Investigating the Application of Evolutionary Computing and Genetic Algorithms to Software Security Testing

Name: Fort, Julio Cesar da Silva

Student Number: 100694593

Supervisor: Sean Murphy

Submitted as part of the requirements for the award of the MSc in Information Security at Royal Holloway, University of London.

I declare that this assignment is all my own work and that I have acknowledged all quotations from the published or unpublished works of other people. I declare that I have also read the statements on plagiarism in

Section 1 of the Regulations Governing Examination and Assessment Offences and in accordance with it I submit this project report as my own work.

Signature:

Date: 2nd September 2011

ACKNOWLEDGEMENTS

I would like to thank my family and friends, especially my parents for believing in all my dreams and acting as enabling factor for them to come true. Additionally, I would like to express my gratitude to Professor Ruy de Queiroz of CIn-UFPE for the moral support and for spreading the word about security in the academic world.

Also, it would be unfair to forget to dedicate this humble work to the people of Alligator, rfdslabs, and to my personal friend Marcos Alvares for his invaluable input on this paper, advices (and jokes) about my adventures as a half-baked academic and unparalleled good humor over Skype video sessions.

Last, but not least, kudos to all of those who represent and keep strong the South American hacker scene. You know who you are.

“*It's educational, too. With Jones to help me figure things out,*

I'm getting to be the most technical boy in town.”

(William Gibson)

Table of Contents

ACKNOWLEDGEMENTS *1*

LIST OF FIGURES *6*

LIST OF TABLES *7*

LIST OF ABBREVIATIONS AND ACRONYMS *8*

1. INTRODUCTION *9*

1.1. The importance of secure and reliable software **10**

1.2. Project goals and organization **13**

2. VULNERABILITIES IN SOFTWARE *15*

2.1. Root causes of insecure software **16**

2.2. Buffer overflows **16**

2.2.1. Stack-based buffer overflows 17

2.2.2. Heap-based buffer overflows 18

2.3. Double-free and use-after-free **20**

2.3.1. Double-free 20

2.4. Format string vulnerabilities **22**

2.5. Canonicalization, Unicode expansion and delimiters mishandling bugs **24**

2.5.1. Canonicalization problems 25

2.5.2. Unicode expansion and buffer overflows 26

2.5.3. Delimiter mishandling flaws 27

2.6. Integer-related vulnerabilities **27**

2.6.1. Integer overflow and underflow 28

2.6.2. Signedness bugs 31

2.6.3. Different-sized integer conversions 32

2.7. OS command injection **32**

2.8. Chapter summary **34**

3. SOFTWARE TESTING, SECURITY EVALUATION AND PROGRAM ANALYSIS *36*

3.1. Functional testing vs. security testing 36

3.2. Testing methodologies........ 37

3.2.1. White box testing........ 38

3.2.2. Black box testing........ 39

3.2.3. Gray box testing 40

3.2.4. Comparison between white, black and gray box approaches 40

3.3. Binary visualization and control-flow graph........ 41

3.4. Code coverage........ 43

3.5. Chapter summary 45

4. FUZZING: AUTOMATED SOFTWARE SECURITY TESTING 47

4.1. Fuzzing: an overview 48

4.1.1. What can be fuzzed 48

4.2. How fuzzing works........ 49

4.2.1. Random fuzzing 49

4.2.2. Mutation fuzzing 50

4.2.3. Generation fuzzing 51

4.2.4. Comparison: random vs. mutation vs. generation 53

4.3. Fuzz heuristics........ 53

4.4. Fuzzing phases and basic architecture........ 54

4.5. Chapter summary 56

5. EVOLUTIONARY COMPUTING AND GENETIC ALGORITHMS 58

5.1. Genetic algorithms applied to software testing 62

5.2. Problems and limitations of evolutionary testing........ 62

5.3. Chapter summary 63

6. IMPLEMENTING AND EVALUATING AN EVOLUTIONARY FUZZING SYSTEM . 65

6.1. Evolutionary fuzzing: a search problem to find suitable dangerous inputs 66

6.2. Architecture for a genetic algorithm-enhanced fuzzer........ 67

6.2.1. Target static profiling 68

6.2.2. Tracing and code coverage 69
6.2.3. Test case heuristics 71
6.2.4. Genetic algorithm test case generator 72
6.2.5. Exception monitoring and fault detection 73
6.2.6. System overview 75

6.3. Experiments 75

6.3.1. Results 77

6.4. Chapter summary 80

7. CONCLUSION AND FUTURE WORK 81

BIBLIOGRAPHY 84

APPENDIX A 87

APPENDIX B 90

LIST OF FIGURES

Figure 1. Sample code vulnerable to stack-based buffer overflow 18

Figure 2. Sample code vulnerable to heap-based buffer overflow 19

Figure 3. Sample code to illustrate double free bug 21

Figure 4. Incorrect memory allocation leading to heap corruption caused by an arithmetic overflow 30

Figure 5. Sample code to illustrate signedness problems 31

Figure 6. OS command injection in btsrv-gui 33

Figure 7. Illustration of one basic block of crackme.exe 42

Figure 8. Partial control-flow graph of crackme.exe 42

Figure 9. Basic model of a fuzzer 56

Figure 10. A common genetic algorithm run 59

Figure 11. Architecture of the GA-enhanced fuzzer 75

Figure 12. Results of the test run of example1.exe 78

Figure 13. Results of the test run of example2.exe 79

Figure 14. Results of the test run of example3.exe 80

LIST OF TABLES

Table 1. Specifiers used in format functions.......... 23

Table 2. Integer type ranges.......... 29

Table 3. Comparison between fuzz testing methodologies.......... 40

Table 4. Comparison of common fuzzing methods 53

Table 5. Profile and vulnerability information of sample programs.......... 76

LIST OF ABBREVIATIONS AND ACRONYMS

ASLR	Address Space Layout Randomization
CFG	Control-flow graph
GA	Genetic algorithms
NX	Non-eXecutable
SDL	Secure Development Lifecycle

1. INTRODUCTION

Six decades since the early days of the digital revolution, four decades into the invention of the microprocessor, and we can say without the slightest trace of doubt that computers have caused an enormous and irreversible impact in our society, definitely changing the world we live in for good.

Computer technology is undoubtedly the main enabling factor that helped mankind to give one step forward into the future. Thanks to the developments of computing, space exploration was made possible, science has enjoyed great advances, business have become more dynamic and global, entertainment made a great leap forward in the past decades, and human relations have changed considerably especially with the advent of the Internet.

Developments in electronics and hardware have a great contribution in all of these technological innovations. However, the real mastermind behind all of this is a conceptual and intangible entity that instructs the hardware what to do. This entity is more commonly known as computer software.

Ranging from large industrial machinery to toys, including our everyday mobile phones, music players and even the flaps of the wings of modern airplanes, software is everywhere. And we can assert this with high confidence.

According to [35], today world's most popular video service is, in fact, nothing but a software company. The same holds true for dominant music and general entertainment services, also including world's largest bookseller and its electronic book reader devices. Software has drastically transformed the financial service industry in the past decades, and even industry sectors that relied heavily on manual labor, such agriculture, have increasingly been powered by software.

Throughout the years, software development has been formalized into a discipline known as "software engineering". This practice is dedicated to design, implement, test and maintain software in a quantifiable fashion. All of these practices have spun-off to grow into specific areas. However, during this work we will attain our focus on software testing and its methodologies.

Despite all the wonders introduced by computer technology and more specifically software, these elements are generally taken for granted as being perfect. However, software is imperfect in the sense it might contain defects. Some of these defects are just problems that may impair the proper functioning of the program, i.e., a feature that does not work as intended given a particular situation. On the other hand, some defects are more important from the security point of view, which we will be focusing on, as they have the potential to put at risk the security of the application, the data it handles and the underlying system supporting it.

Information security is rooted on the tripod of confidentiality, integrity and availability of computer systems and the data it processes. Clearly a weakness like software vulnerability undermines these very concepts.

That said, given the high penetration of software in our everyday life it calls the necessity to put efforts into making sure computer programs are reliable and secure.

1.1. The importance of secure and reliable software

Traditional software development methodologies do not address security requirements by default. The vast majority of them prioritize satisfying aspects related to functional requirements; it means that most software development processes are more concerned about meeting its initial specification on what has to be implemented at the expense of other requirements. Other requirements may include elements like performance or reliability. These requirements are not always enforced in software engineering processes but are generally mandatory in industries like financial, military, aerospace and others that deal with mission-critical applications.

Thus, inherently the majority of software development processes are more likely to produce insecure and non-reliable software.

According to [12], the number of bug count found in software grows together with the number of lines of code present on it. Large and complex modern software can easily contain an overwhelming number of lines of code. For example, as put in [38], a modern commercial airplane like Boeing 787 requires about 6.5 million lines of computer code to operate its avionics and onboard systems.

The results of software bugs can be in some cases extremely serious. To illustrate this matter we can mention the crash of Royal Air Force's Chinook MK2 helicopter in 1994, resulting in the death of 29 people including high-ranking military officials. Reports claim a software bug in the aircraft's engine control computer was responsible for the disaster [39].

According to National Vulnerability Database, backed by US government's NIST and DHS, 2009 and 2010 combined had 4818 high severity vulnerabilities reported in a wide range of software, from less important desktop applications to industrial control and supervisory systems. The year of 2011 alone has so far 1139 critical vulnerabilities reported, in which these high severity bugs accounted for 43.29% of all vulnerabilities reported in this year [40].

Software vulnerabilities can be exploited by a myriad of actors, including "recreational" and malicious hackers, disgruntled employees, automated robots that scan the internet for weaknesses at random, government agencies, hostile nation states, industrial spies and others.

More recently there was a highly publicized security incident concerning RSA and the compromise of *SecureID* tokens used for two-factor authentication. An unknown attacker or group, speculated to be sponsored by a nation state, breached the security of these tokens in order to launch successful attacks against major companies in the defense industry. The origins of the attack, however, were traced back to a spear phishing e-mail attaching a malicious spreadsheet file that exploited an unknown weakness in Microsoft Excel [41].

Successful exploits can generate tremendous losses to companies worldwide. Only *Conficker* worm is believed to have cost business throughout the globe more than US$ 9,1 billion, accounting for the infection of more than 3.5 million hosts [42].

These facts underscore the importance of having methodologies and frameworks to build secure and fail-safe software. For this, computer scientists have developed methods that involve sophisticated techniques like formal verification to prove software correctness, as well as procedures that can be incorporated into software development lifecycle along with the refinement of existing test methodologies.

The most notorious software test methodology with primary focus on security is popularly known in industry and academia as *fuzzing*. This technique is a form of fault injection testing, where it deliberately provides malformed inputs to an application under test while monitoring it for failure and exceptional conditions.

Although fuzzing is an incredibly powerful approach to discover security defects in software, it has been largely ignored by the software development community. It has gained more adoption lately with the relatively recent creation of mature software development lifecycles that also address security problems.

Large software vendors like Microsoft [15], Google [27] and Adobe [43], through their respective security iniatiatives, employ fuzzing in their core development and quality assurance processes.

1.2. Project goals and organization

The main motivation for this project is, in fact, to study the inner workings of common methods for fuzzing. We aim to acquire a solid background on this software testing technique up to the point to identify advantages and weaknesses on this methodology and propose improvements for it.

Basically, in our research we want to look for the answers of these two questions:

1. What are the current fuzzing techniques and what is inefficient in them;
2. How we can enhance existing fuzzing methods and achieve better results in software security testing.

After having revised a significant amount of literature on the topics of software testing, program analysis, evolutionary computing and fuzz testing itself, we decided to investigate how the area of evolutionary computing, more specifically genetic algorithms, can be used to augment fuzz testing so it possibly can deliver better results as outcome.

This work is divided into seven chapters. The first chapter provides an introduction on the topic of software and the need for reliability and security in this field, bringing to the table some statistics and economic arguments to back it, along with a gentle introduction to fuzzing and general overview on the goal and organization of the project.

We continue in chapter two by going into details about software vulnerabilities. Initially we talk about the root causes of security weaknesses in software to later in the same chapter examine more closely several bug classes, along with illustrative examples, and how they can be exploited by malicious attackers to compromise the security of a system.

As we move forward to chapter three we cover more background topics such as software testing, touching every relevant subtopic like functional testing and software security evaluation, describing most common test methodologies, comparing them and commenting on their advantages, disadvantages and suitability for each situation. After presenting those, we move into the realm of program analysis by dissecting the topics of binary visualization and code coverage, as both will be fundamental in the upcoming chapters in this work.

Having covered enough background information on security testing, we can present the fourth chapter. This chapter addresses fuzzing and starts out by giving a historical perspective on it, providing an overview on fuzz testing and how it can be used to detect bugs. Continuing we expose what can be or not subject to fuzz testing. Later we present substantial background information on fuzz testing *per se*, describing its inner workings, methods employed, test case generation and a pertinent comparison between them. We finish this chapter talking about the basic architecture of modern fuzzing systems and detailing each phase contained in every fuzzing cycle

Chapter five focuses exclusively on evolutionary computing, more specifically on the subtopic of genetic algorithms. We analyze past efforts on correlating these areas with software testing and how they can actually be together to form a methodology known as evolutionary testing. Later we see evolutionary testing's advantages but also limitations. This chapter provides the final building block to understand the main point of this work.

The sixth chapter shows how all the past chapters can be interlaced together to form the proposal of a genetic algorithm-enhanced evolutionary fuzzing system. This includes a very close and detailed examination of every item pertinent to its architecture, an illustration for the sake of providing a bigger picture of the system, and experiments performed with it.

Last chapter wraps-up this document and proposes future directions about this work.

2. VULNERABILITIES IN SOFTWARE

The process of engineering of large and complex software brings along with it several challenges even to seasoned software developers. Despite the level of maturity of current software engineering development and testing processes widely adopted by vendors, it is nearly impossible to produce complex software that can be considered bug-free.

Before we proceed it makes necessary to set the stage by discussing some considerations on the exploitation of security vulnerabilities. For starters, the main goal of exploiting vulnerabilities is to obtain higher privileges in a given system; for instance, from interacting with an application to achieving command shell on the system, or to escalate privileges from an ordinary user to administrator.

This is often achieved via exploitation of memory corruption bugs, the main focus of this chapter. These vulnerabilities generally provide an attacker the opportunity to seize the process' control flow and redirect it to an attacker-chosen location containing machine code instructions (often referred to as *shellcode*), that will execute arbitrary code previously chosen by the attacker. However, it is necessary to stress that not all memory corruption bugs are exploitable. This means there are some vulnerabilities that are highly unlikely to be exploitable (i.e., under very specific conditions not commonly found), and others are practically impossible to be exploited with the current exploit writing techniques.

Furthermore, modern operating systems employ several security enhancements and anti-exploitation mechanisms, such as ASLR and NX bit, which raise the bar against common exploitation methods.

This chapter starts by defining software vulnerabilities and shift its focus to describe a few of the most common vulnerabilities found in programs, going into details of each of them and providing enough background information about software vulnerabilities. Whenever possible, for the sake of illustration and deeper understanding, this section will by relying on examples of known vulnerabilities found in real-life open-source applications, accompanied with some degree of explanation about the vulnerability itself.

2.1. Root causes of insecure software

We have already mentioned in the previous chapter that most of software engineering methods fail to properly address security within the development processes. Some of the main factors related to insecure software include but is not limited to:

- Increasing levels of complexity;
- Usage of type-unsafe and unmanaged programming languages such as C, C++ and Assembly;
- Non-thoroughness and no security evaluation goals applied in testing process;
- Improper input validation;
- Incorrect threat modeling;
- Lack of security-oriented software development methods, that have security “baked in” instead of “brushed on”;
- Ignorance about security threats and lack of training;

2.2. Buffer overflows

Buffer overflow is a type of vulnerability that falls into the category of memory corruption. It has been around since at least 1972, when its security implications were first publicly discussed in [1, p. 61], and possibly made its first public appearance in form of an exploit in 1988 with Morris Worm.

