DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2017

Effectiveness of fuzz testing high-security applications

A case study of the effectiveness of fuzz-testing applications with high security requirements.

BALTHAZAR WEST

MARCUS WENGELIN

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION

Effectiveness of fuzz testing high-security applications

BALTHAZAR WEST, MARCUS WENGELIN

Master in Computer Science
Date: June 5, 2017
Supervisor: Dilian Gurov
Examiner: Örjan Ekeberg
Swedish title: Effektiviteten hos fuzztestning av applikationer med hög säkerhet
School of Computer Science and Communication

Abstract

Fuzzing is a testing methodology that is receiving increased attention in the field of software security. The methodology is interesting because almost anyone can download a fuzzer and search for bugs in large and well-tested programs or libraries.

This thesis is a case study which examines the efficiency of fuzzing a library with high security requirements. It was decided that the Mbed TLS, an open source SSL library, would be fuzzed using AFL, a state of the art fuzzer. The steps required to use AFL to fuzz Mbed TLS are outlined along with the results the study yielded.

The fuzzing process did not succeed in finding input that causes crashes. However, there was a clear contrast between the results of the two fuzzed components of the library, and ultimately considered inconclusive primarily due to the fuzzing process being too time-consuming. The thesis is concluded by acknowledging the major takeaways and suggestions for future work.

Sammanfattning

Fuzz-testning är en testmetod som har fått ökad uppmärksamhet inom området mjukvarusäkerhet. Metoden är intressant för att i princip vem som helst kan ladda ner och använda en fuzzer för att hitta buggar i stora samt vältestade program och bibliotek.

Denna rapport är en fallstudie som undersöker effektiviteten av att fuzz-testa ett bibliotek med höga säkerhetskrav. Biblioteket som studerats i denna rapport är mbed TLS, ett open-source SSL-bibliotek. Fuzzern som valdes kallas AFL, en vedertagen fuzzer som visat sig vara mycket effektiv. Stegen som togs för att fuzza mbed TLS beskrivs i rapporten, tillsammans med resultaten.

Under fuzzingen hittades ingen input som genererade krascher. Dock finns en tydlig kontrast mellan resultaten av de två komponenterna av biblioteket som fuzzades, och i slutändan betraktas resultaten som ej slutgiltiga. Rapporten avslutas sedan med en diskussion samt rekommendationer för framtida forskning inom området.

Contents

Chapter 1

Introduction

Fuzzing is the process of inducing program failure or interesting behaviour by feeding a target application malformed or random data. What constitutes interesting program behaviour depends on the fuzzer, but they normally pay close attention to what input causes crashes. Initially, the fuzzer will feed test cases to the target based off of user-supplied seeds. Seeds may vary between different fuzzers, but are typically valid test cases. New tests are created by fuzzing previous tests with respect to these observations. The new tests are then fed to the target and this step will repeat indefinitely until the user terminates the program.

Supplying source code of the target application allows for more sophisticated fuzzing, such as detecting new execution paths which will improve code coverage.

The outcome of a fuzzing process is a set of test cases which trigger interesting behaviour. A good fuzzer will also attempt to minimize these tests while maintaining their characteristics. The goal of fuzzing is finding nuanced bugs and vulnerabilities in software while remaining reasonably ad hoc and at a modest performance cost.

Fuzzing has proven itself as an effective way of finding bugs and vulnerabilities in software. Fuzz testing is part of Microsoft's mandatory policy in the verification phase. Furthermore, fuzz testing makes up a third of all vulnerabilities found in Windows 7. In April 2014, a fuzzing tool discovered the famous Heartbleed vulnerability in a library used by the majority of web servers.

1.1 Problem statement

This paper investigates the use of fuzz testing as a method to easily discover bugs in software with high security requirements.

1.2 Scope

The scope of this paper is limited to fuzzing mbed TLS using AFL.

Mbed TLS was chosen as the target application due to the rigorous testing the library undergoes before each release. DTLS and attack surfaces not accessible remotely are not included in this scope.

The fuzzer that was chosen is called AFL, *American Fuzzy Lop*, and is well known in the IT-security community. It is simple, robust and has an impressive track-record with bugs found in Firefox, Flash and VLC Media Player, among others.

The original source code of both mbed TLS and AFL will be used. If necessary, minimal modifications to mbed TLS will be made in order to augment the fuzzing process.

The fuzzing will be done on a virtual machine in a personal computer. In order to reach deep code paths the fuzzing usually has to be extensive which poses a substantial speed constraint. With faster execution and more time this issue diminishes, but we are limited in both respects.

1.3 Purpose

The purpose of this study is to examine whether AFL can be, with minimal effort, successful in finding bugs in a codebase that is thoroughly tested and deemed secure.

