In-Memory Fuzzing on Embedded Systems

Andreas Reiter

andreas.reiter@student.tugraz.at

Institute for Applied Information
Processing and Communications (IAIK)
Graz University of Technology
Inffeldgasse 16a
8010 Graz, Austria

Master Thesis

Supervisor: Dipl.-Ing. Kurt Dietrich
Assessor: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Karl Christian Posch

April, 2012

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
date (signature)

Abstract

Fuzz testing or Fuzzing is a method to test software for bugs and vulnerabilities. It is an important link in the chain of software-testing and enables automated software tests from an applications view, and not like other testing methods from a source code view.

Fuzzing can be used for different kinds of testing, e.g. for black-box testing where no internals are known, as well as for gray-box testing where some system internals are known which might affect or help to optimize the fuzzing process. Basically, fuzzing puts a high load of malformed input data on the device or software under test, examines the responses and looks for unexpected behaviour or even software crashes. There are many types of fuzzers, with different interfaces, ranging from generic software fuzzers to specific protocol fuzzers.

In this thesis, I analyse the existing fuzzing technologies, specific fuzzer implementations and define the requirements for in-memory fuzzers, especially on embedded devices. In-memory fuzzing is a technology where the fuzzer analyses the target program's memory and tries to isolate regions of interest. These isolated regions can then be tested separately without running through the whole program. For example, the fuzzer could isolate the input parsing parts of a program, and exhaustively test it with thousands of malformed inputs, with only a single application launch. For this purpose it injects some code into the target, which generates test loops and provides callbacks for the fuzzer. Further this thesis introduces an in-memory fuzzing framework which was developed during the research for this thesis.

Keywords: software testing, fuzz testing, fuzzing, in-memory fuzzing

Kurzfassung

Fuzz Testing oder Fuzzing ist eine Methode, um Software auf Bugs und Sicherheitslücken zu überprüfen. Diese Methode ist ein wichtiger Teil in der Kette von Softwaretests und ermöglicht automatisierte Softwaretests aus der Sicht des Benutzers, nicht wie andere Testmethoden, aus der Sicht des Entwicklers.

Fuzzing kann für verschiedene Arten von Tests eingesetzt werden, z.B. für Black-Box-Tests, wo keine Systeminterna erforderlich sind, ebenso wie für Grey-Box-Tests, wo einige Systeminterna bekannt sind und die dazu beitragen, den Prozess zu optimieren. Fuzzing testet Applikationen bzw. Geräte mit einer Menge aus gültigen und ungültigen Daten und beobachtet das Verhalten der Applikation bzw. des Geräts. Es gibt bereits viele verschiedene Arten von Fuzzern mit verschiedensten Schnittstellen, angefangen bei generischen Software Fuzzern bis hin zu sehr spezifischen Protokoll Fuzzern.

Diese Arbeit analysiert die vorhandenen Fuzzing Technologien, spezifische Fuzzer Implementierungen und definiert die Anforderungen an In-Memory-Fuzzer, die vor allem auf Embedded-Geräten Einsatz finden. In-Memory-Fuzzing ist eine Technologie, bei der der Fuzzer den Speicher des Programms bzw. des Geräts analysiert und interessante Bereiche zu isolieren versucht. Diese Bereiche können dann separat getestet werden, ohne jedes Mal das gesamte Programm durchlaufen lassen zu müssen. Der Fuzzer könnte zum Beispiel die Methode zur Eingabeverarbeitung isolieren und diese mit tausenden von gültigen und ungültigen Eingabedaten testen, das Programm aber nur einmal starten. Um dies zu bewerkstelligen werden Testschleifen mit Einhängepunkten in die Applikation eingeschleust, die diese Methode immer und immer wieder mit veränderten Eingangsdaten aufrufen. Weiters beschreibt diese Arbeit ein In-Memory Fuzzing Framework welches im Zuge der Forschung für diese Arbeit erstellt wurde.

Stichwörter: software testing, fuzz testing, fuzzing, in-memory fuzzing

Contents

Introduction **1**

1 Common Testing Methods **3**
1.1 White-Box Testing 4
1.1.1 Unit Testing 4
1.1.1.1 Frameworks 5
1.1.1.2 Example 5
1.1.2 Code Review 6
1.1.2.1 Tools 6
1.2 Black-Box Testing 7
1.2.1 Manual Testing 7
1.2.2 Automated Testing 8
1.3 Grey-Box Testing 8
1.3.1 Binary Auditing 8
1.4 Summary 10

2 Vulnerabilities **11**
2.1 From Bugs to Vulnerabilities 11
2.2 Off-by-One Error 12
2.3 Buffer Overflow 13
2.3.1 Stack-Based Buffer Overflow 15
2.3.1.1 Exploiting the stack-based buffer overflow 17
2.3.2 Heap-Based Buffer Overflow 17
2.3.2.1 Exploiting the heap-based buffer overflow 18
2.4 Integer Overflow 20
2.4.1 What is an Integer? 20
2.4.2 What is an Integer-Overflow 21
2.4.3 Exploiting Integer-Overflows 22

2.5 Summary 24

3 Fuzz-Testing in General 25

3.1 Why Another Testing Methodology? 25
3.2 How Fuzzing Works, a General Overview 26
3.3 Fuzzer Classification 27
3.3.1 Local Fuzzers 28
3.3.1.1 Command-line Fuzzers 28
3.3.1.2 Environment Fuzzers 29
3.3.1.3 File Fuzzers 31
3.3.2 Remote Fuzzers 32
3.3.2.1 Network Protocol Fuzzers 32
3.3.2.2 Web-application Fuzzers 33
3.3.3 Data Generation 34
3.3.3.1 Brute-Force Data Generation 34
3.3.3.2 Intelligent Data Generation 35
3.4 Summary 37

4 In-Memory Fuzzing 38

4.1 Problems of Other Fuzzing Techniques 38
4.2 How In-Memory Fuzzing Works 39
4.2.1 Mutation Loop Insertion 40
4.2.2 Snapshot Restoration Mutation 42
4.2.3 Implementation Aspects 43
4.3 Requirements 44
4.4 Existing Frameworks 45
4.5 Summary 47

5 Implementation Details 48

5.1 Architectural Overview 49
5.2 The Fuzzer Engine 51
5.2.1 Configuration 51
5.2.2 Bootstrapping the Fuzzer 52
5.2.3 Support Library 53
5.2.4 Remote Control 54
5.2.5 Defining Fuzz-Descriptions and Fuzz-Locations 56
5.2.6 Data Generators 59

5.2.7 GDB . . . 61
5.2.7.1 Process Snapshots and Restoration . . . 61
5.2.7.2 The Core-Dump . . . 63
5.3 The Analyser Engine . . . 63
5.3.1 Configuration . . . 64
5.3.2 Analysers . . . 64
5.4 Summary . . . 65

6 Results **67**
6.1 Application Tests . . . 67
6.1.1 HTEditor . . . 67
6.1.2 CoreHttp . . . 68
6.1.3 3Proxy . . . 69
6.1.4 ProFTPD . . . 70
6.2 Performance Test . . . 71
6.3 Known Issues . . . 71
6.4 Summary . . . 72

7 Conclusion and Future Work **73**

A Sample Configuration Files **77**

B Test Setup Configuration **79**
B.1 HTEditor . . . 79
B.2 CoreHttp . . . 80
B.3 3Proxy . . . 81
B.4 ProFTPD . . . 83

List of Figures **87**

List of Tables **88**

Listings **89**

Bibliography **91**

Introduction

Nowadays, software security is more important than ever. Every computer, hand-held, mobile phone and many other devices, even home-automation-systems, heating-installations or cars are interconnected. A single weak component can introduce significant vulnerabilities in a whole system. Activities in the recent years showed, that even global players do not invest enough time in (software-) testing [9][10][19]. The *Second Annual Cost of Cyber Crime Study*[16] gives an overview and comparison to the last years results of the annual costs for committed cyber crimes of the worlds top 50 companies. The results of this study are that the median costs for cyber crimes in 2011 increased by 2.1 million dollars to 5.9 million dollars compared to 3.8 million dollars in 2010. Thus, testing and vulnerability discovery is up-to-date and more important than ever.

What is the right way to go? How can a developer be sure that his product does not have any critical vulnerabilities? He can't, but he can increase its chance to avoid bugs by not relying on a single testing-method.

According to *Fuzzing: Brute Force Vulnerability Discovery[27]* many different vulnerability discovery methodologies exist. They can be categorised in three groups:

- *White-box testing* where all program internals are known, access to the source code is available and even access to the developers is available.
- *Black-box testing* where no extra information is available. A good example of *Black-box testing* is testing a remote service without any documentation.
- *Gray-box testing* resides somewhere in between, there may be some documentation available or there may be some known internals.

This thesis focuses on *fuzz-testing* or *fuzzing*, specifically on *in-memory fuzzing* which is a method of software testing. In general fuzzing puts a high load of malformed input data on the device or software under test, examines the responses and looks for unexpected behaviour or even software crashes. Fuzz-testing can reside in

any *White-box testing*, *Grey-box testing* or *Black-box testing* depending on the information available. Today, many different generic and specific fuzzers exist. However, no informations on in-memory fuzzing nor any ready-to-use implementations are available. This thesis defines the requirements for in-memory fuzzers and describes an in-memory fuzzer implementation that was created during the research for this thesis. It makes heavy use of existing tools and technologies and can use other provided information to ease the process of fuzzing.

Before it focuses on the fuzz-testing approach, the existing testing methods are analysed in chapter 1 and possible vulnerabilities along with their causes and effects are discussed in chapter 2. A description of the different types of fuzz-testing engines and an overview of the existing implementations follows in chapter 3. Chapter 4 focuses on all aspects of in-memory fuzzing, problems when dealing with in-memory fuzzing and its area of application. A detailed description of the fuzzer-architecture, and its components which were implemented during the research for this thesis is shown in chapter 5. Before I conclude in chapter 7 the results are presented in chapter 6.

Chapter 1

Common Testing Methods

This chapter focuses on widely spread and well established testing methodologies and their pros and cons. The terms *White-Box Testing*, *Black-Box Testing* and *Grey-Box Testing* will be used to further structure the testing methods.

Countless testing methodologies exist, discussing all of them would go beyond the scope of this thesis. Therefore commonly used testing methods for each group were selected. Another selection criterion is the metric used by the testing method in order to remain comparable to others and especially to in-memory fuzzing. One white-box testing method that is not included is, for example, *Code Coverage* testing. It rates how many source-lines have been tested by the test-setup in contrast to the complete source lines, therefore a code coverage of 100% is desirable but will not be reached in practice. As you see *Code Coverage* testing has a different metric and the output cannot be compared to the output of other testing methods. Therefore *Code Coverage* testing and other testing methodologies with different metrics would be misplaced here.

None of the described solutions are meant as the one-and-only solution, every class of testing methods discovers other classes of errors. In a production environment a combination of white-, grey- and black-box testing methods should be used.

The subsequent sections define the terms *White-Box Testing*, *Black-Box Testing* and *Grey-Box Testing* and focus on the following testing methods:

- *White-Box Testing: Unit Testing*, *Code Review*
- *Black-Box Testing: Manual Testing*, *Automated Testing*
- *Grey-Box testing: Binary Auditing*

1.1 White-Box Testing

The term *White-Box Testing* refers to testing methodologies that require full access to the source code, in addition access to the developers may be available. This methodology was historically simply referred to as *testing*. A simple scheme of white-box testing is shown in figure 1.1.

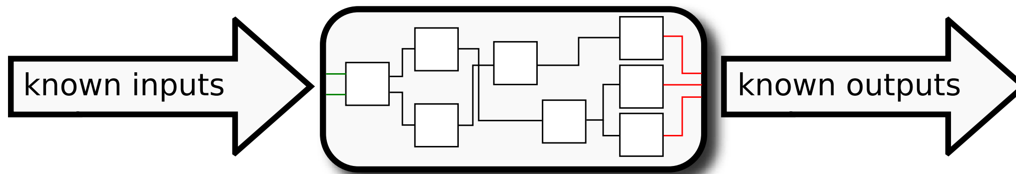

Figure 1.1: White-Box testing scheme

1.1.1 Unit Testing

Using the unit testing approach the program gets separated into small units which can be tested independently. Figure 1.2 shows a basic unit testing scheme with tests.

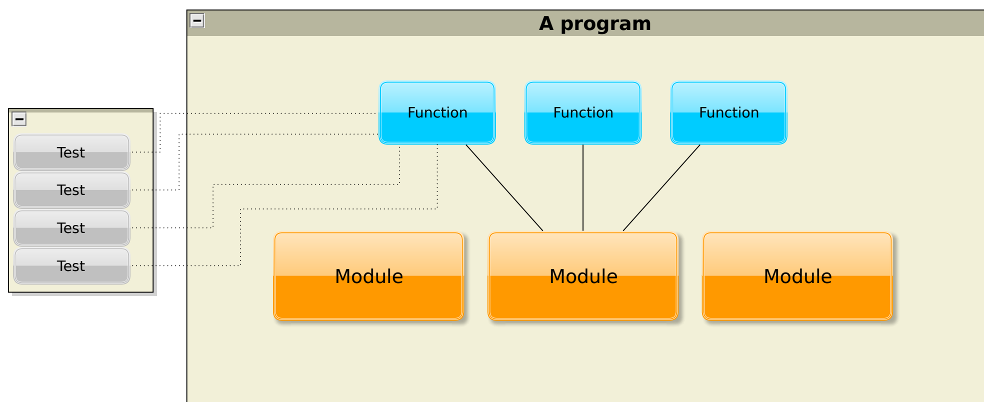

Figure 1.2: Unit testing scheme

Unit tests do not test the interaction of different modules, they only test the target unit, therefore unit tests are only a single part of a testing environment. An important requirement for fast and reliable unit testing is that the execution does not go out-of-scope, therefore a single unit-test should only check the target unit and should not test any other related code (e.g. code from other units) which can lead to unpredictable time constraints. Code from other units is tested by their corresponding unit tests.

1.1.1.1 Frameworks

Countless unit-testing frameworks exist for nearly every programming language available, but the most important framework is called *SUnit[7]* which is a testing framework for *Smalltalk* and the origin of all so called *xUnit* testing frameworks. It introduces itself as:

> *SUnit is the mother of all unit testing frameworks, and serves as one of the cornerstones of test-driven development methodologies such as eXtreme Programming.[7]*

Kent Beck proposed this testing framework in 1989 in the paper *Simple smalltalk testing: With patterns[8]*. Based on his work many different unit testing frameworks appeared.

To name just a few:

- *JUnit* - Unit testing framework for Java-programming language.
- *NUnit* - Unit testing framework for Microsoft .NET and MONO.
- *CppUnit*
- ...

1.1.1.2 Example

A typical example of a unit-test looks similar to the example shown in listing 1.1. Of course the implementation depends on the programming language but the outline of the test will always look the same. First call the unit to test (the unit should be chosen reasonable small), examine the return values and states, catch exceptions and define success- and failure-states.

Listing 1.1: Unit-test example

```
public class FooTest{
 @Test
 public void testFoo(){
   callMethod();
   examineReturnValuesAndState();
   //Define success and failure
   assertTrue(...);
 }
}
```

1.1.2 Code Review

Code Review is a manual or automated inspection of the source code. Practically it is impossible to do a complete manual source code review. If used the review is limited to a highly critical and small area of code. Another technique of manual code review, which is also used in the field is *Pair Programming.* Using this software development technique, two programmers always work at the same workstation, the first one is typing the code (also called the *driver*) and the second one reviews the code (also called the *observer*) and comes up with improvements. This method of course has the advantage that (in most cases) it yields source code of higher quality, but in fact this doesn't say anything about bugs or vulnerabilities. There is a big chance that two developers with the same background and the same point of view, working on the same piece of code, will make the same mistakes.

Automated source code review tools in contrast can inspect the whole source code for known, bad patterns. Many commercial and free review tools for various programming languages are available and can be seamlessly integrated in the development environment. Automated code review tools are able to detect, among others the following common vulnerabilities:

- Methods which are generally considered unsafe, depending on the programming language
- Static and limited dynamic pointer manipulation resulting in buffer overflows during the succeeding program flow.
- SQL-injection vulnerabilities or in general, input validation vulnerabilities as shown in listing 1.2

Listing 1.2: SQL-injection sample

```
SELECT * FROM sometable WHERE something=' " + somevalue + " ' "
```

The main drawback of automated review tools is, that they only inspect the program from a developers point of view, and cannot perform a functional test from the users or attackers point of view.

1.1.2.1 Tools

- *RATS* - Rough Auditing Tool for Security - is an auditing tool for *C*, *C++* and other programming languages.

> *RATS scanning tool provides a security analyst with a list of potential trouble spots on which to focus, along with describing the problem, and potentially suggest remedies. It also provides a relative assessment of the potential severity of each problem, to better help an auditor prioritize. This tool also performs some basic analysis to try to rule out conditions that are obviously not problems.[12]*

- *FxCop[18]* is an external tool for the Microsoft Visual Studio environment. It analyses .NET common language runtime assemblies and takes the review approach even further by assisting the developer in following the given design guidelines like naming conventions, error raising guidelines, security guidelines and many more.
- *Codan[2]* is a light-weight static analysis framework for *Java* and can be compared to *FxCop*. It is fully integrated in the Eclipse development environment.

1.2 Black-Box Testing

Black-Box Testing refers to testing methodologies which do not need to know the inner workings on an application. The only information a black-box testing method has access to is the information it can observe. Figure 1.3 shows a simple scheme of a *Black-Box Testing* mechanism.

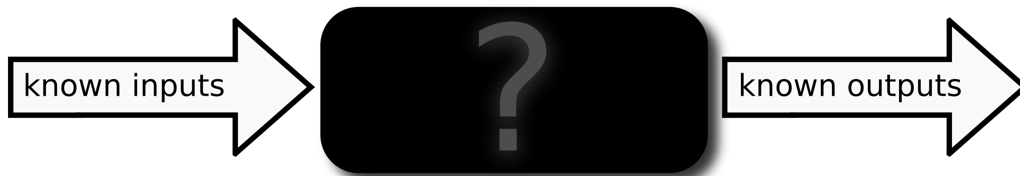

Figure 1.3: Black-Box testing scheme

1.2.1 Manual Testing

Manual testing is software testing with no support of any automated testing tools. This process is not an alternative to an automated testing process, unless you have your own testing-division with hundreds of employees.[27] As recent activities showed, one use-case for manual testing is testing web-pages using content management systems with known vulnerabilities.

1.2.2 Automated Testing

Automated Testing is a general term for many different test methods, but what most of them have in common are that test cases get defined at some point and are running automatically. *Automated Testing* is not fixed to black-box testing, they can also be white-box testing. Take a source module (also called a small unit), define test cases and you have exactly the same set-up as with unit-tests. Test cases for a text processing application could be many different files with all combinations of formatting instructions including invalid instructions and special, non-standard characters. This requires a lot of time for setting up the test-cases, but once created, they can be run without interaction.

Fuzzing, as described in chapter 3, is also an automated testing method, but with a different approach.

1.3 Grey-Box Testing

A *Grey-Box Testing* mechanism can use all the information it can observe (like black-box testing) and all the information it can regenerate from the available (binary-) files. There may be some parts of a binary-program, which can not be regenerated to a human-readable form, as illustrated in figure 1.4.

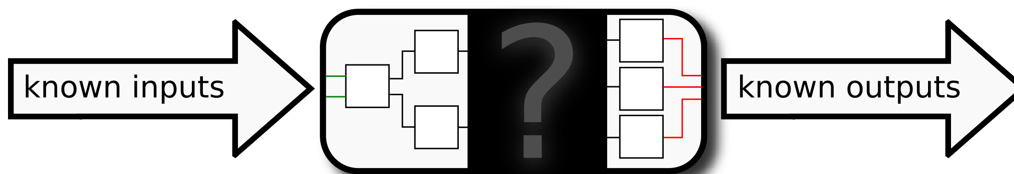

Figure 1.4: Grey-Box testing scheme

1.3.1 Binary Auditing

Binary auditing is closely related to reverse engineering. Reverse engineering is the process of decompiling a binary file to human readable source code and provide assisting information to the developer. Reverse engineering tools are not able to recover the original source code, because all meta-informations like variable names, function names and comments are lost on compilation. The output of these tools is an assembly file combined with a flow chart, to visualize the command flow. There are also reverse engineering tools available (e.g. *Boomerang[1]*) which decompile at least parts of the binary to a higher level language like *C* or *C++*.

Binary auditing is an automated reverse engineering process combined with an analysis phase. In the analysis phase the binary auditor searches for potential vulnerable code constructs and determines whether they can be triggered by the user or not.

A general overview of a binary-auditing process is shown in figure 1.5. The process starts with the reverse engineering process which outputs the disassembled code or regenerated higher language code and the already mentioned flow-chart to assist the developer or analysis tool in inspecting the code. The next step is to find the potential vulnerable code constructs. The output of this phase is a simple listing of vulnerable code constructs, but this table does not contain any information if it can be triggered by the user. This is done by the last step.

