The risks related to our project involve no monetary losses. The damage is therefore gauged by the amount of time lost if a risk occurs (i.e. the time to recover the previous state or the delay caused by the risk). All probabilities are estimates, partly based on previous experience. Table 16.1 shows all risks which were identified at the beginning of the project. The probabilities and damages are as determined at the beginning of the project, with the number of ECTS credits in mind. With each credit requiring 30 hours of work, the resulting overall net working time is at 360 hours per person.

Apart from the risks identified at the beginning, we encountered various minor risks during the project that we handled on a week-to-week basis. Mostly it involved decisions on how to proceed or identifying the most important tasks and features, dropping low priority features if necessary.

The most important decisions were:

- Dropping XML schema processing
- Limiting coverage analysis to plausibility checks
- Bypassing bucketing analysis (partly due to the lack of found bugs in the server)
- Limiting Logger analysis to a conceptual evaluation, without implementing an own custom Logger
- Moving the code freeze deadline

Risk	Max. Damage (h)	Probability (%)	Weighted Damage	Preemptive Measures	Countermeasures
Protocol too complex	400	40	160	Complexity is major factor in decision Avoid proprietary and undocumented protocols.	Implement easy features first Drop optional features Definitive deadlines
Protocol too simple	400	10	40	Complexity is major factor in decision	Switch to other protocol
Fuzzer too complex	400	30	120	Avoid extending single- or multi-protocol fuzzers, use fuzzing frameworks instead Prefer well documented and/or widely used fuzzers Prefer well designed and extensible fuzzers	Switch to other fuzzer
Fuzzer not suitable	400	20	80	Choose fuzzing framework over single-dimensional fuzzer Choose most suitable fuzzer for given protocol and environment	Switch to other fuzzer
Fuzzing reveals no bugs	400	98	392	Choose software with known vulnerability for demo and proof-of-concept Avoid fuzzing widely distributed and deployed software	Change fuzzing strategy, possibly change fuzzer
Unknown Programming Language	180	80	225	Prefer known programming language, else plan language prep	Use time reserves
Missing Assembler Level Debugging Skills	240	95	228	Prefer fuzzing frameworks with easy fuzzer integration Plan debugging prep	Get external advice, don't dig too much into details
Data loss (source, docs)	720	1	7.2	Data is stored in a remote Git repository Individual backups	Parts or all of code or documents have to be rewritten
Other	?	?	?	Time reserves planned	Drop low-priority features, reduce functionality

Table 16.1.: Risk management: Risks and measures

17. Proceedings

All proceedings are externalized to the CD.

Part V.

Appendix

A. CD Content

- Documentation
- Python Scripts
- Peach Extensions
- RFC / XEP
- Schemas
- Fuzzer
- Proceedings

B. List of Abbreviations and Typographic Conventions

B.1. Conventions

Throughout this document we use the following conventions:

- Placeholders: Text in angle brackets (<text>) are placeholders. E.g. <IPAddress> is used to describe a custom IP address and could, depending on the context, be resolved to 192.168.1.1.
- Peach Elements: Peach-specific elements are written in cursive type. For the exact definition and usage of the element refer to the official Peach documentation at [13]. E.g. if we refer to a *String* we do not mean a common string, but a Peach *String* element.
- Peach Concepts: Peach's concepts begin with an uppercase letter. E.g. Logger is the Peach concept of a logger.

B.2. Abbreviations

- ASCII: American Standard Code for Information Interchange
- API: Application programming interface
- CPU: Central processing unit
- DEP: Data Execution Prevention
- DMZ: Demilitarized zone
- DTD: Document Type Definition
- ECTS: European Credit Transfer and Accumulation System
- EPFL: École Polytechnique Fédérale de Lausanne
- IDE: Integrated development environment
- JID: Jabber-ID
- NFS: Network File System
- OSS: Open-Source-Software
- PDU: Protocol data unit
- PID: Process identifier
- QA: Quality assurance
- RFC: Request for Comments
- RPC: Remote procedure call
- RTSP: Real Time Streaming Protocol
- SASL: Simple Authentication and Security Layer
- SIP: Session Initiation Protocol
- SMB: Server Message Block
- TCP: Transmission Control Protocol
- TLS: Transport Layer Security
- Win: Windows
- XEP: XMPP Extension Protocol
- XML: Extensible Markup Language
- XMPP: Extensible Messaging and Presence Protocol

C. Bibliography

Bibliography

[1] Bryan So Barton P. Miller Lars Fredriksen. *An Empirical Study of the Reliability of UNIX Utilities.* Tech. rep. University of Wisconsin-Madison, Dec. 1990.