Prior to this report we knew little about the topic of fuzzing. On top of seeing this as an opportunity to familiarize ourselves in the subject, we wanted to have a hands-on experience of fuzzing a significant codebase not written by ourselves, hence we chose to do a case study.

Chapter 2

Background

This chapter describes the fuzzing process, fuzzing technologies along with benefits, achievements and limitations of fuzzing. Finally, AFL and mbed TLS are also explained in detail.

2.1 The fuzzing process

A typical fuzzing process is illustrated in the diagram below. This section will describe the parts of the diagram in detail.

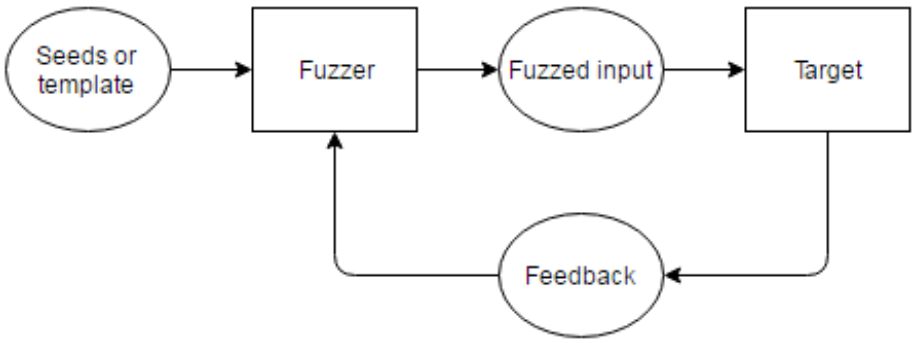


Figure 2.1: *Flowchart of a fuzzing process*

2.1.1 Fuzzer

> "The original work was inspired by being logged on to a modem during a storm with lots of line noise. And the line noise was generating junk characters that seemingly was causing programs to crash. The noise suggested the term 'fuzz'. –bart miller" [1]

Fuzzing is a simple process of feeding unexpected input to a program as a method of inducing crashes and unintended behavior. The specifics of the fuzzer, such as how it generates input and how it takes previous input into account, varies from fuzzer to fuzzer.

The headings below describe the major fuzzing categories in terms of fuzzing approaches and awareness of the target's structure and expectations of the input.

Mutational

A mutational fuzzer creates new data by applying different *mutation strategies* to existing data, one of the basic strategies being flipping bytes or bits. For instance, if a bit that represents a boolean value is flipped, that boolean is consequently inverted. Another strategy is to insert interesting values, such as maximum and minimum values of integers and values close or equal to zero. [2, pp. 8]

Mutational fuzzers require little prior knowledge of what input the target program expects, making them easy to set up and use. However if the input must be intricate like when parsing files, they may never test properly due to the input being rejected outright. [2, pp. 6-7]

Generational

A generational (also called grammar-based) fuzzer generates new data based on a template. The template is a specification of the input structure the target takes. For instance, a template can be split into fields with a name, length and data type.

Having a template solves the problem of mutational fuzzers when fuzzing programs which take highly-structured input, such as protocols and file formats. However, it must be created by the user and making a proper template can be time-consuming. The quality of the template is crucial, as an unfinished template will impede the fuzzer from generating effective and valid data which might cause it to miss interesting results.

Furthermore, there is no standardization of implementing templates, which makes it difficult to use templates across different generational fuzzers. [2, pp. 6]

Black-box

A black-box fuzzer fuzzes a target for which source code is unavailable. Feedback to the fuzzer is usually provided by reading the exit status of the executed target. A naive black-box fuzzer will produce completely random data, but most use this feedback to improve future tests. [3, pp. 2]

Black-box fuzzers are generally less efficient than white-box fuzzers, because it has less awareness of the target's structure. [1]

White-box

A white-box fuzzer fuzzes a target by making use of the source code, often using it to confirm that a certain code path was hit by a certain test case. [1] It remains useful when fuzzing more complex targets such as network applications and parsers whereas black-box fuzzers would likely never reach deep code paths. Because of this, it is reasonable to expect a white-box fuzzer to always yield better results than a black-box fuzzer.

To increase the intelligence of white-box fuzzers, symbolic or concolic execution may be used. Symbolic execution is the execution of a program where variables are given a symbolic meaning instead of a value. By tracing the reads and writes of the symbolic variables, an

execution tree can be built, which can theoretically describe all possible execution paths and how to reach them. The tree can aid the fuzzing process to cover all paths. [4, pp. 386-387] Concolic execution is a variant of symbolic execution where the target binary is also executed, hence the name being a combination of the words *concrete* and *symbolic*.

2.1.2 Seeds

Seeds are the starting point of the fuzzing process and is given by the user. For a mutational fuzzer this means the initial test cases. For a generational fuzzer this means some default values which will remain constant in its template such as a header.