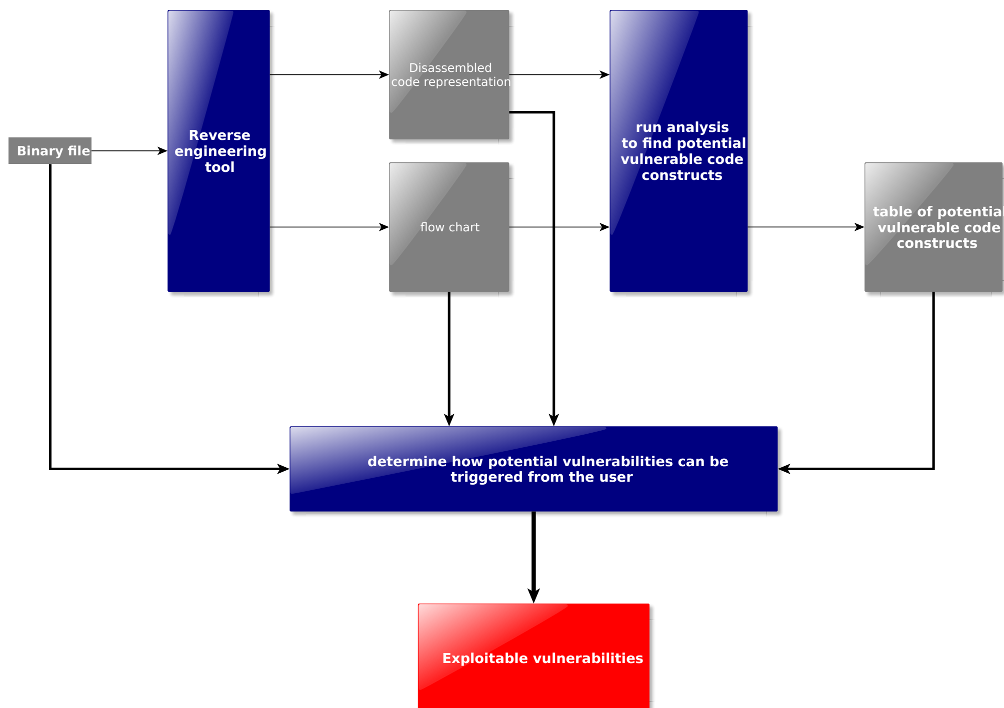

Figure 1.5: Binary-auditing overview

The advantage of this method is its availability, the binary file is available for most of the applications, except remote- or web-applications. The disadvantage is that reverse engineering is a highly complex process and requires highly specialized skills[27].

An all-in-one solution is provided by *IDA - The Interactive Debugger[14]* which is a disassembler and debugger with the capabilities to generate flowcharts and other useful views of the program execution flow.

1.4 Summary

This chapter introduced a few commonly used testing methodologies and grouped them by the information that is required to apply the test-method. The groups are: *White-Box-*, *Grey-Box-* and *Black-Box-Testing*. Each testing method gets assigned to one of these groups by their required knowledge of the program-internals. White-box testing methods have all program internals available including access to the source code. Black-box testing methods in contrast have only the information they can observe and grey-box testing methods additionally use reverse engineering tools and externally supplied information to gather as many program internals (e.g. method boundaries, method parameters, layout of data structures, stack layout,...) as possible. Most of the described methods, regardless of which group they are assigned to, do not test the functionality of the application as a whole, they only test small parts or modules. Only the automated testing approach tests the application from a users point of view but this method is not applicable because it requires too much time and too many resources. All the testing methods have their means of existence and each testing method tests for different kinds of errors. But there is no single-master-testing-method to apply. For a secure system a wisely chosen combination of testing methodologies and other approaches are required.

Chapter 2

Vulnerabilities

A bug is not necessarily a vulnerability. A bug in an applications user interface for example may not be exploitable and therefore is not a vulnerability. This chapter gives an overview of the different exploitable bug-categories and illustrates the way from bugs to vulnerabilities and exploits.

2.1 From Bugs to Vulnerabilities

What is a bug? In general a bug is a software behaviour that was not intended by the developer. For example, a bug may allow users to access resources they are not allowed to access. It may raise the user's privilege level and may allow the user to execute commands that should not be executable by design. Furthermore a bug may cause changes in the user's interface of an application.

In summary we can say, that there are multiple categories of bugs that compromise the system in different ways. A bug in an applications graphical user interface for example may not be critical for the whole system security and may even not be critical for the application itself; a bug in contrast which allows the user of an application to execute user-defined commands, compromises the whole system and is easily exploitable.

Figure 2.1 shows the relationship of *bugs*, *vulnerabilities* and *exploits*. Once bugs are discovered a combination of bugs is chosen by an attacker to create a vulnerability or an attack-path. The vulnerability itself is only dangerous if exploit code exists, so typically an attacker takes an available vulnerability and writes the exploit code which compromises the system. As we can see the real threats are vulnerabilities with existing exploit code.

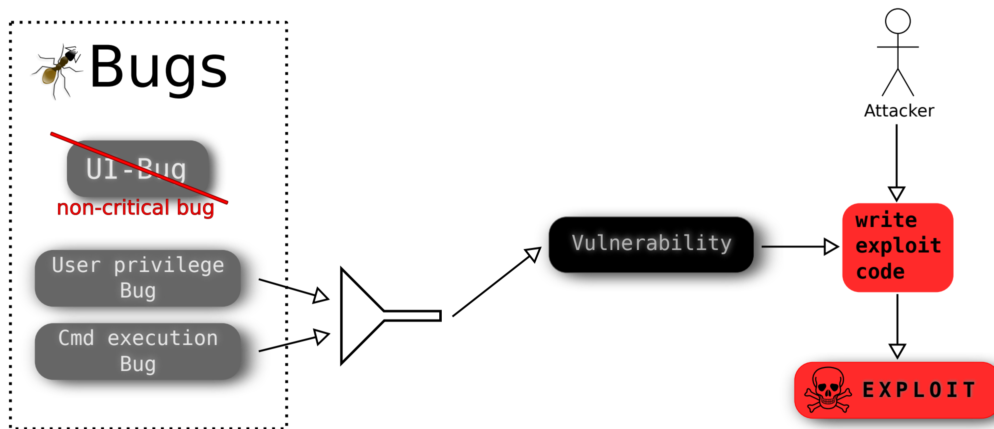

Figure 2.1: From bugs to vulnerabilities and exploits

2.2 Off-by-One Error

The source of off-by-one errors is the way that computers count, *Rathaus and Evron[22]* give a very good example of the problem.

> *How many fence posts are needed to build a fence 25 feet long with posts placed every 5 feet? Well, you can compute this by dividing 25 by 5, getting the answer of 5. While this is the correct answer for the number of fence sections you need (S-1 though S-5), it is not the correct answer for the number of posts needed.[22]*

The problem is illustrated in figure 2.2, to come back to the fence example, it requires six fence posts but only five fences. This is a source of error for every developer.

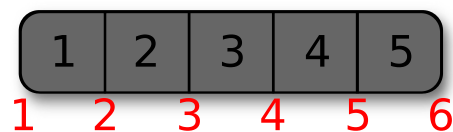

Figure 2.2: Off-by-One Error

How is this related to software development? The prime example for off-by-one errors are array indices in C programming language. They range from *zero* to *length-1*, the index *length* may already be associated with another array.

2.3 Buffer Overflow

This section focuses on buffer overflows and their general workings followed by two concrete buffer overflow implementations:

- Section 2.3.1 focuses on stack-based buffer overflows
- Section 2.3.2 focuses on heap-based buffer overflows

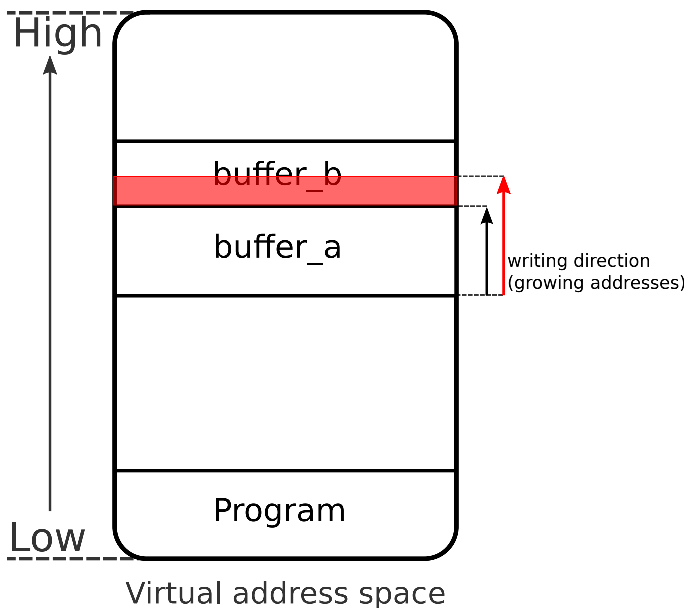

Figure 2.3: General functional principal of a buffer overflow

Buffer overflows are the most common bugs in today's software and can be exploited easily, if no protection mechanisms are applied to the application or the operating system. Figure 2.3 shows the general functional principle of a buffer overflow without going into detail on *Stack-Based* or *Heap-Based* buffer overflows, it shows the virtual address space of an application. The size and proper locations of the buffers and the executable code depends on the architecture and the operating system the application is executed on. What they all have in common is that an allocated block of memory is a consecutive memory region, at least in the context of virtual addresses. If an application writes beyond the bounds of a buffer, call it *buffer a* as shown in figure 2.3 and the operating system or the application has no protection against buffer overflows, it is likely that the application overwrites memory that is allocated by another part of the program or it writes to memory that has not been allocated yet. Both cases result in unpredictable application-states and may crash the application, in the best case. Developers are the main source of buffer-overflow-faults. The following list shows common causes of buffer overflows.

- *User-definable parameters are used without checking:* A user can provide a variable-length buffer which is copied to internal buffers. These buffers may be of fixed size (as it is for many applications handling command line arguments), this may result in buffer overflows for "long-enough" command lines.

- *Use of unsafe methods:* Especially low-level languages like C and C++ provide potentially unsafe methods like *strcpy*. It copies a source string character by character to a destination buffer and continues as long as no null-element is found.

- Many variations of the above exist like reading from files, network streams and so on.

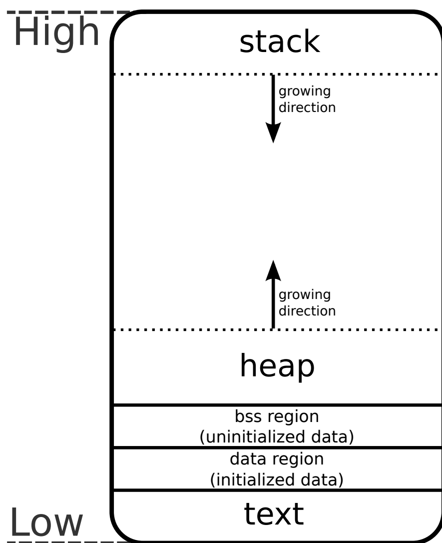

Figure 2.4: Simplified virtual memory layout of current operating systems running on the x86-architecture

Before focusing on dedicated buffer-overflow implementations, I will introduce the virtual-memory-scheme and stack-layout used by current operating systems running on the x86-architecture. Figure 2.4 shows a common and simplified layout of the virtual-memory of a running process. The figure only contains the relevant components, in practice the virtual memory contains some more components like multiple stacks, command line arguments, environment variables, memory mapped IO, a mapped kernel area and depending on the operating system and installed devices other areas. The memory-addresses in figure 2.4 grow from bottom to top, the following sections are contained:

- *text:* This section contains the executable program code. The target of the instruction pointer (ip) is somewhere in this section.

- *data region:* This section also comes from the executable and contains static or constant values.

- *bss section:* This section contains all global variables that have not been initialized statically in source code.

- *heap:* The heap-section contains all dynamically allocated memory and is of variable size. In general the size of the heap is not set by the developer explicitly. A programming language like *C*, for example, contains methods to acquire more dynamic memory (*malloc*). If a program requests more memory,

the *malloc*-method checks if there is enough space available and increases the size of the heap space (using the *brk* or *sbrk* system-call) if required. Analog actions are performed if memory is released. For details on *Heap-based buffer overflows* see 2.3.2.

- *stack:* The stack contains runtime information (local variables, parameters, return address, saved registers) of the method currently executed and of all its predecessors or callers. For details on *Stack-based buffer overflows* see 2.3.1

Sections 2.3.1 and 2.3.2 focus on vulnerabilities based on this simple buffer overflow.

2.3.1 Stack-Based Buffer Overflow

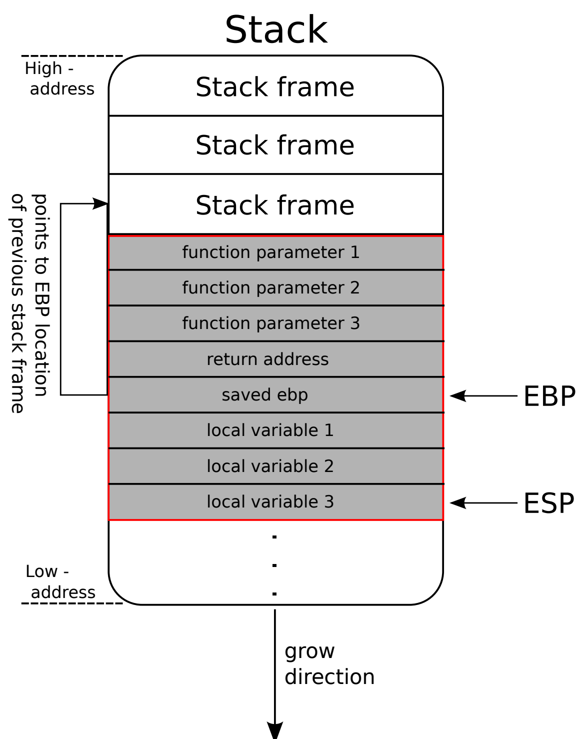

Figure 2.5: Stack layout

To understand the functioning of a *Stack-Based Buffer Overflow* I will first give detailed explanation of the stack as shown in figure 2.5. This represents the stack-layout of a x86-architecture running a current operating system. The stack is built of so-called *stack-frames*, each stack-frame is associated with a corresponding method in code (special cases exist if the compiler applies optimizations and determines that a function does not need to build a stack-frame, but these special cases are ignored for this simplified view) and contains informations required by the associated method to run. Before we look at a complete stack-frame we need to introduce the processor registers that are of interest when talking about stacks:

- ESP - Stack-Pointer: It always points to the last *used* element of the stack, thus to push an element on the stack the *ESP* needs to be incremented first. It gets implicitly manipulated by *POP* and *PUSH* operations.[13]

- EBP - Base-Pointer: It is used to reference the function-parameters and local variables as shown in table 2.1. The *Base-Pointer* gets modified explicitly only.[13]

- EIP - Instruction-Pointer: The *Instruction-Pointer* points to the next instruction that gets executed. If a function returns, the stack-frame is destroyed and the *Instruction-Pointer* is set to the value of the return address (4(%ebp)). As we will see this is one major attack-path.

16(%ebp)	third function parameter
12(%ebp)	second function parameter
8(%ebp)	first function parameter
4(%ebp)	old EIP (the function's "return address")
0(%ebp)	old EBP (previous function's base pointer)
-4(%ebp)	first local variable
-8(%ebp)	second local variable
-12(%ebp)	third local variable

Table 2.1: Base-Pointer offsets[13]

A complete stack frame is shown in figure 2.5 and contains the following elements:

- Function-parameters: The function or method parameters are pushed on the stack by the caller. The call-conventions depend on the cpu-architecture and compiler, but in general it is up to the compiler to also use cpu-registers for passing arguments to the callee. For this example we assume that all arguments are passed using the stack.

- Return address: The return address points to the instruction that is executed right after the current method returns and is also pushed on the stack by the caller.

- Saved registers (EBP): The old value of the *EBP* register is pushed on the stack by the callee to restore the old stack-frame on return.

- Local variables: The local variables are pushed on the stack by the callee.

2.3.1.1 Exploiting the stack-based buffer overflow

Now we have the complete tool-set to understand the technique of overflowing (or smashing) the stack. Figure 2.6 shows an overflowed buffer on the stack and its consequences.

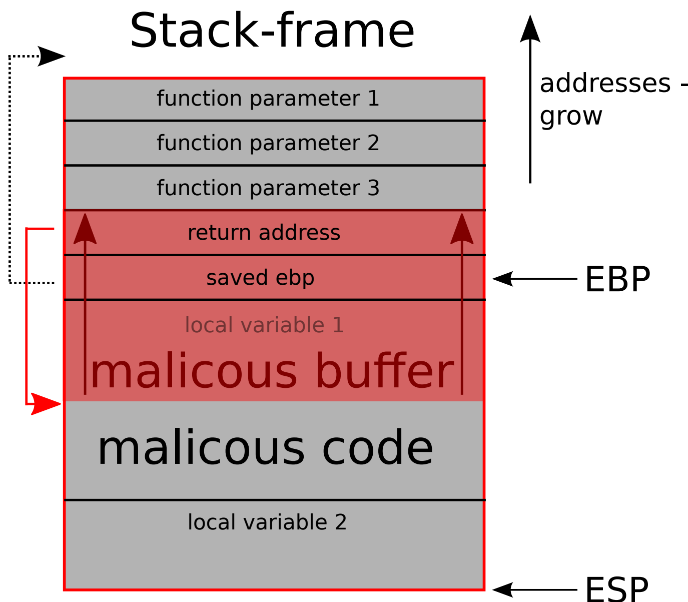

Figure 2.6: Smashed stack frame

Buffer overflows are possible because the program has no information on sizes of allocated buffers at runtime. The example shows a stack-frame with two local variables, where the first is of interest and gets passed from outside of the program. If the passed data gets large enough it writes beyond the bounds of the buffer and overwrites data that resides after the buffer e.g. the return address. In this case the return address gets overwritten with an address that also points to the stack, more precisely it points in the address space of the stack-buffer.

The result of this overflowed buffer is that, once the method returns the program executes code that has been injected by the user.

2.3.2 Heap-Based Buffer Overflow

The heap space has already been introduced in chapter 2.3 and figure 2.3. It is a memory area of dynamic size and contains all memory regions allocated at runtime.

As shown in figure 2.7 the heap-size is managed by the application (or by a library in its address-space). It only requests more memory from the operating system-kernel or releases allocated memory chunks beginning with the last one. The kernel only provides relatively large chunks of memory (e.g. 4kB), the byte-wise memory allocation is managed by the application (or library) itself.

The remaining parts depend on the implementation of the heap-manager, but a common practice is that a header similar to listing 2.1 is added before every allocated memory-block. The header contains a pointer to the next and to the previous

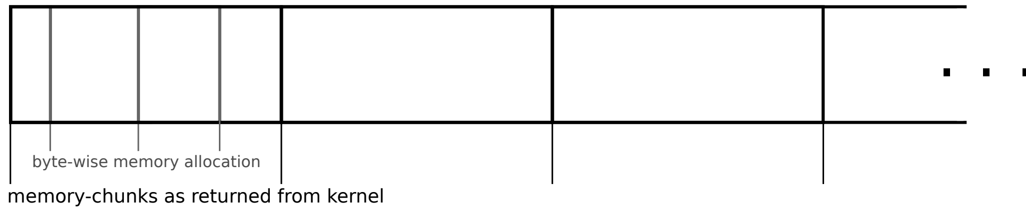

Figure 2.7: Heap space allocation

HeapHeader, hence they are organized in a double-linked list. It also contains the size of the managed memory block and the usage state.

Listing 2.1: Simple heap header[17]

```
typedef struct HeapHeader {
   struct HeapHeader *next;
   struct HeapHeader *prev;

   unsigned int  size;
   unsigned int  used;
   // Usable data area starts here
} HeapHeader_t;
```

Real-world implementations may contain more header-data and will implement a sophisticated algorithm to avoid fragmentation, but this is not within the scope of this thesis. Figure 2.8 shows the byte-wise allocated memory block headers with its links to the next and previous memory block and payload.

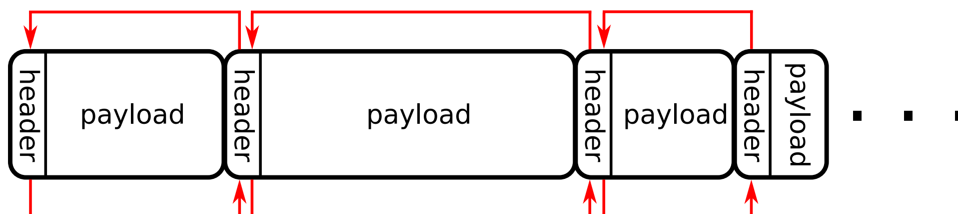

Figure 2.8: Allocated memory blocks with headers and links

2.3.2.1 Exploiting the heap-based buffer overflow

As discussed in chapter 2.3.1.1 exploiting the stack is straight forward and only a few application internals are required. To exploit the heap an attacker needs to know the following application internals amongst others:

- *Heap-management implementation* - It is up to the application developer to

chose one of the available heap-managers or even to manage the heap-space himself.

- *Executable heap* - For the attacker it is important to know if the heap is executable.
- *Jump address to overflow* - To execute injected code, an attacker needs to identify an overflow-able address that is used as a function pointer or similar. Thus the attacker could use it as its entry-point. This can either be an application specific jump-address or a weakness in the implementation of the heap-manager. In case of a stack-based buffer overflow this address was fixed to the stack-frame's return address.

Once all these variables have been identified a heap-based buffer overflow attack can be developed. As shown in figure 2.9 the attack looks similar to a stack-based buffer overflow.

