School of Mathematics and System Engineering

Reports from MSI - Rapporter från MSI

Software Security Testing

A Flexible Architecture for Security Testing

Martin Andersson

Aug
2008

MSI
Växjö University
SE-351 95 VÄXJÖ

Report 08082
ISSN 1650-2647
ISRN VXU/MSI/DA/E/--08082/--SE

Martin Andersson

Software Security Testing

A Flexible Architecture for Security Testing

Master's Thesis

2008

Växjö University

Abstract

This thesis begins with briefly describing a few vulnerability classes that exist in today's software. We then continue by describing how these vulnerabilities could be discovered through dynamic testing. Both general testing techniques and existent tools are mentioned.

The second half of this thesis present and evaluates a new flexible architecture. This new architecture has the ability to combine different approaches and create a more flexible environment from where the testing can be conducted. This new flexible architecture aims towards reducing maintenance and/or adaptation time for existing tools or frameworks. The architecture consists of a given set of plug-ins that can be easily replaced to adapt test as needed. We evaluate this architecture by implementing test plug-ins. We also use this architecture and a set of test plug-ins to generate a fuzzer targeted to test a known vulnerable server.

***Key-words*:** Security testing, dynamic analyses.

Acknowledgments

I would like to thank the people at Combitech AB for providing me with the opportunity to write this work. I would also like to thank my contact person at Växjö University, Jonas Lundberg, for providing me with useful feedback. Last, but not least, I would like to thank my girlfriend for her support and understanding.

Contents

1 Problem description **1**
1.1 Background 1
1.2 Motivation 1
1.3 Purpose 1
1.4 Target group 1
1.5 Problem discussion 1
1.6 Delimitation 1

2 Theory, Related Work **2**
2.1 Software security definitions 2
2.2 Source code vulnerabilities 2
2.3 Software vulnerabilities counter-measure 6
2.4 Software security testing 8
2.5 Dynamic analyses/testing 8
2.6 Test cases 10
2.7 Related Approaches 11

3 A Flexible Architecture for Security Testing **14**
3.1 Introduction 14
3.2 Requirements 15
3.3 Design 16
3.4 Implementation 19

4 Evaluation and Results **23**
4.1 Testing environment 23
4.2 Evaluation 25
4.3 Results 26

5 Summary, Conclusions, and Future Work **27**
5.1 Summary 27
5.2 Conclusions 27
5.3 Future Work 28

References **30**

A Internal format **31**

B Example fuzzer **32**

C Fuzzer execution **46**

D User Interface **48**

1 Problem description

This thesis approaches a large subject and is therefore very restricted in the material. The motivations behind these restrictions and the purpose of the thesis are presented throughout this chapter.

1.1 Background

Each day new software security vulnerabilities are discovered. Vulnerabilities could constitute a great threat against anyone using the concerned software. The security aspects are gaining more and more ground within the software development process. There exist two main methodologies for evaluating software security, namely static and dynamic. While static methods are concerned with evaluating the source code itself, dynamic methods focus on the software during run time.

Evaluating a big and complex application from source code can be a very hard and time consuming task. Dynamic analysis allows the testing team to focus on the most exposed interfaces of the software and take the perspective of a possible attacker.

1.2 Motivation

The area of dynamic analysis and testing has lately been a popular target for research. This area is highly interesting since new techniques are presented on a regular basis. Since new tools and techniques usually has to be modified to fit the target software or the evaluaters needs, alot of human effort in the form of development is often needed when a new approach surface. Minimization of development between different approaches is therefore of high interest.

The subject of dynamic analysis has also been proposed by the company Combitech AB located in Växjö.

1.3 Purpose

The purpose of this work is to present a new flexible architecture for dynamic detection of software security flaws.

1.4 Target group

This thesis can be read by anyone with a background within computer science and a basic knowledge of computer security and software engineering.

1.5 Problem discussion

One of the major problems when doing any kind of security testing/evaluation are the human resources it consumes. Therefore the problem formulation of this thesis is: *How can dynamic analysis be used in software security testing to aid the auditor and not consume a large amount of human resources?*

1.6 Delimitation

This thesis focuses on software written in C or C++. The target system is delimited to Linux systems running on x86 processor architecture. Due to time constraints this thesis focus solely on the network interfaces, which is a commonly provided by software.

2 Theory, Related Work

This chapter tries to refresh the reader's memory or give a brief overview about different security concerns. Due to time constraints, the purpose is not an exhaustive description of all aspects. The reader is encouraged to deeper study the aspects mentioned in this chapter if necessary.

2.1 Software security definitions

Some of the terminology used in this thesis is explained in this Section.

2.1.1 Vulnerabilities

The following citation is taken from [1] in an attempt to explain the term *vulnerability*:

> ”A flaw in a system's security that can lead to an attacker utilizing the system in a manner other than that which the designer intended. This can include impacting the availability of the system, elevating access privileges to an unintended level, complete control of the system by an unauthorized party, and many other possibilities.”

Because of the risks enumerated in the citation above, the ability to discover vulnerabilities is crucial.

2.2 Source code vulnerabilities

This chapter focuses on some of the common vulnerability classes common in C and C++ implementations. The idea is to give the reader a, brief, technical description of these vulnerability classes. For a deeper discussion of these classes the reference material is recommended.

2.2.1 Stack-based buffer overflows

The term *buffer overflow* is constantly repeated in the area of software vulnerabilities. A buffer overflow occurs when too much data is stuffed into memory [2], causing the memory buffer to overflow. Exactly what this means and how it can be used by an attacker depends on where the buffer is located in memory. This Section investigates buffer overflows on the stack memory segment, we will later investigate buffer overflows that occurs on the heap (to be explained later).

Let us have a simple example to understand how this might look like in source code: In Listing 1 a user controlled command (through the Command Line Interface, further referenced as the CLI) is sent to the function *overflow*. The user controlled data is then copied into a buffer, allocated on the stack. Since the buffer only can hold 10 bytes, the buffer will overflow if the user provides a string of 11 or more characters. In the case with 11 characters the last character will be written outside the bound of the allocated memory for *buf*, possibly causing overwrite of data on the subsequent variable.

The source code given in Listing 1 allow us to input any size of the passed character string, this could lead to arbitrary code execution [2]. To see why, let us take a look at Figure 2.1 that shows how the stack looks like when *overflow()* is called. Since the stack grows upwards (on the selected architecture) if we were able to write outside the allocated buffer we would overwrite the saved *ebp* (extended base pointer) and *eip* (extended

Listing 1: Simple stack-based overflow

```
1  #include <string.h>
2
3  void overflow(char *str) {
4      char buf[10];
5      strcpy(buf, str);
6  }
7
8  int main(int argc, char** argv) {
9      overflow(argv[1]);
10     return 0;
11 }
```

instruction pointer). Since the *eip* register points to the next instruction to be executed, if we could overwrite the saved *eip* and set it to whatever value we would like we could decide what could to execute when the saved *eip* is restored. If we somehow had executable instructions somewhere in the memory, *eip* could be set to point to these instructions. The instructions would be executed with the same privileges as the program running [1].

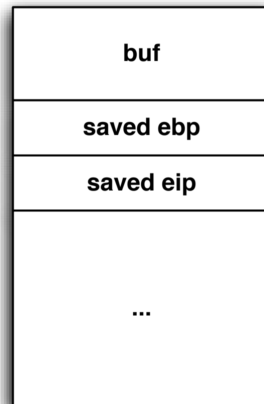

Figure 2.1: Stack when executing overflow function

Although being able to manipulate the *eip* register might be the most dangerous scenario, another threat is the ability to modify the *ebp* register. Since the stored *ebp* defines the local scope, modifying the stored *ebp* modifies the scope and thus variable values could be manipulated during execution.

The key factor for avoiding buffer overflows on the source code level is bounds checking [2]. A piece of software should never copy more data into memory than it have allocated for the buffer. The buffer overflow demonstrated in Listing 1 could have been avoided by using *strncpy()* instead of *strcpy()* since *strncpy()* features bounds checking [3]. The secure version of Listing 1 is shown in Listing 2. In Listing 2 the stack based overflow has been eliminated by not copying more then 10 bytes of data to *buf*. This type of bounds checking is crucial in order to avoid serious overflow bugs in the system.

Listing 2: No stack-based overflow

```
1  #include <string.h>
2
3  void overflow(char *str) {
4      char buf[10];
5      strncpy(buf, str ,10);
6  }
7
8  int main(int argc, char** argv) {
9      overflow(argv[1]);
10     return 0;
11 }
```

Listing 3: Simple heap overflow

```
1  #include <stdlib.h>
2  #include <string.h>
3
4  void heapOverflow(char *string) {
5      char *p1=(char *)malloc(10);
6      char *p2=(char *)malloc(10);
7      strcpy(p1, string);
8      free(p1);
9      free(p2);
10 }
11
12 int main(int argc, char** argv) {
13     heapOverflow(argv[1]);
14     return 0;
15 }
```

2.2.2 Heap overflows

Heap overflows occurs on the heap memory segment. This Section considers the memory segment of an executable where dynamically allocated variables are located. The heap segment may also be used for global and very large variables [1].

Heap overflows occurs, just as stack-based overflows (section 2.2.1), when memory bounds are exceeded [1]. However, when exploiting a stack based overflow our purpose is usually to overwrite the saved *eip* registry and take control over execution. When it comes to heap overflows we need something located after the vulnerable buffer in order to have something to overwrite [1].

Let us look at Listing 3 for an example of how a simple heap overflow could look like. When the code in Listing 3 is executed with an argument exceeding the buffer length (10) it will cause a core dump.

The problem in Listing 3 is that when the buffer *p1* is overflowed, it corrupts the header of the *p2* buffer [4]. The header of buffer *p2* is then read by the *free()* function causing an error since the header data is corrupted.

Exploiting heap overflows can be done by gaining control of the block header, and thus

Listing 4: Format string usage

```
1  #include <stdio.h>
2
3  int main() {
4      int number=10;
5      printf("%d %x \n", number, number);
6      return 0;
7  }
```

Listing 5: Simple format string vulnerability

```
1  #include <stdio.h>
2
3  void vulnFunc(char *str) {
4          printf(str);
5  }
6
7  int main(int argc, char **argv) {
8      vulnFunc(argv[1]);
9      return 0;
10 }
```

control over references to other blocks. Controlling these references usually gives us the control to write controlled data to an arbitrary memory position [4]. This fact could be used by attacks to gain control over execution.

2.2.3 Format strings

A format string is a string used to describe output format in a generic way [1]. As an example, the *printf()* function accepts a format string followed by variables. To demonstrate this fact, consider Listing 4. Listing 4 outputs *10* a when executed. The format string functionality is very handy to format output the way the programmer likes it. This functionality could be misused and therefore might cause serious security vulnerabilities when not used carefully by the software developer.