[2] Microsoft.com. *Microsoft Security Development Life Cycle.* Feb. 2012. URL: http://www.microsoft.com/security/sdl/discover/verification.asp.

[3] Zachary N. J. Peterson Charlie Miller. *Analysis of Mutation and Generation-Based Fuzzing.* Tech. rep. ISE Independent Security Evaluators, 2007.

[4] computerworld.com. *Microsoft runs fuzzing botnet findes 1800 Office bugs.* Feb. 2012. URL: http://www.computerworld.com/s/article/9174539/Microsoft_runs_fuzzing_botnet_finds_1_800_Office_bugs.

[5] blogspot.com. *Google Online Security Blog, Fuzzing at scale.* Mar. 2012. URL: http://googleonlinesecurity.blogspot.com/2011/08/fuzzing-at-scale.html.

[6] softscheck.com. *Fuzzelarbeit - Identifizierung unbekannter Sicherheitsluecken und Softwarefehler durch Fuzzing.* Mar. 2012. URL: http://www.softscheck.com/publications/ProfDrHartmutPohl_Identifizierung_unbekannter_Sicherheitsluecken_und_Software-Fehler_durch_Fuzzing_kes20115.pdf.

[7] Adam Greene Michael Sutton and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery.* Addison-Wesley, 2007.

[8] SoftScheck. Mar. 2012. URL: http://softscheck.inf.h-brs.de/index.php?article_id=9.

[9] dmckinney. *antiparser.* June 2012. URL: http://antiparser.sourceforge.net/.

[10] Martin Vuagnoux. June 2012. URL: http://autodafe.sourceforge.net/.

[11] Tim Brown. June 2012. URL: http://www.portcullis-security.com/16.php.

[12] Jared DeMott. June 2012. URL: http://www.vdalabs.com/tools/efs_gpf.html.

[13] peachfuzzer.com. *Peach Fuzzing Platform.* June 2012. URL: http://peachfuzzer.com/.

[14] Dave Aitel. *SPIKE.* June 2012. URL: http://www.immunitysec.com/resources-freesoftware.shtml.

[15] Pedram Amini. *Sulley.* June 2012. URL: https://github.com/OpenRCE/sulley.

[16] groups.google.com. *peachfuzz internet forum.* Mar. 2012. URL: http://groups.google.com/group/peachfuzz.

[17] OpenRCE. June 2012. URL: http://www.openrce.org/articles/.

[18] xmpp.org. June 2012. URL: http://xmpp.org/xmpp-software/servers/.

[19] Coversant. June 2012. URL: http://www.coversant.com/products/sbs/.

[20] ietf.org. *RFC 6120 - Extensible Messaging and Presence Protocol (XMPP): Core.* June 2012. URL: http://www.ietf.org/rfc/rfc6120.txt.

[21] ietf.org. *RFC 3920 - Extensible Messaging and Presence Protocol (XMPP): Core.* Mar. 2012. URL: http://www.ietf.org/rfc/rfc3920.txt.

[22] ietf.org. *RFC 4422 - Simple Authentication and Security Layer (SASL).* June 2012. URL: http://www.ietf.org/rfc/rfc4422.txt.

[23] xmpp.org. *XEP-0078: Non-SASL Authentication.* June 2012. URL: http://xmpp.org/extensions/xep-0078.html.

[24] peachfuzzer.com. *Peach Fuzzing Platform - Data Modeling.* June 2012. URL: http://peachfuzzer.com/DataModeling.