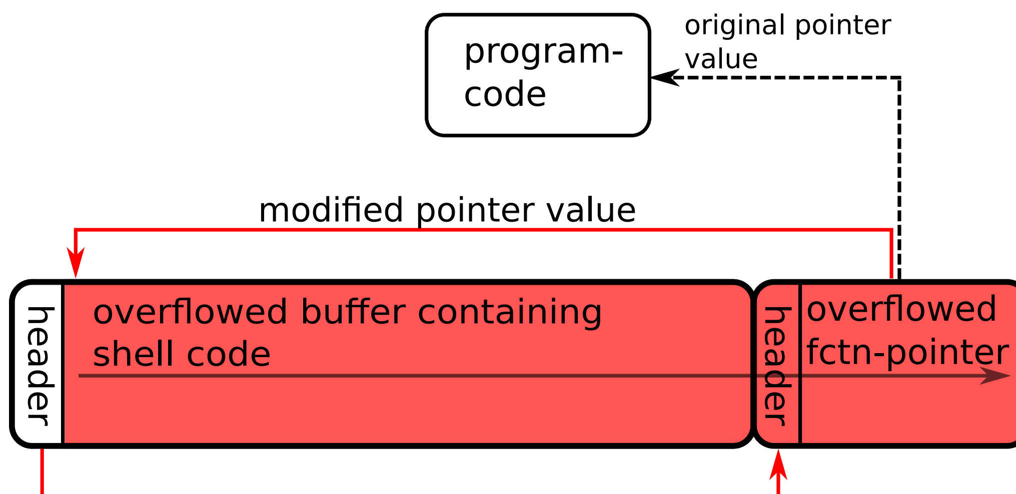

Figure 2.9: Overflowed heap layout

A buffer becomes overflowed with data and overrides the allocated memory block that immediately follows the buffer's data block. The overflow-data is specially structured. It contains malicious code to be executed and the start-address of the malicious code. The malicious code is also called shell-code because in most cases when implementing an overflow attack the first thing to try is to open a remote shell. The function pointer behind the buffer gets overflowed with the modified function pointer. The shell-code gets invoked as soon as the function pointer is called and the buffer is exploited.

2.4 Integer Overflow

Integer overflows can not be directly exploited as it was the case for stack-based and also for heap-based buffer overflows. They can shift the application in an unpredictable application-state and may open the doors for other exploits. But before going into detail on integer-overflow exploits this chapter focuses on the basics.

2.4.1 What is an Integer?

> *Integers, like all variables are just regions of memory. When we talk about integers, we usually represent them in decimal, as that is the numbering system humans are most used to.[21]*

Integers are represented in two's complement because it is not possible to properly handle negative numbers without special treatment of negative signs otherwise. Figure 2.10 shows the number space of an 8-bit integer value. In contrast to the *sign-and-magnitude* binary number representation where a bit is dedicated to the sign representation only, the *two's complement* representation uses the *most significant bit (MSB)* for sign representation and is weighted with the lowest negative number:

- m is the bit-size of the integer-value

$$-2^{m-1} : 1, 0, 0, 0, ..., 0, 0 \tag{2.1}$$

- Thus for an 8-bit integer $m = 8$

$$-2^{7} = -128 : 1, 0, 0, 0, 0, 0, 0, 0 \tag{2.2}$$

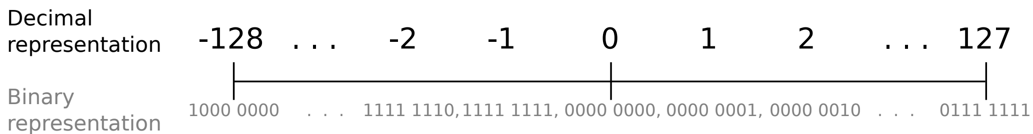

Figure 2.10: Two's complement vs. decimal representation

But how does the two's complement handle the initial problem where special treatment would be required for negative numbers? Equation 2.3 shows the sign and magnitude binary represenation of *1* and *-1*.

$$\begin{aligned}(1)_{10} &= (00000001)_2 \\ (-1)_{10} &= (10000001)_2\end{aligned} \tag{2.3}$$

Imagine adding the binary representation of *-1* to the binary representation of *1*. The expected result is *0* but as seen in equation 2.4 the result is *-2* because the subtraction requires special treatment.

$$\begin{array}{rl} & (00000001)_2 \\ + & (10000001)_2 \\ \hline & (10000010)_2 = (-2)_{10} \end{array} \tag{2.4}$$

When using the two's complement binary representation the addition for an 8-bit integer looks as seen in equation 2.5. Due to the two's complement the addition of negative and positive numbers works without dedicated subtraction operation and without special handling of negative numbers.

$$\begin{array}{rl} & (00000001)_2 \\ + & (11111111)_2 \\ \hline & (00000000)_2 = (0)_{10} \end{array} \tag{2.5}$$

2.4.2 What is an Integer-Overflow

We now know how integers are stored in memory. Again take an 8-bit signed integer with its value set to the highest positive value as shown in equation 2.6.

$$x = (01111111)_2 (= (127)_{10}) \tag{2.6}$$

What happens if another positive value is added to x? The binary addition will be processed as expected and shown in equation 2.7 but the two's complement interpreted result will be incorrect.

$$\begin{array}{rl} & (01111111)_2 \\ + & (00000001)_2 \\ \hline & (10000000)_2 = (-128)_{10} \end{array} \tag{2.7}$$

This is problematic, because after the addition has completed the developer has no chance to tell if the value is the result of an overflow or not. As already mentioned in the beginning, integer overflows cannot be directly exploited, but they may enable for example standard stack-smashing exploits as described in chapter 2.3.1.

The above explanation always assumes signed integers, but overflows are also possible on unsigned integers as stated in equation 2.8, which is slightly modified from equation 2.7.

$$\begin{array}{rl} & (11111111)_2 \\ + & (00000001)_2 \\ \hline & (00000000)_2 = (0)_{10} \end{array} \tag{2.8}$$

The only difference to signed integers is that the resulting value cannot be negative. For an 8-bit unsigned integer the numbers range from (0...255).

2.4.3 Exploiting Integer-Overflows

A straight forward unsigned-integer overflow example taken from [21] is shown in listing 2.2

Listing 2.2: Straight forward integer overflow example[21]

```
1 /* width1.c - exploiting a trivial widthness bug */
2 #include <stdio.h>
3 #include <string.h>
4
5 int main(int argc, char *argv[]){
6   unsigned short s;
7   int i;
8   char buf[80];
9
10  if(argc < 3){
11    return -1;
12  }
13  i = atoi(argv[1]);
14  s = i;
15
16  if(s >= 80){            /* [w1] */
17   printf("Oh no you don't!\n");
18   return -1;
19  }
20
21  printf("s = %d\n", s);
22  memcpy(buf, argv[2], i);
23  buf[i] = '\0';
24  printf("%s\n", buf);
25  return 0;
26 }
```

The integer overflow occurs on line 14. The integer gets assigned to a short-type variable and if the integer value is greater than *65535* (maximum value of unsigned short) it gets truncated, and the check on line 16 is bypassed. The following outputs are produced for the three shown test-cases:

- Valid input data, everything is working as expected. The condition at line 16 is not met.

Listing 2.3: Integer overflow example 1 output[21]

```
./width1 5 hello
s = 5
hello
```

- Valid input data, everything is working as expected. The condition at line 16 is met, program is aborted.

Listing 2.4: Integer overflow example 2 output[21]

```
./width1 80 hello
Oh no you don't!
```

- Invalid input data. The condition at line 16 is bypassed and the buffer given on the command line can override local variables. This example shifts the application in a state where a stack-smashing exploit as described in 2.3.1 is possible.

Listing 2.5: Integer overflow example 3 output[21]

```
./width1 65536 hello
s = 0
Segmentation fault (core dumped)
```

2.5 Summary

This chapter introduced and explained the terms *bug*, *exploit* and *vulnerability* and showed their differences. A bug is a software behaviour that is not intended by the developer but they do not necessarily compromise the system-security. One or more bugs form a vulnerability but the vulnerability is not dangerous as long as no exploit code exists. So, the real threats are vulnerabilities with existing exploit code.

The bugs were categorized into three groups: *Off-by-One-Error*, *Buffer-Overflows* and *Integer-Overflows*. Most of the bugs are located in the *Buffer-Overflow* group, which also offers the greatest exploit flexibility. While the other groups may only enable other failures but are not directly exploitable, the *Buffer-Overflow* category may enable the attacker to execute arbitrary code. Buffer overflows can further be grouped into: *Stack-Based* and *Heap-Based*. Local variables are generally saved on the program's stack next to the function's return address or other saved registers. This clears the way for *Stack-Based Buffer Overflows* and provides a reliable entry point for arbitrary code. The same applies to *Heap-Based Buffer Overflows* but they require more knowledge of the program internals because the heap-structure is not fixed, in contrast to the stack-structure.

Chapter 3

Fuzz-Testing in General

In chapter 1 common testing methods were introduced, in this chapter a raising group of testing methodologies, called *Fuzz-Testing* or *Fuzzing* is discussed. The first fuzzing approach is surrounded by legends, but every legend has its truth, the quintessence of the first fuzzing approach is that some mysterious incidents converged and a program that received truly random generated data crashed. The concept of fuzzing was born, but was not really taken into account as a testing method. It took till the late 90's before the *University of Oulu* picked up the concept and created a fuzzing test suite called *PROTOS*, but more on fuzzing-frameworks in 3.3.

First we will define the term *fuzzing*. Fuzzing is

> *...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. The name comes from modem applications tendency to fail due to random input caused by line noise on "fuzzy" telephone lines.[20]*

This chapter will give an overview of the fuzz-testing techniques available today and will discuss different fuzzing approaches.

3.1 Why Another Testing Methodology?

Chapter 1 focused on commonly used testing methods. The only method that is dedicated to test binary files is the *Binary Auditing* testing method, the disadvantages of this method are: disassembly skills are required and the method limits to

passively find known patterns in the disassembled binary file. No live tests (e.g. corner cases or input data distributed over the whole input data range) are performed. This is how the fuzz-testing technology comes in. It tests a binary application with many different input datasets and can also perform corner-case tests. Depending on the implementation and the type of the fuzzer (see 3.3) it can also take advantages from eventually available source code.

3.2 How Fuzzing Works, a General Overview

According to *Sutton, Greene and Amini[27]* the process of fuzzing can always be divided in the following phases:

- *Identify target.* In the very beginning a target application or library is needed. Consult different security related or bug tracking websites and look for mistakes this vendor already made. It is very likely to find similar mistakes in the vendor's applications. Further, watch out for libraries that are shared across multiple applications. Finding a vulnerability in shared libraries is far more critical than in standalone applications because this bug applies to all programs using this library.

- *Identify inputs.* Fuzzers operate on binary files and try to find triggerable bugs and vulnerabilities. All bugs are caused by insufficient checked user-input, thus it is critical for the fuzzing process to identify all available user-interfaces. Further it is important to identify the possible and valid range of input data, which is not necessarily the same.

- *Generate fuzzed data.* This sounds trivial but is one of the most challenging phases, especially for input data secured by *hash functions* or even *message authentication codes*. Basically two data generation strategies are used: *mutate* existing data or *dynamically create* new data. It depends on the application which strategy can be applied.

- *Execute fuzzed data.* Depending on the application under test, this phase starts an application in the trivial case or simulates a full client in a complex network fuzzing scenario. The most important property of this phase is that the execution performs *fully automated* without user interaction, otherwise the advantage of fuzzing over manual-testing is lost and fuzz-testing would be obsolete.

- *Monitor for exceptions.* The complexity of this phase is similar to the complexity of the *generate fuzzed data* phase, because it is not always clear if an application is in error state, in any undefined state or in a defined state.
- *Determine exploitability.* This phase is a manual process and is performed by security specialists. The outcomes of this phase are exploits for the application or library under test.

The fuzzing process is illustrated in figure 3.1. The generation, execution and monitoring step is performed for each fuzzing dataset.

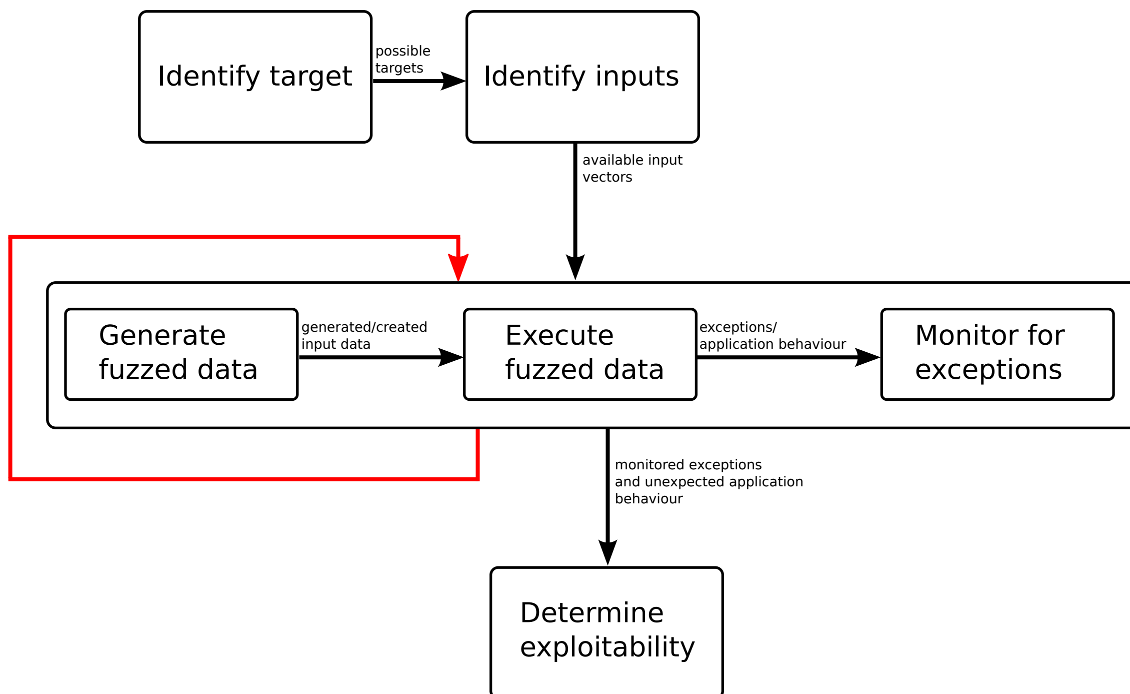

Figure 3.1: Fuzzing overview

This is just a basic view of the fuzzing process. For different applications this set of phases act as a starting point and may get extended, but in general all fuzzing processes have the above phases in common.

3.3 Fuzzer Classification

Many different types of fuzzers and at least as many different classifications are available. For the purpose of this thesis I will introduce a simple classification:

- *Local fuzzers.* Fuzzers that test local code only, e.g. *Command-line fuzzers 3.3.1.1*, *Environment fuzzers 3.3.1.2* or *File fuzzers 3.3.1.3*.

- *Remote fuzzers.* Fuzzers that test remote systems, e.g. *Network protocol fuzzers 3.3.2.1* or *Web-application fuzzers 3.3.2.2*

3.3.1 Local Fuzzers

Local fuzzers test applications on the current system only, thus it is likely that the binary file and the complete output generated by the application is available to the fuzzer. This significantly eases the analysis of the application's current state, by using side-channel data like log-files or application outputs.

3.3.1.1 Command-line Fuzzers

Command-line fuzzers test applications that accept arguments on the command line and try to find faults in the argument parsing code. In general they generate or have a list of arguments and start the program under test for each input and wait for program crashes or unexpected behaviour.

In the case of *honggfuzz[28]* (as an example for a command line fuzzer) a call to the fuzzer would look as shown in listing 3.1. The file *inputfiles/badcode1.txt* contains the input data for the target application *targets/badcode1*.

Listing 3.1: Honggfuzz start command

```
$ honggfuzz -f inputfiles/badcode1.txt -- targets/badcode1
    ___FILE___
```

To clearly reproduce the error the fuzzer writes log-files for each crash or unexpected behaviour of the target application with at least the following information:

- The signal that terminated the application.
- The program counter. (location in the binary file)
- Last executed instruction.
- And some other information depending on the signal.

Afterwards the tester inspects the issue by reproducing the fault e.g. in a debugger and can try to exploit it.

3.3.1.2 Environment Fuzzers

Environment Fuzzers are similar to *Command-line fuzzers* except that they fuzz environment variables that are known to be used by the target application.

For demonstration the test application as shown in listing 3.2 is used.

Listing 3.2: Environment fuzzing test target application

```
#include "stdlib.h"
#include "stdio.h"

int main(int argc, char** argv){
  printf("MYVAR=%s\n", getenv("MYVAR"));
}
```

It just prints the value of the environment variable *MYVAR*.
Basically there are two ways of fuzzing the environment of an application.

- The first way is to simply set the environment on program start as shown in listing 3.3.

  Listing 3.3: Simple environment fuzzing

  ```
  $ MYVAR=myvalue ./targetapplication
  ```

  This overrides *MYVAR* for the context of *targetapplication*. The drawback of this approach is that the fuzzer has no information about whether the target application has read the environment variable at all, and which part of the target application is interested in the environment variable. Thus another tool is required to track the accesses to the variables. For this purpose *ltrace* can be used as described in listing 3.4.

  Listing 3.4: Use of ltrace to track environment variable accesses

  ```
  $ MYVAR="my value" ltrace -e getenv -i ./targetapplication
  [0x40054d] getenv("MYVAR") = "my value"
  MYVAR=my value
  [0xffffffffffffffff] +++ exited (status 15) +++
  ```

  This logs all calls of the *getenv* method and value of the *program counter* (location in the binary file) at the execution time.

- A second way of fuzzing an applications environment is by intercepting the call to the *getenv(...)* method. This of course has the prerequisite that the application uses this method. There are other methods available, but *getenv(...)* is commonly used by most developers.

  If a dynamically linked program gets started, it loads its libraries as they are needed. In general the interception of the *getenv(...)* method (or any other method located in a dynamically loaded library) works by pre-loading the user supplied libraries with the modified method implementations. This means that the methods of the pre-loaded libraries will be used before other methods with the same name from later loaded libraries.

  A simple example *getenv*-replacement library is shown in listing 3.5. It loads the original library and saves a reference to its *getenv(...)* method. The *getenv(...)* implementation returns the fuzzed value for *MYVAR* and the original value for all others.

Listing 3.5: Replacement library for *getenv[11]*

```
#include <stdio.h>
#include <gnu/lib-names.h>
#include <dlfcn.h>
#include <string.h>

static void* glibc;
static char* (*real_getenv) (const char*);
static void load_glibc (void) __attribute__ ((
    __constructor__));
static void load_glibc (void)
{
  glibc = dlopen (LIBC_SO, RTLD_LAZY);
  real_getenv = dlsym (glibc, "getenv");
}

char* getenv (const char* name)
{
  if(strcmp(name, "MYVAR") == 0){
    return "my_value";
  }
  return real_getenv(name);
}
```

A real environment-variable fuzzer may extend the functionality of the replacement-

library by adding e.g. backtracking features to get the origin of the *getenv* method call.

The last step is to pre-load the library. This is an operating system dependant task. For *linux* operating systems the environment variable *LD_PRELOAD* contains all libraries that are loaded before any other. A sample call with and without *LD_PRELOAD* is shown in listing 3.6.

Listing 3.6: Call application with and without *LD_PRELOAD*

```
$ MYVAR=no_value ./targetapplication
MYVAR=no_value

$ LD_PRELOAD=./getenv.so MYVAR=no_value ./targetapplication
MYVAR=my_value
```

As you can see the first case returns the real environment variable and the second case returns the value specified in the *getenv*-replacement library as shown in listing 3.5.

Which way the tester chooses depends on the availability of the methods, because not all flavours of operating systems offer the same possibilities. The results of the methods are similar, although the first method is more an out-of-the-box approach. The second method requires some time to implement the library and all its required features but it is more flexible than the first method.

3.3.1.3 File Fuzzers

The first two fuzzing methods did not require much knowledge of the program that is being fuzzed. In general, all the required information is found in the application documentation. This does not apply to *file fuzzers*. They manipulate existing files, load them back in and wait to see if the file parsing code crashes. Therefore the fuzzer needs to be aware of the basic file format structure and protection mechanisms, e.g. if the application looks for a magic-word to identify the file-type. If the magic words do not match or the file-type contains a checksum and the checksum is not calculated correctly, the application will stop processing the file and the complete fuzzing cycle was useless. According to *The Art of File Format Fuzzing[25]* this approach is generally called *brute-force fuzzing*. It has the advantage that every file type can be fuzzed because no knowledge is required but the success rate is very low. A much more effective method is called *Intelligent fuzzing*. Using this approach the fuzzer is

aware of the file structure and can selectively fuzz single file properties and contents. The fuzzer is also able to maintain file checksums to get a valid document. It has a much higher success and a much lower false-positives rate but the file reconstruction or reverse engineering can get very complex for proprietary file formats.

A well-known file-format fuzzer framework is called *SPIKEfile[24]*, a fork of *Immunity's SPIKE - A network fuzzer creation kit*. It is an API that can be used from the C programming language which provides many pre-built methods and entities that ease the fuzzing process and according to *The Art of File Format Fuzzing[25]* has some success stories of discovered vulnerabilities:

- *MS05-009 Vulnerability in PNG Processing Could Allow Remote Code Execution*
- *MS05-002 - Vulnerability in Cursor and Icon Format Handling Could Allow Remote Code Execution*
- *MS04-041 - Vulnerability in WordPad Could Allow Code Execution*
- *MS04-028 - Buffer Overrun in JPEG Processing (GDI+) Could Allow Code Execution*
- *US-CERT TA04-217A Multiple Vulnerabilities in libpng (Affecting Mozilla, Netscape, Firefox browsers)*
- *CAN-2004-1153 Format String Vulnerabilities in Adobe Acrobat Reader*

3.3.2 Remote Fuzzers

Remote fuzzers increase the level of difficulty compared to local fuzzers. The main difficulty for remote fuzzers is to detect whether the remote system is in an error state or not, because the fuzzer only has the information available that the remote system exposes.