Especially mutational fuzzers rely heavily on the seeds for productive testing. Seeds should feature distinct and large implications to the target's control flow in order to maximize code coverage. If not, large parts of the target may never be exercised, effectively hindering any testing of them. [2, pp. 9]

2.2 Why fuzzing matters

2.2.1 Benefits

Fuzzing has several benefits. First and foremost, it is incredibly accessible. Most fuzzers are convenient and easy to set up, some requiring as little as a target binary and a few test cases. [5] Aside from creating the seeds and or a template, fuzzers can be completely autonomous after their starting point. Furthermore, the fuzzer can be run on one core silently in the background. For these reasons there are hardly any drawbacks considering how little time and resources the user has to commit.

Also, fuzzing complements the shortcomings of conventional testing. Human-written tests are inclined to conform to the ideas that the human already has about the code. Fuzzing is completely free of this bias and can generate unexpected tests that could not have otherwise been thought of by humans. [5] Humans are also much less persistent. After thorough testing, a human tester will get tired and lazy, meanwhile the fuzzer will run indefinitely.

2.2.2 Impact

Fuzzing has proven to be an effective technique for identifying software bugs and vulnerabilities. For instance, the SDL (Security Development Lifecycle) outlines fuzz testing for software verification, a mandatory policy established by Microsoft in 2004. [6, 7] Additionally, white-box fuzzing makes up a third of all vulnerabilities found in Windows 7. [3, pp. 6] In April 2014, the Heartbleed vulnerability was discovered in the OpenSSL library used by the majority of web servers. [8, pp. 2]

2.3 Limitations of fuzzing

While fuzzing is a very good way to find simple bugs, more complicated logic errors will often elude the fuzzer. Consider the example code below:

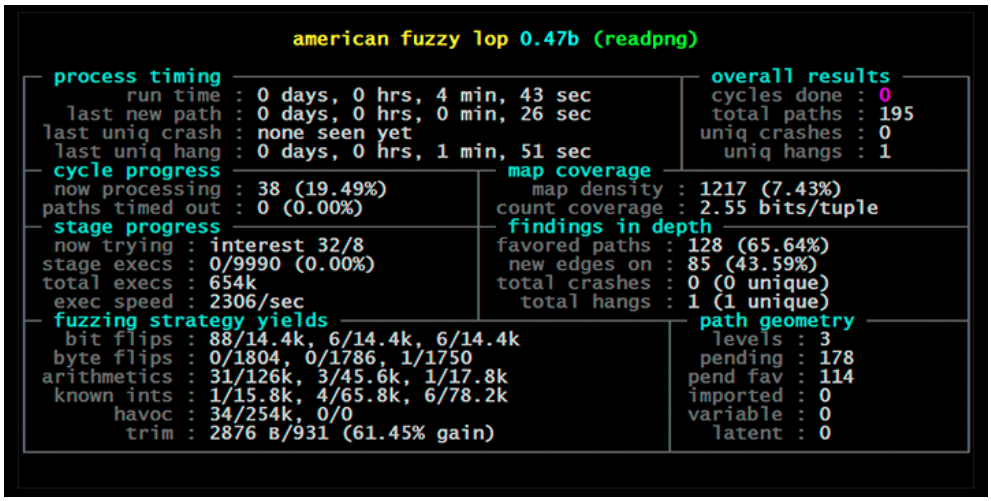
```
void foo(int x) {
    if (x == 5) crash();
}
```

Here, there is only 1 in 2^{32} chance of executing crash(), assuming black-box fuzzing with randomized input data. Keeping in mind black-box fuzzing is usually more sophisticated than in this example, it underlines the inherent limitations of black-box fuzzing in terms of code coverage. [3, pp. 2]

Depending on the specifics of the implementation, white-box fuzzers may not scale well. Path explosion is a common problem related to symbolic execution, where a complex program can not be symbolically executed due to potentially infinite loops. [9, pp. 87]

Although a largely unexplored area, actions can be taken to delay or completely hinder the discovery of bugs and vulnerabilities by fuzzing, known as *anti-fuzzing*. Decreasing performance and masking crashes are two of many approaches to achieve this. [2, pp. 34]

2.4 AFL



AFL (american fuzzy lop) is an open source mutational fuzzer developed by Michal Zalewski. The tool requires minimal tweaking and has a useful UI to track progress and interesting statistics (see picture above). In essence, the algorithm is as follows:

1. Load a queue with seeds.
2. Take the next input from the queue.
3. Trim the input without affecting the target's behaviour.
4. Mutate the input using several fuzzing strategies.
5. Add the mutations deemed interesting to the queue.
6. Go to 2.