A format string bug can occur in functions that accept arbitrary many arguments containing format specifiers. The vulnerability occurs when user supplied data is included in the format string [1]. An example is the *printf()* function. Listing 5 shows a simple vulnerable example. The example in Listing 5 works as expected when a regular string (without any special format specifiers) is sent as input. The problem occurs when format specifiers are included without any variables [1]. The result of executing Listing 5 with the format string *"%x %x %x %x %x"* is given in Listing 6. In Listing 6 we can see the data printed after providing a format string. Although no variables were provided, the

Listing 6: Simple format string vulnerability

```
1  martin@vmubuntu: ./format2 "\%x \%x \%x \%x \%x"
2  bf66b000 bff6af58 8048388 bff6c9be bff6af70
```

Listing 7: Integer overflow example

```
1 #include <stdio.h>
2
3 void main() {
4     unsigned int a=0x7FFFFFFF;
5     printf("a=\%d,␣a+1=\%d", a, a+1);
6 }
```

data shown is gathered from the stack [1]. This fact is a security concern by itself, since sensitive data might be stored on the stack. However, format string vulnerabilities could also be used to gain control over execution flow.

2.2.4 Integer overflows

An integer is represented by bits, determining the current value of the integer. In order to allow negative values, the most significant bit is used to denote the sign (1 for negative values) [7]. But when unsigned integers are used, the most significant bit is used for representing a positive value instead (since the unsigned data type can not be negative).

The overflow occurs when we try to store a value that exceeds the bounds of the data type [7]. In Listing 7 a simple integer overflow is illustrated by incrementing a variable containing the maximum value of a 32-bit integer. Note that even though the value is stored in an unsigned data type it is interpreted by the *printf* function as a signed integer [6].

When the source code shown in Listing 7 is compiled and executed, it will output: *a=2147483647, a+1=-2147483648*. The reason is because the most significant bit was flipped from 0 to 1.

It is actually up to the compiler developers to decide on how to handle this type of overflow. The behavior might change depending on the compiler implementation [7].

From a security perspective, this could be used to bypass bound checks that only checks for to large values. Consider Listing 8 as an example. The user supplied size is only checked to not exceed the *MAX_STRING_SIZE*. By supplying a value larger then *0x7FFFFFFF* it will be interpreted as a negative value in the main function (since size is declared as integer). However, when the copyStr function is called, the size value is interpreted as an unsigned integer. This will cause a large memory copy, overwriting stack data.

2.3 Software vulnerabilities counter-measure

This thesis has so far shown a number of vulnerability classes that can be used to manipulate the software in various ways. Although this thesis has so far shown some examples of how to avoid these threats there also exist dynamic solutions to stop some of these types of attacks. The reason for this is that humans make mistakes. In order to build more safe software, mechanisms are built into the software in order to make it more resistant to such weaknesses as shown so far. Some of these efforts to build security into the application are described in this section.

Listing 8: Insufficient bounds checking

```
1  #include <string.h>
2  #define MAX_STRING_SIZE 20
3
4  void copyStr(unsigned int size, char *str) {
5      char tmpStr[MAX_STRING_SIZE];
6      strncpy(tmpStr, str, size);
7  }
8
9
10 void main(int argc, char **argv) {
11     int size;
12     size=itoa(argv[2]);
13     if (size > MAX_STRING_SIZE)
14         return;
15     copyStr(size,argv[1]);
16 }
```

2.3.1 Stack cookies

The stack cookie protection prevents stack-based overflows by placing a random 32 bit value on the stack [4]. This 32 bit value is located before the saved ebp and eip registers (relatively from the local variables point of view). The reason for this is to prevent overwrite of the two saved registers. Any overwrite would also overwrite the 32 bit value which would then be detected before the vulnerable function returns.

One problem with this approach is that it does not prevent overwrite of the local variables [4]. Also, techniques exist for gaining control of execution before a protected function may return.

2.3.2 Heap protection

There exist different techniques for hardening the heap header and protect it from being overwritten [4]. One method used is to put an 8 bit value in the header that is checked before the header data is used in order to detect corruption. This method has the same problem as the stack cookies; one might still be able to modify data on the heap although not overwriting any header.

2.3.3 Address randomization

Another method that tries to stop exploitation of software is called address space layout randomization (ASLR) [4]. The ASLR method randomizes where the code and data sections are stored in memory, making it unpredictable. This method, in combination with one of the previously mentioned methods yields a quite strong shield against exploitation.

Some of the problems with the ASLR system are that all code might not be rearrangeable and thus need static locations [4]. Another problem is simple brute force, if the randomization space is not that big, one could brute force the attack until it succeeds.

Listing 9: Code coverage

```
1 #define TABLE_SIZE 10
2 #define STRING_SIZE 10
3 void copyStr(char *src, int index) {
4     char table[TABLE_SIZE][STRING_SIZE];
5     strncpy(table[index],src,STRING_SIZE);
6 }
```

2.4 Software security testing

We have so far seen a number of vulnerability classes, and defences aiming towards protecting the execution environment. Since the methods for protecting the environment are not fool proof, we need methods for detecting these vulnerabilities before software reaches the end users. Before we dig into the details of how implementations may be tested dynamically some of the concepts closely related to software security testing and software testing need to be explained.

2.4.1 Attack surface

An important activity when threat modelling an application is to determine the attack surface. The attack surface includes all the entry points an attack might use to interact with the application [5]. As an example [5] mentions some ways that data could be passed to an application, for instance: files, registry, network, remote procedure calls, shared memory, etc. Due to time constraints limitation of this thesis, only the network interaction will be considered. However, the principles examined next (fault injection) can be adopted to other attack surfaces and are thus applicable outside the scope presented by this thesis.

2.4.2 Coverage

When testing an application for various vulnerabilities, the coverage concept is very important. Code coverage has been defined by [10] as follows:

> ”a technique for determining which pieces of code have run during a specific time period, often by instrumenting the code to be tested with a special instructions that run at the beginning and en of a function. When the code runs, the special instructions record which functions were executed. Recording which code a test case covers can indicate how much additional testing is need to cover all the code.”

By using code coverage we can get some metric of how much of the software has been tested. Of course, this metric has it flaws; there is no guarantee that code that has been executed does not contain security vulnerabilities. Consider Listing 2.4.2.

The function *copyStr()* could be executed without any troubles if index is smaller then 10. The fact that this code was covered (when index happened to be legitimate) does not mean we have to discover the flaw.

2.5 Dynamic analyses/testing

The techniques of finding software vulnerabilities can be divided into two categories, static and dynamic [8]. The static techniques aim towards investigating the source code,

without executing it. This can be done by simply reading the source code in order to spot vulnerabilities. For more time efficient static investigations, tools can be used to investigate the source code in a variety of ways.

Dynamic techniques try to discover vulnerabilities through execution of the source code [8]. This means that the source code does not need to be present in order to analyze, since the software could be distributed as a binary. Instead the software is validated through a series of test cases. In order to enhance this technique, the software might be observed during the testing in order to recognize any deviation.

The remaining of this Section investigates different dynamic methods used to discover vulnerabilities.

2.5.1 Fault injection

Fault injection, as it sounds, is the process of introducing faults into a system [1]. Fault injection have been used for a long time to test both hardware and software.

Faults can be inserted at different levels in a computer system [9]. For example, fault producing circuits can be introduced on the hardware level. This thesis focuses on fault injection into software during runtime.

2.5.2 Fuzzing

Peter Oehlert stated in his article [10] the following about *fuzzing*:

> ”A highly automated testing technique that covers numerous boundary cases using invalid data (from files, network protocols, API calls, and other targets) as application input to better ensure the absence of exploitable vulnerabilities. From modem applications tendency to fail due to random input caused by line noise on fuzzy telephone lines.”

In other words, we are using the attack surface to send semi valid data to a program. In some cases, applications might crash or behave in an unexpected way because the software developer failed to implement proper validation of the data.

The problem with testing applications with negative test cases is that there are so many cases [10]. Fuzzing handles this by automating the testing, allowing human resources to be spent elsewhere. However, applying randomness into test cases may result in an infinite field of test cases. In order to counter this fact, to make testing more efficient, only certain boundaries or patterns might be used. There exist two main methodologies for determining the data used through test cases, *data generation* and *data mutation* [10].

- A generation fuzzer generates the test cases by itself by inspecting some specification. This means that a specification needs to written for each piece of software we wish to fuzz. It also means that the fuzzer needs to be able to parse this specification in order to construct the test cases [10].
- The data mutation technique is to insert faults in a known set of valid data. This can be done by inserting the fuzzer between a client and server (figure 2.2). If the target is the server, the fuzzer reads the data sent by the client and then forwards different permutations of this data. This solution might be more appropriate when fuzzing very large or unknown protocols [10].

Both these methods can be used to change random data to create semi valid input. To fine tune this semi valid data some intelligence could be built into the fuzzer.

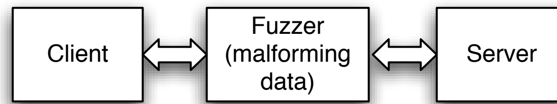

Figure 2.2: Fuzzer intercepting malforming data between client and server.

Fuzzer intelligence

An intelligent fuzzer does not just randomly change bits to produce semi valid data; instead knowledge about the specific format is used to produce data. The big advantage of intelligent fuzzers is the ability produce test cases that might get more code coverage (explained in Section 2.4.2) [10]. However, changing bits on a random basis might produce test cases that are far away from the intelligent test cases allowing discovery of bugs that may occur because the developer did not expect such obscure data.

Intelligence might also include operations such as calculating and appending hash values [10]. These types of operations might have to be included as well in order for the packet to not be discarded immediately by the target.

2.6 Test cases

In previous sections we investigated what vulnerabilities looks like and how they are exploited. This Section tries to explain how test cases can be constructed in order to trigger these vulnerabilities so that they can be discovered. As noted in Section 2.5.2, building some type of intelligent generation of test cases allows more code coverage and thus is more probable of discovering vulnerabilities. Let us examine the test cases for each of the vulnerability classes noted in the background section.

2.6.1 Buffer overflows

Buffer overflows occur when too much data is stuffed into a memory buffer (see Section 2.2.1 and 2.2.2). This type of vulnerability might be triggered by using longer data sequences then expected. As an example Leon Juranic [15] lists the following example in the context of SMTP fuzzing:

> ”mail from: test@localhost
> mailmailmailmailmailmailmailmail from: test@localhost
> mail fromfromfromfromfromfromfrom: test@localhost
> mail from::::::::::::::::::::::::::::: test@localhost
> mail from: testtesttesttesttesttesttesttesttest@localhost
> mail from: test@@@@@@@@@@@@@@@@@@@localhost
> mail from: test@localhostlocalhostlocalhostlocalhost”

The first line denotes the valid command. From the valid command different invalid permutations are derived. These specific derivations are formed in such a way that they might cause complications when parsed or in different ways handled by the target software.

Buffer overflow test cases are also mentioned by Jared Demott [16]. In a simple example a valid part of a command might be replaced by a lot of invalid data (for example 5000 character ”A”). If the target program ever copies this buffer without checking the argument bounds, a buffer overflow is likely to occur.

It should also be noted that since a buffer size might be relative to some user controlled data, many arguments could be fuzzed at the same time. Fuzzing several arguments at once might also cause a different path of execution, which possibly could result in even more code coverage.