3.3.2.1 Network Protocol Fuzzers

As the name suggests *Network Protocol Fuzzers* pay attention to remote services of all kinds including: *mail servers*, *database servers*, *RPC-based services*, *remote access services* or similar. Figure 3.2 shows an overview of the basic fuzzer components.

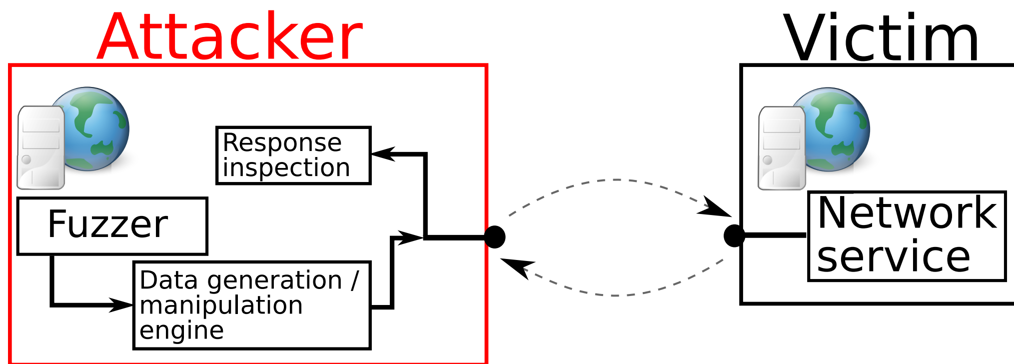

Figure 3.2: Network Protocol Fuzzer overview

The fuzzer basically is built of the following blocks:

- The *Fuzzer block* itself is the control logic. It invokes the *data generation and manipulation engine* and may provide some captured and valid communication data. It also records all exceptions that may occur during the fuzzing process reported by the *response inspection block*.

- The *data generation and manipulation engine block* generates the data, as the name suggests, that is sent to the remote service. In general there are two different approaches to build the data generation block: *Brute force* and *intelligent* approach. They are further described in 3.3.3.

- The *response inspection* block inspects the received responses and tells the Control Unit if something unexpected was discovered, or if there was a response at all.

As you can see, the attacker has no access to log-files or to log-outputs of the target application which complicates the fuzzing process. The attacker may inspect the generated log-files manually or by using available inspection tools but still has to create links from the log-outputs to the generated fuzzer output. This operation is typically not automated by the fuzzer.

A well known representative is the *SPIKE network fuzzer creation toolkit[15]*. Handling *SPIKE* is equal to the handling of *SPIKEfile* as described in 3.3.1.3.

3.3.2.2 Web-application Fuzzers

In general *Web-application Fuzzers* are specialized *Network Protocol Fuzzers* but web applications provide attack surfaces that cannot be covered by generic *Network Protocol Fuzzers*:

- *SQL injection* - SQL Parameters get directly inserted into queries without the use of prepared statements.
- *XSS* - *Cross Side Scripting* - User supplied code gets executed on other users machines
- *Remote code execution* - Supplied code gets executed on the webserver.
- And many more...

3.3.3 Data Generation

A sophisticated, extendable and flexible data generation mechanism is the key to success for every fuzzer. If the data generator only generates ordinary data without focusing on special corner cases, the fuzzer will not generate reasonable results and it would not make any sense to improve the fuzzer itself. Therefore it is important to discuss the available data generation possibilities to get the most out of a fuzz-testing engine.

3.3.3.1 Brute-Force Data Generation

Brute-Force Data Generation is possibly the first but also the worst approach every fuzz-testing engine developer has. As *brute-force* is known from other areas (e.g. password hacking) it tests every possible combination. This is also the case for data generation. A *Brute-Force Data Generator* generates every possible permutation of input data of the given size. This can take a very long time.

Let's assume an 8-bit value. This means *256* different possible values. Assume the fuzzer needs one-second to test a single value (quite optimistic). This results in a test-time of 4-minutes and 16-seconds.

And now a real world example, assume a 32-bit value, 4294967296 different possible values. Applying the previous assumption results in a test-time of approximately 136 years!

As you can see *Brute-Force Data Generation* is not an option as the only data generation method. It may be used to test the response of the application under test to common data and to test for errors that are not dependant on the data itself, but on other properties like the data-size.

3.3.3.2 Intelligent Data Generation

> *Implementation of* how *to generate data is only part of the solution. Equally as important is deciding* what *data to generate.[27]*

Beside the fact that the data generator needs to be aware of its context, there are some globally unique intelligent data generation approaches. *Sutton, Greene and Amini[27]* describe various approaches:

- *Integer values* - For integer values it is obvious to test for their maximum and minimum hexadecimal representation (*0x00000000* and *0xFFFFFFFF*). Probably the size argument is used as an argument to memory allocation routine where it also would make sense to extend the border cases e.g. 1,2,3,...*0xFFFFFFFF-1*, *0xFFFFFFFF-2*, *0xFFFFFFFF-3*... and so on, because the program may extend the size and may therefore trigger an integer overflow.

- *String repetitions* - It generally is a good idea to fuzz string values with long strings. Many programs have buffers of fixed size or dynamically allocated buffers with a maximum length. Without the required sanity checks, long strings will overflow the fixed sized heap or stack buffers and may enable stack-based or heap-based buffer overflow attacks.

- *Field delimiters* - Non alphanumeric characters are often used as field delimiters, therefore it is a good practice to also include field-delimiters like `:|;&_./\<>` and others in fuzzed strings. As an example listing 3.7 shows a *HTTP-Response*. In this example *spaces*, *colons* and *new-lines* can be identified as field separators. The *colons* function as key-value separators. For the fuzzing process it is important to vary all the identified fields (*keys, values and whole lines*) in size.

Listing 3.7: Example HTTP-Response

```
HTTP/1.1 200 OK
Server: Apache/1.3.29 (Unix) PHP/4.3.4
Content-Length: 5000
Content-Language: de
Connection: close
Content-Type: text/html
```

Another well-known example where field-delimiters are exhaustively used are URLs as shown in listing 3.8.

Listing 3.8: Example Request URL

```
http://example.com/page.php?field1=value1&field2=value2&
    field3=value3
```

Again some delimiters can be identified. When fuzzing an URL the keys and the values should be varied in size again.

- *Format strings* - It is always dangerous to use user-supplied strings as arguments to `printf(...)` or similar. Identifying format-string issues is relatively simple by supplying a custom format token e.g. `%d` and watching the output. Exploiting a format-string vulnerability is often done in combination with the `%n` format token, because it is the only format instruction performing write operations. Other format-tokens can only be used to leak sensitive information.

- *Directory traversal* - For web-servers, ftp-servers and other servers accepting paths from remote parties, a potential attack-path is to provide faulty directory values. E.g. a valid path would look like `/path/to/file`. This results in `/server/base/path/path/to/file` on the server and the client is not allowed to access files outside the servers base-directory. A faulty path would look like `/../../secret/path`. If the server does not have the required sanity checks implemented this results in `/server/secret/path`. Therefore the data generator should also fuzz directory paths.

- *Command injection* - They always occur if user supplied input is directly passed to methods like `exec` or similar for other programming languages. An example is shown in listing 3.9.

Listing 3.9: Command Injection Vulnerability[27]

```
directory = socket.recv(1024)
listing   = os.system("ls /" + directory)
socket.send(listing)
```

In normal operation a path is received and the listing is sent back to the sender. But because of the lack of input data checks the received "directory" could also look like `var/lib; rm -rf /` which results in the command `ls /var/lib; rm -rf /`. Therefore the data-generator should also include the command-injection-trigger characters as well.

3.4 Summary

Fuzzing is a highly automated testing method. It puts a high load of malformed but also valid data on the target device or target software and inspects the response. It mainly operates on binary files or on remote application servers depending on the type of fuzzer. Generally fuzzers test the application from a user's point of view and not from a developer's point of view as many other testing methods do. Various different types of fuzzers exist. To understand the concept of fuzzing a network fuzzer suits best. It connects to the remote host as any other client would do and sends malicious requests to the remote side. If the fuzzer can observe any unexpected replies or even no reply it logs the required data to reproduce the error. Afterwards the tester can examine the results and analyse any problems that occurred. This key concept applies to all available fuzzer types.

A key component of every fuzzer is the data generator. Generally two different data generation strategies are available: *Brute-Force Data Generation* and *Intelligent Data Generation*. The first approach simply generates random input data to preferably cover the whole input data range. In fact this approach has a very small chance of success because most protocols (in the case of a network fuzzer) use checksums, replay protections or encryption to protect the communication channel. Using the *Intelligent Data Generation* approach the data generator is aware of the underlying protocol and can generate valid packets that are accepted by the remote side and therefore have a much higher chance of finding an error. It is clear now, that the data generator is a key component of the whole fuzz-testing-setup. Fuzzers can only be as good as their data generation-engine.

Chapter 4

In-Memory Fuzzing

Why does *In-Memory Fuzzing* get its own chapter? In contrast to other fuzzing techniques it is a completely different approach. It does not fuzz user-input, files or any communication channel. It focuses on the functions a binary is composed of and fuzzes the functions parameters. More details on *In-Memory Fuzzing* are shown later in 4.2.

4.1 Problems of Other Fuzzing Techniques

Assume a network fuzzing setup as shown in figure 3.2. The tester may be faced with one or more of the following challenges:

- The victim may be protected against attacks by only allowing a few requests in a given time. This would extremely slow down the fuzzing process and the chance to find bugs.
- Depending on the transmission speed and the size of the requests it may be necessary to transmit a huge amount of data to the remote side. Even if each single test only has a few bytes, multiplied with the required test runs to cover the input data range, again a huge amount of data needs to be transferred.
- The biggest problem arises when it comes to closed source or proprietary protocols. To be able to fuzz such protocols the fuzzer needs to know some details of the protocol.
  - For mutation based fuzzers it may be possible to just mutate some bytes, if the protocol has no checksum, no replay protections and no security at all otherwise at least some internals are needed.

- For generation based fuzzers the data generator needs to know the exact structure of the protocol, which can only be discovered by using reverse engineering techniques, in case of closed source or proprietary protocols. This approach may be practical if you have the source code of the application and write the fuzzer just as another client.

The same facts also apply to other fuzzers, like file-format fuzzers where you may need to reverse engineer the file-format to be able to produce valid files. This is where the *In-Memory Fuzzing* approach comes into account to make the fuzzing process more efficient. Using this approach there is no need to reverse engineer the network-protocol or file-format, to be able to fuzz a proprietary format.

4.2 How In-Memory Fuzzing Works

An *In-Memory Fuzzer* does not focus on file-formats, protocols or command lines, and therefore does not need to know the structure. In contrast it focuses only on the underlying code, the methods and especially its parameters. This has the advantage that the network connection channels are skipped and any form of replay protections, checksums, data encryption and other security mechanisms are not working any more.

In general we can say that the fuzzer hooks at points of interests. A basic scheme looks as shown in figure 4.1. It represents an application with a network-stack which uses a checksum-enabled and encrypted protocol. Before the receiving packet gets processed the network stack receives the packet, calculates the checksum over the payload, compares it to the received checksum and finally decrypts the payload. To attack a networked application with this layout using a network protocol fuzzer the developer needs to reverse engineer the structure of the protocol, the checksum calculation algorithm, the decryption mechanism and finally also needs a valid encryption key.

Using in-memory fuzzing the fuzzer hooks before and after the parsing code. Once the first hook is hit the fuzzer changes the variables of the processing method (e.g. the decrypted payload) and after the parsing code it restores the program to the state it was before. Using this approach the program only gets started once, and many inputs can be tested, saving much time. Moreover the developer does not need to reverse engineer the network stack or any of its components. If a failure occurs, the fuzzer can exactly rebuild the inputs that are necessary to reproduce the error.

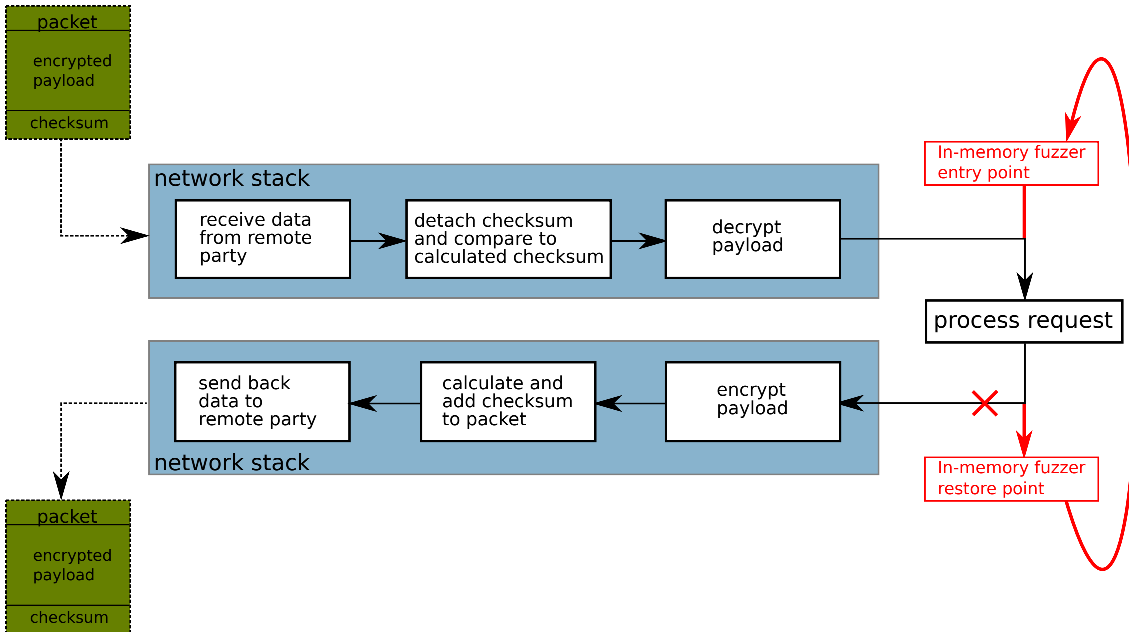

Figure 4.1: Basic In-Memory Fuzzing scheme

Sutton, Greene and Amini[27] propose two fundamental schemes of how to hook the target application.

4.2.1 Mutation Loop Insertion

The goal of the *Mutation Loop Insertion* method is to insert a *mutate* method in the applications binary (or in the running process) which gets called directly after the method under test (call it the *parse* method). The *mutate* method changes the input parameters of the *parse* method on each run by invoking the associated data generator and unconditionally jumps to the *parse* method again. Figure 4.2 shows a simplified program flow with the additions inserted by the *Mutation Loop Insertion* method. Using this approach the in-memory fuzzer can test a method a thousand times with different input parameters by only launching the application once. This bypasses the problems of other fuzzers discussed in 4.1. The *parse* method only needs to be externally triggered once (green path). The inserted loop recalls the method again and again with different input parameters (red path). The (external-) fuzzer logic records occurred problems and other relevant data.

The drawback of the *Mutation Loop Insertion* method is, that not all side-effects caused by the function under test get restored, this may yield a lot of false-positives. Take a simplified pin checking method as shown in listing 4.1. This method allows to check for the correct pin two times and rejects all pins then.

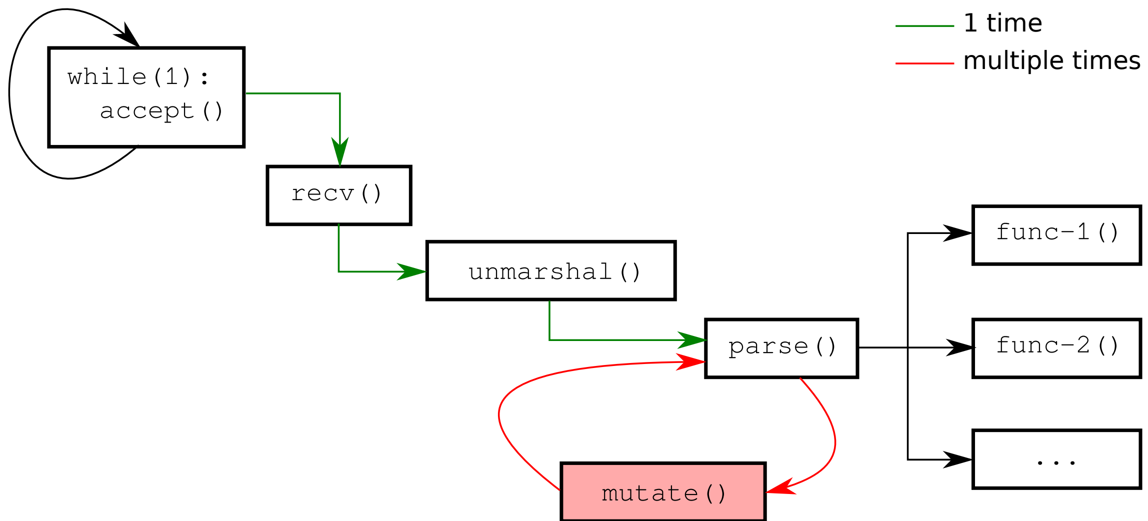

Figure 4.2: Mutation Loop Insertion scheme[27]

Listing 4.1: C-example demonstrating the drawback of the *Mutation Loop Insertion* method

```
int check_pin( char* pin ) {
    static int attempt = 0;

    if ( attempt < 2 )  {
        if(internal_check_pin(pin) == 0){
            attempt++;
        } else {
            attempt = 0;
            return PIN_ACCEPTED;
        }
    }
    return PIN_REJECTED;
}
```

The following list shows the first three fuzzer rounds:

1. The first run triggers the fuzzer start-hook, everything works as expected.
2. The fuzzer inserts a fuzzed pin value. Everything works as expected. The `attempt`-variable has a value of *2*.
3. The fuzzer again inserts a fuzzed pin value, but regardless of the given value the `check_pin(...)` method will always return *PIN_REJECTED* for the third and all subsequent fuzzing-rounds.

The cause of this problem is, that the fuzzer does not restore the original value of the `attempt`-variable because it is of static type and keeps its value over multiple method calls.

To summarize we can say that this approach is only applicable for methods without any side-effects. If the tester can not be sure that the method under test has no side-effects, it is not safe to use this method and may yield many false-positives.

4.2.2 Snapshot Restoration Mutation

The *Snapshot Restoration Mutation* method is similar to the *Mutation Loop Insertion* method, with the difference, that it creates a complete process snapshot at the beginning of the method under test and restores the snapshot before fuzzing the arguments for the next run.

Figure 4.3 shows a simplified program flow with the additions inserted by the *Snapshot Restoration Mutation* method. Again, the *parse* method only needs to be externally triggered once (green path). The inserted *snapshot* and *restore* methods are called on every fuzzer run (red path).

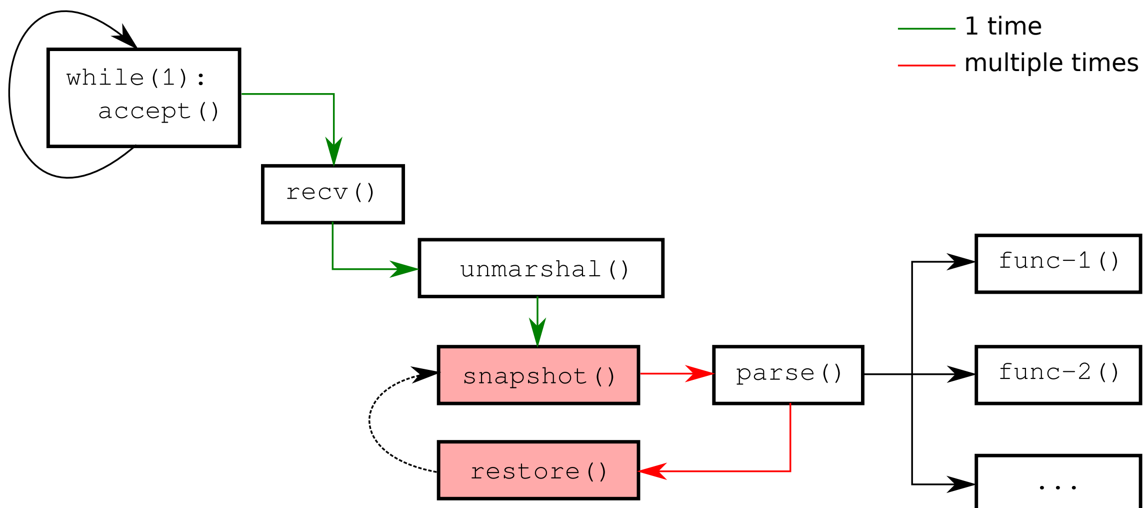

Figure 4.3: Snapshot Restoration Mutation scheme[27]

Using this method every fuzzing-round runs under the same conditions, this effectively eliminates false positives. Because of the continuous snapshot creation and restoration this approach is generally slower than the *Mutation Loop Insertion* approach.