[25] peachfuzzer.com. *Peach Fuzzing Platform - State Modeling.* June 2012. URL: http://peachfuzzer.com/StateModeling.

[26] xsploitedsecurity. *The sulley fuzzing framework! (A basic example walkthrough).* Oct. 2009. URL: http://www.youtube.com/watch?v=6sooEScW07Y.

[27] CVE. *CVE-2002-1120.* June 2012. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-1120.

[28] cve.mitre.org. *CVE-2011-1755.* Apr. 2012. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-1755.

[29] cve.mitre.org. *CVE-2006-1329.* Apr. 2012. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-1329.

[30] cve.mitre.org. *CVE-2004-0953.* Apr. 2012. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-0953.

[31] ietf.org. *RFC 2234 - Augmented BNF for Syntax Specifications: ABNF.* Mar. 2012. URL: http://www.ietf.org/rfc/rfc2234.txt.

[32] ietf.org. *RFC 3490 - Internationalizing Domain Names in Applications (IDNA).* Mar. 2012. URL: http://www.ietf.org/rfc/rfc3490.txt.

[33] w3c.org. *W3C Recommendation, 3.2.7 - dateTime.* June 2012. URL: http://www.w3.org/TR/xmlschema-2/#dateTime.

[34] ietf.org. *RFC 6122 - Extensible Messaging and Presence Protocol (XMPP): Address Format.* June 2012. URL: http://www.ietf.org/rfc/rfc6122.txt.

[35] ietf.org. *RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax.* Apr. 2012. URL: http://www.ietf.org/rfc/rfc2396.txt.

[36] ietf.org. *RFC 2732 - Format for Literal IPv6 Addresses in URL's.* Apr. 2012. URL: http://www.ietf.org/rfc/rfc2732.txt.

[37] xiaoka.com. *jabberd - XMPP Server.* May 2012. URL: http://codex.xiaoka.com/wiki/jabberd2:start#features.

[38] Adam Michael Strzelecki. *A Windows build of jabberd2, popular Jabber server.* June 2012. URL: http://www.nanoant.com/portfolio/jabberd2-win32.

[39] www.python.org. *PEP 8 - Style Guide for Python Code.* Apr. 2012. URL: http://www.python.org/dev/peps/pep-0008/.

[40] Joshua Kerievsky. *Refactoring to Patterns.* Addison-Wesley Professional, 2004.

[41] rosincore.wordpress.com. *Setting up a Sulley Fuzzing Framework on Windows 7.* Sept. 2011. URL: http://rosincore.wordpress.com/2011/09/18/setting-up-a-sulley-fuzzing-framework-on-windows-7/.

[42] louppen.wordpress.com. *Installing the Sulley fuzzer framework on Windows XP Professional - Installtion notes.* June 2011. URL: http://louppen.wordpress.com/2011/06/22/installing-the-sulley-fuzzer-framework-on-windows-xp-professional-installtion-notes/.

[43] OpenRCE. *Windows Installation.* 2012. URL: https://github.com/OpenRCE/sulley/wiki/Windows-Installation.

[44] Emil Gustafsson. *Native C++ Code Coverage reports using Visual Studio 2008 Team System.* May 2012. URL: http://blogs.msdn.com/b/cellfish/archive/2008/11/16/native-c-code-coverage-reports-using-visual-studio-2008-team-system.aspx.

[45] Gadi Evron Noam Rathaus and Robert Fly. *Open Source Fuzzing Tools.* Syngress Media, 2007.

[46] fuzzing.org. *Fuzzing: Brute Force Vulnerability Discovery.* Feb. 2012. URL: http://fuzzing.org/.

[47] Sourceforge.net. *TAOF - The Art Of Fuzzing.* Feb. 2012. URL: http://sourceforge.net/projects/taof/.

[48] Pedram Amini. *PaiMei.* June 2012. URL: https://github.com/OpenRCE/paimei.

[49] Pedram Amini. *PyDbg.* June 2012. URL: https://github.com/OpenRCE/pydbg.

[50] Savant Team. June 2012. URL: http://savant.sourceforge.net/.

D. Declaration of Authorship

E. Copyright & License