2.6.2 Format strings

The format string vulnerability class were described in Section 2.2.3. The same fundamental concept as in buffer overflow test cases (previous section) can be used. Instead of inserting a huge amount of *%n* a few, well chosen, are probably enough to trigger the vulnerability.

These format strings can be placed instead of valid data or inside valid data. As an example, let us consider the SMTP originally used by [15]. But this time we make permutations suited to test the target for format strings instead of buffer overflows (which were used in the original version):

> mail from: test@localhost
> ma%n%n%n%n%n%n%n%n%nil from: test@localhost
> mail fr%n%n%%n%n%n%n%n%nom: test@localhost
> mail from%n%n%n%n%n%n%n%n test@localhost
> mail from: te%n%n%n%n%n%n%n%n%nst@localhost
> mail from: test%n%n%n%n%n%n%n%n%nlocalhost
> mail from: test@local%n%n%n%n%n%n%n%n%nhost

Note that in the above enumerated test cases the format string is always inserted in the middle of each command being fuzzed. Inserting the fuzz string in the middle is of course not required. It could be inserted before, after or on an arbitrary location. The more locations we try, the more convinced we can be that the targeted command does not contain a format string vulnerability. It is important that we do fuzz each of the strings we are able to supply to the target with these format strings in order to cause faulty behaviour on the target.

2.6.3 Numbers

Not only strings should be changed to contain unexpected values, this should also include numbers. As we have seen from Section 2.2.4 large numbers should be included in the test data in order to trigger integer overflows. Values for fuzzing integers were mentioned by [10]:

> ”If your input includes a 4-byte signed integer that should be between 1 and 10, for example, boundary cases would include 0, -1, 11, 12, large negative numbers, and cases around the byte boundaries (2^8, 2^{16}, 2^{24}, 2^{31}).”

The same principle is of course applicable for other type of numbers as well. Since numbers might be used as buffer sizes they are of great importance to test.

2.7 Related Approaches

This Section enumerates and gives a short introduction to a few implemented tools for performing dynamic testing. There exist many different implementations in this area, due to time constraints only a few implementations could be investigated. Note that the

implementations considered here are widely known, or implements new features which are recognized as important. Consider the references throughout this Section for further details about each specific implementation mentioned.

2.7.1 SPIKE

SPIKE is a framework for creating fuzz network clients in order to test target software [11]. SPIKE can be used by a user performing security testing by writing test cases for a specific protocol in SPIKE's script language.

This type of framework enables code reuse for the security tester writing a specific protocol fuzzer. This is very helpful when constructing specific fuzzers and allows the fuzzer developer to focus on formulating test cases, allowing more rapid development.

2.7.2 Peach

Peach is a fuzzing framework created to support various forms of fuzzing, as stated in [12]:

> ”Peach is fuzzer framework intended to provide a good mix of flexibility, code reuse and fast development time. Peach is object oriented in design and was implemented in the Python language for portability. This initial version is subject to change as the code is further refined and tested. This is the first implementation of this code.”

2.7.3 General Purpose Fuzzer

General Purpose Fuzzer (GPF) has the ability to replay network traffic captured by Ethereal [13]. The GPF package allows converting Ethereal captures to human readable files, which allows for human interaction. It also supports plug-ins for calculations on payload and other vital tasks.

2.7.4 SNOOZE

SNOOZE (Stateful NetwOrk prOtocol fuzZEr) is a tool for building network protocol fuzzers for stateful fuzzing [14]. The SNOOZE tool takes a set of specifications, provided by the user (the security tester), and uses them as a base for test case creation. Therefore the user has to specify the protocol description based on available documentation, the user can also supply default values to be used.

The SNOOZE tool keeps track of the protocol state using a state machine module [14]. The state machine tracker makes SNOOZE suitable for fuzzing stateful protocols.

2.7.5 Summary

This Section provides a brief summary over the implementations we have investigated so far. Some of the major differences are provided in the following table.

Name	Type	SRC available
SPIKE	Fuzzing framework	Yes
PEACH	Fuzzing framework	Yes
GPF	Automated tool	Yes
SNOOZE	Automated tool	Unknown

Fuzzing frameworks can be used to produce specific fuzzers in a faster way then writing one from scratch. Frameworks allows for code reuse by providing a common base for all fuzzers the test team might have to code. The exact test cases and how does are derived are left to the tester.

The new generation of fuzzers are toward more automation in the process of creating a fuzzer for a target implementation. Also, complete tools implement some kind of logic to derive test cases from specifications.

3 A Flexible Architecture for Security Testing

In the previous Section some source code vulnerabilities, concepts and dynamic tools were investigated. One problem with the existing approaches today is the lack of flexibility. Either you use a framework that will reduce development time compared with developing a fuzzer from scratch. Even if the overhead is decreased by the framework, development time is still an issue. You can also use automated tools that are locked to one or several built in test case generators. What if we want to try a new method? That will probably cause development of a new tool or framework.

The lack of flexibility in today frameworks and tools causes problems, (e.g.):

- Initilialization time. Each time a new protocol or interface is about to be fuzzed a bit of development effort is needed. Any adjustments to a protcol will also cause development time.

  In the case of an automated tool, it might be required to provide a protocol description. This can be a time consuming task.

- When a new approach surface, the development effort starts again.

In order to enhance auditing work we are presenting a new architecture with the goal of providing the auditor with a flexible solution that, to a large extent, can be reused even if the exact method changes over time. This architecture is presented in Figure 3.1 and a short description follows in Section 3.1. The components of the picture and the motivation behind this architecture will be motivated throughout this section. Thus, this Section focus on describing a flexible architecture that could be used to discover vulnerabilities described so far.

3.1 Introduction

Before we dig into the specifics (requirements, design decisions and details) of the architecture we would like to present a very short description.

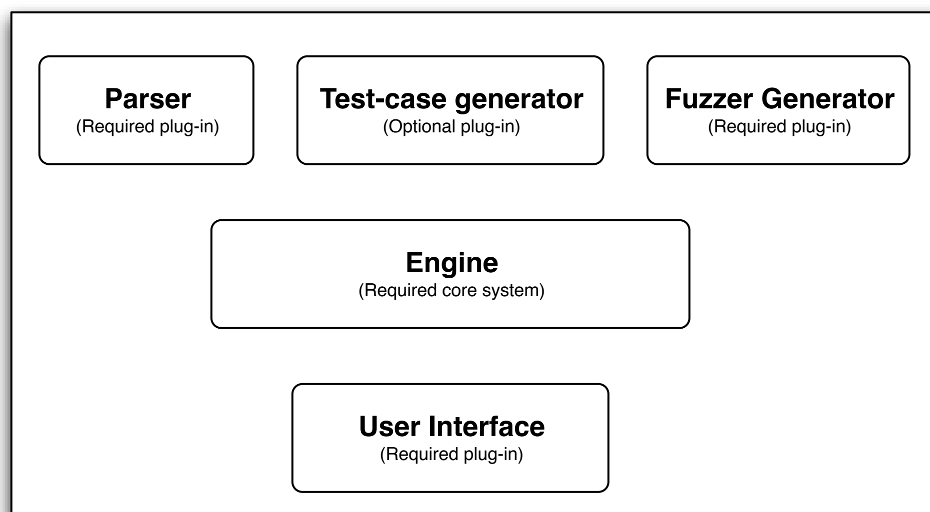

Figure 3.1: Flexible architecture overview

The architecture, shown in Figure 3.1, consists of the following modules:

Module name	Type	Dependence
Parser	Plug-in	Required
Test-case generator	Plug-in	Optional
Fuzzer generator	Plug-in	Required
Engine	Core system	Required
User Interface	Plug-in	Required

Table 3.1: Framework overview

Table 3.1 needs some explanation. When a module is of type plug-in and is required it means that at least one implementation has to be present in order for the architecture to make sense. The module flagged as the core system could be replaced, but the module is considered to be much more work than a simple plug-in. Each plug-in has a well defined interface and limited functionality which makes it easily replaceable. All communication between plug-ins is done through the engine, which eliminates any dependency between plug-ins.

Let us briefly go through each module in the framework (shown in Figure 3.1). The parser is required since we will need some input data in order to produce anything. The parser is responsible for reading a certain format and convert it to an internal format used by the engine.

A test-case generator, or fuzzer, takes the internal format and creates test cases based on the input. A test-case generator is not required since the destined output format might be a fuzzing framework (handling test-case creation on its own). Thus, the framework presented here does not have to create test cases on its own, but has the ability to do it.

The fuzzer generator creates a runable fuzzer. As mentioned previously, this can be done by generating source code dependent on another framework or it can be source code for a complete fuzzer program.

A core system is needed to enable communication between these modules, this core system is called the engine. The base system handles user requests by interacting with the present modules.

Last, the user interface provides the end user with a way of interacting with the framework. The user interface could be implemented in any way that fit the current needs.

The remainder of this chapter will go deeper into the details of the architecture that has so far been introduced by motivating this solution.

3.2 Requirements

This Section tries to gather and motivate the requirements on which the architecture is built.

The main motivation for this thesis is to allow a more flexible approach, that does not rely solely on one particular solution. For instance, a network fuzzer that was built using a specific framework to reduce development time is rather static. Imagine we would like to add some protocol features, more development time is required. Another example, if we audit protocols that in the passed have suffered vulnerabilities we will probably want to adopt similar test cases from successful runs. In this case, all the fuzzers have to be updated. This would consume resources for development but also making sure each fuzzer is the latest version and that the target is tested with the right test cases.

To allow adaptations, such as described above, flexibility is required. Flexibility in this sense refers to the ability to easily adapt to new/different methods.

It is not only important to reduce the time consumed by changing the underlying framework or method. When one, or several, methods are in use it is also important to allow modification without too much trouble. One of the shortcomings for current implementations is Initilialization time, as mentioned above. The Initilialization time becomes even a bigger problem when it occurs over and over again. It is therefore important that the architecture supports new protocols. This requirement is called modifiability, since the existing solution needs to be modified in order to support several protocols.

Since reuse of code and simplicity are keywords in this architecture it should be easy to use. When there does not exist any need for implementing new support, it should not be required for the person operating the software to be a professional programmer.

Because of the reasons mentioned throughout this section, the following key requirements have been recognized:

- *Flexibility*. This requirement should allow for easy generation modifications. If new testing techniques or generation procedures should be added it should not affect the rest of the system.
- *Modifiability*. Increasing the protocol support should not be a large development effort.
- *usability*. Since it should not be required that a professional security auditor use the tool, it is important that it is easy to use for anyone that does not have a deep knowledge within the vulnerability research area.

The next step is to design a system architecture that considers these requirements.

3.3 Design

Throughout this Section we try to establish an architecture that will capture and satisfy the requirements mentioned in Section 3.2. Each requirement and its corresponding architectural solution are motivated one by one.