4.2.3 Implementation Aspects

When it comes to the implementation of either of the two methods you will notice that the requirements of both methods are similar:

- Choose the hook points
- Hook at specific points in the application
- Access (read and write) memory in the applications memory space
- Restore process snapshots and/or insert mutation loop
- Catch process exceptions

All of the requirements are best satisfied by implementing custom debuggers or extending existing debuggers. Software- or hardware-breakpoints are used to hook at specific points. Most debugging APIs and debugging frameworks provide methods to read and write into the target application's memory space. The snapshot and restoration feature may not be included in the debugging API or framework, although some frameworks have this feature built-in, but it can be implemented by using standard debugging features.

For Windows operating systems the easiest way of implementing a custom debugger is by using the *PyDbg[6]* debugging framework. It is a highly flexible wrapper around the Windows debugging API with many additional functions. The snapshot capability is provided directly by *PyDbg*. To choose the hook-points for Windows applications, as it is for other operating systems, some reverse engineering skills and tools are required.

- *OllyDbg[30]* is a Windows 32-bit assembler-level binary code debugging and analysis tool. It is ideal for examining binary files where no source code is available, hence it is ideal to find hook points on Windows operating systems.
- *IDA - Interactive Disassembler[23]* is a cross-platform disassembler, debugger and program flow analyser. It can visualize the disassembled output to assist the reverse engineer in finding the hook points.

For Linux/Unix operating systems there are several debugging APIs and toolkits available:

- The *Ptrace* system-calls provide the tools necessary to implement breakpoint debugging and system call tracing. This is a low-level API, therefore whenever possible pre-built frameworks and tools should be used.

- *GDB: The GNU Project Debugger[4]* is a well-known debugger, which uses *ptrace* if available. The *GDB* provides all functions necessary to implement an in-memory fuzzer. Chapter 5 focuses on the implementation details and technologies used. In fact *GDB* is an all-in-one solution because it also functions if no source code is available and breakpoints can be set based on assembly code. Compared to *IDA* it lacks the reverse engineering assistance, however *IDA* and *GDB* are a good combination.

4.3 Requirements

To create a fuzzer that can be effectively used in production environments following minimum requirements must be met:

- *Modularity and expandability* - It should be possible to easily exchange important components of the fuzzer. E.g.:
  - *Extend the fuzzer with other connection methods.* This is especially important for embedded systems, because they may be connected through any proprietary port and may require a special application to establish a connection to the target.
  - *Plug-in other fuzz-targets.* There may be an application with a data structure that can not be mapped with the existing fuzz-targets.
  - *Plug-in new data generation methods.* An easy way is required to create new data-generators which ideally tune their generated data for the target application.
- *Portability* - As discussed in chapter 4.4 currently no in-memory fuzzer is available for Linux/Unix operating systems. The only (freely-) available implementations are based on *PyDbg* and therefore are not usable on operating systems other than Windows. For *In-Memory Fuzzing on Embedded Devices* it is crucial to be portable among all major operating systems.
- *Flexibility* - Some embedded systems have more and some less memory available. It should be possible to use the complete fuzzer feature-set on every system, regardless of whether the system has enough memory for running the complete suite or not. This can be accomplished by only running the necessary components on the device. They can communicate with the host system with

a full-blown setup. If the device has enough resources it has the option to run the complete fuzzer.

- *Small in size* - This requirement goes hand in hand with the *Flexibility*-requirement. It should be possible to run the fuzzer with a very low footprint.

- *Speed* - Depending on the resources of the target device a single fuzzing-round should be quick. With huge round-times the advantages of in-memory fuzzers fall off.

- *Usability* - It is clear that moderate skills in various disciplines like reverse engineering are required, but once these challenges are mastered the fuzzer should be easily configurable.

- *Improve results with additional information* - For many applications additional information in the form of documentation or debugging symbols is available. This information should be used by the in-memory fuzzer to improve the usability and the results. Debugging symbols can be used to significantly ease the hook point extraction and may make the reverse engineering step obsolete or at least considerably ease it. The application documentation may describe some internal structures, therefore it should be possible to include manually extracted information in the reverse engineering step.

4.4 Existing Frameworks

In-memory fuzzing is still a field of heavy research. As of today, there are some proof-of-concept fuzzers but none of them are commercially available or have ever left alpha stage. The requirements for a ready-to-use in-memory fuzzer are defined in chapter 4.3, but none of the currently available fuzzers meet all requirements:

- *In Memory Fuzz PoC[26]* - As the name suggests this is a proof-of-concept implementation from the authors of *Fuzzing: Brute Force Vulnerability Discovery[27]*. It uses *PyDbg[6]* as debugger backend and clearly shows the fuzzing-concept. It was never meant to be a fuzzing framework or ready-to-use fuzzing tool.

- *HyperTest* - Not much information is available concerning *HyperTest*, only some source code without documentation. Testing the implementation showed that it is working. Examining the source code showed that it uses the Windows

debugging API directly. Compared to the fuzzer above the only difference is that it bypasses *PyDbg* and invokes the debugging API directly.

- *Corelan In-Memory Fuzzer[29]* - Currently the most advanced in-memory fuzzer implementation (freely-) available. It is also implemented using python and the *PyDbg* toolkit, therefore it is only available for Windows operating systems. This fuzzing toolset is separated into two components: the *Tracer* and the *Fuzzer Engine*.

  The *Tracer* eases the process of identifying the methods that process the user input and its parameters. The workflow of the *Tracer* is shown in figure 4.4.

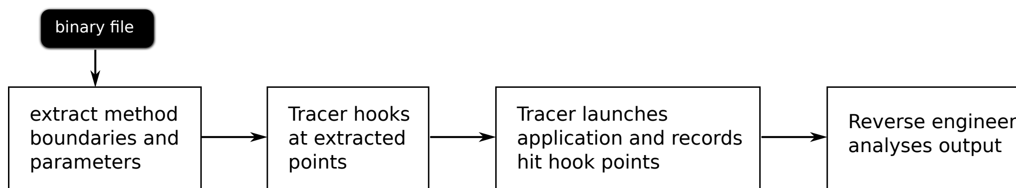

Figure 4.4: Corelan In-Memory Fuzzing Tracer workflow

The first step is to extract a function list from the binary file, this can either be done with the source code (if available) or with the help of disassembler tools like *OllyDbg* and *IDA*. They process the binary file and look for known patterns identifying function boundaries. The output is a list with all identified function start addresses, their return addresses and the discovered function parameters. Then the *Tracer* hooks at all of the discovered methods, for example, by setting software- or hardware-breakpoints and launches the application with known and valid input parameters. Every time a discovered and hooked method is called, the *Tracer* breaks and logs the call and all of its parameters. A Sample log output is shown in listing 4.2.

Listing 4.2: Sample Tracer log output

```
Log: function 0x10001800(
       [ESP+8] 0x1b33184  "AAAA (heap)"
);
```

It shows a single call of a hooked method. The known input was *"AAAA"* in this case. Using this approach, the reverse engineer is able to reconstruct the flow of the supplied input data (*"AAAA"*) through the complete program and then decides which method to fuzz. Using any method without tracing

the user input may also result in discovering bugs of internal methods, but generally they are not triggerable by the user.

Once the snapshot and restore-points are defined, the *Fuzzing Engine* works as described in 4.2.1 and 4.2.2.

The drawbacks of this fuzzer are:

- It heavily builds on *PyDbg*, therefore it is only usable on Windows operating system. No Linux/Unix or embedded system support is available.
- It only uses a rudimentary data generator and does not implement the guidelines as described in 3.3.3.2.
- The fuzzer only detects program crashes, but there may also be overflows (stack- or heap-based) or other errors that may not result in a crash.

4.5 Summary

In this chapter a different approach of fuzz-testing called *In-Memory Fuzzing* was introduces and described why another method is required. Once approaching closed source or proprietary protocols and file formats, other fuzzers may require a lot of reverse engineering to rebuild the protocol- and file-structures. An in-memory fuzzer does not need to know the structure, it focuses only on the underlying code, the methods and especially its parameters. It hooks before and after relevant methods and changes the variables of the method-under-test before it is called, restores the original program state afterwards and tests the method again. Two different implementation approaches were discussed: the *Mutation Loop Insertion* and the *Snapshot Restoration Mutation* method. When it comes to the implementation of one of the methods I discovered that the requirements for both methods were nearly the same. Both need access to the applications memory space and need to insert code or hook at specific hook-points. Debuggers fulfil all the requirements and are suited best to implement a custom in-memory fuzzer. At the end of this chapter the existing in-memory fuzzing frameworks were analysed. None of the available frameworks can be used in production environments because all of the available fuzzers are either proof-of-concept implementations or never left pre-alpha stage. The best results were achieved using the *Corelan In-Memory Fuzzer*, but the implementation is based on the *PyDbg*-debugging framework, which is only available for Windows. The architectural idea of the the *Corelan In-Memory Fuzzer* is adopted and advanced in chapter 5.

Chapter 5

Implementation Details

This chapter focuses on the implementation of an in-memory fuzzer (further referenced as *IMF*), that was created during the research for this thesis. The requirements for embedded systems are separately highlighted and all components are described in detail.

What are the goals of a newly developed in-memory fuzzer especially designed for embedded systems? Most embedded systems use operating systems based on Linux, but also bare embedded systems are available. One goal of *IMF* is to operate on both approaches and still be extensible to future embedded systems with other operating systems. Most existing fuzzers only monitor the application under test for crashes or unexpected behaviour. In-memory fuzzing enables the tester to find bugs which do not result in an application crash. Take an application with a stack-based buffer overflow bug. Once the buffer is overflowed it may not necessarily result in an application crash. Therefore one goal of an in-memory fuzzer is to also find bugs which do not instantly result in application crashes but may be exploitable under special conditions.

The required implementation effort to create an in-memory fuzzer from scratch is enormous and would result in feature or usability compromises. Therefore it is beyond dispute to reuse existing applications and components to realize the requirements as specified in 4.3. As already discussed in 4.4 the most promising available in-memory fuzzer is the *Corelan In-Memory Fuzzer*, although it is only available for Windows operating systems it is built on an extensible architecture. It also provides an external tool to ease the process of reverse engineering and of finding the hook-points. In its early stage the *IMF* implementation has been oriented towards the *Corelan In-Memory Fuzzer* architecture, but has been heavily improved and extended later on.

5.1 Architectural Overview

IMF is an implementation of a *Snapshot Restoration Mutation* based fuzzer as described in 4.2.2. The architectural overview of *IMF* is shown in figure 5.1. It shows a setup where the fuzzer footprint on the target device is as small as possible for use on embedded devices with limited resources.

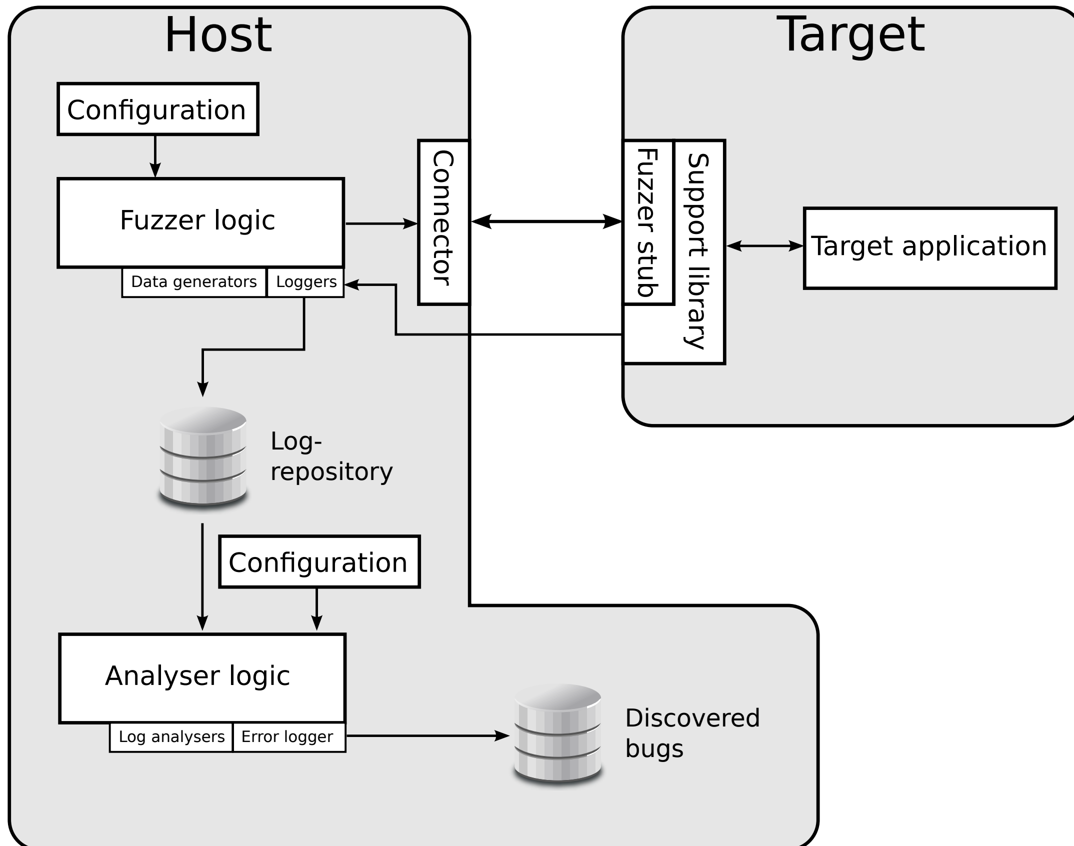

Figure 5.1: Architectural overview of *IMF*

The fuzzing suite is separated into two parts, the fuzzer and the analyser. The fuzzer is built of various components again. To be as flexible as possible and to be able to run the fuzzer on all devices, regardless of their available resources it uses a *stub* - *connector* approach. The *stub* only receives commands from the fuzzer logic which connects using an exchangeable connector. If host and target are the same machine the *connector* and *stub* can meld to a single component. On the target device a *support library* is used to perform device or architecture dependant tasks and log target specific events. The *support library* connects back to the host (or uses a prebuilt communication channel) to communicate with a dedicated *support libary* logger.

The *fuzzer logic* gets extended with various pluggable components, which are provided by the *configuration* component. For the *fuzzer logic* the pluggable components are:

- *The data-generators* - Some predefined data-generators are available. They can be extended by providing external libraries or by simple script-files attached to the main configuration file.
- *Loggers* - The speed of the fuzzing process varies depending on the captured information. Depending on the attached loggers the fuzzer may only detect certain vulnerabilities. E.g.: If the fuzzer is required to detect heap-based buffer overflows a specific logger and support from the *support library* to log all memory allocations is required. If the target application crashes because of a heap based buffer-overflow and the logger was not activated, there is no way to determine the real reason of the crash, nevertheless the tester will be able to reproduce the error with the recorded input data.

  The desired loggers can be attached to the fuzzing process using the configuration file. It may also be necessary to include custom loggers for custom target devices.
- *Fuzz Targets* - Easily extend the fuzzer to support custom data structures.

While the fuzzer is running the *loggers* write their log-files to a *log-repository.* For each fuzzing round an associated array of log-files is written to the repository. Depending on the attached loggers this includes:

- *Fuzzdata* - Contains the data that has been generated for this specific round. If an error occurs, this data can be used to reproduce the error or debug the error manually.
- *Errorlog* - Contains information about obvious errors in this fuzzing round, e.g.: segmentation fault.
- *Execlog* - Contains all read and write operations during this fuzzing round.
- *Stackframeinfo* - Contains all information that can be extracted from the current stack frame in the end of each fuzzing round.

The second part of the fuzzer suite is the analyser. Its configuration is similar to the configuration scheme used for the fuzzer. The analyser does not require a

connection to the target device and only operates on the log-files generated in the fuzzing process. This has the advantage that particular analysers can be improved later on, when the fuzzing log-files have already been recorded without the need to run the fuzzer again. Various *log-analysers* can be attached to the *analyser-logic* using the configuration file. They are further discussed in 5.3.

The output of the analyser is a single file containing all discovered errors and additional information, such as: in which round did the error occur, the current execution address and additional logger specific values.

5.2 The Fuzzer Engine

Before the implementation can start an agreement on the programming language must be made. It might seem obvious to use a low-level programming language because it is a strict requirement that the fuzzer is usable on embedded systems. But the required implementation effort would significantly increase. This would result in compromises which decrease the feature-set and lower the flexibility. Therefore *C#* compiled with *Mono* was chosen. This enables the complete flexibility, resulting in an engine which can dynamically load components. Because of the high modularity it is possible to implement only small parts (or stubs) using C or C++ programming languages, which are already ready-to-use as discussed in 5.2.7.This brings us directly to the components of the fuzzer.

5.2.1 Configuration

Currently the fuzz-configuration is stored in an xml-file, an example is shown in listing A.1.

The file can contain *Include*-tags. The values of the tags are supposed to be filenames. The included files contain key-value pairs that can be used in path specifier to parametrize the path values. Using this approach it is simple to adapt an existing fuzz-configuration to other machines.

In general the configuration file is divided into three groups: *TargetConnection*, *FuzzDescription* and *Logger*.

The *TargetConnection* group specifies how to create a target connection. This can either be a remote connection, or a local connection to local running target applications. Currently the only implemented *TargetConnection* is the `general/gdb` connector which uses *GDB* as its backend. But *GDB* can be used in multiple modes:

- It can be used in normal operation mode where *GDB* starts the target application on its own in the background. To use this mode the *target* value needs to be set to *run_local*.
- To attach *GDB* to a local running process the *target* value needs to be set to *attach_local* and the value *target-options* with the process ID of the running process needs to be inserted.
- Another operation mode is *extended-remote host:1234* where *GDB* connects to a remotely running *GDBserver* instance on port *1234*.

More details on *GDB* and its internals are shown in 5.2.7.

The *FuzzDescription* group specifies where to set the snapshot- and restore-points and which values should be fuzzed. Moreover the *FuzzDescription* tag can contain multiple *FuzzLocation* tags. Every *FuzzLocation* can contain triggers which specify when to start fuzzing, when to stop and which data generator to use. Details on data generators are discussed in 5.2.6. Chapter 5.2.5 focuses on *FuzzLocations* especially on the available data region specifiers.

The *Logger* group defines all attached loggers. More loggers may slow down the fuzzing process but yield more log output and increase the chance of discovering bugs and vulnerabilities. Currently the log-repository is a directory where all log files, prefixed with the current fuzzing-round, are saved.

5.2.2 Bootstrapping the Fuzzer

The heart of the fuzzer is the *FuzzFactory* and the *FuzzController*. The first step is to parse the configuration and build all necessary structures. This is where the *XmlFuzzFactory* is invoked to perform the following steps:

- Initialize connection to the remote support library. For the purpose of this implementation the proposed architecture was slightly modified. The *support library* was enhanced to get a lightweight *remote-control* program. Further details on the *remote-control* are discussed in 5.2.4.
- Initialize target connection. According to the configuration the symbol table gets loaded (if available), the target connection gets initialized and the connection gets established. More information on the target connection and the inner workings can be found in 5.2.7.

- Initialize fuzz targets. All the available *FuzzDescriptions* are iterated and the specified snapshot and restore points are set. An in-depth discussion of *FuzzDescriptions* and *FuzzLocations* can be found in 5.2.5.

- Initialize loggers. All specified loggers are attached to the fuzzing engine and the target directory gets initialized. Loggers and the produced data are discussed in 5.3.

5.2.3 Support Library

The *Support Library* is a replacement library for *malloc, calloc, realloc* and *free* and is tightly coupled with the *Remote Control 5.2.4* application. In fact it intercepts the call to the memory-allocation methods and outputs the following information:

- The values of the parameters. Especially for *malloc* (and similar) calls it is important to know the size of the allocated memory block.

- Internally the replacement library calls the 'real' methods and outputs its return value.

- Finally the replacement library outputs a complete stack trace (if available). This is important for the analysis phase to identify where a specific call came from.

The output of the *Support Library* is written to a pipe as specified in the *LOG_MEM_PIPE* environment variable or to *stderr* if none is specified.

A sample call using this library is shown in listing 5.1.

Listing 5.1: Sample call of the *Support Library*

```
LOG_MEM_PIPE=/tmp/mem_alloc LD_PRELOAD=./log_memory_allocations.
    so ./targetapplication
```

The call depends on the operating system and its capabilities. A user may not be allowed to use the *LD_PRELOAD* environment variable as discussed in 3.3.1.2.

The output is structured as shown in listing 5.2.

Listing 5.2: Output of the *Support Library*

```
<function name>: args=[<arg name>=value,...return=value] bt[0x<
    address>,...]
```

- *Function name* can be one of *malloc, calloc, free* and others depending on the implementation.
- *Args* specifies all arguments supplied to the corresponding method including its return value as the last argument seperated by spaces.
- *Bt* - complete backtrace. The instruction pointer addresses are listed from the inner frame to the outer, they are preceded by 0x and seperated by spaces.

5.2.4 Remote Control

The *Remote Control* application enhances the *support library.* For best embedded device compatibility it is written in the C programming language. It listens for connections on a TCP socket at the configured port and controls the remote side of the fuzzer. The protocol is a simple bidirectional packet based protocol as shown in listing 5.3.

Listing 5.3: Remote Control protocol specification

```
<FUZZ><receiver subsystem 4-chars><data length 2-bytes><data>
```

Each packet starts with the magic number FUZZ (4 bytes) followed by the receiving subsystem identifier with 4 characters. Each receiver specifies a subclass of packets and is freely defined as needed. The receiver also specifies the structure of *data.* The maximum length of a single packet may not exceed 10kB, so the data can only have a maximum length of 10KB-10B.