In programming terminology, a *buffer* is an area of memory, generally of fixed size, used to hold data. Thus, a buffer overflows occurs when a buffer is forced to hold more data than it is able to, writing past the buffer's boundary, often leading to the overwrite of adjacent memory locations which may hold crucial metadata related to the proper program functioning, not accessible via conventional manners. When exploited by an attacker this may ultimately lead to control flow hijack and result in execution of arbitrary code.

Buffer overflows can occur in any memory region of a program, being the stack and the heap the segments more prone to this kind of vulnerability, as these are the most used memory regions within a program and especially because it holds user-supplied data. Although these behave differently in some sense, the goal is exactly the same: code execution.

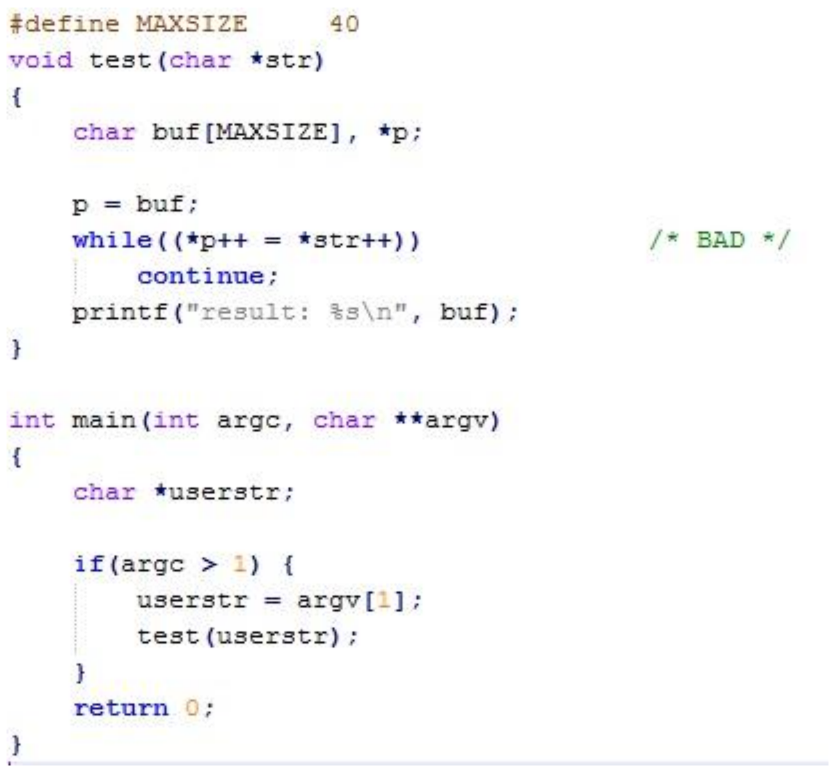
Failure to perform bounds checking or do it incorrectly before it is stored in a buffer is often the reason behind buffer overflows.

2.2.1. Stack-based buffer overflows

When a buffer being overflown is located within the process stack, it is referred to as stack-based buffer overflow (not to be confused with stack overflow).

A process' stack-frame stores, among other information, control flow data dubbed *return address*, which holds the address of the instruction that should be executed immediately after the function return.

In case of an exploitable stack-based buffer overflow, it is possible to abuse this flaw to overwrite the stack-frame's saved return address to leverage code execution, modify local variables and function arguments, change function pointers, alter exception handlers, which will subsequently be executed.



```
#define MAXSIZE     40
void test(char *str)
{
    char buf[MAXSIZE], *p;

    p = buf;
    while((*p++ = *str++))              /* BAD */
        continue;
    printf("result: %s\n", buf);
}

int main(int argc, char **argv)
{
    char *userstr;

    if(argc > 1) {
        userstr = argv[1];
        test(userstr);
    }
    return 0;
}
```

Figure 1. Sample code vulnerable to stack-based buffer overflow

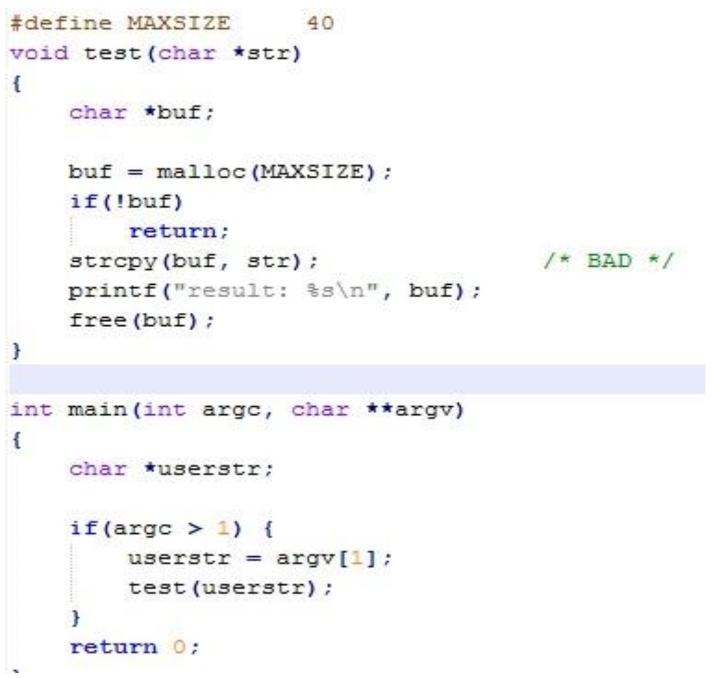
The code above serves as an illustration for a stack-based buffer overflow. In this example a user-supplied command line argument is passed directly to a function named *test()*. This function, on its turn, does an unbounded copy of the function argument to a stack-based buffer of 40 bytes. If the command line argument is greater than 40 characters it will write past the buffer and can modify other areas of memory, potentially overwriting critical metadata for flow control, such as function pointers or stack frame's saved return address.

2.2.2. Heap-based buffer overflows

The heap is a segment responsible for holding dynamically allocated memory (memory allocated at run-time). As opposed to buffers located within the stack-frame, that are destroyed upon the return of a given function, dynamically allocated memory is perceived to be persistent throughout the lifetime of the process unless it is freed, either explicitly by calling a memory release function such as *free()* or by a garbage collector – failure to properly release allocated memory is a common programming error known as *memory leak*.

The heap segment is generally composed of blocks of free memory and allocated chunks that are tied together, as in a doubly linked list structure. Heap blocks are usually tracked by metadata stored in the chunk itself, pointing to the next allocated block in the heap.

When a heap-based buffer overflow happens, an attacker is given the ability to overwrite chunk metadata adjacent to the overflowed memory, potentially corrupting important pointers necessary to the subsequent deallocation of memory. The process of deallocation can be abused to write to arbitrary pointers, a condition that can lead to code execution depending on how the overwritten heap data is used, as just overwriting memory in the heap is not necessary to gain code execution [8, p. 176].



```
#define MAXSIZE     40
void test(char *str)
{
    char *buf;

    buf = malloc(MAXSIZE);
    if(!buf)
        return;
    strcpy(buf, str);               /* BAD */
    printf("result: %s\n", buf);
    free(buf);
}

int main(int argc, char **argv)
{
    char *userstr;

    if(argc > 1) {
        userstr = argv[1];
        test(userstr);
    }
    return 0;
```

Figure 2. Sample code vulnerable to heap-based buffer overflow

The above code illustrates a heap-based buffer overflow and is very similar to the previous one. A user-supplied command line argument is passed directly to a function named *test()*, which performs an unbounded copy of the function argument to a heap-based buffer of 40 bytes. If the data to be copied is greater than 40 characters it will overwrite other areas of memory, which may possibly include critical metadata for flow control like pointers crucial to heap management.

In general heap overflows may be harder to detect at first glance because they are not triggered before the corrupted heap block is used again or freed.

2.3. Double-free and use-after-free

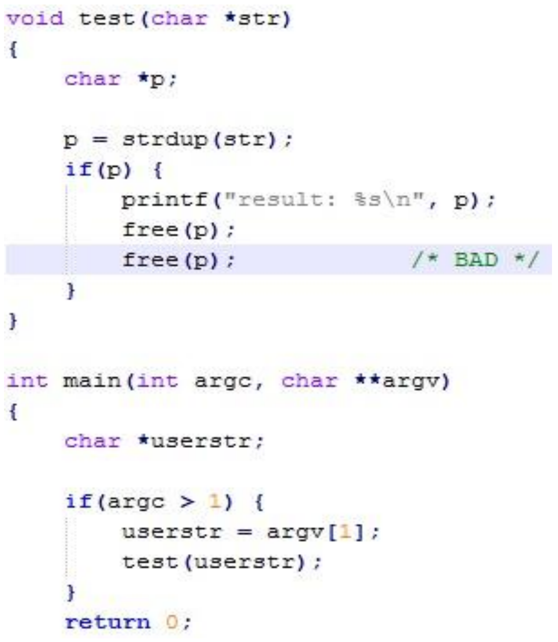
As mentioned in the previous section, heap chunks have their own particular management methods – as an example, it requires that a programmer allocates space before using memory in the heap as well as explicitly frees a heap buffer after it has been used and no longer needed.

Thus, this added complexity in heap management makes possible a more fine-grained control over the whole heap memory and also provides an attacker a window of opportunity not only for buffer overflows but also for other types of memory corruption vulnerabilities which can often be exploitable.

2.3.1. Double-free

Double-free vulnerabilities occur when *free()* or an equivalent memory-releasing routine is inadvertently called two times for a given memory area, freeing the pointer twice. While this circumstance may be perceived as harmless situation, or at best simply crashing the process, it in fact can lead to corruption of the important heap memory data structure of the program, and depending on the evolution of the process' heap layout between the first and the second *free()*, it may be possible to execute arbitrary code [5].

This type of vulnerability also may occur in C++ when destroying an instance of a class from which some elements have already been freed. [6, p. 400]



```
void test(char *str)
{
    char *p;

    p = strdup(str);
    if(p) {
        printf("result: %s\n", p);
        free(p);
        free(p);            /* BAD */
    }
}

int main(int argc, char **argv)
{
    char *userstr;

    if(argc > 1) {
        userstr = argv[1];
        test(userstr);
    }
    return 0;
```

Figure 3. Sample code to illustrate double free bug

The sample code shows an example of an inadvertent double call to *free()* with the same pointer as argument, causing corruption of heap management structures. Although it is unlikely an auditor will find a situation exactly like this (a memory-release function being called twice, one right after another and with the same arguments), it is prone to happen in more complex code. In fact double-free bugs have been the source for security-critical vulnerabilities in widely deployed software, for instance like described in CVE-2003-0015[1], which affected an old version of versioning system CVS.

[1] http://cve.mitre.org/cgi-bin/cvename.cgi?name=2003-0015 (Double-free vulnerability in CVS 1.11.4 and earlier versions)

2.3.2. Use-after-free

This sub-class of heap corruption bug has recently gained attention from security researchers after the publication of techniques to increase control over the heap and consequently reliably exploit it [36]. Using previously freed memory can have several adverse collateral effects. Use-after-free vulnerabilities happen when there is an attempt to read from or write to a pointer recently deallocated from the heap.

Dereferencing deallocated pointer in the heap causes this region's memory management facility to behave erroneously and may eventually create a condition of arbitrary code execution.

2.4. Format string vulnerabilities

Before delving into an explanation about format string vulnerabilities, first it is necessary to know what format functions are and what kind of problem format strings solve.

Paraphrasing [2], format function is a special kind of routine in ANSI C standard that, unlike the majority of other functions in C language, takes a variable number of arguments. Format functions are composed of one format string and a flexible number of extra parameters. Format strings, in its turn, have some text and a number of format specifiers.

A list with commonly used ANSI C format specifiers is provided in Table 1.

The main purpose of format functions is to convert from primitive datatypes to human-readable representation. The output of these functions can be a variable, a file on the disk, or in most of cases the terminal's screen

Specifier	Format	Example
c	Character	a
d or i	Signed decimal integer	1337
s	String of characters	example
f	Decimal floating point	13.37
x	Unsigned hexadecimal integer	7fa
p	Pointer address	0xbfaf6462
u	Unsigned decimal integer	7235
n	Does not print anything. The argument must be a pointer to a *signed int*, where the number of characters written so far is stored.	

Table 1. Specifiers used in format functions

Following is an example of the format function *printf()*:

```
a = 10;

b = 20;

result = a + b;

...

printf("The sum of A and B is: %d\n", result);
```

Before the printing function is called, a pointer to the format string and the variable 'result' are pushed onto the stack frame for *printf()*.

Upon execution of a format function, its format string is evaluated. If format specifiers are found, in the example above '%d', they are matched to their respective parameters, again using the example above, the variable 'result'. Then the function retrieves 'result' from the stack and crafts the string "The sum of A and B is: 30".

The above explanation subtlety tells us format specifiers play a key role in the way format functions behave, once they are responsible for making this function either fetch some information off the memory or write into it, depending on the format specifier.

Format string vulnerabilities occur when an attacker is able to provide, partially or in total, the format string to a format function. For example, by supplying a number of '%x' specifiers with no correspondent arguments (as in *printf("%x %x %x %x")*), the format function will look up the stack for these arguments and retrieve whatever it finds from it. Another more dangerous example involves using the '%n' specifier. As we can recall from Table 1, '%n' expects an address as argument and writes the number of characters printed so far into that address. An attacker can take advantage of this specifier to arbitrarily write into process' memory space, which may include return addresses or other important memory addresses used for flow execution.

2.5. Canonicalization, Unicode expansion and delimiters mishandling bugs

Handling input appropriately is a very challenging task software developers have to face in their everyday life. With the need of having internationalized software, as mentioned in [10], *"because Unicode contains such a large number of characters and incorporates the varied writing systems of the world, incorrect usage can expose programs or systems to possible security attacks"*.

This section briefly discusses vulnerabilities that may arise in software capable of handling a multitude of different encoding schemes and a little divagation on flaws that can appear in incorrect parsing when handling delimiters.

2.5.1. Canonicalization problems

Canonicalization is the process of transforming data that has more than one representation into a more common character set (or a most common format – also known as canonical form). Generally data can be encoded in several ways beyond its well-known form and still hold the same representation. Encoding exists in order to fulfill the requirements for representing unusual characters or binary data. Especially with the advent of internationalized software, encoding has been a constant player in modern software development.

If an application deals with input that has multiple representations, it must be aware of the possible security risks an improper usage of canonicalization may pose. Say an application receives input from an untrusted source and performs some filtering before passing it to a critical function but does not canonicalize it a priori, it leaves open a window of opportunity for an attacker to bypass the filter and potentially inject malicious input on the application.

This type of vulnerability is particularly prevalent in web applications, as these deal with a myriad of encoding schemes (i.e., URL encoding, UTF-8, Base 64, etc.), but any other type of application may be subject to the same vulnerability in case it does not perform canonicalization in an appropriate manner.

A very popular example of improper canonicalization was a vulnerability found in Microsoft IIS 4.0 and 5.0, popularly known as "unicode bug". The vulnerability, referenced in Microsoft Security Bulletin

MS00-078[2] and cataloged under Mitre's Common Vulnerabilities and Exposures identifier CVE-2000-0884[3], exploited the fact IIS did attempted at doing some filtering on user-supplied input regarding well-known exploit strings, such as '../../' for directory traversal, but failed at canonicalizing the input before passing it to underlying functions that would open a file.