3.3.1 Flexibility

Let us start with flexibility, which should allow for easy generation modifications. Since we would like to allow flexible fuzzer creation, generation is two folded:

1. Test case generation. The method for generating test cases is likely to change over time. Therefore, the actual generation of test cases should be easy to replace.
2. Fuzzer generation. Different frameworks for network communication might contain different functionality. We should not limit our self to one specific framework as it might be an obstacle in the future.

The arguments listed above leads us to the conclusion that in order to support easy to change test case and fuzzer generation, both of these generators has to be easy to replace modules in our architecture. With this decision, one could generate the same type of fuzzer (with respect to test cases) for any given framework and therefore in any given source code language. Also, several test-case generators can be used to cover different areas of vulnerabilities.

We also recognize that a target framework (for fuzzer generation) may already contain the functionality to fuzz data. Therefore, the internal test-case generation plug-in (figure

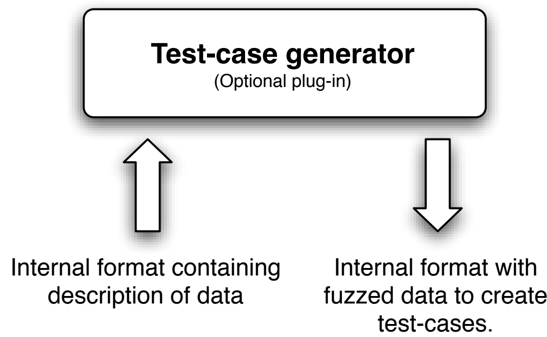

Figure 3.2: Test-case generator

3.2) is made optional. Figure 3.2 is a high level design picture of the interface provided by each implementation of a test-case generator. All implementations need to implement a method that takes the internal description format as a parameter and returns a fuzzed version in the same internal format.

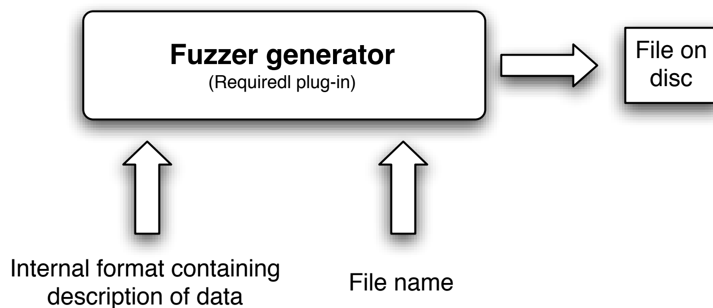

Figure 3.3: Fuzzer generator

The fuzzer generator (figure 3.3) is required in order to create a file that can be launched to test a specific target. An implementation of a fuzzer generator has to have two in parameters; an internal data description and a file name. The generator will, based on the input, create a file containing source code that can be executed. This source code may or may not be dependent on another framework, depending on the specific implementation.

3.3.2 Modifiability

The next requirement that our architecture needs to satisfy is modifiability which should be able to handle new protocols. What we really need is a mechanism to read different protocols, a parser. Since each protocol is unique the modifiability has to be a concern for the parser. Because parsers should be changeable to allow several protocols to be parsed, parsers are implemented as plug-ins. This decision also allow different type of protocol decisions, we can develop new modules when new descriptions are invented that might fit our needs better then what is currently in use.

Figure 3.4 shows the required input and output of a parser implementation. When a parser plug-in is called a file name has to be provided. The file is parsed and converted to an internal format.

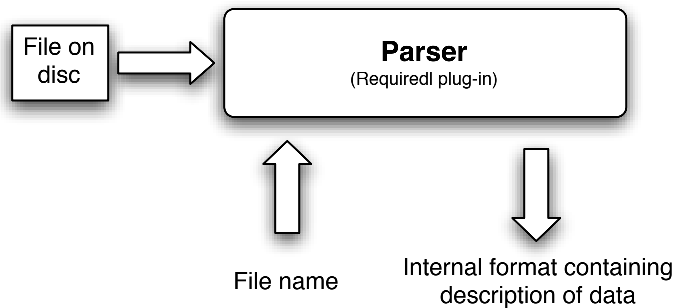

Figure 3.4: Parser

3.3.3 Usability

The third, and last, requirement is usability. The solution for the usability requirement should add to the simplicity of the architecture. Since different users usually have different needs, one GUI might not be the best solution. Instead, even the GUI is made flexible and could therefore be adjusted as needed.

We also need a core system that is the link between different modules providing the opportunity to use different modules without changing the source code. The link between modules is called the engine and is the backbone of the architecture. All communication between modules goes through the engine.

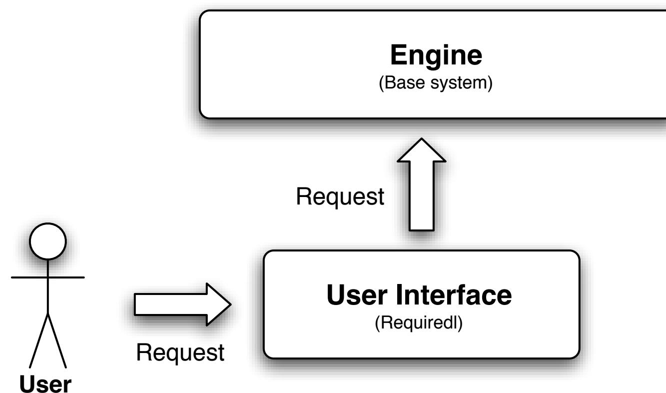

Figure 3.5: User Interface Engine

The interaction between user, user interface and engine is shown in Figure 3.5. The UI provides the user with some options, e.g.: providing file names to be parsed or written. These requests are then forwarded to the engine that calls a plug-in to handle the specific request. Because options such as log level can be provided a user interface can be more or less advanced.

3.3.4 Putting it all together

We have now seen how each of the established requirements has been met by architectural decisions. However, the architecture is not entirely completed. Since we would like each plug-in to be flexible they should not depend on each other (e.g.: replacing a parser should not imply replacing any other plug-in). Because of this all the plug-ins communicates directly to the engine. The communication in the architecture is shown by Figure 3.6.

We have previously mentioned an internal format that plug-ins has to either accept as in parameter or return back to the engine (except for the UI). The internal format is used to share information between plug-ins and consists of two things:

- *Sequence*. An example of a sequence can be found in Appendix A where the target protocol is FTP.
- *Types*. Each sequence is parsed to determine what types it consists of, such as strings, numbers, special chars and so on. This information is used to construct test cases.

When the engine receives a sequence from a parser plug-in it performs a type parse on it and associates each sequence entry with a specific type. Determining the specific type helps when construction test cases since we might want to test numbers in different ways compared with strings.

3.4 Implementation

The architecture has been implemented with Python. The reason for this is that python is known for its rapid development and is also platform independent. The only requirement on the language environment is that it should work on Linux, which certainly is true for python.

The architecture consists of five different parts: engine, user interface, parser, test-case generator and fuzzer generator. Figure 3.6 shows the different parts and how they communicate internally, this will be further explained throughout this section.

3.4.1 Engine

Let us start with the engine. The engine can be instantiated as an object and provides the following functionality:

- *Parse a file*. The ability to actually parse a file is provided by the parser plug-in. The engine will call a plug-in method to parse the file into a sequence.
- *Generate test cases from a sequence*. The engine is not required to provide this funtionalty since the test-case generator plug-in is optional. When a sequence has been obtained it is used for generating test cases. If test-case generation is supported and desired by the user, the engine calls a method belonging to a test-case generator module in order to obtain test cases from the specified sequence.
- *Generate a fuzzer based on test cases*. The test cases generated by the test-case generator are used to build an actual fuzzer. The engine handles this by supplying the fuzz generator with the test cases.

Each of the modules mentioned here are explained in deeper detail through this chapter.

Since the engine is instanciated and its methods are used to interact with the different plug-ins, the last plug-in of the architecture is also needed; namely the user interface (UI). The UI handles the interaction between the user and the engine, making it possible for the user to request actions.

The communication described in these sections is also shown in Figure 3.6 where arrows represent requests and responses. What a request or response contains is not included in Figure 3.6 but can be found in Section 3.3.

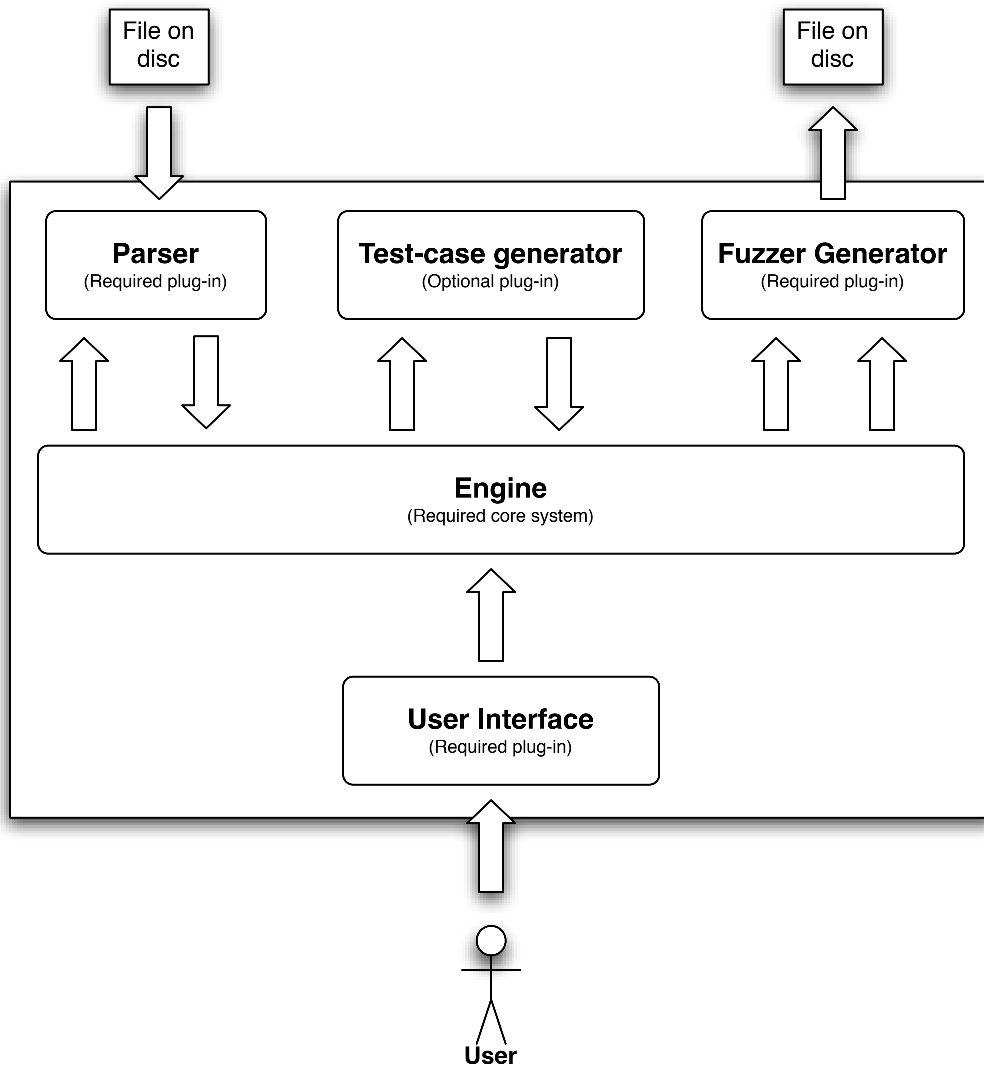

Figure 3.6: TFW architectural details.