It can be used for both white-box fuzzing and black-box fuzzing. It can white-box fuzz C, C++ and Objective-C. Additionally, variants of AFL allow fuzzing of Python, Go, GCJ Java, Rust and more. [10] [11]

2.4.1 AFL's life cycle

Below is a flowchart of the core components in AFL in relation to each other.

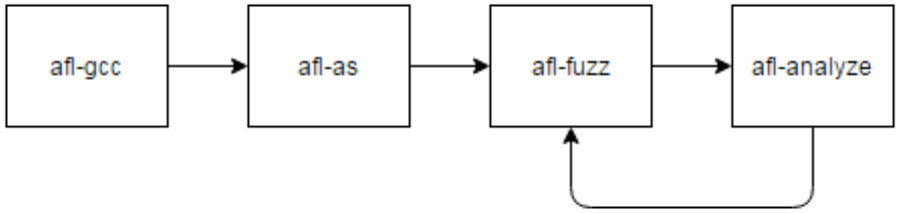


Figure 2.2: *Flowchart of AFL*

Afl-gcc is a drop-in replacement for GCC or clang and passes its output to afl-as. Source code of the target is required for this component and enables white-box fuzzing.

Afl-as is a wrapper for as and instruments the compiled target by injecting assembly code which captures branch coverage. It outputs an executable binary file which is then used by afl-fuzz.

Afl-fuzz, as expected, is the fuzzing component. It takes a binary and fuzzes its input according to the findings of afl-analyze. For black-box fuzzing a target, this component should be used directly. Afl-fuzz is also responsible for printing the UI and attempting to trim data while maintaining the same behaviour.

Afl-analyze uses an input file, changes it and observes how the execution path is affected, using the instrumentation from afl-as if provided. The findings are communicated to afl-fuzz for further and improved mutation.

The cycle of mutating data, sending it to the target, analyzing the execution and repeating continues forever until AFL crashes or is interrupted somehow, normally by the user. [12] [10]

2.4.2 AFL's trophy case

AFL's official website lists the many vulnerabilities and interesting bugs discovered by AFL. The findings include security critical software as OpenSSL, OpenSSH and nginx, in the web browsers Mozilla Firefox, Internet Explorer and Apple Safari, and other well-known software such as Libre Office and Adobe Flash. [10]

2.4.3 Use case with AFL

To learn how to use AFL in practice, consider the following small program *sample.c*:

```
#include <stdlib.h>
void vuln(char* dst) { gets(dst); }
int main( void ) {
    char buf[24];
    vuln(buf);
    return 0;
}
```

The program above first allocates a buffer, consisting of 24 characters, and then makes a call the the function *vuln*. The function reads a string from stdin, and terminates when it encounters a nullbyte or end-of-file.

If the user supplies more than 24 characters to the program, the return address saved on the stack by the caller of main, will be overwritten, as well as the stack pointer. This is known as a stack-based buffer overflow.

The next step is compiling the program with afl-gcc.

```
~$ afl-gcc sample.c -o sample
```

Now all that is needed is to provide a valid sample input, and the fuzzing can begin.

```
~$ mkdir inp
~$ echo "AAAA" > inp/test1
~$ afl-fuzz -i inp -o out -- $(pwd)/sample
```