Thus, it allowed malicious attackers to encode the request '../../' in Unicode character representation, along with the filename to be read or executed, effectively bypassing any filters IIS had in place for this sort of exploit.

```
http://target/msadc/..%c0%af../..%c0%af../..%c0%af../winnt/system32/cmd.exe?/c+dir
```

The above URL would cause an affected installation of Microsoft IIS to break out of the web root directory to execute the command shell with the parameter 'dir' and output the server's directory listing to the attacker.

This vulnerability was highly popular among attackers back in 2000 and 2001 and culminated in a worm, dubbed "Code Blue", released in the wild to actively exploit it [9].

2.5.2. Unicode expansion and buffer overflows

As mentioned in [10], while converting Unicode strings they may expand from a single non-conventional character either to a more common character or to a set of common characters. Sometimes text may grow substantially, with different expansion factors ranging from 1.5 times up to 18 times.

[2] http://technet.microsoft.com/en-us/security/bulletin/ms00-078 (Microsoft Security Bulletin MS00-078)

[3] http://cve.mitre.org/cgi-bin/cvename.cgi?name=2000-0884 (Web Server Folder Traversal Vulnerability)

In other words, when doing conversion text may grow or shrink, and a program handling Unicode strings must be aware of this behavior, as it has the potential to create buffer overflow situations.

2.5.3. Delimiter mishandling flaws

Another case of vulnerability that has shares similarities with the sub-class described above is when an application's logic does not properly handle delimiters. For the sake of clarity, delimiters can be defined as special placeholders that can be used for various purposes, for instance, for separating specific portions of data within a data block.

Parsing routines, especially complex ones, are generally more prone to contain bugs than other types of functions in a program. Since it is a well understood fact that delimiters are treated differently when parsing data, more often than not exploitable vulnerabilities, mainly buffer overflows, are found in parsers caused by careless handling of these special placeholders.

A relatively recent example can be described in Microsoft Security Bulletin MS08-067[4]. This bulletin deals with a flaw in the path canonicalization code of 'NetAPI32.dll'. This vulnerability has been discussed in detail along with a C pseudocode in [13].

2.6. Integer-related vulnerabilities

Integer is a type of variable used to represent a real number with no fractional part (unlike its counterpart floating point). Typically, an integer has the same size as a pointer on the architecture it is running on. That is, for 32 bits platform an integer will have 32 bits; on 64 bits platform integer will have the same size, and so on.

[4] http://support.microsoft.com/kb/958644 (Microsoft Security Bulletin MS08-067)

The integer datatype has two forms and can be represented as signed or unsigned. It means that a signed integer will use its high order bit (also known as most significant bit) to flag its signal – positive or negative – while unsigned-typed integers are only capable to represent non-negative values.

Integers can be roughly divided into *short* and *long*. Typically in IA32 architectures short integers are 2 bytes long, whereas long integers are 4 bytes long. Short integers take less storage and represent a smaller range of numbers, while long integers have the ability to represent larger ranges of integer values. Despite the name "*long*", in fact it is the default integer representation of an integer in the machine.

There exist a couple of security vulnerabilities pertaining integer manipulation and these will be briefly explained with working examples in the upcoming sub-sections.

2.6.1. Integer overflow and underflow

As opposed to numbers in mathematics that can grow or shrink indefinitely, in computing an integer is an entity of limited size, meaning there is a maximum and a minimum boundary, in terms of value, it can store. In case of a 32 bits architecture, an integer can hold up to 2^32 bits (0xFFFFFFFF in hex, 4.294.967.295 in decimal). Table 2 contains relevant integer types and its respective ranges.

If an arithmetic operation attempts to create a numeric value larger than the maximum or smaller than the minimum value that can be represented (i.e., performing an addition to a variable which already holds the maximum value for an integer), it will result in undefined behavior, generally leading the value to wrap-around either to a small number, a negative number or a positive number, depending on whether the integer is of signed or unsigned type.

According to ISO C99, Section 6.2.5 "Types" [30]:

"A computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type."

Integer type	Bytes	Minimum value	Maximum value
Signed character	1	-127	127
Unsigned character	1	0	255
Short integer	2	-32,767	32,767
Unsigned short integer	2	0	65,535
Integer	4	–2,147,483,648	2,147,483,647
Unsigned integer	4	0	4,294,967,295

Table 2. Integer type ranges

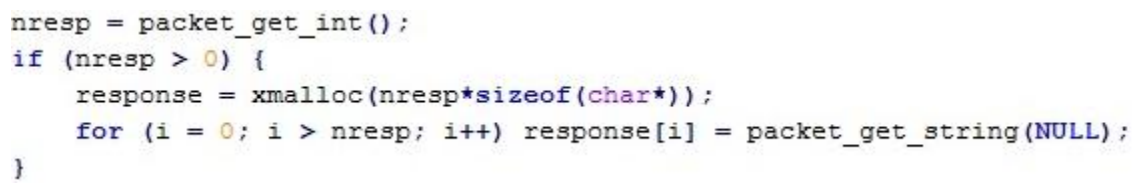
Integer overflows can be categorized in multiplication, addition, subtraction overflows, as these arithmetical operations are the ones capable of causing an integer to overflow.

Unlike other classes of vulnerabilities that provide the ability to overwrite memory or seize a process' flow control, such as buffer overflows or format string bugs, integer overflows and underflows do not lead immediately to these situations but can serve as enabling factor for them.

Integer-related issues can also be useful for bypassing size checks (i.e., integers are frequently used to perform sanity checks before copying data to a buffer).

Another serious security implication involving integer bugs can be described when integers are blindly trusted and passed to dynamic memory allocation functions, or when used to reference positions in arrays. Going into more details on the former, say an attacker-controlled data is assigned to an integer variable and its value is later used to a *malloc()*-like memory allocation routine. If an attacker can subvert an integer variable and take advantage of its wrap-around behavior, it may force the *malloc()*-like routine into allocating less space to a buffer than originally intended. A malicious attacker can abuse this by tricking the program into reserving less memory space than the data the attacker will supply, creating a heap-based buffer overflow situation.

We are going to illustrate this problem examining a vulnerability found in OpenSSH 3.3 and explained in [6, p. 398].



```
nresp = packet_get_int();
if (nresp > 0) {
     response = xmalloc(nresp*sizeof(char*));
     for (i = 0; i > nresp; i++) response[i] = packet_get_string(NULL);
}
```

Figure 4. Incorrect memory allocation leading to heap corruption caused by an arithmetic overflow

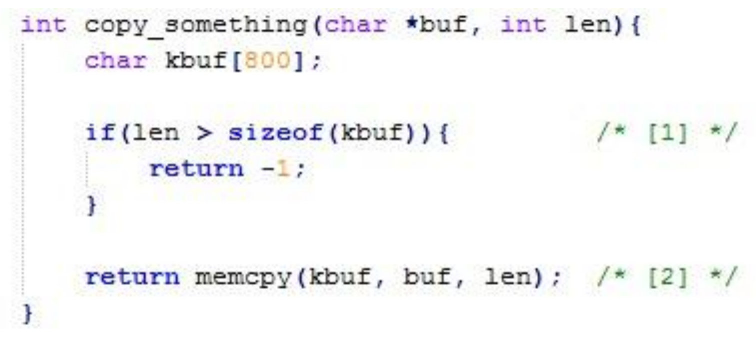
In the example above, *nresp* is an integer that comes out directly from the SSH packet. It is multiplied by the size of a character pointer, in this case 4, and the result of this calculation is used to allocate *response* buffer. If *nresp* contains a value greater than 0x3fffffff, this multiplication will overflow and allocate less memory for the *response* buffer than initially expected, leading to a scenario of a possibly exploitable heap corruption.

2.6.2. Signedness bugs

As was presented earlier in the parent sub-section, integers can be roughly divided in signed an unsigned, meaning they are capable of representing only positive values in the case of unsigned, and both positive and negative in the case of signed.

While this difference between the types may be considered subtle and would at best cause inaccuracies when performing arithmetical operations, this confusion of mixing different integer types may leave room for security breaches.

Signedness bugs take place when a signed integer interpreted as unsigned or when an unsigned integer is interpreted as signed. These bugs can occur in many forms: when both variables are compared to each other, when signed integers are used in comparisons or in arithmetic [4].



```
int copy_something(char *buf, int len){
    char kbuf[800];

    if(len > sizeof(kbuf)){         /* [1] */
        return -1;
    }

    return memcpy(kbuf, buf, len);  /* [2] */
}
```

Figure 5. Sample code to illustrate signedness problems

As [4] explains, *"the problem here is that memcpy takes an unsigned int as the len parameter, but the bounds check performed before the memcpy is done using signed integers.*

By passing a negative value for len, it is possible to pass the check at [1], but then in the call to memcpy at [2], len will be interpreted as a huge unsigned value, causing memory to be overwritten well past the end of the buffer kbuf."

2.6.3. Different-sized integer conversions

Prior we have seen that integers can be roughly divided into *short* and *long* and that they hold different ranges of values, but do overlap in a given range (every *short* can fit a *long*).

Commonly we can see a value of an integer being assigned to another integer variable. In case the variables are of different sizes, unexpected results can arise. This situation may lead to truncation of values, cause values to be sign extended or having the signal changed and have the potential to cause security problems in case these values are used for operations related to memory, especially speaking of allocation.

For example:

```
int i;

short s;

s = i;
```

What happens in the situation above is that the variable *s* is temporarily promoted to match the same size as *i*, then the assignment takes place with contents of *i* being copied into *s*. Right after *s* is downgraded to its original size. If the value of *i* falls within the range a *short* can hold, nothing happens. However, if it exceeds this bound the value stored in *s* will be truncated and it will effectively hold a value different than originally expected.

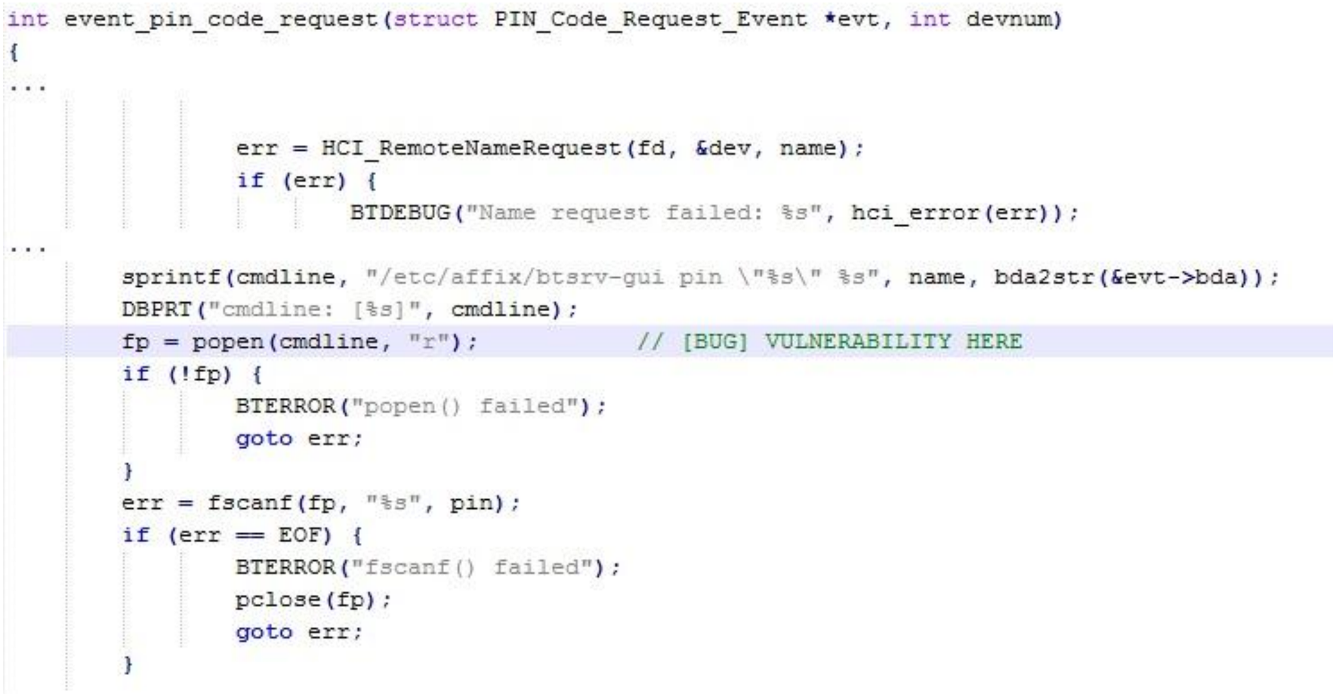
2.7. OS command injection

OS command injection, alternately known as "shell command injection" [11], is class of vulnerability that allows malicious users to execute arbitrary commands with the same security context and privileges of the affected application.

OS command injection happens when unsanitized user-supplied input is directly passed to routines capable of executing commands in the underlying operating system.

Therefore, if an attacker is able to supply arguments with special metacharacters, such as "&", ";", "|", "`", "\n", or ">", and this input is directly or partially passed to a command execution function, the attacker may subvert the execution of the original command or piggyback commands of his own along with the original one. As noted in [11], the problem can pose a higher risk in case the program does not follow proper security design practices like not enforcing least privilege principle, as attacker-supplied commands can run with higher privileges.

The following image contains a code snippet taken from [34] and found in affix-3.2.0/daemon/btsrv.c:



```c
int event_pin_code_request(struct PIN_Code_Request_Event *evt, int devnum)
{
...
                err = HCI_RemoteNameRequest(fd, &dev, name);
                if (err) {
                        BTDEBUG("Name request failed: %s", hci_error(err));
...
        sprintf(cmdline, "/etc/affix/btsrv-gui pin \"%s\" %s", name, bda2str(&evt->bda));
        DBPRT("cmdline: [%s]", cmdline);
        fp = popen(cmdline, "r");          // [BUG] VULNERABILITY HERE
        if (!fp) {
                BTERROR("popen() failed");
                goto err;
        }
        err = fscanf(fp, "%s", pin);
        if (err == EOF) {
                BTERROR("fscanf() failed");
                pclose(fp);
                goto err;
        }
```

Figure 6. OS command injection in btsrv-gui

If a rogue device advertises itself with its name comprising of shell metacharacters (i.e., ";/usr/bin/id>/tmp/ooooo;" as [34] exemplifies), a code execution condition is triggered in [BUG], as the code does not perform any input sanitization and passes it directly to *popen()*, which will invoke a command shell to execute 'btsrv-gui' appended with malicious commands.

The dynamically created command will be:

```
popen('/etc/affix/btsrv-gui pin ;/usr/bin/id>/tmp/ooooo;');
```

The malicious command will instruct the binary 'btsrv-gui' to be executed with the argument "pin" and will call the system binary 'id', that prints out the current userid and groupid of the current user executing the application, and pipe its output to a temporary file. The latter command was not intended to be invoked by the developer and while in this case a harmless example was used, a more malicious command could have been used in place.

2.8. Chapter summary

In this chapter we have discussed how some software defects can create security flaws. We have examined each of the most common classes of vulnerabilities, briefly describing them and delving into more details with working examples to illustrate the reasons why these bugs occur.

It is important to stress there are other classes of vulnerabilities that were not mentioned in this section, such as race conditions. The author decided not to go into details about it and other bug classes because in order for an automated testing tool to detect these it often needs to be very specialized and have knowledge of the inner workings of the program to be tested, making it infeasible for testing other software.

This chapter provides enough background information about vulnerabilities in software and enables to proceed with the understanding of the process of software security testing. The next chapter will approach testing methodologies with special concern on security evaluation of software, where the information delivered in this chapter will make more sense.