Listing 10: Type parse

```
1  USER anonymous\r\n
2  <string><space><string><special><special>
```

Listing 11: Parser interface

```
1  # fileName [in] - String containing the file name to be
2  #                 parsed.
3  # sequence [out] - A sequence of events (internal format).
4  sequence parse(fileName)
```

3.4.2 Internal format

The internal format design were described in Section 3.3.4, the focus here is to describe how the format is implemented in source code. A sequence in the internal format is implemented as a matrix, e.g.:

$$sequence[MAX_SEQUENCE][MAX_STR_LEN] \tag{3.1}$$

This sequence can capture the interaction with a given network protocol, such as shown in Appendix A.

The second part of the internal format, the type parse, contains the type of a given sequence entry. Let us take an example: As one can see, the first entry in the sequence shown in Appendix A contains five types. The functionality of parsing a séquence is provided by the engine but is passed and accessible by the plug-ins (except for the parser plug-in).

3.4.3 Parser

A parser object is required to have a parse method. Each implementation of the parser plug-in has to comply with the interface description shown in Listing 11. The parse method parses the file into a sequence of events (an example of a sequence is shown in Appendix A). This sequence is then returned and handled by the engine.

The sequence returned is what possible generated test cases are built upon and is used to generate fuzzers at the end stage.

3.4.4 Test-case generator

When a test case generator is implemented it needs one method that takes a sequence as an argument. The interface is further described in Listing 12. This sequence, containing

Listing 12: Test-case generator interface

```
1  # fuzzedSequence [out] - String containing the file name
2  #                          to be parsed.
3  # sequence [in] - A sequence of events (internal format).
4  # types [in] -  Type of each sequence element.
5  fuzzedSequence fuzzAll(sequence, types)
```

Listing 13: Test cases

```
1 USER anonymous\r\n
2 <string><space><string><special><special>
3
4 USAAAAAAAAAAAAAAAAER anonymous\r\n
5 USER AAAAAAAAAAAAAAAAAAAAAAAAanonymous\r\n
6 USER anonymous\r\r\r\r\r\r\r\r\n
7 %n%n%n%n%nUSER anonymous\r\n
```

Listing 14: Fuzzer generator interface

```
1 # fuzzedSequence [in] - A sequence of events (internal
2 #                                                 format).
3 void fuzzAll(fuzzedSequence)
```

format information, is then used to generate test cases. Take Listing 13 as an example.

Line one contains a sequence instance, while line two contains the format information for line one. A sequence instance and it is format specifier is used to produce test cases, example of test cases are found on line four through line eight.

3.4.5 Fuzzer generator

A fuzzer can be generated from either test cases or a regular sequence. Since a fuzzing framework (which might be the target for the fuzzer generator) may contain built-in functionality for fuzzing data sent one could generate a fuzzer using only a regular sequence. It is the up to the target framework to create test cases. Data could also be fuzzed, creating test cases that are then used to generate a fuzzer.

Since the generation is implemented as a module, the target of the generation could be anything. It could, for example, be a source code file that is compiled with its intended framework in order to create a fuzzer.

The interface, described ín source code, can be found in Listing 14. All plug-ins of a fuzzer generator has to implement this interface.

3.4.6 User Interface

The user interface creates a engine object that is used to interact with the other modules. The specific user interface is in its simplest form just a script that creates a engine object, and feed the engine methods with filenames, target information, etc. The user interface could therefore be completely adoptable to its users needs.

The simplest possible UI is just a file instancing the engine and passing some initial variables to it. This simple UI can be found in Appendix D.

4 Evaluation and Results

This Section tries to evaluate the architecture presented in the previous section. Before we present the details of the evaluation we would like to describe what we intend to evaluate and how this evaluation will be performed.

According to our requirements (presented in 3.2), our architecture satisfy the requirements if we can:

- change test-case generation procedure without affecting any other part of the system.
- change fuzzer generation procedure without affecting any other part of the system.
- increase to protocol support without causing a large development effort.
- support a user interface that could be adapted to the user.

We would also like to see if our architecture could work in the real world.

Our technique for evaluation will be to implement two of each possible plug-in. We will start by inserting one of each plug-in and validate that the architecture behaves as expected. We will then replace each plug-in, one by one, to validate that a change of plug-in does not affect the rest of the architecture. If plug-ins can be replaced, one by one, without causing development work in any other plug-in or in the engine, the architecture meets the requirements listed above.

In order to test if this architecture could work in the real world we will select a test implementation and launch it against a real service. We hope that we will be able to trigger some vulnerability which could be a hint that our implementation could work.

Since we need a test target to fuzz, we have chosen to install Red Hat 6.2 which contains the vulnerable FTP daemon wuftpd. However, the focus is not to find all possible vulnerabilities in the exposed software. Since the test-case generator plug-in could easily be replaced it is a time consuming task to develop a good test-case generator. Instead we will show an example, hopefully proving that this could work.

4.1 Testing environment

This Section gives a short description of the plug-ins that has been implemented for testing.

4.1.1 Parsers

We have implemented two parsers for testing, namely:

- *ftp-wireshark* for parsing of FTP data in wireshark XML format.
- *http-wireshark* for parsing of HTTP data in wireshark XML format.

Both parsers targets XML based Wireshark captures. The fact that we use an XML format provides us with the possibility of using an already existing XML parsing library to reduce development time of the parser. An XML Wireshark capture is parsed and turned into a sequence to describe the user interaction.

4.1.2 Test-case generator

Two test-case generators are currently implemented, which we will call *simple-random* and *peach-based*. As the name suggest, our simple generator has only been developed for testing purposes (in order to verify that two implementations work). The funtionality provided by *simple-random* is to insert a random amount of A's on a random position in the payload.

The *peach-based* implementation on the other hand is more advanced, see Table 4.1 for a short summary of its functionality. The *peach-based* is based on a peach class for

Type	Test-cases applied	Short description
String	Buffer overflow	Strings are padded with A's in front, back and random middle position.
String	Format string	Format strings are inserted on random locations. Strings are also replaced with format strings.
Numbers	Over/underflows	Numbers are replaced with numbers that could possibly trigger an overflow condition.

Table 4.1: peach-based generator support

deriving test cases, thus peach needs to be installed in order for this test-case generator to work.

The test cases listed in Table 4.1 are thought of as enough for an example. We could of course make more test cases, and more advanced. However, due to time constraints we wont dig deeper into it.

4.1.3 Fuzzer generator

Two fuzzer generators have been implemented for testing:

- *tcp-client-python* to generate a TCP client that is a executable python script.
- *dummy-client* that generates the sequence to be sent to a text file.

The tcp-client-python generator is implemented to generate a python script that can be used together with a self developt framework for network communication. Part of an example fuzzer using this simple framework can be found in Appendix B.

Since we need at least one more generator to validate if the generator plug-in could be easily exchanged a dummy-client generator has been implemented. The dummy-client simply writes the payload to be sent into an ordinary text file. Although this text file can not be directly executed it is easy to see that this data could be sent using some other network framework.

4.1.4 User Interface

The user interface consists of a main python script instantiating the engine. The variables passed to the engine are changed by simply editing the python script. This is the simplest way of providing a user interface and has been chosen to reduce development time.

The user interface is not implemented in two versions as the rest of the plug-ins. The reason is that the engine has a specified interface (see Section 3.4.1). The user interface

creates a engine object and uses its interface to reach different plug-ins. Since all we need to do is to create an engine object and call methods its trivial to see that multiple user interfaces would work.

4.2 Evaluation

Testing a target application is made in several steps, according to the specified architecture. First, we need to describe the target protocol so that the description can be handled and converted to the internal format by our present parser. Second the internal format is used to generate scripts. Third, these scripts are executed and the result from each script is investigated. The overall sequence of events for testing is shown in Figure 4.1.

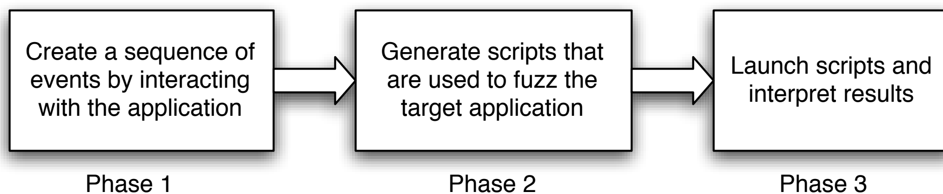

Figure 4.1: Application testing workflow

We will use phase 1 and phase 2 (in Figure 4.1) with our test plug-ins to evaluate our architecture. We will then use phase 3 to determine whether this architecture could detect security vulnerabilities.

The exact detail how each of these steps is performed depends on the current implementation. In the remainder of this Section we shall see how these steps are handled in our test implementation (described in 4.1).

4.2.1 Creating protocol description

Our test parsers (described in Section 4.1.1) both rely on Wireshark. We can create a protocol description (which contains a subset of a protocol) by recording a normal interaction with the application.

In the ftp-parser case, we will catch FTP-traffic to a server using the Wireshark packet sniffer. The catching is done as follows:

- *Start wireshark.* Start the packet sniffer to catch data, in this case we will use wireshark.

- *Start gFTP*. Launch an ftp client, we will use gFTP to interact with our ftp server as a normal user might use it.

- *Exit and save.* Close the ftp client and export the capture file in Wireshark to a XML-file.

The same technique can be used for any network based protocol and is also used for our http-parser.

The evaluation passes if the module can be loaded and parse a capture into the internal format.

4.2.2 Creating scripts

Creating scripts consists of two parts, namely:

- generating test cases, and
- generating scripts.

Each generation has two possible plug-ins to test. We will first examine the test-case generation.

We will try to generate test cases using both plug-ins (one at a time) in our architecture. The evaluation is said to pass if the internal format can be passed to the test-case generator to generate test cases as expected.

Last we will try both generation implementations. These implementations are evaluated as passed if they can generate files as expected independently on parser or test-case generator.

4.2.3 Running scripts and interpreting results

After the scripts have been created they can simply be executed by changing directory to the script directory (controlled by the UI). Each script should be executed in order.

By default, payload sent over the network connection is printed on screen. Any returned data or timeout's occurring while waiting for an answer are also shown on the screen. When an actual fault occurs, execution for the script in question is halted and the last sent message is printed as the probable cause of triggering the fault. In more complex bugs, the last sent message might not be enough to trigger the fault by itself, it might therefore be a good idea to rerun the script with full debug equipment to find out what exactly happened.

Our scripts are evaluated as passed if ”end of script” is printed after executing a script. A successful bug is found if we get the target to crash before ”end of script” is reached and this crash can be reproduced.