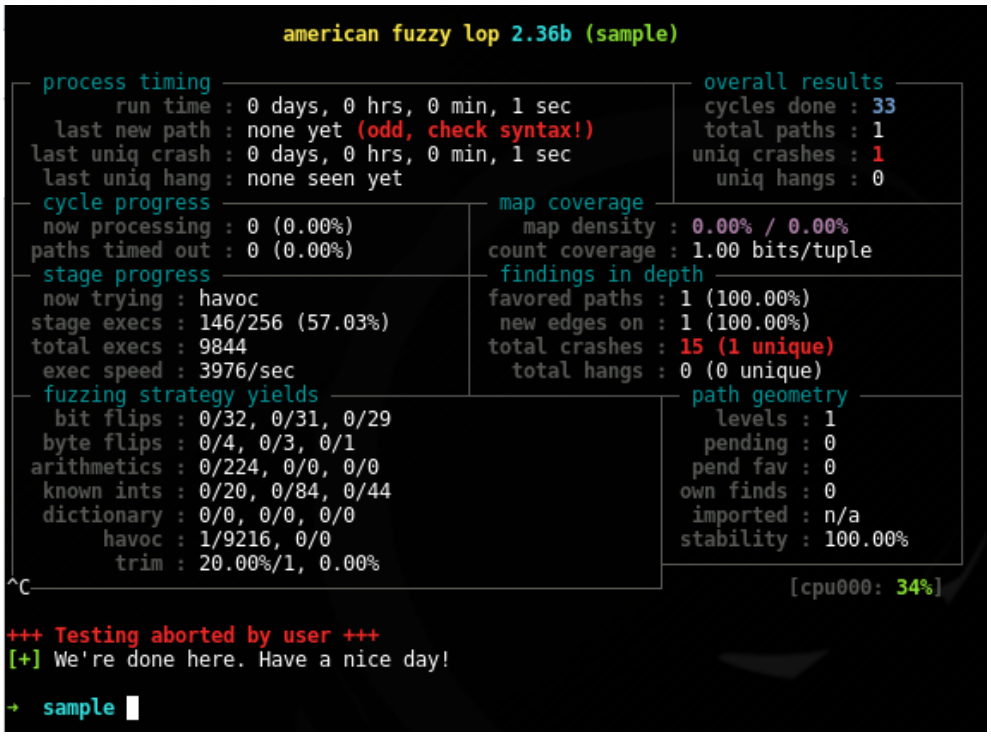


Figure 2.3: *AFL status screen after fuzzing sample.c*

As expected, AFL finds the bug in the matter of a second.

2.5 SSL/TLS

SSL, an abbreviation for Secure Socket Layer, is a standard security protocol for establishing encrypted communication between a server and a client. An example of SSL usage is a website (server) and a browser (client). SSL allows for secure transmission of sensitive information such as passwords. TLS (Transport Layer Security) is the successor of SSL and they are frequently referred to interchangeably. [13]

2.5.1 OpenSSL

The need for a secure SSL library becomes apparent when assessing OpenSSL. OpenSSL is an open-source SSL library which is used by many applications to communicate securely over the internet. Over the years, a plethora of bugs have been found in the library. A prime example is the Heartbleed bug which allowed attackers to read sensitive and supposedly protected information. [8, pp. 2] Several memory corruption bugs in OpenSSL enabled a major virus known as *Slapper Worm* to propagate itself between computer systems. [14, pp. 4]

2.5.2 mbed TLS

Mbed TLS, formerly known as PolarSSL, is an open source SSL library supported by ARM Limited. It started as an official continuation of XySSL in 2008. Mbed TLS aims for an easy implementation of cryptographic and SSL functionality in embedded applications. It is written in the portable language C and the components in the library are loosely coupled which allows developers pick which features they wish to use. [15]

As one would expect from a security-related library, Mbed TLS uses extensive testing and quality assurance. Among the over 6000 automated framework tests, there are tests using live SSL connections, memory checks and more. These tests are performed on different operating systems, compilers and IDEs. The same fuzzing tool which uncovered the famous Heartbleed vulnerability is also used to test Mbed TLS. [16, 17]

Chapter 3

Method

This chapter outlines which methods were used to fuzz the target application, as well as their respective pros and cons.

3.1 Preface

Fuzzing can be a very powerful tool, but it has some disadvantages, especially when working with a large codebase.

Before throwing the fuzzer at the target in question, a few things have to be considered. Fuzzing the entire project at once may not be cost-effective and difficult to set up. An example of this would be non-linear programs with a high degree of parallelism (such as servers), which can have a large memory footprint.

One must also be aware of the flow of user-supplied input in the target, especially if the interest in the application is primarily security related. A null pointer dereference may be a bug, but if no user-supplied input is involved, it is hardly security-critical.

Another problem, which is frequent if one decides to split up the fuzzing into different functions, is that the target application may expect certain fields to be properly initialized.

3.2 Choosing fuzzer

When selecting a fuzzer we assessed ease of use, features, compatibility and cost. Ideally the fuzzer would also be quick to set up, require no prior knowledge of the target and make use of the source code to improve the fuzzing results. AFL was deemed an excellent fit for these requirements. Beyond being a mutational white-box fuzzer, it was well established, compatible with our systems and completely free.

3.2.1 Acquiring fuzzer

AFL version 2.36b was downloaded using the system's package manager on 2017-04-28. The system the fuzzer ran on was a virtual machine running the Linux distribution Debian Jessie.

3.3 Choosing targets

There were many more options when choosing the target. We wanted the target to be significant in terms of size and security meanwhile remaining relatively easy to fuzz. Of course, its source code had to be available for the white-box fuzzing. In the end the mbed TLS library was chosen. It was open-source and its plethora of test cases would help with getting started. Additionally, it claimed to be largely decoupled which would make fuzzing individual components easier.

3.3.1 Acquiring target

The source code for mbed TLS was downloaded from GitHub using a git utility. The source code was downloaded on 2017-04-28 from the development branch at `https://github.com/ARMmbed/mbedtls`. Two target applications were chosen from the sample programs provided in the mbed TLS library.

x509 Certificate Parser

The first one, *crl_app*, was of interest mostly due to the fact that it is very easy to fuzz. The applications purpose is to parse certificate files and output information about them. Finding a vulnerability in this application would be interesting because in certain situations the client has to provide a certificate to prove authenticity.