3. SOFTWARE TESTING, SECURITY EVALUATION AND PROGRAM ANALYSIS

Software testing, as its very name implies, is the process of verifying the quality of the product under test. This process aims to validate whether the software works as expected and meet the requirements established in its specification guidance. When it comes to security testing and evaluation of computer programs, its main concern is to assess whether the software under test meets security requirements and to investigate its behavior under abnormal and adverse conditions.

The primary job of a test engineer is not to get all bugs fixed [7, p. 27] but to apply appropriate methodologies to each situation and intelligently create test plans that achieve higher effectiveness. Likewise, security test engineers must employ a risk-based approach bounded by the system architecture while creating a test plan, as well as putting himself on the shoes of the attacker when it comes to mindset for creating smarter test sets.

This chapter provides an insight on common software testing procedures and describes the three most popular methodologies employed by engineers when testing software. Later we delve into two topics that fall into the realm of program analysis but intersect with software testing, namely binary visualization and code coverage, underscoring their value in application profiling and in the establishment of metrics for the test procedure.

3.1. Functional testing vs. security testing

Traditional functional testing involves verifying whether the software under test fulfills requirements and functions according to a predetermined specification. Basically, it validates if the functionality does what it is supposed to do (i.e., "the user is able to add a new appointment on his personal calendar"). In software testing this is often referred to as "positive testing", because its main focus is on positive hypotheses.

Security testing, also known as non-functional test, on the other hand, is sometimes considered an aspect of functional testing but due to its very nature needs to be considered and planned separately, as it has almost a completely opposite focus [7, p. 30]. Therefore, software security testing moves towards the realm of negative testing. This kind of testing is labeled "negative" because it involves testing negative hypothesis, like for example "the user cannot modify other user's personal information". As said in [23 p. 36], *"this has led to security testers to search for exceptions to security requirements. Such exceptions are, of course, vulnerabilities, opportunities to realize malicious functionality"*.

Additionally, it is very common in security testing to make use of a technique named *boundary testing*. It generates test sets based on *corner cases*, namely test cases using extreme values within a domain like maximum, minimum or out of bound values. This kind of testing is very important at exercising uncommon conditions and verifying software's resiliency to exceptional cases.

3.2. Testing methodologies

This section seeks presenting a brief overview of common software testing methods. Typically they are divided in white and black box, but it is becoming more often to find in literature mentions to gray box testing as well. These approaches reflect the point of view that a test engineer has when building test sets for testing a system.

The upcoming subsections will go into more details regarding the approaches mentioned above, discuss common cases in where each approach is used and also talk about the pros and cons of each method.

3.2.1. White box testing

White box testing is an evaluation method in which the auditor has a significant amount of knowledge about the internals of system under test. In white box testing a test engineer often has access to design documentation, specifications and more importantly the source code of the application. This greatly augments the testing process especially because it enables a tester to choose appropriate inputs to exercise a larger number of paths, increasing coverage in tests. Code coverage will be discussed later in this chapter.

In general white box testing involves source code auditing. This process can be accomplished either manually or with automated tools. Manual code review needs an experienced auditor to read through the code and pinpoint vulnerabilities on it – although this is an expensive process, it may uncover bugs no automated tool can. On the other hand, automated source code analysis tools aid in reducing the task of poring over thousands of lines of code and find potentially vulnerable areas within the audited code. However, a manual auditor is still required to verify whether the findings are indeed valid.

This method of testing falls into the category of static analysis as it does not involve having the actual program being executed.

Earliest source code analyzers were very rudimentary in the sense of looking for vulnerabilities.

Basically they contained a database with well-known dangerous functions and signatures of bad and potentially insecure coding practices. These tools generated large amounts of false positives and could not catch vulnerabilities that did not fit into the pattern-matching used by these tools. Modern source code scanners, on the other hand, are better at pinpointing more subtle bugs and outputs less false positives.

Given its very nature, as [8, p. 9] mentions, white box testing in its source code auditing form is capable of achieving total coverage as the source code as a whole is available and all possible code paths can be audited for vulnerabilities.

3.2.2. Black box testing

While one may be inclined into thinking the absence of source code would render a software security assessment useless, this is not necessarily true. Arguably it is easier to analyze source code than binary code and if the source is available it will be a plus, but it is still possible to carry out security tests without source code involved.

Unlike the previous testing method, black box testing does not make use of any knowledge about the target application but exclusively relies on what the auditor can observe (i.e., inputs and outputs sent and received) and does not exactly know what is happening behind the scenes.

In the black box approach test cases are created around specifications and requirements, such as RFCs or other standards defining a protocol or file format. Black box security testing involves injecting faulty inputs into the application under test and monitors its behavior with a debugger or by analyzing other elements of the system, like log files, file system or CPU usage.

From a security auditing perspective, in order to uncover vulnerabilities, black box testing lends itself into sending random inputs or using a predetermined set of known bad inputs for common vulnerabilities and boundary cases. As said in [23, p 42], alternative methods for black box testing *"include the construction of malicious clients, facilitating manual parameter manipulation"* (i.e., an attack proxy for web application vulnerability testing).

Usually this method is considered as dynamic analysis because in order to conduct a black box test it is necessary to have the program running and preferably under monitoring.

3.2.3. Gray box testing

As the name implies, gray box testing is a mixture between white and black box approaches.

Basically, gray box takes advantage of reverse engineering to shed a light into some internal workings of the target application (for instance, to have an idea about a proprietary protocol), to gather profile information about the target or discover, via binary auditing, potentially vulnerable areas in the code, among other benefits.

Information gleaned through this hybrid analysis can assist black box testing, turning it a little less "blind" by providing a tester or a tool with more contextual information and significantly enhance its effectiveness.

3.2.4. Comparison between white, black and gray box approaches

White box	Black box	Gray box
Source code required	No source code required	No source code required; some reverse engineering needed
High code coverage	Limited code coverage	Improved code coverage

Table 3. Comparison between fuzz testing methodologies

3.3. Binary visualization and control-flow graph

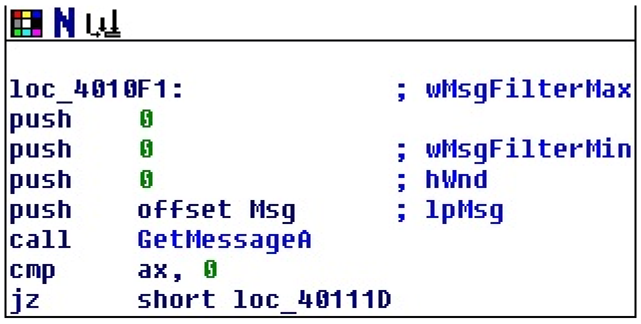
Disassembled binaries can be viewed as an abstraction commonly referred to as call graphs. This abstraction provides a visualization of a binary as a big picture, as well as outlining the relationships among functions within the executable. On the other hand, disassembled functions can be visualized as control-flow graphs. As put by [14], *"[the] CFG is a directed graph where the vertices represent basic blocks and edges represent possible transfer of control flow from one basic block to another"*.

CFG's most fundamental elements are basic blocks and directed edges. A basic block is a straight-line sequence of instructions where each instruction within it is guaranteed to run uninterruptedly and in order.

Typically basic blocks begin at the start of a function, or in control transfers (when it is the target of a branch instruction) or right after a branch instruction. Likewise, they terminate at instructions of branch (i.e., conditional or unconditional jumps) and return. Direct edge is an abstract representation of the possible control flow transfers that can happen in the end of a basic block to the beginning of another block.

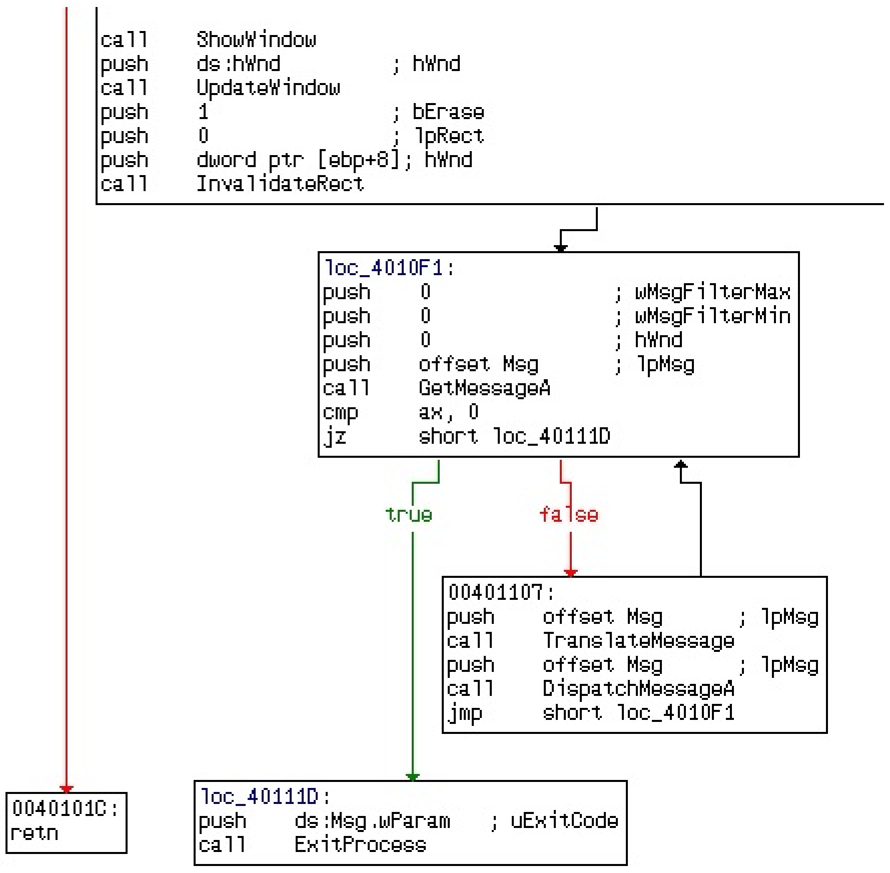
Common terminology concerning basic blocks include *entry* and *exit* block, which are, respectively, blocks that represent entry and exit points of a function. There are a large number of other definitions and properties based on graph theory regarding CFGs, especially when it comes to compiler optimizations, but there is no need to get into details about them for this work and they will not be discussed.

Figures 7 and 8 illustrate the concept of basic blocks and CFGs using a small closed-source program named *crackme.exe*:



```
loc_4010F1:                 ; wMsgFilterMax
push    0
push    0                   ; wMsgFilterMin
push    0                   ; hWnd
push    offset Msg          ; lpMsg
call    GetMessageA
cmp     ax, 0
jz      short loc_40111D
```

Figure 7. Illustration of one basic block of crackme.exe



```
call    ShowWindow
push    ds:hWnd           ; hWnd
call    UpdateWindow
push    1                 ; bErase
push    0                 ; lpRect
push    dword ptr [ebp+8]; hWnd
call    InvalidateRect
```

```
loc_4010F1:
push    0                   ; wMsgFilterMax
push    0                   ; wMsgFilterMin
push    0                   ; hWnd
push    offset Msg          ; lpMsg
call    GetMessageA
cmp     ax, 0
jz      short loc_40111D
```

```
00401107:
push    offset Msg       ; lpMsg
call    TranslateMessage
push    offset Msg       ; lpMsg
call    DispatchMessageA
jmp     short loc_4010F1
```

```
loc_40111D:
push    ds:Msg.wParam    ; uExitCode
call    ExitProcess
```

```
0040101C:
retn
```

Figure 8. Partial control-flow graph of crackme.exe

The main reason for discussing CFGs and some of their properties is that this kind of abstraction plays an essential role in profiling a binary application and provides enough information for the development of core parts of any tool that relies on program profiling. Later in this work we will emphasize how the usage of profiling data can be used for constructing smart fuzzers.

3.4. Code coverage

Code coverage is a quantitative metric employed in software testing that measure the amount of code (source or machine code) that has been executed by a set of test cases. This metric is often applied in software's quality assurance process to ascertain the quality of tests that had been carried out and to learn which areas of the program have not been exercised by the tests, giving a test engineer the opportunity to create additional enhanced test cases to increase coverage.

While one may be inclined of thinking coverage test can be perceived as a measurement of code quality as well, many seasoned testing practitioners disagree with this definition. However, it is a consensus that high coverage reflects thoroughness in testing.

Coverage analyzers, also known as coverage monitor, are tools used by test engineers to analyze code coverage of software under test. Most of coverage analysis techniques require access to the test program source code, or need the test application to be compiled with special options. Nevertheless, in black box approach code coverage can as well be measured in machine code level based on the number of basic blocks hit during the test execution.

Typically basic block level coverage can be achieved through a program analysis technique commonly known as *instrumentation*, more specifically binary instrumentation for the case of black box testing.

Instrumentation, in its more broad sense, is a method often employed to monitor an application for measuring performance, log trace data and other general diagnostics and profiling information about the process under observation. It works by inserting instructions in specific areas of the code for gathering whatever information one is interested in. Generally binary instrumentation is accomplished in three manners: patching the executable binary and inserting the monitoring instructions directly on it; utilizing a debugger to track down memory, register states and setting breakpoints and hooks on predetermined areas of the code; employing a technique known as dynamic binary instrumentation (often referred as DBI), which adds instrumentation code in run-time without the need of modifying the target executable.

Of all the aforementioned methods for instrumentation, DBI is the one who has the best performance and is recommended for industrial-grade instrumentation. DBI can be accomplished via frameworks such as Pin[5], DynamoRIO[6] and Valgrind[7].

Information about coverage can provide security testers an invaluable insight on the testing process of a given application. Just like regular testers, vulnerability researchers can benefit from this metric to improve test cases in an attempt to navigate through less exercised areas of the code and uncover a larger number of security defects within the program under evaluation.

A fuzzer can take advantage from coverage data if this information is incorporated to it through some sort of feedback mechanism – an evolutionary fuzzer can use this to greatly enhance the quality of test cases, ultimately leading to the increase of the overall test coverage.

In addition, code coverage also brings along with it reasonable criteria to answer the question on when to stop the testing process.

[5] Pin: http://www.pintool.org/

[6] DynamoRIO: http://www.dynamorio.org/

[7] Valgrind: http://www.valgrind.org/

Mature secure software development processes often include test sets with optimized test cases with increased coverage as mandatory requirement. As quoted in Microsoft's Security Development Lifecycle [15]:

"*Optimized templates have been shown to double fuzzing effectiveness in studies. A minimum of 500,000 iterations, and have fuzzed at least 250,000 iterations since the last bug found/fixed that meets the SDL Bug Bar*"

A caveat that must be taken into consideration when talking about this topic is that achieving maximum code coverage may be nearly impossible in large programs. This is due to the fact some code is internal to the application and cannot be influenced via the external input [31], and sometimes finding the correct combination of inputs to satisfy complex constraints to execute particular code paths may be extremely costly [28].

3.5. Chapter summary

In this chapter we have walked through a handful of topics relevant to software testing and program analysis. We started out underscoring the value of security assessment in modern software, touching on the topic of software testing, making clear the differences between functional and non-functional test (the latter is where security testing fits in), along with a brief examination of common software testing methodologies – white, black and gray box – highlighting their advantages, drawbacks and suitability for each situation. It is important to stress one can hardly state a certain methodology is better than another – an experienced tester will chose the most appropriate methodology for each situation.

Later we augmented the discussion by talking about binary visualization and code coverage. The former was examined more closely and we presented how a compiled executable can be represented as a graph abstraction and how this can be useful for profiling a closed-source application.