Currently the following features are available:

- *ECHO* - Command to test the device connectivity.
- *EXEC* - Executes the specified program using *fork/execve* and sets the specified arguments and environment variables. The data is structured as shown in listing 5.4.

  Listing 5.4: Remote Control *EXEC*-data structure

  ```
  data: <program_name_length int16><program_name>
        <program_path_length int16><program_path>
        <argument_count int16>
         [<arg1_length int16><arg1>,...]
        <env_count int16>
         [<env_name1_length int16><env_name1>,...]
  ```

The *program_name* field can be freely chosen and is included in every response concerning this particular program instance.

Once the process is forked a response is sent containing the newly spawned process id. The response is structured as shown in listing 5.5

Listing 5.5: Remote Control *EXEC* response data structure

```
REXS (Response exec status)
data: <program_name_length int16><program_name><pid int32>
      <status int32>
```

If the execution was successful the status is 0, otherwise status is an error code and the process id does not contain a valid value.

- *PROC* - Reads all running processes, process IDs and their associated users and sends it to the remote side. This command does not accept any data but sends a response with the structure shown in listing 5.6.

Listing 5.6: Remote Control *PROC* response data structure

```
RPRC (Response PROC)
data: <number_of_processes int32>
      [
       <number_of_values_for_single_process int16>
       <length int16><value>,...
      ]
```

Multiple values for each process are sent, they always have the form of *key=value* where only the first '=' is of interest. Unknown keys should be ignored by the host, currently known keys are: *user:* donates the associated user ID, *cmd:* gives the command that was executed to spawn this process, *pid:* contains the associated process ID.

- *PIPE* - Opens the specified pipe for reading and sends all received data to the host. The pipe request packet's data only contains the pipe-name to listen on. The response packet as shown in listing 5.7, is sent multiple times, as long as the remote side of the pipe remains open. It only contains the plain data read from the pipe.

Listing 5.7: Remote Control *PROC* response data structure

```
RPIP (Response pipe)
data: <int16 pipe_id><pipe_data>
```

This command is especially used to send the log-output generated by the *support library 5.2.3* to the host machine.

The *Remote Control* application can be seamlessly integrated into the fuzzing process, by extending the configuration file as shown in listing 5.8. The *Remote-Control*-section defines the host and port to connect to (where the application is running) and defines one or more *Exec*-tags. The command represented by the *Exec* tag gets executed on the remote side with respect to the trigger value. Currently the following trigger values are available: *immediate* - executes the program right after the connection has been established, *onFuzzStart* - executes the program at the beginning of every fuzzer run and *onFuzzStop* - executes the program after every fuzzer run.

Listing 5.8: Remote Control integration

```
<RemoteControl>
  <Host>localhost</Host>
  <Port>8899</Port>
  <Exec trigger="immediate">
    <Cmd>commandline</Cmd>
    <Arg>arg1</Arg>
    <Arg>arg2</Arg>
  </Exec>
</RemoteControl>
```

With the help of the *Remote Control* an automated, remote fuzzing process can be modelled.

5.2.5 Defining Fuzz-Descriptions and Fuzz-Locations

Fuzz-Descriptions and *Fuzz-Locations* define the exact region and arguments to test. A class hierarchy of the *Fuzz-Description* and *Fuzz-Location* related classes is shown in figure 5.2.

The *Fuzz-Description* is composed of a snapshot and restore point, modelled as a breakpoint. The *IBreakpoint* interface is the base interface for all connector-specific

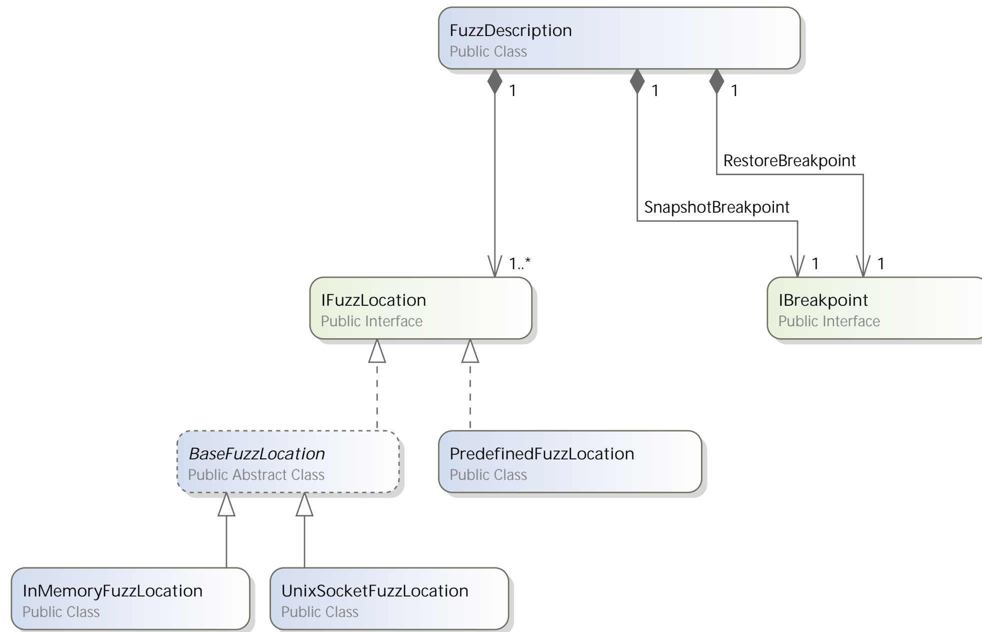

Figure 5.2: Fuzz-Description and Fuzz-Location class hierarchy

breakpoints. According to the configuration listing in A.1 multiple breakpoint specifiers are available, which get translated to a connector specific breakpoint object:

- `method|methodname` - Sets a breakpoint at the beginning of the specified method if symbols for the loaded binary are available.
- `address|0x00000000` - Sets a breakpoint at the specified address. This specifier can also be used if no symbol table is available.
- `source|file,line` - Sets a breakpoint in the specified source-file and line if symbols are available.

Every time the snapshot breakpoint is hit, the *Fuzz-Locations* are invoked in the same order they are specified in the configuration file. A *Fuzz-Location* specifies the data-region to fuzz in the context of the snapshot breakpoint. *IMF* implements a very flexible scheme and in fact is not locked to in-memory fuzzing only, even though the main focus is on in-memory fuzzing. Nevertheless, for some applications it might be required to use a secondary fuzzing channel to build a fuzzing configuration. To cover this case the *FuzzerType* can be specified in the configuration. The currently available fuzzer implementations are: *InMemoryFuzzLocation* (`fuzzer/in_memory`) and *UnixSocketFuzzLocation* (`fuzzer/unix_socket`). For details on the *UnixSocketFuzzLocation* see the source code documentation.

For the fuzzer it is required to resolve the target-region to a valid address. But for usability reasons multiple specifiers are available where human readable variable names or expression can be specified. The following listing gives an overview of the available specifiers and how the fuzzer resolves them to a valid address:

- `variable|name` - Resolves the specified variable name using the symbol table in the context of the snapshot breakpoint as its target, therefore local variables are valid too.
- `address|0x00000000` - Uses the specified static address as its target.
- `calc|${reg:rbp}+24` - Calculates a given expression and uses the resulting value. Special variables are substituted by the preprocessor: *${reg:regname}* - gets replaced by the value of the specified CPU register; *${0xDEADBEEF}+1234* - to directly specify hexadecimal values.
- `cstyle_reference_operator|struct->val` - Depending on the connector, it is not always possible to resolve an address using the above methods. This especially applies to single fields of structures or classes. For this purpose the *cstyle_reference_operator*-method is available. It prepends the *addressOf* (&) operator to the specified expression and evaluates the result.

Another important feature of the *InMemoryFuzzLocation* is to fuzz data in different ways. The requirements for fuzzing an integer value and for fuzzing a dynamically allocated string value are entirely different. To fuzz an integer value, only the address and size (which is implicitly defined by the data generator) of the value are required. The fuzzed value can be written directly to the resolved address. For dynamically allocated strings the situation is completely different. The generated fuzz-data may vary in size and therefore the original buffer may be too small, thus a new buffer needs to be allocated (and freed afterwards) and the variable pointing to the old buffer needs to be set to the address of the newly allocated buffer. To solve this issue, the fuzzer supports multiple fuzz technologies: *single_value* - Writes all fuzzed data to the resolved address. With this implementation it is possible to fuzz all primitive types and static arrays. *pointer_value* - Interprets the resolved value as pointer. New memory gets allocated where the fuzzed data gets written to. The pointer is set to the address of the allocated buffer.
As already mentioned earlier, the *Fuzz-Locations* are always invoked when the snapshot breakpoint is hit. But if multiple locations are specified the tester may not

want to invoke all of the *Fuzz-Locations* on each hit. For this purpose the *Trigger* and *StopCondition* capability were introduced.
The *Trigger* value tells the fuzzer when to invoke a specified *Fuzz-Location*. Three different trigger types are available: *start* - always invokes the trigger after the snapshot has been restored and before the fuzzing-round starts; *end* - invokes the trigger in the end of a fuzzer run; *location* - the trigger specifies an address where the trigger gets invoked.
After the trigger-type specifier a number can be specified which tells the fuzzer how often to invoke the *Fuzz-Location*. E.g.: a number of *3* tells the fuzzer to invoke the *Fuzz-Location* only at every third run, therefore a number of *1* (or no number) tells the fuzzer to invoke it every time.
The *StopCondition* in contrast specifies when to not invoke the fuzzer. Currently the only implemented *StopCondition* is `count|num` where the fuzzer runs the associated *FuzzLocation* *num*-times. If no *StopCondition* is specified the *FuzzLocation* gets invoked as long as the fuzzer gets interrupted by the user.

5.2.6 Data Generators

Data generators are one of the most important components and need to be as flexible as possible. *IMF* uses an approach that enables the tester to easily extend the fuzzer engine with new data generators. The configuration of the data generators is illustrated in listing A.1. All specified *DataGenArg* key-value pairs are passed to the data generator, therefore writing a new data generator does not require the developer to write any parsing code. The predefined and ready-to-use data generators are:

- `datagen/fixed_bytes` - This data generator always outputs the same sequence of bytes, which is specified in the configuration.

- `datagen/random_bytes` - This one generates random byte sequences with respect to the specified type and length constraints. The following type-constraints are available and should be self explanatory: `All`, `PrintableASCII`, `PrintableASCIINullTerminated` and `AllNullTerminated`.

  For the length constraint three different types are available: `random` - data of random length between the minimum and maximum length is generated; `increase` - data length is increased by the specified value starting with the minimum length; `decrease` - data length is decreased by the specified value starting with the maximum value.

- `datagen/replay` - This data generator replays the data from another fuzzing run. It reads previously applied data from the generated log output.

To include custom data generators without having to write custom libraries, a scripting engine has been included which supports slightly modified C#-style code. The scripting data generator is identified by `datagen/scripted`. A sample data generation script is shown in listing 5.9. The difference to common C# code is, that no surrounding class or method declarations are required. Imports can be done with the special command `#import` followed by `#endheader`.

The fuzzer defines some methods that can be used directly without importing anything:

- `SetValue(string name, object value):void` - sets a data generator specific value in its environment which can be used across multiple calls.
- `GetValue(string name):object` - retrieves a previously set value.
- `IsValueSet(string name):boolean` - checks if the specified value is set.

Listing 5.9: Remote Control integration

```
#import System.IO
#endheader

int size = 1;
if(IsValueSet("last_size"))
  size = (int)GetValue("last_size") + 1;
SetValue("last_size", size);

byte[] data = new byte[size];
for(int i=0; i<size; i++){
  data[i] = (byte)(i%256);
}
SetData(data);
```

The sample script generates data which increases in size on every call. The data gets filled with values 1 - 255.

5.2.7 GDB

GDB is an advanced debugger supporting multiple platforms and it is a key-component of the *IMF*. As shown in figure 5.3 *GDB* serves two requirements, the symbol table and the debugging connector. The symbol table is used to resolve named variables, breakpoint specifications involving source files and many others. The fuzzer is also functional without a symbol table and with a binary file without attached symbols, but symbol tables significantly ease the fuzzing process.

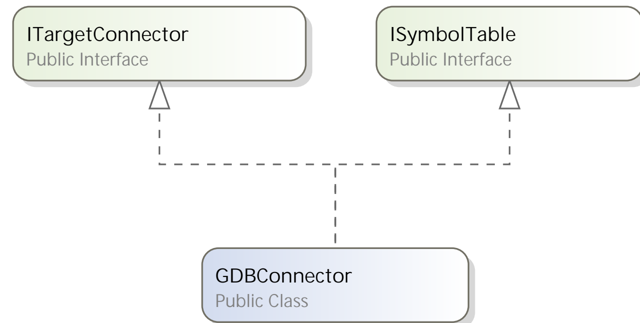

Figure 5.3: Basic class diagram of the GDBConnector

The connector is linked to a *GDB* sub process and communicates with it by parsing command line output, the same way a developer would use *GDB*.

5.2.7.1 Process Snapshots and Restoration

The whole fuzzer is based on a component that has a snapshot and restore feature for processes. After heavy research it was discovered, that *GDB* provides two features that can be used to achieve this requirement:

Newer versions of *GDB* have a *checkpoint* capability. This feature enables the tester to create a *checkpoint* and to restore it later. Internally the checkpoint creation works by forking an exact copy of the original process including randomized address spaces. Creating a checkpoint looks as shown in listing 5.10.

Listing 5.10: GDB checkpoints

```
(gdb) info checkpoint
  1 process 11004 at 0x40065c, file gdb_checkpoint_debugging_test
     .c, line 24
* 0 process 11003 (main process) at 0x40066d, file
   gdb_checkpoint_debugging_test.c, line 29
```

The checkpoint command creates two different system-processes and *GDB* switches between the processes. This approach is great for local debugging, but has the

disadvantage that it is not supported by remote *GDB* sessions using *GDBserver*. However this approach is not of any use for the *IMF*.

Another feature that has been introduced to *GDB* recently is called *reverse debugging*. Using this feature *GDB* records every single CPU instruction and its effective side-effects. If the user wants to restore a former state of the application he can go back step-by-step.

Internally *GDB* needs to be aware of all available instructions for the current CPU architecture and all its side effects. The drawback is, that this is a heavily architecture dependent feature but *GDB* already implements this feature for various architectures including: linux_x86, linux_x86-64, linux_arm (pre alpha) and some others.

Figure 5.4 shows a single recorded `mov` instruction. The side-effect of the `mov` operation is that the value of the target register gets overwritten. The recording is done before the instruction is executed and yields a *ReplayRecord* which contains a list of changed memory regions and registers with its values before the instruction is executed. The *ReplayRecords* are created for every single instruction and are linked together to form a list of changes. If the tester wants to go back to a former process state the changes are applied in reverse order as recorded in the *ReplayRecords* beginning at the last. So if the *mov* operation needs to be undone, the recorded value for `ax` is written to `ax`, the program counter is set correctly and the process state is restored.

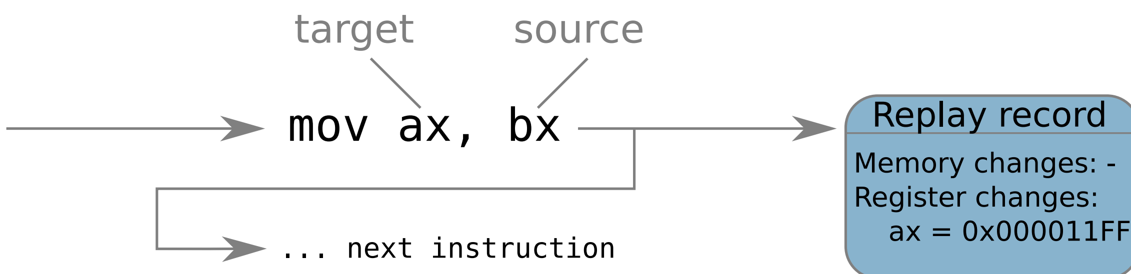

Figure 5.4: Single instruction recorded by the GDB reverse debugging feature

Another advantage of this approach is that the *ReplayRecords* can be exported and read by an analyser discussed in 5.3 to discover various overflows and other vulnerabilities.

5.2.7.2 The Core-Dump

As discussed, *GDB* is able to export the *ReplayRecords*. This is done by performing a core dump. The core dump is divided in various segments where the current state of the process is saved. One of these sections is dedicated to save the *ReplayRecords*. To be able to read core dump files using C#, the wrapper library *libbfd.net* has been developed around the native *libbfd* library. The class hierarchy is shown in figure 5.5. The heart of the wrapper library is the *GDBCoreDump* class, it uses the *BfdStream* to select a particular segment in the core dump file and interprets its content as *GDBProcessRecordSection* which contains all register and memory changes grouped by the associated instruction, using the native wrapper. The analysers described in 5.1 and 5.3 make heavy use of this feature to detect overflows and unauthorized data access.

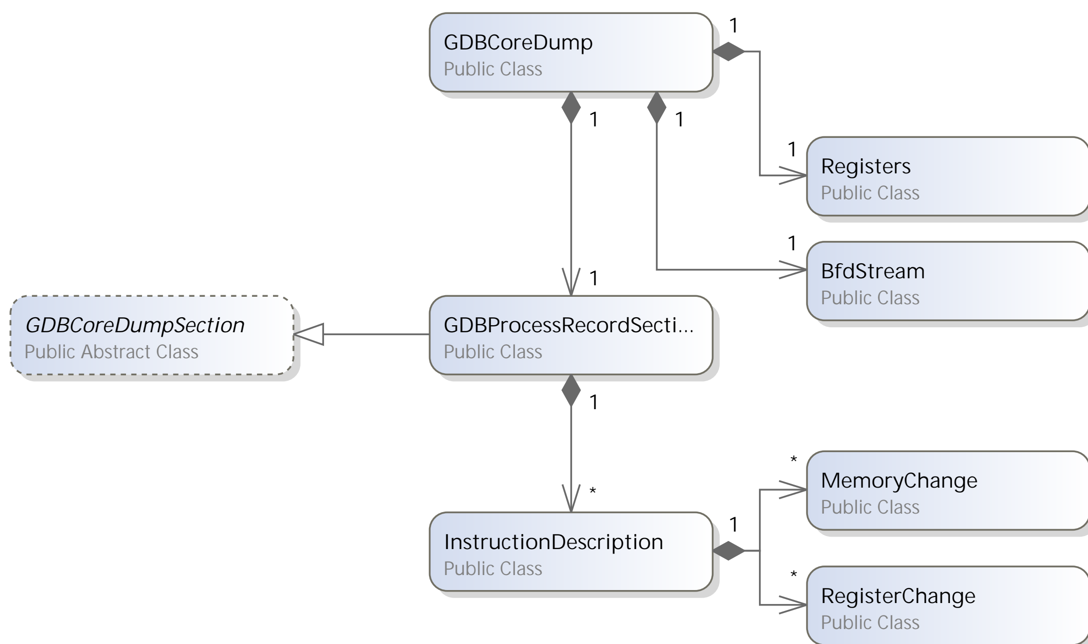

Figure 5.5: libbfd.net class hierarchy

5.3 The Analyser Engine

The *Analyser Engine* has already been introduced in 5.1. This chapter will focus on its configuration and internals.

5.3.1 Configuration

The configuration is similar to the configuration of the *Fuzzer Engine* but not that large. An example configuration is shown in listing A.2.

The *Include* tags have the same functionality as for the *Fuzzer Engine*. The included paths can be referenced in the rest of the configuration file. The next two values (*RegisterTypeResolver* and *RegisterFile*) haven't appeared so far and are architecture dependent. Because every architecture has different register names and register numberings for registers that have the same function, the *RegisterTypeResolver* has the ability to resolve target independent register names (e.g. *ProgramCounter*) to target specific register names (e.g. *eip* or *rip*). The *RegisterFile* in contrast contains all available registers for a specific architecture in increasing order. This information is needed by the analysers because some of them analyse *ReplayRecords* and reference the registers only by their numbers.

The rest of the configuration is straight forward and adds specific analysers to the engine, which are executed in the same order as they are specified when running the analyser.

5.3.2 Analysers

Program error analyser(`analyzers/program_error`) - Analyses the log-files for obvious program-crashes and also writes some additional informations like: stop reason, exit status and exit address to the resulting log-file. The information is read from the *errorlog* files.

Saved register analyser(`analyzers/saved_registers`) - As shown in figure 2.6 registers, return addresses and function parameters are saved on the stack. This analyser uses the core dump file to find memory writes which overwrite values on stack (e.g. return address or saved *ebp*) using the *stack frame info* file.

Heap overflow analyser(`analyzers/simple_heap_overflow`) - This analyser uses the core dump file and all recorded dynamic memory allocations (if available) to find simple heap overflows. Currently it is only possible to find heap overflows where the write operation goes out of bound and the target area is not associated with another buffer or the write operation goes to unallocated memory, resulting in write access errors. If a buffer is overflowed and the write operation goes to another allocated buffer the heap overflow is not detected because from the analysers point of view it looks like a valid write operation.

Memory zone analyser(`analyzers/memory_zones`) - To be sure that the target application does not write in a specific, very critical memory section, a tester can define so called *red-zones*. Every write-access to *red-zones* gets logged in the error-log. This analyser scans through the core dump and logs all memory writes that go to *red-zones*. On the other hand it provides so called white-zones, where only memory write operations in the specified zone are allowed.