4.3 Results

The aim for this chapter was, as mentioned initially, to ”... evaluate if our architecture is indeed flexible and if it could work in the real world”. We have so far described what has been implemented in order to evaluate our architecture. We have also described how evaluation will be performed and what will result in successful evaluation. Based on the information provided throughout this chapter we present Table 4.2 which contains the evaluation results. In order to try if this could work in the real world we have selected the first setup of plug-ins (ftp-wireshark, peach-dependent, tcp-client-python) to target the wuftpd in RedHat Linux 6.2. This setup did trigger a vulnerability and the successful discovery of a format string vulnerability is shown in Appendix C.

In order to give reader an idea of the work effort needed to develop a plug-in we list the amount of code that each test plug-in consists of in Table 4.3. It should be noted that since only test plug-ins have been developed we can only give the reader an idea of the work needed to develop a plug-in of similar quality.

Parser	Test-case generator	Generator	Outcome
ftp-wireshark	peach-dependent	tcp-client-python	Pass
http-wireshark	peach-dependent	tcp-client-python	Pass
ftp-wireshark	random	tcp-client-python	Pass
http-wireshark	random	tcp-client-python	Pass
ftp-wireshark	peach-dependent	dummy-client	Pass
http-wireshark	peach-dependent	dummy-client	Pass
ftp-wireshark	random	dummy-client	Pass
http-wireshark	random	dummy-client	Pass

Table 4.2: Result

Plug-in	Lines of code
ftp-wireshark	100
http-wireshark	100
random	40
peach-dependent	250
tcp-client-python	110
dummy-client	10

Table 4.3: Lines of code

5 Summary, Conclusions, and Future Work

This Section will discuss what could have been done differently and what future investigations might consider.

5.1 Summary

This thesis has shown several vulnerability classes that are present today, despite of the environment protection that are available. We have also seen how dynamic security testing can be used in order to discover these vulnerabilities. There already exist a lot of tools for dynamic testing, but these tools are usually not compatible with each other. Thus, switching between testing tools/frameworks might include severe development time and is a time consuming task.

In order to create an even higher abstraction layer we have presented a new flexible architecture which support parsers, test-case generators, script generators and user interfaces as plug-ins in order to allow more reuse.

This new architecture has been evaluated by implementing plug-ins and switching between them in order to make clear that the plug-ins are independent. We also tried one setup of plug-ins against a known vulnerable FTP server, and a format string vulnerability was discovered. This indicates that the architecture could work in the real world.

5.2 Conclusions

Throughout this Section we will present our conclusions based on the content of this work. Our conclusions touche different aspects of this thesis and are therefore divided into subsections to avoid confusing the reader.

5.2.1 Architecture

We have conducted a limited evaluation by replacing plug-ins. This evaluation has shown that the requirements listed for the architecture has been met. That is:

- We can change test-case generation procedure without affecting any other part of the system.
- We can change fuzzer generation procedure without affecting any other part of the system.
- We can increase to protocol support without causing a large development effort.
- We support a user interface that could be adapted to the user.

Of course, to be even more certain that these requirements are fulfilled we could conduct additional testing. Since one could conduct evaluations in infinity we realise, since two plug-ins of each possible plug-in works, that these could be modified to create more plug-ins that also confirms to the same interface.

Our conclusion is that our architecture satisfies the requirements and is thus a flexible architecture.

5.2.2 Could more vulnerabilities be detected?

We have so far shown one successful vulnerability discovery. The reason that not more vulnerabilities have been discovered is because of the test-case generator(s). Since the test-case generator creates the test cases that trigger vulnerabilities, more vulnerabilities could be found with better generators. Due to time constraints, this is left as future work.

5.2.3 Goals

We reproduce the purpose of this thesis in order to remind the reader ”The purpose of this work is to present a new flexible architecture for dynamic detection of software security flaws”. This is exactly what we have done and we consider the purpose to be fulfilled.

We also had problem formulation that this thesis aimed to answer, namely: ”How can dynamic analysis be used in software security testing to aid the auditor and not consume a large amount of human resources?”. We would like to answer this question with the following formulation: Dynamic analyses can be conducted by adopting our flexible architecture in order to reduce human resources.

5.3 Future Work

As mentioned previously, the area of dynamic security testing is a very expansive area where new discoveries and approaches are presented regularly. A lot has happened since we begun writing this thesis. Among other things, completely automated tools have been presented. This does of course affect the need of a flexible architecture in a negative way. If approaches are made completely automated there is no longer a need to reduce development time for switching between tools.

Since we are not sure if this approach is even needed for the future, the need for this architecture would need to be revaluated. If we can conclude that we would benefit from this architecture there are several aspects that need future work:

- The *internal format* should be evaluated and perhaps replaced by something more efficient in order to allow even more advanced fuzzers. For example, in the current version we do not care for relationships between data.
- Develop efficient test-case generators and evaluate them in order to find more vulnerabilities in applications.
- Add support for generators, specially a generator that produces a server instead of a client.
- Produce even more plug-ins in order to continue evaluation of the current architecture.

There are many different aspects that would need future work. However, we think that items mentioned above are the most crucial parts and should therefore receive attention first.

References

[1] Jack Koziol, David Litchfield, Dave Aitel, et al., *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*, Wiley Publishing, Inc., Indianapolis, 2004.

[2] Aleph One, *Smashing The Stack For Fun And Profit*, Phrack 49, Volume Seven, Issue Forty-Nine, November 08, 1996.

[3] Linux man pages *STRCPY(3)*

[4] Mark Dowd, John McDonald, Justin Schuh, *The art of software security assessment*, Pearson Education, Inc., 2007.

[5] Chris Wysopal, Lucas Nelson, Dino Dai Zovi, Elfriede Dustin, *The Art of Software Security Testing*, Pearson Education, Inc., Upper Saddle River, 2007.

[6] Linux man page, *PRINTF(3)*.

[7] Stuart McClure, Joel Scambray and George Kurtz, *Hacking Exposed Fifth Edition*, McGraw-Hill/Osborne, Emeryville, California, 2005.

[8] Evan Martin, Karen Smiley, *Experiment and Comparison of Automated Static Code Analyzers and Automated Dynamic Tests*, North Carolina State University, 2005.

[9] Mei-Chen Hsueh, Timothy K. Tsai, Ravishankar K. Iyer, *Fault Injection Techniques and Tools*, University of Illinois at Urbana-Champaign, 1997.

[10] Peter Oehlert, *Violating Assumptions with Fuzzing*, IEEE Security & Privacy, March/April, 2005.

[11] Dave Aitel, *The Advantages of Block-Based Protocol Analysis for Security Testing*, Immunity,Inc., February, 2002.

[12] Michael Eddington, *Peach Fuzzing Framework 1.0*, http://peachfuzz.sourceforge.net/README.txt, 2007-09-16.

[13] Jared Demott, *GPF Readme File*, 2007-03-24.

[14] Greg Banks, Marco Cova, Viktoria Felmetsger, et al., *SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr*, University of California, Santa Barbara, 2006.

[15] Leon Juranic, *Using fuzzing to detect security vulnerabilities*, Infigo IS, INFIGO-TD-01-04-2006, 2006.

[16] Jared Demott, *The Evolving Art of Fuzzing*, June, 2006.

[17] FTP, RFC 959, http://www.faqs.org/rfcs/rfc959.html, 2007-06-05.

A Internal format

A simple FTP sequence that is used by the engine as an internal format.

Listing 15: Example FTP sequence

```
1  "USER␣anonymous\r\n"
2  "PASS␣anonymous\r\n"
3  "SYST\r\n"
4  "TYPE␣A\r\n"
5  "PWD\r\n"
6  "PASV\r\n"
7  "LIST␣-aL\r\n"
8  "CWD␣/etc\r\n"
9  "PWD\r\n"
10 "PASV\r\n"
11 "LIST␣-aL\r\n"
12 "CWD␣/\r\n"
13 "PWD\r\n"
14 "SITE␣EXEC␣ls\r\n"
15 "PASV\r\n"
16 "LIST␣-aL\r\n"
```

B Example fuzzer

In this Appendix a sample fuzzer is shown that has been generated using test plug-ins developed for our flexible architecture.