We concluded discussing code coverage and reinforced its importance as a metric for software testing, as well as explaining how it can be accomplished via a method known as instrumentation and briefly mentioning how an automated testing platform can leverage this information to learn more about the testing process.

All the information presented in this chapter serves as background information to create the building blocks of our proposed system in Chapter 6. The next section will examine a software security testing technique popularly known as “fuzzing”, which forms the very core of this work.

4. FUZZING: AUTOMATED SOFTWARE SECURITY TESTING

The first appearance of fuzzing for software quality assurance purposes dates back from 1983 when

Steve Capps, an engineer with Apple Inc., developed a program named "The Monkey" to feed random events into Mac programs [20]. However, the term "fuzzing" was not used before 1988. It was coined by Barton Miller in a project assessment for his “Advanced Operating System” class, and established in the seminal paper [16].

Although Miller *et. al* did not pioneer this technique for software testing, they undoubtedly were the first ones to document the process and attempt at creating a formal testing regime based on the importance of fault injection in software testing practices.

The goal of the project led by Miller was to assess the robustness and resiliency of various UNIX applications when fed with random inputs. Again, this project was focused in quality assurance instead of security. Nevertheless, it helped uncover potential security bugs and, more interesting, unearthed vulnerability classes unknown at the time and widely ignored until early 2000s.

The results approximately 33% of the UNIX applications tested in the experiment crashed when fed with random input.

In 1995 Miller and his team repeated the experiments under similar conditions. This time over 80 applications were tested and the crash rate had worst case of 40% for basic programs and over 25% for X-Window applications, suggesting lack of secure coding and testing practices and also that very little effort was put by programmers into fixing the crashing applications (for instance, at least one bug discovered in the original research was still present seven years later).

The outcome of these experiments, as noted in [23, p. 45], *"[...] inadvertently defined a general model of a practical fuzzer"*. Although developments have been made to fuzzing throughout the years, the basic model remained the same. The general model of a practical fuzzer is going to be discussed in further detail later in this chapter.

Nevertheless, these advances have made fuzzing to be no longer being perceived as a non-methodical approach and draw attention to robustness testing to the formal software engineering methods.

4.1. Fuzzing: an overview

Fuzzing is a technique employed in software testing that involves deliberately injecting faulty inputs in a program under test. Faulty inputs can come in form of random, invalid, unexpected or non-conformant data. The program subject to test is monitored for exceptional conditions, such as hangs or memory violation faults. This process is typically fully automated or semi-automated, and hardly requires intensive human intervention after it is running.

Fuzzing tools are generally divided into custom standalone scripts, often crafted to test a specific program or protocol, and fuzzing frameworks, which already includes features most of fuzzers need and exposes on its API enabling the rapid construction of fuzzers on top of it.

4.1.1. What can be fuzzed

One of the very first steps a tester must take before fuzzing is to correctly identify what can be fuzzed. Typically, as pointed out in [24], *"any point where there is some communication with an application can potentially be fuzzed"*. Namely, a tester is interested in mapping the application's *attack surface*. This surface can be characterized as any data processed by the application that can be influenced directly or indirectly.

Potential inputs to be fuzzed include APIs, data traversing network communication, command line arguments, registry keys, files, kernel drivers, environment variables, and more.

It is imperative to say this phase is extremely important when doing fuzz testing, as an improperly mapped attack surface can leave parts of the code untested, potentially leaving behind security-critical bugs.

4.2. How fuzzing works

As already said earlier in this chapter, fuzzing works by providing faulty inputs to a program under test. However, the manner these inputs are generated can differ greatly between fuzzers, with each having different levels of difficulty when implementing and consequently achieving different results.

The way fuzzers work can roughly be divided in random-, mutation- and generation-based. Below are detailed descriptions about each type of fuzzer, regarding the process of generating test cases.

4.2.1. Random fuzzing

Random-based fuzzing is considered the simplest form of fuzzing and consists in sending a stream of random data to the software subject to testing. The main advantage of this “zero-knowledge” approach is that it requires no prior knowledge about the target application. Moreover, the effort and skills needed to craft a custom script to start random testing are minimum.

While it has the potential to discover bugs, as presented in [16], there are several drawbacks in this methodology. Random fuzzing generally catches only “low-hanging fruits”, missing lots of bugs that could have been found with a slightly more elaborated fuzz testing. Furthermore, code coverage during test is considerably poor if compared to other types of fuzzers.

4.2.2. Mutation fuzzing

Fuzz testing with deliberate or carefully chosen modifications on a given data is called mutation fuzzing. This type of fuzzing borrows inspiration from a formal software testing methodology named *mutation-based testing*. Mutation testing evaluates correctness of an implementation and involves small modifications in a program's byte code or source code.

Likewise, mutation fuzzing typically begins with the acquisition of valid data samples, like for instance with collecting network captures and files. A large number of samples is preferred as diversity is more likely to contain samples that lead to higher code coverage. Then test cases are constructed by mutating the samples, randomly or heuristically.

Usually random mutations are achieved through bit-flipping (negating the boolean value of a bit – i.e., a bit value of 0 becomes 1), whereas heuristic mutations take place by either inserting predefined values known for causing programs to fail (i.e., long strings, format specifiers – will be discussed in detail later), moving data from one location within the sample to another or swapping data within it.

Mutation fuzzing is relatively simple to set up, as it requires very little or no knowledge about the file format or protocol being tested. Writing a program to flip bits or do heuristic mutations in a given sample is considered inexpensive.

Furthermore, there are several ready-to-use file mangling programs, like FileFuzz[8] and Egurra[9], that can be easily adapted for specific needs.

[8] FileFuzz: http://www.fuzzing.org/wp-content/FileFuzz.zip

[9] Egurra: https://code.google.com/p/egurra/

Disadvantages of mutation fuzzing include limited code coverage. In addition, this approach may generate test cases that are too invalid to be processed by the application under test. Also, this method is hardly useful when testing applications that rely on checksums, cyclic redundancy checks (CRCs) or compression to ensure the validity of the processing data, unless the fuzzer is protocol-aware and recalculates these values for the modified file; otherwise many CPU cycles will be wasted with data that cannot even be processed.

Despite the aforementioned limitations, this approach has been very successful at finding major security vulnerabilities.

To cite relevant discoveries using this technique, we can refer to [25], in which security researcher Charlie Miller, employing a simple mutation scheme, found hundreds of crashing test cases and dozens of potentially exploitable bugs in widely used products such as Adobe Reader, Apple Preview, OpenOffice and Microsoft Office.

More recently, Google Security Team led a similar effort [27], this time with large computing power that resulted in the discovery of 106 exploitable vulnerabilities in Adobe Flash Player.

4.2.3. Generation fuzzing

Generation-based fuzzing is regarded to be the most advanced among common methods used for fuzzing. Unlike its random and mutation-based counterparts, generation-based fuzzing does not involve only feeding random data to a program nor requires the collection of several samples in order to commence the test run.

On the other hand, generation fuzzers start out by gathering information off a specification, such as a RFC. This specification is later used to create the protocol grammar used by the fuzzer. Doing so is necessary to identify parts of the data that should remain static and others that can be fuzzeable.

The construction of test cases in this approach differs from the other methods. Here after building the grammar and identifying datatypes of fields and which ones can be fuzzed, the generation-based fuzzer commonly resorts into using fuzzing heuristics – it often employs values known for exercising corner cases, long strings and other data known for causing programs to fail. The success of this approach is dependent on the ability to properly map the protocol into a grammar.

Generation fuzzers are considerably more difficult to implement if compared to the others we mentioned before. This is due to the fact a thorough research on the file format or network protocol has to be done up-front, and this can take a reasonable time for complex protocols. Moreover, employing this method for fuzzing proprietary protocols is feasible, although can be extremely costly as reverse engineering or considerable network monitoring, in case of a network protocol, will be required.

This method of fuzzing is hardly capable of exercising “obscure”, undocumented features and backdoors in software. An example of memory corruption vulnerabilities in undocumented features are CVE-2010-3653[10] and CVE-2008-5416[11], whereas CVE-2006-2369[12] serves as a good illustration of backdoor in the sense of authentication bypass.

[10] http://cve.mitre.org/cgi-bin/cvename.cgi?name=2010-3653 (Adobe Shockwave rcsL Memory Corruption)

[11] http://cve.mitre.org/cgi-bin/cvename.cgi?name=2008-5416 (MS SQL Server sp_replwritetovarbin Memory Corruption)

[12] http://cve.mitre.org/cgi-bin/cvename.cgi?name=2006-2369 (RealVNC Remote Authentication Bypass Vulnerability)

Despite of the limitations mentioned earlier, this approach is more likely to discover bugs the others would miss. Generation-based fuzzing's protocol-aware nature is considered to have considerable superior code coverage, as quantified in [26]. This paper shows that test cases created by generation fuzzers covered 229% more code than simple mutation fuzzer in *libpng*, a popular library for parsing PNG files.

4.2.4. Comparison: random vs. mutation vs. generation

This table provides an easy reference to compare the three most common test case generation methodologies we have discussed prior:

Random fuzzing	Mutation fuzzing	Generation fuzzing
Trivial to implement	Relatively easy to implement	Harder to implement; reverse engineering may be required
Limited code coverage	Improved code coverage	Higher code coverage, but still deficient at exercising deep program logic
Limited at discovering bugs; finds only "low-hanging fruits"	Improved at finding bugs; may miss more subtle ones	Discovers more bugs than the other approaches

Table 4. Comparison of common fuzzing methods

4.3. Fuzz heuristics

From what has been examined so far in this chapter we already can notice generating data for fuzz testing is a crucial element in the whole process. Equally as important is the need to have a strategy to define what data to generate in order to maximize the effectiveness of the testing process in terms of bug count and other applicable metrics, and reduce the time spent during test.

In [37] the author demonstrated how even very small programs can possibly have test sets that would require millions of years to run entirely. Thus, testing all possible permutations in a given input space is not only inefficient but also infeasible.

For instance, let’s say we need to build a test set for testing a numeric field in a protocol. Surely we can use all possible numbers within a *short* integer or *long* integer range, but this is largely ineffective as the number of test cases will explode and the test run will take too long to finish.

Therefore, we need to choose an intelligent subset of inputs to be part of our test cases. To do so we need to include in our list of fuzz strings values considered to be potentially dangerous, also known as fuzz heuristics.

We enumerate a few categories of vulnerabilities and which data are more likely to trigger software faults, giving us preference to include them in our smarter test set:

- **Buffer overflows**: Long character sequences, string repetitions;
- **Integer overflows**: Border and near-border integer values, border values divided by 2, 3, 4 and values around it;
- **Delimiter problems**: Non-alphanumeric characters and common characters used to delimit fields;
- **Unicode expansion**: Characters known for getting expanded when converted, preferably those with higher expansion ratio;
- **Canonicalization problems**: Encoded malicious input;
- **Format strings**: Format specifiers ‘%n’ and ‘%s’;
- **OS command injection**: Common shell metacharacters followed by a harmless OS command.

4.4. Fuzzing phases and basic architecture

Regardless of the process used by fuzzer to create test cases, all fuzzing tools share similarities when it comes to their architecture. Figure 9 illustrates a very basic model of a fuzzer architecture. Moreover, the fuzzing process commonly employs the very same basic phases:

1. **Enumerating attack surface**: This first step is essential to the success of fuzzing. Failing to properly map attack surface and other potential sources of input can limit testing as it will provide an inaccurate picture about the resiliency of the program against faulty inputs.

2. **Create test cases (fuzzed data)**: Every fuzzer must somehow create test cases to be fed as input to a program. This process must be automated and the method used to generate test cases must be chosen accordingly to the needs of the tester.

3. **Execute fuzzed data**: This step involves automatically sending the input to the target application. For instance, in a network protocol fuzzing it consists into sending packets with mangled data to the target; in a file format fuzzing it spawns the target executable and makes it open the fuzzed file, etc.

4. **Monitor for exceptions**: A crucial element of any fuzzing system is its capability to monitor for exceptions. If a fuzzer is capable of triggering bugs but not noticing them, it is of less value as many potentially exploitable bugs will be missed. Common methods for detecting these conditions include attaching a debugger to the program subject to testing and monitoring for exceptions, or performing "liveness detection", a method that involves checking whether the program or service under test responds accordingly – if it does not, a hang or a crash may have occurred.

A key feature in any well-designed fuzzing system is a way to correlate crashes and anomalous conditions to test cases – these must be recorded to guarantee the reproducibility of the fault and assure the repeatability of the process. A fault-induction test case and its respective *crashdump* (a sample report containing low-level information about the problem) are of great support for further investigation on the root cause of the bug.

Determining exploitability can be considered an optional step while fuzzing. Depending on the goal of the audit, when a bug is discovered it may be necessary to verify whether the bug is a security vulnerability or a non-exploitable bug. This process can be automated to some degree with tools like !exploitable[13] or CrashWangler[14], but a serious audit should involve manual verification of crash reports when regarding exploitability, as these tools can generate false positives and false negatives.

More often than not a fuzzer can trigger the exact same bug, eventually leading to a duplicate crash report. Thus, another optional asset a fuzzing toolkit may contain is a way to identify whether a crash is unique or was triggered before. Undoubtedly, uniqueness of crash reports is important because it reduces the number of reports to analyze and helps focus only on unique defects.

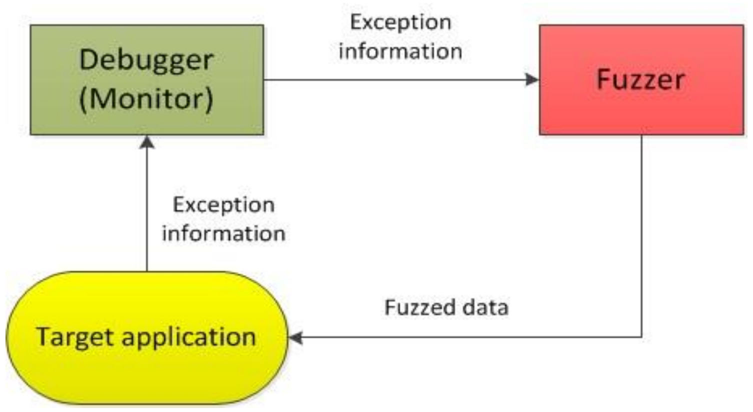


Figure 9. Basic model of a fuzzer

4.5. Chapter summary

[13] !exploitable: https://msecdbg.codeplex.com/

[14] CrashWangler: http://connect.apple.com/

This chapter examined fuzzing beginning with its origins, general overview and what can be fuzzed. This serves to set the scene to discuss the inner workings of this test technique and get into detail on the most common methods used to generate malformed data as support stage of fuzz testing, along with a comparison on these methods, stressing their suitability for each situation.

Additionally, we highlighted the importance of fuzzing heuristics in the sense of having a strategy to intelligently generate data for maximizing the effectiveness and reducing the time of the testing process. Furthermore, we detailed the basic model of a modern fuzzer, enumerating its mandatory features and optional assets, along with a diagram explaining it in an illustrated fashion.

The next chapter will examine developments on the field of genetic algorithms and evolutionary computing to augment the required background for understanding the proposal in Chapter 6.

5. EVOLUTIONARY COMPUTING AND GENETIC ALGORITHMS

Evolutionary computing is a field of computer science that tries to solve problems inspired on concepts very similar to evolutionary mechanisms found in nature. These concepts were introduced and formalized by Charles Darwin, and according to his theory life on Earth is the result of a process of selection of the fittest and better adapted individuals, and these have more chances to reproduce.