5.4 Summary

During the research of this thesis an *In-Memory Fuzzer* (further referenced as *IMF*) was created. It is the implementation of a *Snapshot Restoration Mutation* based fuzzer. Generally, the architecture is split into two parts: the fuzzer engine and the analyser engine. The fuzzer engine collects data by fuzzing the target device or target application and the analyser engine inspects the generated data afterwards. This approach has the advantage that the analysers can be adopted afterwards without rerunning the fuzzing process. The objectives of this fuzzer are to fulfil all the requirements defined in 4.3. To implement this feature set from scratch an enormous effort is required, therefore *IMF* heavily reuses existing components without compromising the usability or feature set. The current implementation builds on *GDB* and its capabilities. As discussed in chapter 4 the *Snapshot Restoration Mutation* method requires a mechanism to create a process snapshot and restore it afterwards. This is achieved by using *GDB's* reverse debugging feature. This feature records all executed CPU instructions and its side effects. To restore a process' state the recorded instructions are executed in the reverse order. At the end of every fuzzer run the recorded instructions are saved to disk (along with other log output) and can be used later in the analysis phase.

The following list shows all requirements and how they are fulfilled with the created implementation:

- *Modularity and expandability:* This requirement is covered by various implementation aspects. *IMF* uses XML configuration files where the implementation for every single component can be exchanged. This goes hand in hand with the fact that *IMF* builds on C# using the .NET/MONO framework, therefore it requires zero effort to include new libraries and exchange existing components with custom implementations. For the data generation component this approach even goes a step further. For the purpose of creating custom data generators a simple scripting engine has been created which dynamically

compiles C# style code. So, the data generator logic can be directly written in the configuration file (or also in external XML files).

- *Portability:* The fuzzer logic is entirely written using the C# programming language and therefore is available for all operating systems where the .NET- or MONO framework is available.

- *Flexibility* and *Small in size:* *GDB* can basically operate in two modes: a local mode and a remote mode. For embedded devices with only little memory it is possible to only run a small stub (*GDBserver*) and a remote control application on the device. The stub only receives commands and executes them, all processing is done on the host device. If the device has enough resources it can also run the full-blown fuzzer setup locally.

- *Speed:* The speed of the fuzzing process heavily depends on the hardware under test. Remote fuzzing setups are slower than local fuzzing setups, but no general prediction concerning fuzzing speed is possible.

- *Usability:* Depending on the available information for the application under test the tester needs moderate reverse engineering skills. To ease this process external tools like *IDA* or *OllyDbg* can always be integrated in the fuzzing workflow seamlessly.

- *Improve results with additional information: IMF* or more precisely *GDB* has the capability to load external symbol tables for a release-build-binary without any debugging symbols attached. External tools can be used to regenerate as much information from the binary or other sources as possible and can be integrated in the fuzzing process to ease the hook point detection or argument discovery.

Chapter 6

Results

6.1 Application Tests

This chapter contains fuzzing test-results for four applications with known vulnerabilities. The selection criteria for the test applications are as follows: A test application should be runnable on embedded devices and should contain known vulnerabilities. The existence of vulnerabilities can be determined by using various security related websites, e.g. *CVE - Common Vulnerabilities and Exposures[3]* The errors are reconstructed using *IMF* to illustrate the functioning of the toolkit.

6.1.1 HTEditor

- **URL:** `http://hte.sourceforge.net`

  HTEditor is an editor/viewer/analyser application. Various security related websites claim that this application contains a stack-based buffer overflow related to command-line arguments. The application was fuzzed using multiple data-generator patterns but the fuzzer was not able to monitor any unexpected behaviour on various architectures.

- **Test setup:** *HTEditor* uses a local only setup as shown in listing B.1. The application is executed directly by the GDB sub process. It fuzzes the first command line argument `argv[1]` with randomly generated data and increasing length starting at 3000 bytes. The length gets extended by 1000 bytes on every fuzzing round.

- **Fuzzed parameter:** argv[1]

- **Fuzzing rounds:** 1000

- **Data length:** 3000 - 1000000 Bytes

6.1.2 CoreHttp

- **URL:** `http://corehttp.sourceforge.net/`

  CoreHttp is a lightweight web server used on embedded systems.

- **Fuzzed parameter:** Incoming request buffer, which is directly taken from the HTTP-request.

- **Test setup:** *CoreHttp* uses a remote control setup as shown in listing B.2. The application gets executed by the remote control application on the remote side. The fuzzer connects to the remote side using the *GDBserver*. The fuzz target is the http-server's receive buffer `parentsprock->buffer`. This buffer requires special handling and therefore can not be resolved using the `variable` fuzz target specifier. It uses the *addressOf* (`&`) operator to resolve the address of the buffer. The data generator again produces random, printable and null terminated data starting at a length of one byte up to a length of 2000 bytes. The size gets increased by 100 bytes on every fuzzing round.

- **Fuzzing rounds:** 21 (cancelled due to incoming data length restrictions)

- **Data length:** 1 - 2000 Byte, increasing data-length

The test-results are shown in table 6.1. The data shows that saved registers can be overwritten, therefore a classical stack-smashing attack can be performed.

Round	Date-length	Result
1 - 6	0 - 500 Byte	No analyser detected any issues
7 - 8	600 - 700 Byte	**Stack frame analyser:** Saved registers got overwritten, stack based buffer overflow.
9 - 21	800 - 2000 Byte	**Program error analyser:** A segmentation fault was detected **Stack frame analyser:** Saved registers got overwritten, stack based buffer overflow.

Table 6.1: Test results for CoreHttp

6.1.3 3Proxy

- **URL:** `http://www.3proxy.ru/`

  3Proxy is a proxy server, which is also suited for embedded devices.

For the first fuzzing attempt the following parameters where selected:

- **Test setup:** *3Proxy* again uses a remote control setup as shown in listing B.3. The fuzz target is the request-line buffer for the proxy server. The first fuzzing attempt does not clearly show the source of the error, therefore another attempt was performed with a modified fuzz-region as shown in listing B.4
- **Fuzzed parameter:** Incoming request buffer from the receive method
- **Fuzzing rounds:** 60
- **Data length:** 1 - 6000 Byte, increasing data-length

The test results are shown in table 6.2. Fuzzing round 21-56 result in a segmentation fault, this can have various causes. Either a pointer in the method-under-test gets overwritten and dereferenced later on, or a sub method produces the error.

Round	Date-length	Result
1 - 21	0 - 2000 Byte	No analyser detected any issues
22 - 56	2000 - 5600 Byte	**Program error analyser:** A segmentation fault was detected
57 - 60	5700 - 6000 Byte	**Program error analyser:** A segmentation fault was detected **Stack frame analyser:** Saved registers got overwritten, stack based buffer overflow. (rbp, rbx, r12, rip)

Table 6.2: Test results for 3Proxy, first attempt

Using *GDB* in interactive mode shows that a sub method produces this error, therefore the fuzzing configuration is adapted and a second fuzzing attempt, using the inner method, is performed with the following parameters:

- **fuzzed parameter:** Incoming request buffer from the inner method.
- **fuzzing rounds:** 77
- **data length:** 1 - 7700 Byte, increasing data-length

The results are shown in table 6.3. They show that starting with round 22 (as it was for the previous attempt) saved registers get overwritten, which enables a stack smashing attack again.

Round	Date-length	Result
1 - 21	0 - 2000 Byte	No analyser detected any issues
22 - 77	2000 - 7700 Byte	**Stack frame analyser:** Saved registers got overwritten, stack based buffer overflow. (rbp, rip)

Table 6.3: Test results for 3Proxy, second attempt

6.1.4 ProFTPD

URL: `http://www.proftpd.org/`

ProFTPD is a highly configurable FTP server available on various platforms. Security related websites report some vulnerabilities concerning the 'Controls' module of *ProFTPD*[5]. The region to fuzz is entered once a Unix socket connects to the module. A socket cannot be handed over from one application to another, therefore the fuzzer needs a way to trigger the module using a Unix socket and combine the in-memory fuzzing capabilities with the Unix socket fuzzing capabilities. Analysing the configuration file in listing B.5 reveals the following advanced structures:

- *Predefined fuzzers:* The configuration predefines fuzzers which can be used in multiple places later on. The predefinition is required because a single Unix socket cannot be opened multiple times, but needs to be fuzzed with multiple triggers.
- *Pre conditions:* The configuration defines two pre conditions or two actions that are executed before the fuzzing process is started but after the connection has been established: `fuzz_helper/delay -` delays the further fuzzer execution by the specified time. This gives the victim time to start-up and prepare for requests. `fuzzer/predefined -` uses the predefined fuzzer and specifies a custom data generator (included from an external file and shown in listing B.5). It triggers the hook-point, but the socket need to remain open or the current session is lost.
- *Multiple fuzzing channels:* In the fuzz description area of the configuration two different fuzz locations with different types of fuzzers are defined. They are invoked in every fuzzing round (according to the trigger value) in the same order as they are specified in the configuration file.

Test results: The results of this test are that the fuzzed argument `reqarglen` is controlled by the attacker. A standard stack smashing attack can be performed where the fixed sized request buffer gets overflowed with attacker supplied data.

6.2 Performance Test

Another important aspect besides the functionality is the performance aspect. Because there is not a single suite with the same functionality, no inter-framework comparison can be made. But because the *GDBConnector* supports multiple modes they can be compared, and as shown in this section, this comparison yields interesting results.

As discussed in section 5.2.1 the *GDBConnector* supports a local and a remote mode. It is obvious that the remote mode is slower, but as a result of how the reverse debugging feature is implemented the performance values are worse than expected as shown in table 6.4.

	GDBserver		*Local GDB*	
	Time	**Time per instruction**	**Time**	**Time per instruction**
Program run	50s	0.78ms/Instr.	2.2s	0.034ms/Instr.
Snapshot restoration	18s	0.28ms/Instr	2.0s	0.031ms/Instr.

Table 6.4: Performance results

The results are based on a region with about *64000* instructions and show that using the *GDBserver* mode is about 16-times slower than using *GDB* in local mode. This leaves much room for improvement as discussed in chapter 7.

6.3 Known Issues

Beside the fact that using the remote mode is significantly slower than using the local mode as described in section 6.2 some other issues exist.

Using *GDBs* reverse debugging feature to implement a snapshot/restore capability has the drawback that *GDBs* memory usage grows with every executed instruction. Therefore *GDB* sets a hard limit for executed instructions with a default value of 200000. This limit can be changed by the tester but just the existence of this limit tells us that this *GDB* feature is not suitable to test whole program flows or large methods. Another problem concerning the limit for executed instructions are loops. If a loop runs long enough, every limit is reached.

Currently a huge effort is required for applications where no debugging symbols or source code are available, to isolate regions of interest and to define fuzz-targets.

The Corelan team[29] has built a simple trace application, to follow the users input through the complete application. To ease the process of identifying the regions of interest, an enhanced version of this *Tracer* is required.

6.4 Summary

The results show the trend that most exploitable vulnerabilities are stack-based buffer overflows. Heap-based buffer overflows are hard to detect in practice and require a more in-depth analysis of the memory allocation and memory write operations. If the problem of limited maximum executed instructions, and the problem of finding errors (preferably buffer overflows) in sub-methods, as discussed in chapter 7, was solved, then it would even be possible to test whole program flows. Currently the test area limits to a relatively small code area but especially the *ProFTPD* test shows the flexibility and capabilities of the *IMF* framework.

Chapter 7

Conclusion and Future Work

This thesis reviews common testing methods and groups them by the information that is required to apply the test-method. The following groups were introduced: *White-Box-*, *Grey-Box-* and *Black-Box-Testing*. Each testing method is a assigned to one of these groups by their required knowledge of the program-internals (e.g. method boundaries, method parameters, layout of data structures, stack layout,...). *White-Box-Testing* methods have all program internals available including access to the source code. *Black-Box-Testing* methods in contrast have only the information they can observe and *Gray-Box-Testing* methods additionally use reverse engineering tools and externally supplied information to gather as many program internals as possible. Most of the methods, regardless of which group they are assigned to, do not test the functionality of the application as a whole, they only test small parts or modules. Only the automated testing approach tests the application from a users point of view but this method is not applicable because it requires too much time and too many resources.

After the discussion of currently available and well established testing methods this thesis focuses on the distinction of the terms *bug*, *vulnerability* and *exploit*. A bug is a software behaviour that was not intended by the developer but they do not necessarily compromise the system-security. One or more bugs form a vulnerability but the vulnerability is not dangerous as long as no exploit code exists. So, the real threats are vulnerabilities with existing exploit code. The bugs are categorised into three groups: *Off-by-One-Error*, *Buffer-Overflows* and *Integer-Overflows*. Most of the bugs are located in the *Buffer-Overflow* group, which also offers the greatest exploit flexibility. While the other groups may only enable other failures but are not directly exploitable, the *Buffer-Overflow* category may enable the attacker to execute arbitrary code. Buffer overflows can further be grouped into: *Stack-Based*

and *Heap-Based*. Stack-based buffer overflows provide a reliable entry point for arbitrary code because the stack layout is almost the same for all applications on a specific architecture. Heap-based buffer overflows, in contrast are hard to detect and hard to exploit because the heap structure is not fixed.

With all this knowledge another testing method called fuzzing or fuzz-testing is introduced. The main concept of fuzzing is to put a high load of data (malformed and valid data) on the target device or the target application and inspect the response. If the application or device responses unexpectedly the fuzzer records the input data for later verification and continues the fuzzing process. Afterwards the tester examines the recorded input data and tries to recreate the response. A deeper analysis of the behaviour may enable the tester to exploit the failure on the target application or device.

The main part of this thesis focuses on a different approach to fuzz-testing called *In-Memory Fuzzing*. Once approaching closed source or proprietary protocols and file formats, other fuzzers may require a lot of reverse engineering to rebuild the protocol- and file-structures. An in-memory fuzzer does not need to know the structure, it focuses only on the underlying code, the methods and especially its parameters. It hooks before and after relevant methods and changes the variables of the method-under-test before it is called, restores the original program state afterwards and tests the method again.

Two different implementation approaches were discussed: the *Mutation Loop Insertion* and the *Snapshot Restoration Mutation* method. When it comes to the implementation of one of the methods I discovered that the requirements for both methods were nearly the same. Both need access to the applications memory space and need to insert code or hook at specific hook-points. Debuggers fulfil all the requirements and are best suited to implement a custom in-memory fuzzer. None of the currently available frameworks can be used in production environments because all of the available fuzzers are either proof-of-concept implementations or never left pre-alpha stage. The best results were achieved using the *Corelan In-Memory Fuzzer*, but the implementation is based on the *PyDbg*-debugging framework, which is only available for Windows.

During the research of this thesis an *In-Memory Fuzzer* (further referenced as *IMF*) was created. It is the implementation of a *Snapshot Restoration Mutation* based fuzzer. The objectives of this fuzzer are to fulfil all the requirements: *Modularity and expandability*, *Portability*, *Flexibility*, *Small in size*, *Speed*, *Usability* and *Improve results with additional information*. Enormous effort is required to imple-

ment all components from scratch, therefore *IMF* reuses existing components and uses *GDB* as its debugger backend. The following list shows all requirements and how they are fulfilled with the created implementation:

- *Modularity and expandability:* *IMF* is built using C# and the .NET/MONO framework. This enables a highly modular design and requires zero additional effort to include new libraries. For configuration *IMF* uses XML configuration files where the implementation for every single component can be specified. For data generation *IMF* even goes a step further and provides a scripting engine where new data generator logic can be directly written in the XML configuration file.

- *Portability:* .NET framework is available for Windows operating systems and MONO is a highly portable framework and can be used on various architectures, therefore *IMF* can be used on all operating systems where the .NET or MONO framework is available.

- *Flexibility* and *Small in size:* Out of the box *IMF* is highly flexible concerning its footprint. This is accomplished by the integration of *GDB* which can basically operate in two modes: a local mode and a remote mode. With the local mode a full-fledged GDB and *IMF* installation is used on the target device which provides the highest possible speed. For devices with only little memory and resources the remote mode can be used where only a small stub (*GDBserver*) and a remote control application runs on the target device. The stub only receives commands from the fuzzer on the host device and executes them. If even more flexibility is required a tester always has the possibility to create a custom target connector and therefore has full control of the footprint on the target device.

- *Speed:* The speed of the fuzzing process highly depends on the hardware under test and its available resources. The only general assumption that can be made here is that remote fuzzing setups are slower than local setups.

- *Usability:* One of the primary design goals of *IMF* is usability but, depending on the available information of the application under test, in-memory fuzzing is a highly complex testing method. Therefore at least moderate reverse engineering skills are required. External tools like *IDA* or *OllyDbg* can always be used to ease this process.

- *Improve results with additional information:* Often additional information for an application is provided in the form of source code, debugging symbols or any other documentation by the vendor or third-parties. *GDB* provides the capability to attach provided information to binary files out of the box. For other connectors where *GDB* is not involved *IMF* provides the capability to use this extra information.

Some applications with known vulnerabilities and suitable for embedded systems have been selected to test with *IMF*:

- *HTEditor:* An editor/viewer/analyser application.
- *CoreHttp:* A lightweight web server used on embedded systems.
- *3Proxy:* A proxy server, which is also suited for embedded systems.
- *ProFTPD:* A highly configurable FTP server available for various platforms.

The test cases show the flexibility of the *IMF* framework and the results confirm the assumption that most exploitable vulnerabilities are stack-based buffer overflows. This does not mean that heap-based buffer overflows do not exists but they are hard to detect in practice and require a more in-depth analysis.

The current implementation only supports a single target connector with multiple modes. Future enhancements might therefore include new connectors, for example to directly communicate with emulators like QEMU. Using GDBs reverse debugging feature to implement the snapshot/restore capability has the drawback that GDB's memory usage grows with every executed instruction. Therefore GDB sets a hard limit for executed instructions. This limit can be changed, but future work may take alternatives into account and may use the reverse debugging feature only for in-depth analysis of an application. Another drawback of the current implementation is that e.g. stack-based buffer overflows are only visible to the fuzzer if they occur in the outer stack-frame. If they occur in any sub method the application may crash but the fuzzer cannot determine the cause of the crash because it has no information on the inner stack-frame. Future enhancements may take this issue into account.

Appendix A

Sample Configuration Files

Listing A.1: Basic sample fuzzer configuration file

```
<Fuzz>
  <Include>config_paths.xml</Include>
  <TargetConnection>
    <Connector>general/gdb</Connector>
    <Config key="gdb_exec">{[gdb_exec]}</Config>
    <Config key="gdb_log">stream:stderr</Config>
    <Config key="gdb_max_instructions">4000000</Config>
    <Config key="target">run_local</Config>
    <Config key="file">{[test_source_root]}corehttp</Config>
    <Config key="run_args">{[test_source_root]}/../chttp.conf</Config>
  </TargetConnection>
  <FuzzDescription>
    <RegionStart>source|http.c,19</RegionStart>
    <RegionEnd>source|http.c,83</RegionEnd>
    <FuzzLocation>
      <Trigger>start</Trigger>
      <StopCondition>count|1000</StopCondition>
      <FuzzerType>fuzzer/in_memory</FuzzerType>
      <FuzzerArg name="data_region">cstyle_reference_operator|parentsprock->
          buffer</FuzzerArg>
      <FuzzerArg name="data_type">fuzzdescription/single_value</FuzzerArg>
      <DataGenerator>datagen/random_bytes</DataGenerator>
      <DataGenArg name="minlen">1</DataGenArg>
      <DataGenArg name="maxlen">2000</DataGenArg>
      <DataGenArg name="lentype">increase|100</DataGenArg>
      <DataGenArg name="type">PrintableASCIINullTerminated</DataGenArg>
    </FuzzLocation>
  </FuzzDescription>
  <Logger>
    <Destination>{[log_root]}</Destination>
    <UseLogger name="datagenlogger" />
    <UseLogger name="connectorlogger" />
    <UseLogger name="stackframelogger" />
    <UseLogger name="remotepipelogger">
      <PipeName>logmem_pipe</PipeName>
    </UseLogger>
  </Logger>
</Fuzz>
```

Listing A.2: Example analyser configuration

```
<Analyze>
  <Include>config_paths.xml</Include>

  <RegisterTypeResolver name="registertypes/x86_64" />
  <RegisterFile>{[register_root]}x86-64.registers</RegisterFile>
  <FuzzLogPath>{[log_root]}</FuzzLogPath>
  <ErrorLog>{[log_root]}errorlog.xml</ErrorLog>

  <AddAnalyzer name="analyzers/program_error" />
  <AddAnalyzer name="analyzers/saved_registers" />
  <AddAnalyzer name="analyzers/simple_heap_overflow" />

  <AddAnalyzer name="analyzers/memory_zones">
    <WhiteZone>0x602010:0x60201F</WhiteZone>
    <RedZone>0x600000:0x700000</RedZone>
  </AddAnalyzer>
</Analyze>
```

Listing A.3: Example config_paths.xml file

```
<Options>
  <Path name="gdbserver_exec">/opt/gdb/bin/gdbserver</Path>
  <Path name="gdb_exec">/opt/gdb/bin/gdb</Path>
  <Path name="log_root">./log/</Path>
  <Path name="test_source_root">./src/</Path>
  <Path name="register_root">../registers/</Path>
  <Path name="lib_logmem">../liblog_memory_allocations.so</Path>
</Options>
```

Appendix B

Test Setup Configuration

B.1 HTEditor

Listing B.1 shows the fuzz configuration used to produce the test-results in 6.1.1.