SSL Server

The second one, *ssl_server*, was of interest due to the fact that it is the meat of the library. The SSL server application utilizes a lot of the code in the library, and interacts with remote users.

3.4 Approach

The initial step in the fuzzing process was to find sample programs which show users how to use the library. Mbed TLS includes some samples with their libraries. To minimize the amount of work required to start fuzzing the target application, it was decided that the sample binaries would be fuzzed instead of custom written programs targeting specific library functions.

3.4.1 Benefits

This particular approach has a few benefits. First and foremost, it requires a minimal amount of work. Secondly, for the fuzzer to find a vulnerability we may need some context around the function we are fuzzing. A simple example to strengthen this point would be a scenario where we have an overwrite of a function pointer in some function, and then use the very same function pointer outside of the aforementioned function. Fuzzing the function in which the overwrite occurs would yield no results, while fuzzing the entire application would.

3.4.2 Drawbacks

The approach that was chosen also has some drawbacks. Firstly, it can be very slow. Every time a test is fed to the target application it may have to set up structures, initialize memory, read configuration files and so forth. Secondly, an SSL-library is especially difficult to fuzz, since it is not deterministic. SSL and TLS rely heavily on random number generation and timestamps. We are also faced with the problem of feeding the application with input, since AFL does not support sockets.

3.4.3 Solutions

Fortunately, it is possible to preload libraries in Unix. Preloading libraries will allow the user to override certain library functions. There is a library for this available on `https://www.github.com/zardus/preeny`. The library's most relevant components are desock, desrand and derand. These will ensure that the program reads input from stdin instead of sockets and also ensure that any random number generation is deterministic.

However, this is not enough to make the target application execute in a deterministic fashion, since timestamps play an important role in the SSL protocol. To counter this problem a small detime utility was written:

```
#include <unistd.h>
time_t time(time_t* t) {
    return (time_t)1493569431;
}
```

Now the process is more or less deterministic. To get sample input for the application derand, desrand and detime were loaded. A successful SSL handshake was captured, and the TCP data dumped. Now running the application and feeding it the prerecorded SSL handshake, it successfully completes the handshake.

Chapter 4

Results

This chapter presents and briefly compares the results achieved by fuzzing the two target applications using the previously described method. If the target program returns any abnormal exit code, it is considered a crash. A hang occurs when the program takes longer to execute than the timeout specified when running AFL. Program stability is the deviation in execution path between multiple test runs using the same test data. This means that the process is not entirely deterministic.

4.1 x509 Certificate Parser

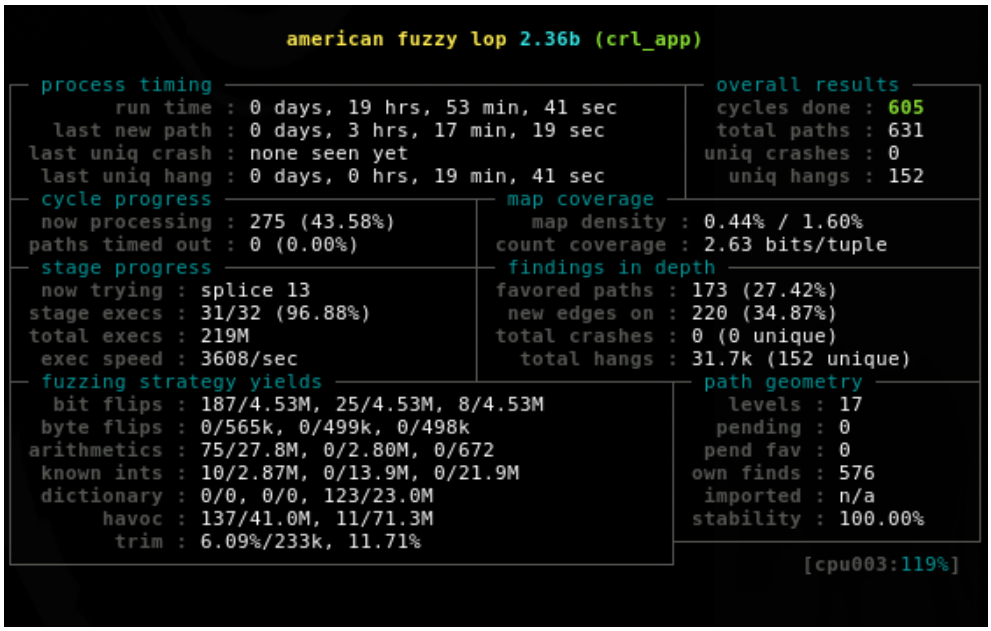


Figure 4.1: *AFL status screen after fuzzing crl_app*

The crl_app was fuzzed for approximately 20 hours. As shown by the image above, the application stopped traversing new paths approximately 17 hours in.