In biology every living organism is composed of genes, which in turn are tied together to form chromosomes. Each gene represents a specific trait and may have different settings (i.e., the genes for eye color may have settings like blue, green, brown and black). Genes and their settings are the organism's genetic payload and is usually referred to as genotype. The physical embodiment of the genotype is called phenotype. In evolution the process known as recombination happens when two organisms mate and share their genes.

The outcome is an offspring that may end up having half the genes from one parent and the other half from the other. Occasionally a gene may be mutated, and although in general it does not affect the phenotype it can sometimes be expressed in the organism as a completely new trait.

A genetic algorithm is an approximating search technique often used for optimization. It is inspired in the evolutionary computing paradigm and essentially mimics the same process found in nature by making use of combination (often referred to as *crossover* in GA), mutation and selection to evolve a solution of a given problem.

A GA performs combinations and mutations on a base population of individuals while maintaining desired hereditary traits in future offspring. Then rules of natural selection are applied to choose which organisms are fit for the environment in place. Finally the fittest individuals are combined and eventually mutated to serve as base to form future generations. In GA the process continues until either satisfactory solution is achieved[15] or an *a priori* number of generations is reached.

Before one can use genetic algorithms to start solving optimization and search problems, some basic components are needed to define a GA. Figure 10 provides a general overview of these elements and they are listed and described below:

- **Encoding**: Potential solutions must be encoded accordingly. Typically it is a binary bit string or a string with real numbers;
- **Representation of a solution**: A representation of what the solution (or a set of solutions) might look like. This is used as stop condition;
- **Fitness function**: Evaluates each candidate solution to select the most "fit". It is dependent on the problem to be solved;
- **Reproduction**: Functions responsible for crossover, mutation and selection. This component is responsible for passing hereditary information and ensures diversity in future generations.

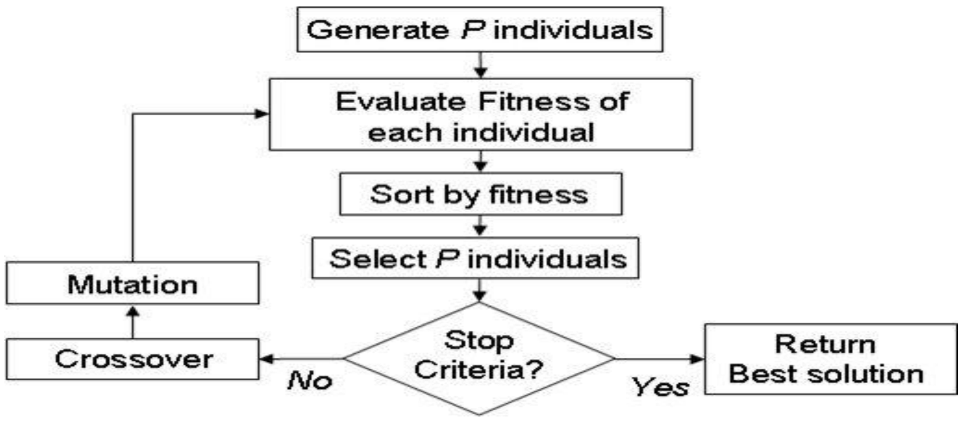


Figure 10. A common genetic algorithm run

[15] The final output of a genetic algorithm run does not mean a solution was found, but depicts an approximated optimal solution. It can, however, be the expected solution for the problem.

For the sake of clarification, following is an illustration of a sample genetic algorithm run taking into consideration a simple problem. The example is taken from [8, p. 431] with no modifications at all. The problem to be solved in this example involves maximizing the number of 1s in a binary string of length 10. We can easily realize that fitness function and representation are pretty straightforward (fitness will count the number of 1s within a binary string and select the fittest solutions while representation of the solution is a binary string full of 1s). For reproduction this example will rely on a simple method for crossover and mutation. It arbitrarily swaps the two binary strings at positions 3 and 6 and to randomly flip a bit in the offspring.

After having all components well-defined the GA is now ready to evolve. It does so as follows:

1. Initialization phase: generates random population of candidate solutions;
2. Tests candidate solutions against the fitness functions to select the most fit ones;
3. Reproduction step – applies crossover and mutations to the selected candidate solutions;
4. The resulting generation replaces the previous population, forming the base for the next round of evolution. This process loops until it finds a solution or reaches a limit in the number of generations.

► First generation

0100100000	2
<u>1000001010</u>	**3**
<u>1110100111</u>	**7**
0000001000	1

The highlighted chromosomes above are considered the most fit, therefore chosen to mate. Bounded by the reproduction rules, the following offspring is created:

► Second generation (crossover using single-point method)

100**<u>0001010</u>**	3	100 + 0100111 -> 1000100111
111**<u>0100111</u>**	7	111 + 0001010 -> 1110001010
100000**<u>1010</u>**	3	100000 + 0111 -> 1000000111
111010**<u>0111</u>**	7	111010 + 1010 -> 1110101010

After crossover, the algorithm will keep evolving the solution to find better candidates:

► Third generation (random mutation using bit-flipping)

1000100111 -> 10**<u>1</u>**0100111	6
1110001010 -> 111000**<u>0</u>**010	4
1000000111 -> 100000**<u>1</u>**111	5
1110101010 -> 1110101**<u>1</u>**10	7

By the third generation we are already able to see the GA increased the average fitness of the population. Given a reasonable threshold for generations, GA will continue to search for better solutions until it eventually finds an actual solution or an approximation to it.

5.1. Genetic algorithms applied to software testing

Previously in this work we have seen that finding desired inputs to properly exercise specific code within a software under test may be a hard task. Furthermore, while random test may succeed in some cases it is highly unlikely it is going to have the same performance if there is a need to satisfy complex constraints, thus achieving poor coverage [17].

In the past decades several researchers have attempted to correlate the areas of machine learning, more specifically genetic algorithms, and software testing with the intent to embed some "intelligence" in the testing process in order to enable the generation of more effective test cases. Test cases effectiveness, on its turn, can be measured through different criteria, like path coverage, number of defects encountered, and others.

When discussing about a few relevant prior efforts to intersect the domains of software testing and genetic algorithms, we can mention [17] in which the authors have developed a test case generator that relied on genetic algorithm. The experiments presented in the aforementioned paper suggested that GA-enhanced tests outperformed tests based on random generators. Another interesting and older development in this field is discussed in [18] also suggests GA can aid in the creation of test cases that yield better coverage.

5.2. Problems and limitations of evolutionary testing

Up to this point we have only discussed the power and benefits that a stochastic search algorithm like GA can bring to software testing. In fact, genetic algorithms are currently considered one of the most efficient search algorithms. However, genetic algorithms (and consequently evolutionary testing) suffer from a couple of problems and limitations that must be taken into account.

As suggested in [21, p. 202] of the problems encountered in GAs one is especially of our concern. In many problems genetic algorithms may be biased to converge towards local optima rather than the global optimum point, yielding misleading results. Furthermore, depending on the search

landscape GAs can be stuck in *flat landscapes*, when the search has no higher point to converge to and end up degenerating itself into a random search, or in *deceptive landscapes*, when the search believes to be converging to an optimal point but in fact *"the basins of attraction lead away from the global optimum"* [19].

These problems are closely related to the choice of the fitness function applied in the algorithm.

Another limitation regarding fitness criteria is that GA cannot solve problems in which the only fitness measure available to the algorithm is a right/wrong result [22, p. 1], for instance like in a decision problem.

5.3. Chapter summary

This chapter covered evolutionary computing and a search optimization technique known as genetic algorithm. Initially it offered a general overview of evolutionary computing to start analyzing the topic of genetic algorithms. Later on we went through each relevant point within GA and provided a diagram for the sake of illustration and understanding. In addition, we clarified it by giving out a sample GA run, describing each pertinent point in every step of the process.

We also described how genetic algorithms can be used in software testing to form a methodology known as evolutionary testing, referring to past research in this area, as well as discussing problems and limitations with this methodology.

The information exposed and discussed in this section provides a solid background for the sake of understanding the next chapter, in which all these concepts will be bound together in order to form a proposal for a GA-capable fuzzing tool.

6. IMPLEMENTING AND EVALUATING AN EVOLUTIONARY FUZZING SYSTEM

It is already clear from what we have seen so far that random, non-smart fuzzing lacks efficiency in many important aspects regarding software security testing. We have previously underscored the fact dumb fuzzing methods are effective at uncovering "low hanging fruits", as they are generally better at scratching the attack surface but lack the ability of going deeper within the program's logic, rendering this type of fuzzer unable to exercise code that could contain exploitable bugs.

The past chapters discussed a myriad of topics concerning software testing, common vulnerabilities found in software and automated vulnerability discovery with fuzzing. All this background information is necessary to build foundational knowledge in order to fully understand the proposed architecture of this work for a genetic algorithm-enhanced fuzzer.

This chapter provides a discussion on the investigation carried out by the author on the application of genetic algorithms with the aim to optimize input generation for achieving better coverage, and consequently expose more bugs. It goes into the inner workings on the proof-of-concept implementation, describing in details each relevant point on the proposed architecture.

The proof of concept fuzzing system we intend to propose is written in the Python programming language and relies on two other external reverse engineering tools, namely PaiMei[16] and IDA Pro[17], to support some low-level tasks required by our system and Pyevolve[18], an open-source genetic algorithm library. These support tools and libraries were chosen mainly because of its easy usage, large quantity of manuals and support material and the endorsement of a large and active community on them.

[16] PaiMei: https://code.google.com/p/paimei/

[17] Hex-Rays IDA Pro: http://www.hex-rays.com/idapro/

[18] Pyevolve: http://pyevolve.sourceforge.net/

6.1. Evolutionary fuzzing: a search problem to find suitable dangerous inputs

Evolutionary fuzzing is a relatively new approach to fuzzing. As one may guess, this approach brings together evolutionary testing methods and software security testing. It mainly consists into leveraging code coverage data or other kind of information useful for “fuzzer tracking” to generate appropriate inputs with the aid of GA, aiming to exercise a larger number of regions within a program or reach specific code within it.

Only very recently security researchers have started using genetic algorithms to improve fuzz testing despite of the fact similar techniques have been employed to regular software testing for some time.

Most notably, efforts led by academic and industry researchers in [32] and [31] served as starting point for security testers realize the potential of GA and other search algorithms applied for fuzzing.

In [32] the authors have developed a tool named "Sidewinder" capable of automatically generating inputs to force program's execution into a following a certain path. The tool finds potentially vulnerable code sections (i.e., calls to functions like *strcpy()*) and a subgraph between the input interface and the pinpointed vulnerable location. Then a Markov process is applied to the subgraph to calculate the probability of traversing certain paths – this step is used to calculate the fitness function, and inputs with higher fitness are mated to create inputs that reach regions closer to the target section. The process continues until the vulnerable section is reached.

The authors of the aforementioned work brought to the table a very powerful and novel concept with “Sidewinder”. However, this approach has some drawbacks that impair its efficiency. First of all, the heuristic employed in finding potentially vulnerable code sections has severe limitations. We have shown before that vulnerabilities can manifest in many code constructions, especially in parsers, not only in insecure function calls like *strcpy()* or *gets()*. Thus, this approach misses many other bugs that do not fit into these very simple cases. Second, accurately extracting CFG from a binary can be difficult, especially if obfuscation, compression or encryption has been applied to it. An erroneous CFG will certainly mislead the algorithm.

The work presented in [29] shares many similarities with what Sherri Sparks *et al* have proposed, including the initial triage to find potentially vulnerable APIs and functions. However, instead of using Markov probabilities, it simply gives higher fitness to inputs that have more similar predicates to the “CDPPath” – the path constraint from the root node until the leaf containing the vulnerable code.

Another related work, but slightly less sophisticated, is presented in [33]. The author of this work developed a proof of concept tool that works in two rounds: the first one it runs a common random fuzzing and annotates which portions of the target were covered, while the second uses genetic algorithm to create inputs capable of reaching portions of code left uncovered in the previous run. Essentially it uses GA to brute force an input to drive the execution to locations that have not been covered before.

The research presented in [31] suggests a more relatively superior approach to the works above. DeMott *et al.*, have developed a tool named “Evolutionary Fuzzing System”. EFS is a full-fledged tool capable to generate fuzz data in both ASCII and binary formats. Additionally, and more importantly, it has the ability to evolve fuzzing sessions to a level that it can learn the protocol employed in the application being fuzzed. This system uses a complex representation for “fuzzies”; data it is separated in tokens, legs, sessions and pools. A pool is composed of sessions, which in its turn is composed by legs, and so on. These are used to leverage more complex breeding schemes, possibly achieving better results.

6.2. Architecture for a genetic algorithm-enhanced fuzzer

The proposed system has the same level of simplicity as the one presented in [33] but takes an approach similar to [31] when it comes to the final objective: instead of targeting specific areas of code by creating inputs that will drive execution to it, our only goal is to increase code coverage and consequently exploring more areas of the program, regardless these regions contain calls to insecure functions or not.

In this section we outline the main components of our proof of concept genetic algorithm-enhanced fuzzer, highlighting the differences between a normal fuzzer and a GA-based fuzzing system and diving into details concerning the implementation of the modules of this tool.

6.2.1. Target static profiling

A preliminary step in building our fuzzing system is to do a profiling of the target binary. By profiling an executable we are able to gather information about its inner workings; for instance we can learn the total number of instructions in the target application, what sections of code are in charge for handling a given task, which are the entry points of functions, how many basic blocks the executable is composed of, and more.

For this phase we will be relying on 'pida_dump.py', a custom script provided in PaiMei for IDA Pro. It gives us the ability to convert static information from an executable extracted by IDA Pro to a portable format composed of separate classes for representing instructions, basic blocks and functions. This allows navigation through a binary and extraction of information from it.

As mentioned in previous chapters, disassembled executables can be visualized in an abstraction known as control-flow graphs. This abstraction has the ability to represent a disassembled binary in form of basic blocks, with each block being a node in the graph and the branches among the blocks serving as edges.

Therefore, we can start this task by extracting the target program's CFG, as it allows us to retrieve information about the total number of basic blocks, start and end addresses of basic blocks and functions.

This stage is carried out manually by loading the target executable in IDA Pro and running the static profiling script. The script creates a '.pida' file, which contains all the harvested information about the binary in a searchable and easy to navigate format. Later this file is loaded by the fuzzer and it will consume some of the information obtained from the target application.

It is necessary to highlight the importance of this stage to the system as a whole. This step plays a fundamental role in the intelligent fuzzing approach as the outcome of this phase will serve as input for both the code coverage tool we will develop and discuss in the upcoming section, and also for modeling the genetic algorithm's fitness function.

6.2.2. Tracing and code coverage

The information acquired in the profiling step is used during this stage to track the execution of instructions and determine which basic blocks within the target application are being hit while fuzzing.

We have seen earlier in this work that code coverage is an important metric in software testing, as it measures the quantity of the program's code that has been tested. This metric arguably can be very useful for our purposes to provide feedback to other modules in our fuzzer about the extent of the code covered so far; in other words the code that could be reached given our set of inputs.

This data – the amount of code executed – will be subsequently used to serve as basis to create the fitness function that will drive the execution of the genetic algorithm and test case generation.

For fitness evaluation we must take into account solutions that lead to better coverage.

This section details step by step the creation of a rudimentary but effective tool for measuring code coverage.

Initially we make use of the .pida file generated by the static profiling phase. As we have discussed before, this file contains valuable data harvested from the binary, including information about its functions and basic blocks addresses. Thus, we are able to navigate through the file and extract the total number of functions and basic blocks, in addition to their starting addresses. We are also capable of identify which functions are pertaining to the target binary, as the 'pida_dump' script may also include addresses for statically linked library functions, which turn out to be irrelevant for our testing purposes as these are code located outside our target.