Listing 16: Example of a generated fuzzer

```
###################################################
#This file was auto-generated using the TFW framework.#
###################################################
from network import tcpclient
from datalib import data
#import time #Use this to manually add wait statements
#if needed

###################
#       INIT      #
###################
TIMEOUT=5
client = tcpclient.TCPclient("192.168.44.129",21,
TIMEOUT)
cmd = data.Data()

###################
#    Sequence     #
###################
cmd.clear()
cmd.appasc("USER␣anonymous\r\n")
client.sendrecv(cmd)

cmd.clear()
cmd.appasc("PASS␣anonymous\r\n")
client.sendrecv(cmd)

cmd.clear()
cmd.appasc("SYST\r\n")
client.sendrecv(cmd)

cmd.clear()
cmd.appasc("TYPE␣A\r\n")
client.sendrecv(cmd)

cmd.clear()
cmd.appasc("PWD\r\n")
client.sendrecv(cmd)

cmd.clear()
cmd.appasc("PASV\r\n")
client.sendrecv(cmd)

cmd.clear()
```

```
45 cmd.appasc("LIST -aL\r\n")
46  client.sendrecv(cmd)
47
48 cmd.clear()
49 cmd.appasc("CWD /etc\r\n")
50  client.sendrecv(cmd)
51
52 cmd.clear()
53 cmd.appasc("PWD\r\n")
54  client.sendrecv(cmd)
55
56 cmd.clear()
57 cmd.appasc("PASV\r\n")
58  client.sendrecv(cmd)
59
60 cmd.clear()
61 cmd.appasc("LIST -aL\r\n")
62  client.sendrecv(cmd)
63
64 cmd.clear()
65 cmd.appasc("CWD /\r\n")
66  client.sendrecv(cmd)
67
68 cmd.clear()
69 cmd.appasc("PWD\r\n")
70  client.sendrecv(cmd)
71
72 cmd.clear()
73 cmd.appasc("AAAAAAAAAAAASITE EXEC ls\r\n")
74  client.sendrecv(cmd)
75
76 cmd.clear()
77 cmd.appasc("SIAAAAAAAAAAAATE EXEC ls\r\n")
78  client.sendrecv(cmd)
79
80 cmd.clear()
81 cmd.appasc("SITEAAAAAAAAAAAA EXEC ls\r\n")
82  client.sendrecv(cmd)
83
84 cmd.clear()
85 cmd.appasc("SITE AAAAAAAAAAAAEXEC ls\r\n")
86  client.sendrecv(cmd)
87
88 cmd.clear()
89 cmd.appasc("SITE EXEAAAAAAAAAAAAC ls\r\n")
90  client.sendrecv(cmd)
91
92 cmd.clear()
93 cmd.appasc("SITE EXECAAAAAAAAAAAA ls\r\n")
```

```
94  client.sendrecv(cmd)
95
96  cmd.clear()
97  cmd.appasc("SITE EXEC AAAAAAAAAAls\r\n")
98  client.sendrecv(cmd)
99
100 cmd.clear()
101 cmd.appasc("SITE EXEC ls\r\n")
102 client.sendrecv(cmd)
103
104 cmd.clear()
105 cmd.appasc("SITE EXEC lsAAAAAAAAAA\r\n")
106 client.sendrecv(cmd)
107
108 cmd.clear()
109 cmd.appasc("AAAAAAAAAAAAAAAAAAAASITE EXEC ls\r\n")
110 client.sendrecv(cmd)
111
112 cmd.clear()
113 cmd.appasc("SAAAAAAAAAAAAAAAAAAAAITE EXEC ls\r\n")
114 client.sendrecv(cmd)
115
116 cmd.clear()
117 cmd.appasc("SITEAAAAAAAAAAAAAAAAAAAA EXEC ls\r\n")
118 client.sendrecv(cmd)
119
120 cmd.clear()
121 cmd.appasc("SITE AAAAAAAAAAAAAAAAAAAAEXEC ls\r\n")
122 client.sendrecv(cmd)
123
124 cmd.clear()
125 cmd.appasc("SITE EXEAAAAAAAAAAAAAAAAAAAAC ls\r\n")
126 client.sendrecv(cmd)
127
128 cmd.clear()
129 cmd.appasc("SITE EXECAAAAAAAAAAAAAAAAAAAA ls\r\n")
130 client.sendrecv(cmd)
131
132 cmd.clear()
133 cmd.appasc("SITE EXEC AAAAAAAAAAAAAAAAAAAAls\r\n")
134 client.sendrecv(cmd)
135
136 cmd.clear()
137 cmd.appasc("SITE EXEC ls\r\n")
138 client.sendrecv(cmd)
139
140 cmd.clear()
141 cmd.appasc("SITE EXEC lsAAAAAAAAAAAAAAAAAAAA\r\n")
142 client.sendrecv(cmd)
```

```
143
144 cmd.clear()
145 cmd.appasc("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAASITE␣EXEC
146 ls\r\n")
147  client.sendrecv(cmd)
148
149 cmd.clear()
150 cmd.appasc("SIAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAATE␣EXEC
151 ls\r\n")
152  client.sendrecv(cmd)
153
154 cmd.clear()
155 cmd.appasc("SITEAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA␣EXEC
156 ls\r\n")
157  client.sendrecv(cmd)
158
159 cmd.clear()
160 cmd.appasc("SITE␣AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAEXEC
161 ␣ls\r\n")
162  client.sendrecv(cmd)
163
164 cmd.clear()
165 cmd.appasc("SITE␣EXAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAEC
166 ␣ls\r\n")
167  client.sendrecv(cmd)
168
169 cmd.clear()
170 cmd.appasc("SITE␣EXECAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
171 ␣ls\r\n")
172  client.sendrecv(cmd)
173
174 cmd.clear()
175 cmd.appasc("SITE␣EXEC␣AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
176 Als\r\n")
177  client.sendrecv(cmd)
178
179 cmd.clear()
180 cmd.appasc("SITE␣EXEC␣ls\r\n")
181  client.sendrecv(cmd)
182
183 cmd.clear()
184 cmd.appasc("SITE␣EXEC␣lsAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
185 AAA\r\n")
186  client.sendrecv(cmd)
187
188 cmd.clear()
189 cmd.appasc("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
190 AASITE␣EXEC␣ls\r\n")
191  client.sendrecv(cmd)
```

```
192
193  cmd.clear()
194  cmd.appasc("SAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
195  AAAITE␣EXEC␣ls\r\n")
196   client.sendrecv(cmd)
197
198  cmd.clear()
199  cmd.appasc("SITEAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
200  AAAAAA␣EXEC␣ls\r\n")
201   client.sendrecv(cmd)
202
203  cmd.clear()
204  cmd.appasc("SITE␣AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
205  AAAAAAAEXEC␣ls\r\n")
206   client.sendrecv(cmd)
207
208  cmd.clear()
209  cmd.appasc("SITE␣EXEAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
210  AAAAAAAAAAC␣ls\r\n")
211   client.sendrecv(cmd)
212
213  cmd.clear()
214  cmd.appasc("SITE␣EXECAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
215  AAAAAAAAAAA␣ls\r\n")
216   client.sendrecv(cmd)
217
218  cmd.clear()
219  cmd.appasc("SITE␣EXEC␣AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
220  AAAAAAAAAAAAls\r\n")
221   client.sendrecv(cmd)
222
223  cmd.clear()
224  cmd.appasc("SITE␣EXEC␣ls\r\n")
225   client.sendrecv(cmd)
226
227  cmd.clear()
228  cmd.appasc("SITE␣EXEC␣lsAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
229  AAAAAAAAAAAAA\r\n")
230   client.sendrecv(cmd)
231
232  cmd.clear()
233  cmd.appasc("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
234  AAAAAAAAAAAAASITE␣EXEC␣ls\r\n")
235   client.sendrecv(cmd)
236
237  cmd.clear()
238  cmd.appasc("SIAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
239  AAAAAAAAAAAAAAATE␣EXEC␣ls\r\n")
240   client.sendrecv(cmd)
```

```
241
242 cmd.clear()
243 cmd.appasc("SITEAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
244 AAAAAAAAAAAAAAAAA␣EXEC␣ls\r\n")
245  client.sendrecv(cmd)
246
247 cmd.clear()
248 cmd.appasc("SITE␣AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
249 AAAAAAAAAAAAAAAAAAAEXEC␣ls\r\n")
250  client.sendrecv(cmd)
251
252 cmd.clear()
253 cmd.appasc("SITE␣EXAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
254 AAAAAAAAAAAAAAAAAAAEC␣ls\r\n")
255  client.sendrecv(cmd)
256
257 cmd.clear()
258 cmd.appasc("SITE␣EXECAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
259 AAAAAAAAAAAAAAAAAAAAAA␣ls\r\n")
260  client.sendrecv(cmd)
261
262 cmd.clear()
263 cmd.appasc("SITE␣EXEC␣AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
264 AAAAAAAAAAAAAAAAAAAAAAAls\r\n")
265  client.sendrecv(cmd)
266
267 cmd.clear()
268 cmd.appasc("SITE␣EXEC␣ls\r\n")
269  client.sendrecv(cmd)
270
271 cmd.clear()
272 cmd.appasc("SITE␣EXEC␣lsAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
273 AAAAAAAAAAAAAAAAAAAAAAAA\r\n")
274  client.sendrecv(cmd)
275
276 cmd.clear()
277 cmd.appasc("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
278 AAAAAAAAAAAAAAAAAAAAAAAASITE␣EXEC␣ls\r\n")
279  client.sendrecv(cmd)
280
281 cmd.clear()
282 cmd.appasc("SITAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
283 AAAAAAAAAAAAAAAAAAAAAAAAAAAE␣EXEC␣ls\r\n")
284  client.sendrecv(cmd)
285
286 cmd.clear()
287 cmd.appasc("SITEAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
288 AAAAAAAAAAAAAAAAAAAAAAAAAAA␣EXEC␣ls\r\n")
289  client.sendrecv(cmd)
```

```
290
291 cmd.clear()
292 cmd.appasc("SITE␣AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
293 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAEXEC␣ls\r\n")
294  client.sendrecv(cmd)
295
296 cmd.clear()
297 cmd.appasc("SITE␣EAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
298 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAXEC␣ls\r\n")
299  client.sendrecv(cmd)
300
301 cmd.clear()
302 cmd.appasc("SITE␣EXECAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
303 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA␣ls\r\n")
304  client.sendrecv(cmd)
305
306 cmd.clear()
307 cmd.appasc("SITE␣EXEC␣AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
308 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAls\r\n")
309  client.sendrecv(cmd)
310
311 cmd.clear()
312 cmd.appasc("SITE␣EXEC␣ls\r\n")
313  client.sendrecv(cmd)
314
315 cmd.clear()
316 cmd.appasc("SITE␣EXEC␣lsAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
317 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\r\n")
318  client.sendrecv(cmd)
319
320 cmd.clear()
321 cmd.appasc("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
322 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAASITE␣EXEC␣ls\r\n")
323  client.sendrecv(cmd)
324
325 cmd.clear()
326 cmd.appasc("SAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
327 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAITE␣EXEC␣ls\r\n")
328  client.sendrecv(cmd)
329
330 cmd.clear()
331 cmd.appasc("SITEAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
332 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA␣EXEC␣ls\r\n")
333  client.sendrecv(cmd)
334
335 cmd.clear()
336 cmd.appasc("SITE␣AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
337 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAEXEC␣ls\r\n")
338  client.sendrecv(cmd)
```

```
339
340 cmd.clear()
341 cmd.appasc("SITE␣EXEAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
342 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAC␣ls\r\n")
343  client.sendrecv(cmd)
344
345 cmd.clear()
346 cmd.appasc("SITE␣EXECAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
347 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA␣ls\r\n")
348  client.sendrecv(cmd)
349
350 cmd.clear()
351 cmd.appasc("SITE␣EXEC␣AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
352 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAls\r\n")
353  client.sendrecv(cmd)
354
355 cmd.clear()
356 cmd.appasc("SITE␣EXEC␣ls\r\n")
357  client.sendrecv(cmd)
358
359 cmd.clear()
360 cmd.appasc("SITE␣EXEC␣lsAAAAAAAAAAAAAAAAAAAAAAAAAAAA
361 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\r\n")
362  client.sendrecv(cmd)
363
364 cmd.clear()
365 cmd.appasc("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
366 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAASITE
367 EXEC␣ls\r\n")
368  client.sendrecv(cmd)
369
370 cmd.clear()
371 cmd.appasc("SITAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
372 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAE
373 EXEC␣ls\r\n")
374  client.sendrecv(cmd)
375
376 cmd.clear()
377 cmd.appasc("SITEAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
378 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
379 EXEC␣ls\r\n")
380  client.sendrecv(cmd)
381
382 cmd.clear()
383 cmd.appasc("SITE␣AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
384 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAEXEC
385 ␣ls\r\n")
386  client.sendrecv(cmd)
387
```

```
388 cmd.clear()
389 cmd.appasc("SITE␣EXEAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
390 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAC
391 ␣ls\r\n")
392 client.sendrecv(cmd)
393
394 cmd.clear()
395 cmd.appasc("SITE␣EXECAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
396 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
397 ␣ls\r\n")
398 client.sendrecv(cmd)
399
400 cmd.clear()
401 cmd.appasc("SITE␣EXEC␣AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
402 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
403 AAls\r\n")
404 client.sendrecv(cmd)
405
406 cmd.clear()
407 cmd.appasc("SITE␣EXEC␣ls\r\n")
408 client.sendrecv(cmd)
409
410 cmd.clear()
411 cmd.appasc("SITE␣EXEC␣lsAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
412 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
413 AAAA\r\n")
414 client.sendrecv(cmd)
415
416 cmd.clear()
417 cmd.appasc("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
418 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
419 AAASITE␣EXEC␣ls\r\n")
420 client.sendrecv(cmd)
421
422 cmd.clear()
423 cmd.appasc("SITAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
424 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
425 AAAAAAE␣EXEC␣ls\r\n")
426 client.sendrecv(cmd)
427
428 cmd.clear()
429 cmd.appasc("SITEAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
430 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
431 AAAAAAA␣EXEC␣ls\r\n")
432 client.sendrecv(cmd)
433
434 cmd.clear()
435 cmd.appasc("SITE␣AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
436 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
437 AAAAAAAAEXEC␣ls\r\n")
438  client.sendrecv(cmd)
439
440 cmd.clear()
441 cmd.appasc("SITE␣EXAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
442 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
443 AAAAAAAAAAEC␣ls\r\n")
444  client.sendrecv(cmd)
445
446 cmd.clear()
447 cmd.appasc("SITE␣EXECAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
448 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
449 AAAAAAAAAAAA␣ls\r\n")
450 <client.sendrecv(cmd)
451
452 cmd.clear()
453 cmd.appasc("SITE␣EXEC␣AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
454 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
455 AAAAAAAAAAAAAAls\r\n")
456  client.sendrecv(cmd)
457
458 cmd.clear()
459 cmd.appasc("SITE␣EXEC␣ls\r\n")
460  client.sendrecv(cmd)
461
462 cmd.clear()
463 cmd.appasc("SITE␣EXEC␣lsAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
464 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
465 AAAAAAAAAAAAAAA\r\n")
466  client.sendrecv(cmd)
467
468 cmd.clear()
469 cmd.appasc("%n%n%n%n%n%nSITE␣EXEC␣ls\r\n")
470  client.sendrecv(cmd)
471
472 cmd.clear()
473 cmd.appasc("SI%n%n%n%n%n%nTE␣EXEC␣ls\r\n")
474  client.sendrecv(cmd)
475
476 cmd.clear()
477 cmd.appasc("SITE%n%n%n%n%n%n␣EXEC␣ls\r\n")
478  client.sendrecv(cmd)
479
480 cmd.clear()
481 cmd.appasc("%n%n%n%n%n%n␣EXEC␣ls\r\n")
482  client.sendrecv(cmd)
483
484 cmd.clear()
485 cmd.appasc("SITE␣%n%n%n%n%n%nEXEC␣ls\r\n")
```

```
486  client.sendrecv(cmd)
487
488  cmd.clear()
489  cmd.appasc("SITE␣EX%n%n%n%n%n%nEC␣ls\r\n")
490  client.sendrecv(cmd)
491
492  cmd.clear()
493  cmd.appasc("SITE␣EXEC%n%n%n%n%n%n␣ls\r\n")
494  client.sendrecv(cmd)
495
496  cmd.clear()
497  cmd.appasc("SITE␣%n%n%n%n%n%n␣ls\r\n")
498  client.sendrecv(cmd)
499
500  cmd.clear()
501  cmd.appasc("SITE␣EXEC␣%n%n%n%n%n%nls\r\n")
502  client.sendrecv(cmd)
503
504  cmd.clear()
505  cmd.appasc("SITE␣EXEC␣ls\r\n")
506  client.sendrecv(cmd)
507
508  cmd.clear()
509  cmd.appasc("SITE␣EXEC␣ls%n%n%n%n%n%n\r\n")
510  client.sendrecv(cmd)
511
512  cmd.clear()
513  cmd.appasc("SITE␣EXEC␣%n%n%n%n%n%n\r\n")
514  client.sendrecv(cmd)
515
516  cmd.clear()
517  cmd.appasc("%n%n%n%n%n%n%n%n%n%n%n%nSITE␣EXEC␣ls\r\n")
518  client.sendrecv(cmd)
519
520  cmd.clear()
521  cmd.appasc("SI%n%n%n%n%n%n%n%n%n%n%n%nTE␣EXEC␣ls\r\n")
522  client.sendrecv(cmd)
523
524  cmd.clear()
525  cmd.appasc("SITE%n%n%n%n%n%n%n%n%n%n%n%n␣EXEC␣ls\r\n")
526  client.sendrecv(cmd)
527
528  cmd.clear()
529  cmd.appasc("%n%n%n%n%n%n%n%n%n%n%n%n␣EXEC␣ls\r\n")
530  client.sendrecv(cmd)
531
532  cmd.clear()
533  cmd.appasc("SITE␣%n%n%n%n%n%n%n%n%n%n%n%nEXEC␣ls\r\n")
534  client.sendrecv(cmd)
```

```
535
536 cmd.clear()
537 cmd.appasc("SITE␣E%n%n%n%n%n%n%n%n%n%n%nXEC␣ls\r\n")
538  client.sendrecv(cmd)
539
540 cmd.clear()
541 cmd.appasc("SITE␣EXEC%n%n%n%n%n%n%n%n%n%n%n␣ls\r\n")
542  client.sendrecv(cmd)
543
544 cmd.clear()
545 cmd.appasc("SITE␣%n%n%n%n%n%n%n%n%n%n%n␣ls\r\n")
546  client.sendrecv(cmd)
547
548 cmd.clear()
549 cmd.appasc("SITE␣EXEC␣%n%n%n%n%n%n%n%n%n%n%nls\r\n")
550  client.sendrecv(cmd)
551
552 cmd.clear()
553 cmd.appasc("SITE␣EXEC␣ls\r\n")
554  client.sendrecv(cmd)
555
556 cmd.clear()
557 cmd.appasc("SITE␣EXEC␣ls%n%n%n%n%n%n%n%n%n%n%n\r\n")
558  client.sendrecv(cmd)
559
560 cmd.clear()
561 cmd.appasc("SITE␣EXEC␣%n%n%n%n%n%n%n%n%n%n%n\r\n")
562  client.sendrecv(cmd)
563
564 cmd.clear()
565 cmd.appasc("%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%nSITE
566 EXEC␣ls\r\n")
567  client.sendrecv(cmd)
568
569 cmd.clear()
570 cmd.appasc("SIT%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%nE
571 EXEC␣ls\r\n")
572  client.sendrecv(cmd)
573
574 cmd.clear()
575 cmd.appasc("SITE%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n
576 EXEC␣ls\r\n")
577  client.sendrecv(cmd)
578
579 cmd.clear()
580 cmd.appasc("%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n␣EXEC
581 ␣ls\r\n")
582  client.sendrecv(cmd)
583
```

```
584 cmd.clear()
585 cmd.appasc("SITE␣%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n
586 EXEC␣ls\r\n")
587  client.sendrecv(cmd)
588
589 cmd.clear()
590 cmd.appasc("SITE␣EX%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n
591 %nEC␣ls\r\n")
592  client.sendrecv(cmd)
593
594 cmd.clear()
595 cmd.appasc("SITE␣EXEC%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%
596 n%n␣ls\r\n")
597  client.sendrecv(cmd)
598
599 cmd.clear()
600 cmd.appasc("SITE␣%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n
601  ls\r\n")
602  client.sendrecv(cmd)
603
604 cmd.clear()
605 cmd.appasc("SITE␣EXEC␣%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n
606 %n%nls\r\n")
607  client.sendrecv(cmd)
608
609 cmd.clear()
610 cmd.appasc("SITE␣EXEC␣ls\r\n")
611  client.sendrecv(cmd)
612
613 cmd.clear()
614 cmd.appasc("SITE␣EXEC␣ls%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n
615 %n%n%n\r\n")
616  client.sendrecv(cmd)
617
618 cmd.clear()
619 cmd.appasc("SITE␣EXEC␣%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n
620 %n%n\r\n")
621  client.sendrecv(cmd)
622
623 cmd.clear()
624 cmd.appasc("PASV\r\n")
625  client.sendrecv(cmd)
626
627 cmd.clear()
628 cmd.appasc("LIST␣-aL\r\n")
629  client.sendrecv(cmd)
630
631
632 ###################
```

```
633 #    End of script #
634 ###################
635 client.discon()
636 print "Reached end of script. Quiting."
```