Listing B.1: HTEditor fuzzer configuration

```
<Fuzz>
  <Include>config_paths.xml</Include>
  <TargetConnection>
    <Connector>general/gdb</Connector>
    <Config key="gdb_exec">{[gdb_exec]}</Config>
    <Config key="gdb_log">stream:stderr</Config>
    <Config key="gdb_max_instructions">4000000</Config>
    <Config key="target">run_local</Config>

    <Config key="file">{[test_source_root]}ht</Config>
    <Config key="run_args">dummyarg</Config>
  </TargetConnection>


  <FuzzDescription>
    <RegionStart>source|main.cc,262</RegionStart>
    <RegionEnd>source|main.cc,380</RegionEnd>

    <FuzzLocation>
      <Trigger>start</Trigger>
      <StopCondition>count|1000</StopCondition>
      <FuzzerType>fuzzer/in_memory</FuzzerType>
      <FuzzerArg name="data_region">variable|argv[1]</FuzzerArg>
      <FuzzerArg name="data_type">fuzzdescription/pointer_value</FuzzerArg>

      <DataGenerator>datagen/random_bytes</DataGenerator>
      <DataGenArg name="minlen">3000</DataGenArg>
      <DataGenArg name="maxlen">1000000</DataGenArg>
      <DataGenArg name="lentype">increase|1000</DataGenArg>
      <DataGenArg name="type">PrintableASCIINullTerminated</DataGenArg>
    </FuzzLocation>
  </FuzzDescription>
```

```
    <Logger>
      <Destination>{[log_root]}</Destination>
      <UseLogger name="datagenlogger" />
      <UseLogger name="connectorlogger" />
      <UseLogger name="stackframelogger" />
      <UseLogger name="remotepipelogger">
        <PipeName>logmem_pipe</PipeName>
      </UseLogger>
    </Logger>
  </Fuzz>
```

B.2 CoreHttp

Listing B.2 shows the fuzz configuration used to produce the test-results in 6.1.2.

Listing B.2: CoreHttp fuzzer configuration

```
<Fuzz>
  <Include>config_paths.xml</Include>
 <RemoteControl>
    <Host>hostname</Host>
    <Port>8899</Port>

    <Exec trigger="immediate">
      <Cmd>{[gdbserver_exec]}</Cmd>
      <Arg>--wrapper</Arg>
      <Arg>env</Arg>
      <Arg>LD_PRELOAD={[lib_logmem]}</Arg>
      <Arg>LOG_MEM_PIPE=logmem_pipe</Arg>
      <Arg> --</Arg>
      <Arg>:1234</Arg>
      <Arg>{[test_source_root]}corehttp</Arg>
      <Arg>{[test_source_root]}/../chttp.conf</Arg>
    </Exec>
  </RemoteControl>

  <TargetConnection>
    <Connector>general/gdb</Connector>
    <Config key="gdb_exec">{[gdb_exec]}</Config>
    <Config key="gdb_log">stream:stderr</Config>
    <Config key="gdb_max_instructions">4000000</Config>
    <Config key="target">extended-remote hostname:1234</Config>
    <Config key="file">{[test_source_root]}corehttp</Config>
  </TargetConnection>


  <FuzzDescription>
    <RegionStart>source|http.c,19</RegionStart>
    <RegionEnd>source|http.c,83</RegionEnd>

    <FuzzLocation>
      <Trigger>start</Trigger>
      <StopCondition>count|1000</StopCondition>
      <FuzzerType>fuzzer/in_memory</FuzzerType>
      <FuzzerArg name="data_region">cstyle_reference_operator|parentsprock->
          buffer</FuzzerArg>
```

```
      <FuzzerArg name="data_type">fuzzdescription/single_value</FuzzerArg>

      <DataGenerator>datagen/random_bytes</DataGenerator>
      <DataGenArg name="minlen">1</DataGenArg>
      <DataGenArg name="maxlen">2000</DataGenArg>
      <DataGenArg name="lentype">increase|100</DataGenArg>
      <DataGenArg name="type">PrintableASCIINullTerminated</DataGenArg>
    </FuzzLocation>
  </FuzzDescription>

  <Logger>
    <Destination>{[log_root]}</Destination>
    <UseLogger name="datagenlogger" />
    <UseLogger name="connectorlogger" />
    <UseLogger name="stackframelogger" />
    <UseLogger name="remotepipelogger">
      <PipeName>logmem_pipe</PipeName>
    </UseLogger>
  </Logger>
</Fuzz>
```

B.3 3Proxy

Listing B.3 shows the fuzz configuration used to produce the test-results for the first attempt in 6.1.3.

Listing B.3: 3Proxy (first attempt) fuzzer configuration

```
<Fuzz>
  <Include>config_paths.xml</Include>
  <RemoteControl>
    <Host>hostname</Host>
    <Port>8899</Port>

    <Exec trigger="immediate">
      <Cmd>{[gdbserver_exec]}</Cmd>
      <Arg>--wrapper</Arg>
      <Arg>env</Arg>
      <Arg>LD_PRELOAD={[lib_logmem]}</Arg>
      <Arg>LOG_MEM_PIPE=logmem_pipe</Arg>
      <Arg> --</Arg>
      <Arg>:1234</Arg>
      <Arg>{[test_source_root]}3proxy</Arg>
      <Arg>{[test_source_root]}/../cfg/3proxy.cfg.sample</Arg>
    </Exec>
  </RemoteControl>-->

  <TargetConnection>
    <Connector>general/gdb</Connector>
    <Config key="gdb_exec">{[gdb_exec]}</Config>
    <Config key="gdb_log">stream:stderr</Config>
    <Config key="gdb_max_instructions">4000000</Config>
    <Config key="target">extended-remote hostname:1234</Config>
    <Config key="file">{[test_source_root]}3proxy</Config>
  </TargetConnection>
```

```
  <FuzzDescription>
    <RegionStart>source|proxy.c,873</RegionStart>
    <RegionEnd>source|proxy.c,878</RegionEnd>

    <FuzzLocation>
      <Trigger>start</Trigger>
      <StopCondition>count|100</StopCondition>
      <FuzzerType>fuzzer/in_memory</FuzzerType>
      <FuzzerArg name="data_region">variable|req</FuzzerArg>
      <FuzzerArg name="data_type">fuzzdescription/pointer_value</FuzzerArg>

      <DataGenerator>datagen/random_bytes</DataGenerator>
      <DataGenArg name="minlen">1</DataGenArg>
      <DataGenArg name="maxlen">20000</DataGenArg>
      <DataGenArg name="lentype">increase|100</DataGenArg>
      <DataGenArg name="type">PrintableASCIINullTerminated</DataGenArg>
    </FuzzLocation>
  </FuzzDescription>

  <Logger>
    <Destination>{[loge_root]}</Destination>
    <UseLogger name="datagenlogger" />
    <UseLogger name="connectorlogger" />
    <UseLogger name="stackframelogger" />
    <UseLogger name="remotepipelogger">
      <PipeName>logmem_pipe</PipeName>
    </UseLogger>
  </Logger>
</Fuzz>
```

For the second attempt only the hook points are adapted as shown in listing B.4.

Listing B.4: 3Proxy (second attempt) fuzzer configuration

```
.
.
  <FuzzDescription>
    <RegionStart>source|proxy.c,873</RegionStart>
    <RegionEnd>source|proxy.c,878</RegionEnd>
.
.
.
  </FuzzDescription>
.
.
```

B.4 ProFTPD

Listing B.5 shows the fuzz configuration used to produce the test-results shown in 6.1.4. Listing B.6 shows the associated data generation script.

Listing B.5: ProFTPD fuzzer configuration

```
<Fuzz>
  <Include>config_paths.xml</Include>
  <RemoteControl>
    <Host>hostname</Host>
    <Port>8899</Port>

    <Exec trigger="immediate">
      <Cmd>{[gdbserver_exec]}</Cmd>
      <Arg>--wrapper</Arg>
      <Arg>env</Arg>
      <Arg>LD_PRELOAD={[lib_logmem]}</Arg>
      <Arg>LOG_MEM_PIPE=logmem_pipe</Arg>
      <Arg> --</Arg>
      <Arg>:1234</Arg>
      <Arg>{[test_source_root]}proftpd</Arg>
      <Arg>-n</Arg>
      <Arg>-d 8</Arg>
    </Exec>
  </RemoteControl>

  <TargetConnection>
    <Connector>general/gdb</Connector>
    <Config key="gdb_exec">{[gdb_exec]}</Config>
    <Config key="gdb_log">stream:stderr</Config>
    <Config key="target">extended-remote hostname:1234</Config>

    <Config key="file">{[test_source_root]}proftpd</Config>
  </TargetConnection>


  <!-- Some Fuzzer types can only be instantiated once and need to be used as
      PreCondition AND as a
       FuzzLocation afterwards.
       Define the Fuzzers here, name them and use them as pre condition and fuzz
            location
    -->
  <DefineFuzzer>
    <Id>proftpd_socket</Id>
    <FuzzerType>fuzzer/unix_socket</FuzzerType>
    <FuzzerArg name="socket_file">/opt/proftpd-vulnerable/var/proftpd/proftpd.
        sock</FuzzerArg>
    <FuzzerArg name="enable_scripting">1</FuzzerArg>
    <FuzzerArg name="script_lang">CSharp</FuzzerArg>
    <FuzzerArg name="script_code">
      #ref Mono.Posix
      #import Mono.Unix
      #import System.IO
      #endheader
      if(HookType() == UnixSocketHookType.AfterSocketCreation)
      {
        //Associate the socket with a local file,
```

```
      //the server inspects the socket file and checks its permission

      string sockFile = "/tmp/tmp.sock";
      if(File.Exists(sockFile))
      {
        Console.WriteLine("sockfile: '{0}' already existing, unlinking",
            sockFile);
        File.Delete(sockFile);
      }

      Console.WriteLine("Binding unix socket to: '{0}'", sockFile);
      fuzzLocation.Connection.UnixSocket.Bind(new UnixEndPoint(sockFile));
      Console.WriteLine("Setting sock file '{0}' protection", sockFile);
      new UnixFileInfo(sockFile).Protection = Mono.Unix.Native.FilePermissions
          .S_IRWXU;
    }

  </FuzzerArg>
</DefineFuzzer>

<!-- Tries to send some data to the victim regardless of the position, to
    reach the
     start of the area of interest and the end. It can be seen as a test run.

     It has quite the same structure than a Fuzz Location, but the pre
         condition is only invoked once.

     Multiple PreCondition can be specified, they are invoked in the same
         order as they appear in the
     configuration file.
  -->
<PreCondition>
  <FuzzerType>fuzz_helper/delay</FuzzerType>
  <Delay>2000</Delay>
</PreCondition>
<PreCondition>
  <FuzzerType>fuzzer/predefined</FuzzerType>
  <FuzzerArg name="id">proftpd_socket</FuzzerArg>

  <DataGenerator>datagen/scripted</DataGenerator>
  <DataGenArg name="enable_scripting">1</DataGenArg>
  <DataGenArg name="script_lang">CSharp</DataGenArg>
  <DataGenArg name="script_file">script_datagen_mod_ctrls.cs</DataGenArg>
  <DataGenArg name="scriptval_generate_valid_data">1</DataGenArg>
</PreCondition>


<FuzzDescription>
  <RegionStart>source|ctrls.c,532</RegionStart>
  <RegionEnd>source|ctrls.c,545</RegionEnd>

  <FuzzLocation>
    <Trigger>start</Trigger>
    <StopCondition>none</StopCondition>
    <FuzzerType>fuzzer/in_memory</FuzzerType>
    <FuzzerArg name="data_region">variable|reqarglen</FuzzerArg>
    <FuzzerArg name="data_type">fuzzdescription/single_value</FuzzerArg>

    <DataGenerator>datagen/scripted</DataGenerator>
```

```
      <DataGenArg name="enable_scripting">1</DataGenArg>
      <DataGenArg name="script_lang">CSharp</DataGenArg>
      <DataGenArg name="script_code">
        #import System.IO
        #endheader
        int val = 100;
        if(IsValueSet("last_val"))
          val = (int)GetValue("last_val") + 10;
        SetValue("last_val", val);


        using(MemoryStream sink = new MemoryStream())
        {
          StreamHelper.WriteInt32(val, sink);    //status
          SetData(sink.ToArray());
        }
      </DataGenArg>
    </FuzzLocation>
    <FuzzLocation>
      <Trigger>start</Trigger>
      <StopCondition>none</StopCondition>
      <FuzzerType>fuzzer/predefined</FuzzerType>
      <FuzzerArg name="id">proftpd_socket</FuzzerArg>

      <DataGenerator>datagen/scripted</DataGenerator>
      <DataGenArg name="enable_scripting">1</DataGenArg>
      <DataGenArg name="script_lang">CSharp</DataGenArg>

      <!-- On process snapshot restoration the file descriptor is also restored
          to its previous position
        -->
      <DataGenArg name="script_code">
        SetData(new byte[10]);
      </DataGenArg>
    </FuzzLocation>

  </FuzzDescription>

  <Logger>
    <Destination>{[log_root]}</Destination>
    <UseLogger name="datagenlogger" />
    <UseLogger name="connectorlogger" />
    <UseLogger name="stackframelogger" />
    <UseLogger name="remotepipelogger">
      <PipeName>logmem_pipe</PipeName>
    </UseLogger>
  </Logger>
</Fuzz>
```

Listing B.6: ProFTPD data generation script

```
#import System.IO
#endheader

using(MemoryStream sink = new MemoryStream())
{
  StreamHelper.WriteInt32(0, sink);    //status
  StreamHelper.WriteUInt32(1, sink);   //number of arguments
```

```
    if(IsValueSet("scriptval_generate_valid_data") &&
       (string)GetValue("scriptval_generate_valid_data") == "1")
    {
      StreamHelper.WriteUInt32(100, sink); //data length

      //Write 100 bytes of data, no overflow
      byte[] data = new byte[100];
      sink.Write(data, 0, data.Length);
    }
    else
    {
      uint lastLength = 100;
      if( IsValueSet("last_length") )
        lastLength = (uint)GetValue("last_length");

      //increase the length of data
      lastLength += 50;
      SetValue("last_length", lastLength);

      byte[] data = new byte[lastLength]; //data
      Random r = new Random();
      r.NextBytes(data);

      StreamHelper.WriteUInt32(lastLength, sink);
      sink.Write(data, 0, data.Length);
    }


    SetData(sink.ToArray());
  }
```

List of Figures

1.1 White-Box testing scheme 4
1.2 Unit testing scheme 4
1.3 Black-Box testing scheme 7
1.4 Grey-Box testing scheme 8
1.5 Binary-auditing overview 9

2.1 From bugs to vulnerabilities and exploits 12
2.2 Off-by-One Error 12
2.3 General functional principal of a buffer overflow 13
2.4 Simplified virtual memory layout of current operating systems running on the x86-architecture 14
2.5 Stack layout 15
2.6 Smashed stack frame 17
2.7 Heap space allocation 18
2.8 Allocated memory blocks with headers and links 18
2.9 Overflowed heap layout 19
2.10 Two's complement vs. decimal representation 20

3.1 Fuzzing overview 27
3.2 Network Protocol Fuzzer overview 33

4.1 Basic In-Memory Fuzzing scheme 40
4.2 Mutation Loop Insertion scheme[27] 41
4.3 Snapshot Restoration Mutation scheme[27] 42
4.4 Corelan In-Memory Fuzzing Tracer workflow 46

5.1 Architectural overview of *IMF* 49
5.2 Fuzz-Description and Fuzz-Location class hierarchy 57
5.3 Basic class diagram of the GDBConnector 61

5.4 Single instruction recorded by the GDB reverse debugging feature . . 62
5.5 libbfd.net class hierarchy 63

List of Tables

2.1 Base-Pointer offsets[13] 16

6.1 Test results for CoreHttp 68
6.2 Test results for 3Proxy, first attempt 69
6.3 Test results for 3Proxy, second attempt 70
6.4 Performance results 71

Listings

1.1 Unit-test example 5
1.2 SQL-injection sample 6
2.1 Simple heap header[17] 18
2.2 Straight forward integer overflow example[21] 22
2.3 Integer overflow example 1 output[21] 23
2.4 Integer overflow example 2 output[21] 23
2.5 Integer overflow example 3 output[21] 23
3.1 Honggfuzz start command 28
3.2 Environment fuzzing test target application 29
3.3 Simple environment fuzzing 29
3.4 Use of ltrace to track environment variable accesses 29
3.5 Replacement library for *getenv[11]* 30
3.6 Call application with and without *LD_PRELOAD* 31
3.7 Example HTTP-Response 35
3.8 Example Request URL 36
3.9 Command Injection Vulnerability[27] 36
4.1 C-example demonstrating the drawback of the *Mutation Loop Insertion* method 40
4.2 Sample Tracer log output 46
5.1 Sample call of the *Support Library* 53
5.2 Output of the *Support Library* 53
5.3 Remote Control protocol specification 54
5.4 Remote Control *EXEC*-data structure 54
5.5 Remote Control *EXEC* response data structure 55
5.6 Remote Control *PROC* response data structure 55
5.7 Remote Control *PROC* response data structure 56
5.8 Remote Control integration 56
5.9 Remote Control integration 60

5.10 GDB checkpoints 61
A.1 Basic sample fuzzer configuration file 77
A.2 Example analyser configuration 78
A.3 Example config_paths.xml file 78
B.1 HTEditor fuzzer configuration 79
B.2 CoreHttp fuzzer configuration 80
B.3 3Proxy (first attempt) fuzzer configuration 81
B.4 3Proxy (second attempt) fuzzer configuration 82
B.5 ProFTPD fuzzer configuration 83
B.6 ProFTPD data generation script 85

Bibliography

[1] Boomerang. A general, open source, retargetable decompiler of machine code programs. Available online at: http://boomerang.sourceforge.net/.

[2] Codan. Available online at: http://wiki.eclipse.org/CDT/designs/StaticAnalysis.

[3] Common vulnerabilities and exposures. Available online at: http://cve.mitre.org/.

[4] Gdb: The gnu project debugger. Available online at: http://www.gnu.org/software/gdb/.

[5] Proftpd exploit. Available online at: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-6170.

[6] Pydbg. Available online at: http://code.google.com/p/paimei/.

[7] Sunit. The mother of all unit testing frameworks. Available online at: http://sunit.sourceforge.net/.

[8] Kent Beck. Simple smalltalk testing: With patterns, 1989. Available online at: http://www.xprogramming.com/testfram.htm.

[9] MITRE Corporation. *CVE - Apple/Iphone OS/Security Vulnerabilities Published In 2011.* Available online at: http://www.cvedetails.com/vulnerability-list/vendor_id-49/product_id-15556/year-2011/Apple-Iphone-Os.html.

[10] MITRE Corporation. *CVE - Google/Android/Security Vulnerabilities Published In 2011.* Available online at: http://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-19997/year-2011/Google-Android.html.

[11] Ludovic Courts. Inspecting getenv(3) calls. Available online at: `http://www.mail-archive.com/nix-dev@cs.uu.nl/msg02516.html`.

[12] Inc. Fortify Software. Rats. Available online at: `https://www.fortify.com/ssa-elements/threat-intelligence/rats.html`.

[13] Stephen J. Friedl. Intel x86 function-call conventions - assembly view. Available online at: `http://unixwiz.net/techtips/win32-callconv-asm.html`.

[14] Hex-Rays. Ida - the interactive disassembler. Available online at: `http://www.hex-rays.com/products/ida/index.shtml`.

[15] Immunity Inc. Spike. Available online at: `http://immunityinc.com/resources-freesoftware.shtml`.

[16] Ponemon Institute. Second annual cost of cyber crime study, 2011. Available online at: `http://www.arcsight.com/collateral/whitepapers/2011_Cost_of_Cyber_Crime_Study_August.pdf`.

[17] Felix Lindner. A heap of risk. Available online at: `http://www.h-online.com/security/features/A-Heap-of-Risk-747161.html`.

[18] Microsoft. Fxcop. Available online at: `http://msdn.microsoft.com/en-us/library/bb429476.aspx`.

[19] Microsoft. Virus alert about the win32/conficker worm, 2008. Available online at: `http://support.microsoft.com/kb/962007/en-us`.

[20] P. Oehlert. Violating assumptions with fuzzing. *Security Privacy, IEEE*, 3(2):58 – 62, march-april 2005.

[21] Phrack. Basic integer overflows. Available online at: `http://www.phrack.org/issues.html?issue=60&id=10`.

[22] Noam Rathaus and Gadi Evron. *Open Source Fuzzing Tools*. Syngress Publishing, 2007.

[23] Oleh Hex-Rays SA. Ida - interactive disassembler. Available online at: `http://www.hex-rays.com/products/ida/index.shtml`.

[24] Securiteam. Spikefile. Available online at: `http://packetstormsecurity.org/files/download/39625/SPIKEfile.tgz`.

[25] Michael Sutton and Adam Greene. The art of file format fuzzing. Available online at: `http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-sutton.pdf`.

[26] Michael Sutton, Adam Greene, and Pedram Amini. In memory fuzz poc. Available online at: `http://fuzzing.org/`.

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

[28] Robert Swiecki. Honggfuzz. Available online at: `http://code.google.com/p/honggfuzz/`.

[29] Corelan Team. Corelan inmemory fuzzing. Available online at: `http://www.corelan.be/index.php/2010/10/20/in-memory-fuzzing/`.

[30] Oleh Yuschuk. Ollydbg. Available online at: `http://www.ollydbg.de/`.