The fuzzing yielded 152 unique hangs and no unique crashes. The execution speed was approximately 3600 executions per second and program stability was 100%.

4.2 SSL Server

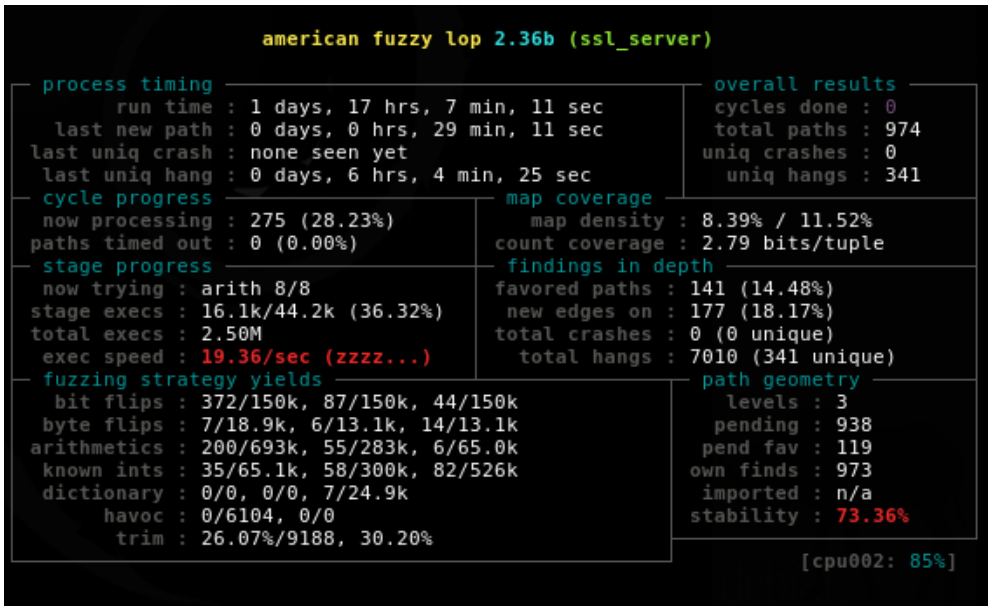


Figure 4.2: *AFL status screen after fuzzing ssl_server*

The ssl_server was fuzzed for approximately 41 hours. The application had not stopped traversing new paths when the fuzzer was terminated.

The fuzzing yielded 341 unique hangs and no unique crashes. The execution speed was approximately 20 executions per second and program stability was 73.36%.

Chapter 5

Discussion

This chapter reflects upon the results presented in the previous chapter.

5.1 Execution Speed

One thing to note is that the execution speed is vastly different between the two applications. This difference is due to the fact that running the ssl_server requires a lot of setup to work properly. It has to read test certificates from files and initialize structures.

AFL provides a few solutions to increase speed, such as the persistent mode. This mode enables AFL to run multiple different test cases on a single execution of the program. Consider a web server. It may start by reading a configuration file and setting up memory, which takes a lot of time. The web server then moves on to the segment which parses incoming requests. Through persistent mode, AFL can run multiple test cases on the same instance of this web server and restart after a certain number of executions. Unfortunately, this mode is still in its experimental stage, and did not work for our particular setup. The execution speed increased, but the path discovery was reduced.

There are other ways to improve execution speed, such as the deferred forkserver mode and parallel fuzzing. These methods were also tested but did not improve our results and were therefore not included.

5.2 Hangs

Both applications experienced multiple unique hangs. These cases are not as interesting as crashes, since they most likely depend on the program getting stuck while attempting to perform some blocking IO-operation, like trying to read input from a socket.

However, they deserve some degree of attention, since a hang may be caused by some other memory issue. For instance, an overwrite of a file descriptor or a corruption might cause the program to get stuck in an infinite loop. A hang could also potentially be used for denial of service attacks.

There are a few ways to solve this issue. Manual analysis of the tests can be done using some memory error detector, e.g. valgrind. Valgrind will, among other bugs, show out-of-bounds read or write as well as usage of uninitialized variables. Another possible way would be to run the application under ltrace or strace, two programs which trace the library calls and system calls of the application respectively. This could reveal where the application stops working. All tools mentioned above are most likely available in your Linux distribution's package manager.

Another important thing to note is that not all hangs reproduce outside of AFL and a lot of test cases could be discarded by automatically executing the programs with a high timeout and seeing if they terminate normally.

5.3 Crashes

No crashes were encountered during the fuzzing. This is probably due to the extensive testing that mbed TLS undergoes before each release. However, since the fuzzing of the SSL server was not completed, there may still be unexplored code paths which may yield crashes.