In possession of this data we can choose whether we will measure coverage based on the number of hits of instructions, basic blocks or functions during the program's run. As noted in [8, p. 444], tracking a program's execution based on instructions is not only extremely slow but unnecessary. We immediately discard this approach and are left with two other options to consider.

Tracking execution via basic blocks hits gives us a considerable better performance and leads us to achieve the very same results – because of the very nature of basic blocks we are tracking instructions execution too. However, when tracing programs with a large number of basic blocks the performance is likely not to be satisfactory and we may want to shift to another approach.

Doing our tracing based on functions gives us even better speed. On the other hand, tracking the execution relying on functions' addresses may provide our fuzzer a less accurate picture about coverage.

For our implementation we opted to do code coverage at basic block level because the sample target executable that will serve for test purposes does not contain a large number of basic blocks.

Right after pinpointing the addresses we are interested in, there must be a way to tell when they were hit during execution. Using PyDbg to monitor for breakpoints is fairly simple and straightforward. Basic usage includes creating a custom function to handle the types of events we are interested in (memory breakpoints, in this case), registering this custom handler as a callback function to be called in case of an event of our interest happens (in this particular situation we are interested in the type *EXCEPTION_BREAKPOINT*) and either attaching the debugger script to the target process or load it from disk.

This custom breakpoint handler verifies whether the exception address is within a list of basic blocks for the executable and if affirmative, increases the hit counter. We subsequently remove the recently hit breakpoint's address from the list for the sake of speed at the expense of more contextual information. Nevertheless, the removal of breakpoint addresses is not believed to impair our analysis.

Appendix A contains the source code of the whole implementation, including the code coverage tool deployed in this fuzzer.

6.2.3. Test case heuristics

For our prototype we are relying on some of the very same test case heuristics generally employed by fuzzers described earlier in this paper. We will include potentially dangerous values in our list of test cases – in other words, values that are more likely to trigger buffer overflows, format bugs and so on.

As we are just intending do basic experiments, this proof of concept will only feature long ASCII strings.

6.2.4. Genetic algorithm test case generator

The creation of test cases is backed up by a genetic algorithm-powered string generator. The main idea behind it is to have the ability of learning desirable information from inputs that have been tried before by assessing its raw score and use this knowledge to create better inputs. The parameters for our GA engine are described below:

- **Genome**: Internally, the genome we are using in our example is nothing but a unidimensional list of integers. This structure is based on Python list datatype and inherits all characteristics from it.
  We generate a list of random integers ranging from 48 to 126, as these represent the spectrum of printable ASCII characters. Later on we do convert this list of integers into a text string (we will refer to it as GA-created string), leveraging the fact we are using numbers that correspond to the representation of characters in the ASCII table.
- **Fitness**: The objective function employed in our system returns as fitness score the number of basic blocks hit after the program is fed with the GA-created string. This will have our program to keep chasing higher scores by generating more suitable inputs, thus traversing more of the code and achieving higher coverage.
- **Reproduction**: The scheme we use for reproduction is based on the defaults of Pyevolve. We employ both crossover and mutation for mating purposes.

  1) **Crossover method**: Single point crossover
     a. **Crossover rate**: 80%
  2) **Mutation method**: Swap mutator
     a. **Mutation rate**: 2%
  3) **Selection method**: Rank selector

- **Stop condition**: We use a maximum number of generations as sole criteria for stopping the search. We arbitrarily choose 10,000 as default value but it can be easily adjusted.

6.2.5. Exception monitoring and fault detection

We have already outlined the importance of fault detection when we examined the architecture of a basic fuzzer. After all, a fuzzer is of less value if it is capable to trigger a fault in the target application but unable to catch the exception it triggered, potentially leaving unnoticed a handful of vulnerabilities.

Earlier in this paper were mentioned a couple of ways to detect and monitor exceptions, from liveness detection to process crash due to the raise of a fatal exception. One popular way to monitor exceptions

in a binary program is to attach a debugger to it. Attaching a debugger to our target application allows us to intercept any events and exceptions our fuzzer may have triggered. However, most of the popular debuggers are not suitable for our particular case as they cannot be easily automated.

On the other hand, although the Windows operating system exposes a rich debugging API in which we can work on top of and achieve the desired automation, getting familiar the API would take us some time plus it could take a longer time for developing a debugger with all the functionality one would be interested in when doing fuzzing and vulnerability research.

Instead we resorted again to PaiMei framework and found out it had a much better solution to the problem of assisting our fuzzer in exception monitoring. One of PaiMei's components is PyDbg, a module that gives us the ability to create custom debugger scripts. PyDbg offers most of debugging functionality we will need, such as setting breakpoints, reading from and writing to memory and handling events/exceptions.

Likewise, this component provides a range of features for vulnerability analysis, like stack and SEH unwinding (snapshot of the call stack and SEH chain at the moment the exception occurred), crash recording and more. These optional features can be of great help when an auditor is faced with the task of pinpointing the precise cause of an exception or when weeding out duplicate crash reports.

We already know it is very trivial to make PyDbg monitor for exceptions. Again, we lend ourselves into creating a custom function to handle access violations, register this custom handler as a callback function to be called in case of *EXCEPTION_ACCESS_VIOLATION* occurs and attach our debug to the target or load it from the disk.

As opposed to just flagging an auditor in the event of an access violation, our implementation goes an extra mile and takes advantage of PyDbg's vulnerability analysis support components by making use of stack/SEH unwinding, disassembly around (disassembly of the five instructions before and after the instruction that triggered the exception) and crash recording features.

Appendix B contains detailed information about the implementation of the exception handler module of the fuzzer.

6.2.6. System overview

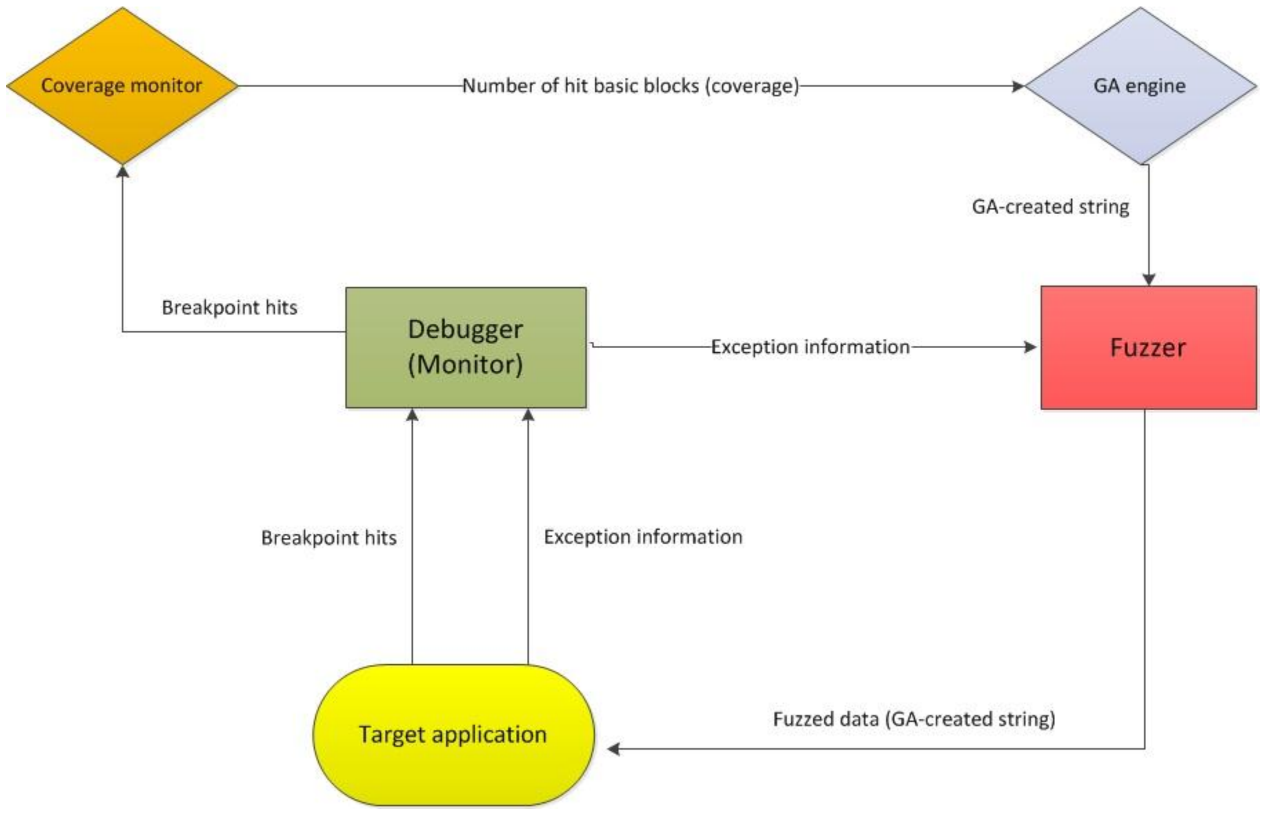


Figure 11. Architecture of the GA-enhanced fuzzer

6.3. Experiments

After developing the proof of concept tool there is the obvious need to measure its effectiveness. This section presents the results of the experiments carried out to evaluate the system.

The first step of our test plan includes the creation of three sample programs written in C and designed on purpose to contain well-known security vulnerabilities that can be uncovered by a fuzzer. Each of them has a “level of difficulty”, in the sense that more constraints need to be satisfied so the input can penetrate deeper in their respective CFGs to finally reach the vulnerable code. These levels of complexity are used to demonstrate the ability of the genetic algorithm search to perform in different scenarios, from the most basic control flow graphs to more elaborated ones that involve more conditions to be met so the program can go further.

Table 5 shows profile and vulnerability information of sample programs. The source code for all sample programs used in our tests can be found in Appendix A.

	Example 1	Example 2	Example 3
Total basic blocks	5	6	10
Minimum basic blocks hits	3	3	3
Vulnerability	Buffer overflow and format bug	Buffer overflow	Buffer overflow

Table 5. Profile and vulnerability information of sample programs

The second phase of experimentation was conducted in an ordinary netbook computer running Microsoft Windows 7 and Python 2.5. All binaries were compiled using Dev-C++ 4.9.9.2 with no special compiler flags.

We decided to carry on with a very simple test plan. Our hypothesis was to run our GA-capable fuzzing system against each target three times and note the number of basic blocks hit versus the average number of individuals needed to do so.

The author is aware of the fact this test plan is not the most appropriate one for benchmarking such system. However, it has proved to be valuable in learning some interesting behaviors and pitfalls that concern evolutionary testing.

6.3.1. Results

In this section we analyze the results we obtained with the proof of concept tool proposed in this work. These results will make us able to draw conclusion about evolutionary fuzzing in the next chapter.

For *example1.exe*, as we can recall from the Table 5, we have a total of five basic blocks. Running this executable with any input, regardless if it is suitable for traversing the constraints or not, its minimum block hit count is three.

Following our test methodology, we ran the fuzzer three times against *example1.exe*. This is the sample which contains the simplest of the conditionals to proceed into deeper instructions within the code. Naturally we expect our fuzzer to generate a smaller number of individuals in order to achieve more coverage if comparing with other examples that contain more complex conditionals.

In the first run it was needed 48 individuals to increase one block in coverage; the second took 39 individuals, while the third needed 70. All of these accounted for an average of 52.6 individuals in order to achieve one point in coverage and cause the program to crash with our malicious input. The information on the number of individuals vs. basic blocks hit can be visualized in the graph of Figure 12.

One interesting fact we noted while performing this experiment was that sometimes the algorithm was getting stuck in a local optima, like described in Chapter 5, section 5.2. In one particular case the algorithm degenerated rapidly and even with more than 2000 individuals it had not achieved one single extra unit in coverage.

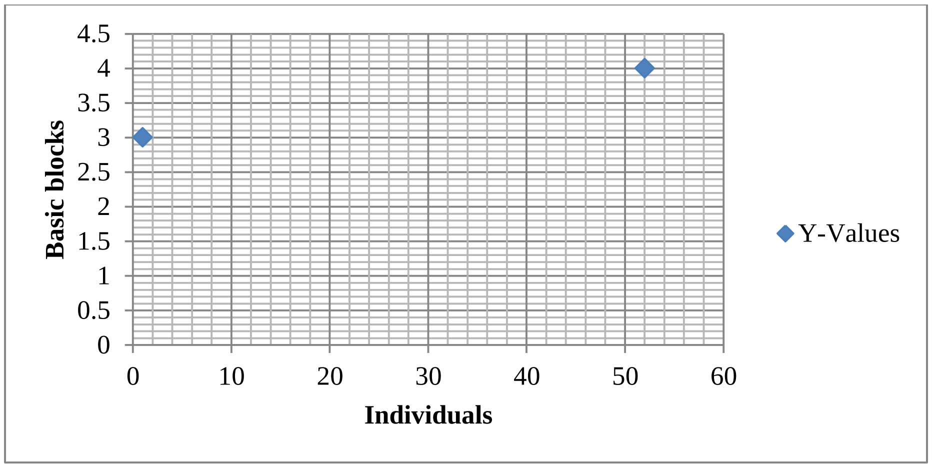


Figure 12. Results of the test run of example1.exe

We proceeded with testing *example2.exe* with the same methodology employed in the prior example. This sample has a more complex set of conditionals to be met in order to penetrate deeper in the code and we expected our fuzzer to generate a larger number of individuals compared to the previous test in order to achieve higher coverage. However, we were surprised to have been proven wrong at least in a few cases.

In our first test run we were able to achieve four blocks coverage in the 16th individual. In the second the 5th individual; in the third in the 2nd individual. This accounts for an average of 7.6 individuals, much lower than the previous example, which contains a simpler conditional to be met. In the same run we achieved five blocks coverage in the individual number 110, number 45 and surprisingly enough, number 3. When achieving such coverage our fuzzer is capable of triggering a buffer overflow vulnerability present in the code. Figure 13 shows a visualization of this data.

An earlier and unsuccessful experiment with the same binary had our search converging in the individual number 15548 without reaching the vulnerable block.

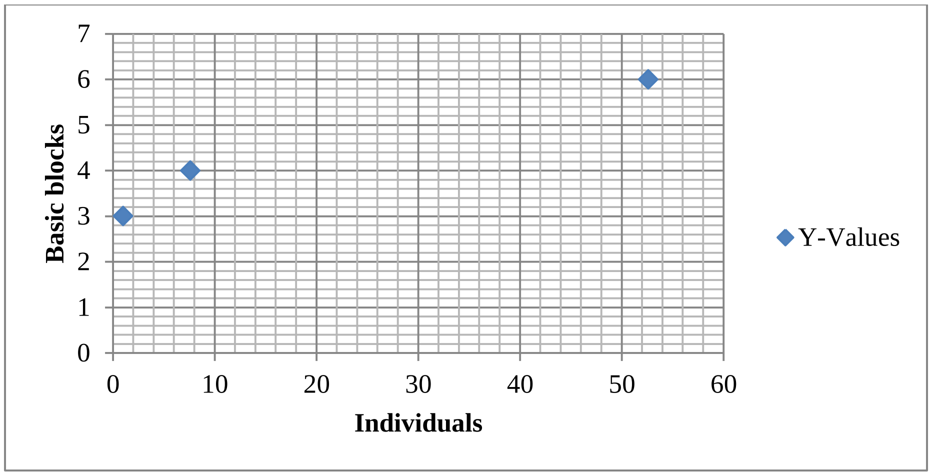


Figure 13. Results of the test run of example2.exe

Analyzing *example3.exe* means our system will have to satisfy constraints that are a lot more complicated than the ones presented in the previous examples. However, in theory genetic algorithms are capable to do that.