C Fuzzer execution

This is an example of a fuzzer executing that result in a vulnerability discovery. Only the tail of the execution log is shown since rather much information is displayed. This is the result of executing the fuzzer shown in Appendix B against the vulnerable wu-ftpd 2.6.0.

Listing 17: Successfull discovery of a format string vulnerability

```
> SI%n%n%n%n%nTE EXEC ls

Waiting for result...
[RECV] 500 'SI%N%N%N%N%NTE␣EXEC␣ls': command not
understood.

> SITE%n%n%n%n%n EXEC ls

Waiting for result...
[RECV] 500 'SITE%N%N%N%N%N␣EXEC␣ls': command not
understood.

> %n%n%n%n%n EXEC ls

Waiting for result...
[RECV] 500 '%N%N%N%N%N␣EXEC␣ls': command not
understood.

> SITE %n%n%n%n%nEXEC ls

Waiting for result...
[RECV] 500 'SITE␣%N%N%N%N%NEXEC␣ls': command not
understood.

> SITE EX%n%n%n%n%nEC ls

Waiting for result...
[RECV] 500 'SITE␣EX%N%N%N%N%NEC␣ls': command not
understood.

> SITE EXEC%n%n%n%n%n ls

Waiting for result...
[RECV] 500 'SITE␣EXEC%N%N%N%N%N␣ls': command not
understood.

> SITE %n%n%n%n%n ls

Waiting for result...
[RECV] 500 'SITE␣%N%N%N%N%N␣ls': command not
understood.

> SITE EXEC %n%n%n%n%nls
```

```
Waiting for result...
******************************************
*                Result                  *
******************************************
Connection closed after sending:
SITE EXEC %n%n%n%n%n%n1s
```

D User Interface

This Appendix shows an easy user interface that has been used throughout this thesis.

Listing 18: Simple user interface

```
1  ########################
2  #   Includes           #
3  ########################
4  from tfw import engine
5
6
7  ########################
8  #   Settings           #
9  ########################
10 TARGET_HOST = "192.168.44.129"
11 TARGET_PORT = 21
12 RECV_SIZE = 4056 #Specify max data to recieve.
13 LOG_DIR = "./logs" #dir to write log file(s)
14 SCRIPT_DIR =
15 "/home/martin/programming/python/wrapper3/scripts/"
16         #dir to store scripts
17 SERVER_MODE = False #Server mode not implemented yet
18 STATEFUL = True #Set this to true if the target
19             #protocol is a stateful protocol
20 TIMEOUT = 5
21
22 PARSER = "ftp-wireshark" #Available: http-wireshark,
23                 #ftp-wireshark
24 TCGEN = "test" #Available: test, simple-random
25 FUZZGEN = "tcp-client" #Available tcp-client, dummy
26
27
28 ######################
29 # Engine interaction #
30 ######################
31 eng = engine.Engine(PARSER,TCGEN,FUZZGEN)
32 eng.parse("/home/martin/xml/ftp/ftp.xml")
33 eng.fuzz()
34 eng.genscripts(TARGET_HOST,TARGET_PORT,TIMEOUT,
35 SCRIPT_DIR)
```

Växjö
University

Matematiska och systemtekniska institutionen
SE-351 95 Växjö

Tel. +46 (0)470 70 80 00, fax +46 (0)470 840 04
http://www.vxu.se/msi/