To improve the results and increase the probability of finding bugs, more time should be spent on setting up the environment, improving execution speed and also fuzzing more targets.

If a crash had been found, it could have had serious implications. Some crashes could allow remote code execution systems running mbed TLS, and some crashes could be used in denial of service attacks.

5.4 Stability

When fuzzing, optimally a program stability of 100% should be maintained, otherwise AFL cannot discern between meaningful effects and random changes when tweaking some input file. We were unable to account for all sources of randomness which caused the stability when fuzzing ssl_server to be 73.36%. Mbed TLS, in addition to using native sources of random, also implements its own pseudo-random number generator. With that said however, this PRNG was only used for the transmission of messages, and the SSL handshake was deterministic.

5.5 Other statistics

The rest of the statistics supplied by AFL are not interesting for our problem statement. AFL shares a lot of statistics with the end user, but they are mostly for trivia. The curious reader may take a look at `https://github.com/stribika/afl-fuzz/blob/master/docs/status_screen.txt` for an in-depth explanation of each field.

Chapter 6

Conclusion

To relate the results to the problem statement, the answer is no. AFL can not be used to easily find bugs in a well-tested library such as mbed TLS. AFL has previously been used to find bugs in mbed TLS, but how much setup was required is unknown. This outcome could have been anticipated considering our lack of experience and the highly tested target library.

6.1 Future work

There are many things that can be improved upon. First, the platform on which the fuzzing was done posed one significant problem. Since it was executed on a virtual machine on a normal desktop PC, the time spent on fuzzing was quite limited. AFL should have cycled over the input queue at least a few times before being terminated.

The second improvement would be execution speed. Due to time constraints, the sample programs supplied by mbed TLS were used. To improve execution speed, we suggest writing your own test programs. This reduces redundant code which will decrease startup time and therefore increase execution speed. Another benefit is that you can tailor those tests to also fuzz test more specific parts of the library. One could also look into using AFL's built-in features for improving execution speed that were discussed in the previous chapter.

The third improvement would be program stability. When fuzzing the ssl_server application, the program stability was not optimal. This caused non-deterministic fuzzing meaning that different behaviour was exhibited by the target despite being fed the same input.

Bibliography

[1] John Neystadt. Automated Penetration Testing with White-Box Fuzzing. *Unknown*, 2008.

[2] Emil Edholm David Göransson. Escaping the Fuzz. *Unknown*, 2016.

[3] David Molnar Patrice Godefroid, Michael Y. Levin. SAGE: White-box Fuzzing for Security Testing. *ACM*, 2012.

[4] James C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 1976.

[5] Parul Garg. Fuzzing - Mutation vs. Generation. *Exploit Development*, 2012.

[6] Unknown. Evolution of the Microsoft Security Development Lifecycle, n.d.. URL `https://www.microsoft.com/en-us/SDL/Resources/evolution.aspx`. [Accessed 2017-05-01].

[7] Unknown. Microsoft Security Development Lifecycle, n.d.. URL `https://www.microsoft.com/en-us/sdl/default.aspx`. [Accessed 2017-05-01].

[8] Zakir Durumeric et al. The Matter of Heartbleed. *ACM*, 2014.

[9] Koushik Sen Cristian Cadar. The challenges - and great promise - of modern symbolic execution techniques, and the tools to help implement them. *ACM*, 2003.

[10] Unknown. american fuzzy lop homepage, n.d.. URL `http://lcamtuf.coredump.cx/afl/`. [Accessed 2017-05-01].

[11] Unknown. american fuzzy lop readme, n.d.. URL http://lcamtuf.coredump.cx/afl/README.txt. [Accessed 2017-05-14].

[12] Unknown. Technical "whitepaper" for afl-fuzz, n.d.. URL http://lcamtuf.coredump.cx/afl/technical_details.txt. [Accessed 2017-05-14].

[13] Unknown. SSL Library from mbed TLS: Easy to use open source SSL in C, n.d.. URL https://tls.mbed.org/ssl-library. [Accessed 2017-05-14].

[14] Peter Szor Frederic Perriot. Analysis of the Slapper Worm Exploit. *Unknown*, 2003.

[15] Unknown. SSL Library mbed TLS/PolarSSL, n.d.. URL https://tls.mbed.org/. [Accessed 2017-05-01].

[16] Unknown. mbed TLS automated testing and Quality Assurance - Knowledge Base, n.d.. URL https://tls.mbed.org/kb/generic/what-tests-and-checks-are-run-for-mbedtls. [Accessed 2017-05-14].

[17] Unknown. Fuzz Testing to Find Hidden Vulnerabilities | Synopsys, 2017. URL https://www.synopsys.com/software-integrity/security-testing/fuzz-testing.html. [Accessed 2017-05-14].