Just like all the other executables, this one also has the minimum execution of three basic blocks regardless of the input. For some reason unknown to the author, but possibly the manner in which the conditionals are disposed in the code, increase in code coverage is very irregular. For instance, we jumped from three blocks to six (individuals 26, 1599 and 678, respectively), but never saw it increasing in a linear way. After that we have reached eight blocks in the first run with the individual number 1905. All the other runs had the search converging to local optimas before they can reach this point. Figure 14 provides a visualization of this data.

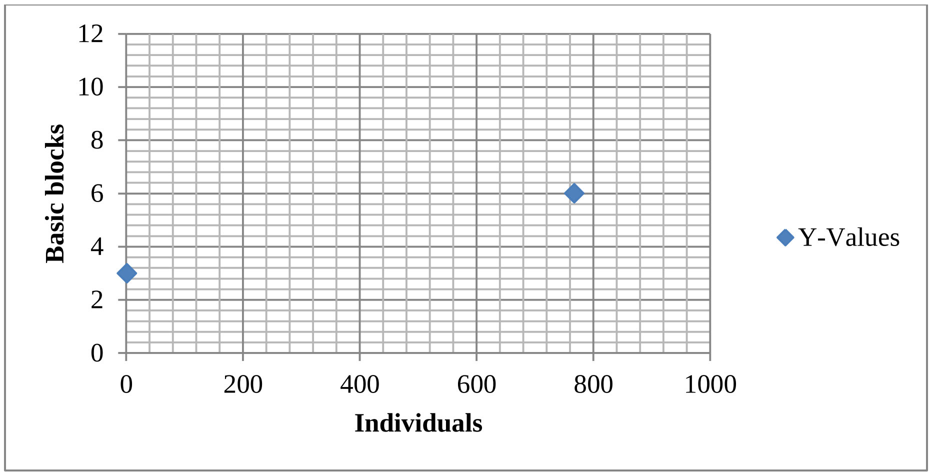


Figure 14. Results of the test run of example3.exe

One important aspect we observed during our test runs is that the fuzzer is more likely to go deeper within the code is directly linked with how soon the search algorithm finds more suitable inputs. To exemplify, the fuzzer has more chances to go deeper if the first increase in coverage was reported in the individual number 5, let's say, instead of the individual number 839, as it has less chance to degenerate the search.

6.4. Chapter summary

In this chapter we have put together all the concepts that were explored throughout this work. We initiated by laying down the concepts of evolutionary fuzzing and mentioning related research in this field. Right after we have formed the proposal for the architecture of a GA-enhanced fuzzing system, including a very close examination of all elements and what is our strategy to implement each of these and finally turn it into an evolutionary fuzzing tool.

We then proceeded to evaluate our system by performing some basic experiments. The results, however, were modest and sometimes did not reflect what was expected. Nevertheless, these outcomes suggest this technique has the potential to become relevant in the security testing scenario but needs more research and refinement for that.

7. CONCLUSION AND FUTURE WORK

This work had as core motivation bringing to the table a discussion regarding improvements to common software security testing methodologies. In the very beginning of this research a few months ago, we had in mind we would look for the answers to the questions of what are the current fuzzing technologies and identify weak points in them and what can be our contribution to enhance existing methodologies.

Initially we set the scene with an informative explanation on the impact of software in the modern society and why there is a real necessity to create robust and secure software. We outlined the fact that security is very rarely taken as priority in most of software development methodologies, and sometimes even completely neglected. The author personally believes that the recent security awareness mindset and great leap forward in the sense of security being baked into the software development processes will little by little foster more awareness among developers community. Nevertheless, this will certainly not be immediate as we would like to, but at least will slowly reduce the number of developers uneducated in writing secure code.

We proceeded on to argue that even mature software engineering processes cannot guarantee bug-free code, especially in large and complex software projects. This was the opened for providing a thorough discussion on different classes of software vulnerabilities with examples of past security weaknesses in real-life software, accompanied with a small analysis of each bug we used to illustrate.

Later we examined methodologies widely used in software testing practices, commenting on each of them and highlighting its strengths and weaknesses, as well as providing information on what is code coverage, how it can be measured and what is its value in the realm of software testing. We have used this as a “hook” to bring fuzz testing to the spotlight, describing the inner workings of this test technique and exploring common methodologies employed when fuzzing. Furthermore, we have presented a basic model of a fuzzer and talked about different types of fuzz data generation encountered in real world.

By now hints to the answers to the questions that drove us into this research started to appear. We then proceeded into investigating evolutionary computing and genetic algorithms to try to find how this can be beneficial for enhancing existing testing processes. We found out prior research into this field, but more focused on functional test instead of non-functional test, which is our priority, and also underscored the problems these search optimization algorithms are subject to. At this point we look at fuzzing from a different angle and started to believe finding suitable inputs can be molded as a search problem and fed into a genetic algorithm engine.

All the background information provided in this report culminated into our proposal for an evolutionary fuzzing system that used a genetic algorithm-powered engine to create better test sets with higher capability of traversing more of the target program under test.

As the main outcome of this research we have been able to develop a proof of concept system to illustrate and sustain our idea. This report also documents all architectural points and details required for the implementation of this system.

The outcomes of the experiments we have carried out in this research can be considered satisfactory from the proof of concept point of view. Although the results obtained were modest and sometimes unexpected, they suggest this technique can be refined to help improve current methodologies employed in fuzz testing. The author firmly believes evolutionary testing has a great potential and may become an important testing standard in the future.

Future directions for this research should first of all include the creation of a full-fledged fuzzing system instead of a simple prototype and capable of covering a larger number of vulnerability classes. After building it, especially in a modular fashion so features can be added or removed without much work from the software writing perspective, the genetic algorithm engine should be fine-tuned. For instance, we have encountered and reported many problems with the search algorithm converging prematurely and this could not have happened depending on the parameters fed into the GA engine.

Another architectural enhancement could be by using a DBI framework to replace many of tasks our system does with the aid of the debugger. For example, we could do code coverage regardless of having to rely on the program's CFG as we are currently doing. Furthermore, these frameworks have fine-tuned memory inspection capabilities that could detect other memory problems our userland debugger is incapable to, providing a higher degree of confidence and accuracy on tests.

Last but not least, the experiments should be carried out in a more appropriate way. First the test plan should be more elaborate and include real-life programs to see how the system performs in large applications. Second, the fuzzer should be benchmarked and compared against other fuzzing systems like mutation-based and generation-based and see whether it outperforms those or not.

BIBLIOGRAPHY

[1] James P. Anderson, *Computer security technology planning study - volume II.*, Hanscom AFB, Bedford, MA, 1972

[2] scut, *Exploiting Format String Vulnerabilities*, 2001
http://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf (last accessed Aug 10 2011)

[3] anonymous, *Once upon a free()…*, Phrack Magazine issue 57 article 9, 2001
http://www.phrack.org/issues.html?issue=57&id=9 (last accessed Aug. 19 2011)

[4] blexim, *Basic Integer Overflows*, Phrack Magazine issue 60 article 10, 2002
http://www.phrack.org/issues.html?issue=60&id=10 (last accessed Aug. 19 2011)

[5] jp, *Advanced Doug lea's malloc exploits*, Phrack Magazine issue 61, article 6, 2003
http://www.phrack.org/issues.html?issue=61&id=6 (last accessed Aug. 21 2011)

[6] Jack Koziol, et al. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*, Wiley & Sons, ISBN 0764544683, 2004

[7] Maura van der Linden, *Testing Code Security*, Auerbach Publications, ISBN 0849392519, 2007

[8] Michael Sutton, Adam Greene and Pedram Amini, *Fuzzing: Brute Force Vulnerability Discovery*, Addison Wesley, ISBN 0321446119, 2007

[9] ISS X-Force, *Internet Security Systems Alert – Code Blue Worm*, 2001
http://www.iss.net/threats/advise96.html (last accessed Jul. 17 2011)

[10] Mark Davis and Michel Suignard, *Unicode Security Considerations*, 2010
http://unicode.org/reports/tr36/ (last accessed Jul. 17 2011)

[11] Common Weakness Enumeration, *CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection').*
http://cwe.mitre.org/data/definitions/78.html (last accessed Jul. 24 2011)

[12] Greg Hoglund and Gary McGraw, *Exploiting Software: How to Break Code*, Addison-Wesley Professional, ISBN 0201786958, 2004

[13] Alexander Sotirov, *Decompiling the vulnerable function for MS08-067*, 2008
http://www.phreedom.org/blog/2008/decompiling-ms08-067/ (last accessed Jul. 24 2011)

[14] GNU Compiler Collection (GCC) Internals, *Control flow graph*
http://gcc.gnu.org/onlinedocs/gccint/Control-Flow.html (last accessed Jul. 27 2011)

[15] Microsoft Corporation, *Microsoft Security Development Lifecycle (SDL) – Process Guidance*,
http://msdn.microsoft.com/en-us/library/cc307418.aspx (last accessed Aug. 21 2011)

[16] Barton P. Miller, L. Fredriksen, and B. So, *An Empirical Study of the Reliability of UNIX Utilities*, Communications of the ACM 33, 12, 1990

[17] Michael, C.C., McGraw, G.E., Schatz, M.a., Walton, C.C., *Genetic algorithms for dynamic test data generation*, Proceedings 12th IEEE International Conference Automated Software Engineering, 1997

[18] Pargas, Roy P., Harrold, Mary Jean, Peck, R. Robert, *Test-Data Generation Using Genetic Algorithms*, Journal of Software Testing, Verification and Reliability, 1999

[19] Tim Kovacs, *Fitness landscapes*
http://www.cs.bris.ac.uk/Teaching/Resources/COMSM0305/04-landscapes.pdf (last accessed Aug. 12 2011)

[20] Andy Hertzfeld , *Monkey Lives*, 1983
http://www.folklore.org/StoryView.py?story=Monkey_Lives.txt (last accessed Aug. 18 2011)

[21] Ari Takanen, Jared D. Demott and Charlie Miller, *Fuzzing for Software Security Testing and Quality Assurance*, Artech House, ISBN 1596932147, 2008

[22] Marko Bozikovic, Marin Golub and Leo Budin, *Solving n-Queen problem using global parallel genetic algorithm*, Eurocon 2003 Ljubljana, Slovenia, 2003

[23] Toby Clarke, *Fuzzing for software vulnerability discovery*, 2009

[24] Ilja Van Sprundel, *Fuzzing: Breaking software in an automated fashion*, 2005

[25] Charlie Miller, *Babysitting an army of monkeys*, CanSecWest, 2010
http://securityevaluators.com/files/slides/cmiller_CSW_2010.ppt (last accessed Aug. 29 2011)

[26] Charlie Miller and Zachary Peterson, *Analysis of Mutation and Generation-Based Fuzzing*, 2007
http://www.securityevaluators.com/files/papers/analysisfuzzing.pdf (last accessed Aug. 25 2011)

[27] Google Security Team, *Fuzzing at scale*, 2011
http://googleonlinesecurity.blogspot.com/2011/08/fuzzing-at-scale.html (last accessed Aug. 29 2011)

[28] Scott Stender, *Blind Security Testing – An Evolutionary Approach Functional vs. Non-Functional Testing A Better Way to Create Test Cases Test Case Generation*, Black Hat USA, 2007
https://www.blackhat.com/presentations/bh-usa-07/Stender/Whitepaper/bh-usa-07-stender-WP.pdf (last accessed Aug. 31 2011)

[29] Wu, L I U Guang-hong, Zheng, Gang, Shuai, Tao, *Vulnerability Analysis for X86 Executables Using GA and Fuzzing*, Third International Conference on Convergence and Hybrid Information Technology, 2008

[30] ISO/IEC 9899, *Programming Languages – C*, 1999
http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf (last accessed Aug. 31 2011)

[31] Jared DeMott, Richard Enbody and William Punch, *Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing*, Black Hat USA, 2007

[32] Sherri Sparks, Shawn Embleton, Ryan Cunningham and Cliff Zou, *Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting*, Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), 2007

[33] Kevin Yu Zhang, *Evaluating Software Security Aspects through Fuzzing and Genetic Algorithms*, 2008

[34] Kevin Finisterre, *DMA[2005-0826a] - 'Nokia Affix Bluetooth btsrv poor use of popen()'* , 2005
http://www.digitalmunition.com/DMA%5B2005-0826a%5D.txt (last accessed Aug 31 2011)

[35] Marc Andreessen, *Why Software Is Eating The World*, Wall Street Journal, 2011
http://online.wsj.com/article/SB10001424053111903480904576512250915629460.html (last accessed Aug 31 2011)

[36] Wikipedia, *Heap spraying*
https://secure.wikimedia.org/wikipedia/en/wiki/Heap_spraying (last accessed Aug 31 2011)

[37] Glenford J. Meyers, *The Art of Software Testing – Second Edition*, Wiley, ISBN 0471469122, 2004

[38] IEEE Spectrum, *This Car Runs on Code*, 2009
http://spectrum.ieee.org/green-tech/advanced-cars/this-car-runs-on-code (last accessed Aug 31 2011)

[39] Simon Rogerson, *The Chinook Helicopter Disaster*, IMIS Journal Volume 12 No 2, 2002

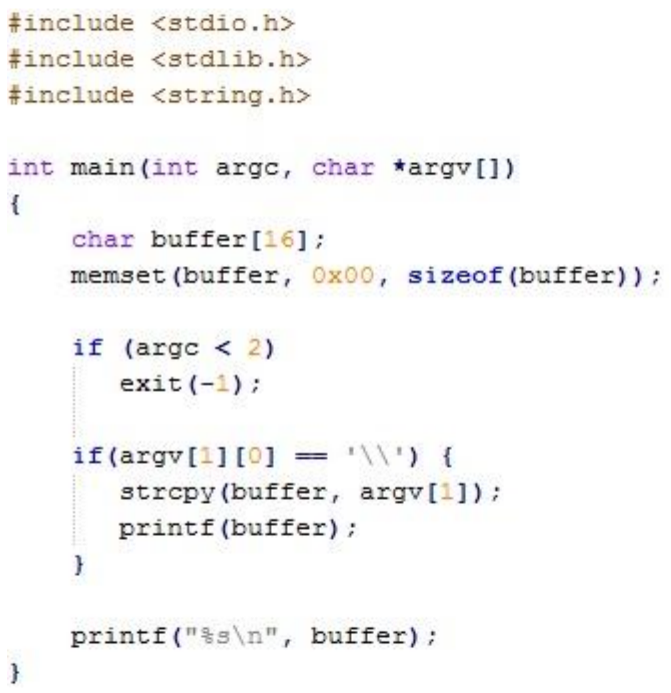
[40] NIST National Vulnerability Database, *Statistics Results Page*, 2011
http://web.nvd.nist.gov/view/vuln/statistics-results?cves=on&query=&cwe_id=&pub_date_start_month=-1&pub_date_start_year=2006&pub_date_end_month=-1&pub_date_end_year=2011&mod_date_start_month=-1&mod_date_start_year=-1&mod_date_end_month=-1&mod_date_end_year=-1&cvss_sev_base=HIGH&cvss_av=&cvss_ac=&cvss_au=&cvss_c=&cvss_i=&cvss_a= (last accessed Aug. 31 2011)

[41] F-Secure, *How we found the file that was used to Hack RSA*, 2011
http://www.f-secure.com/weblog/archives/00002226.html (last accessed Aug. 31 2011)

[42] ZDNet, *Conficker's estimated economic cost? $9.1 billion*, 2009
http://www.zdnet.com/blog/security/confickers-estimated-economic-cost-91-billion/3207 (last accessed Aug. 31 2011)

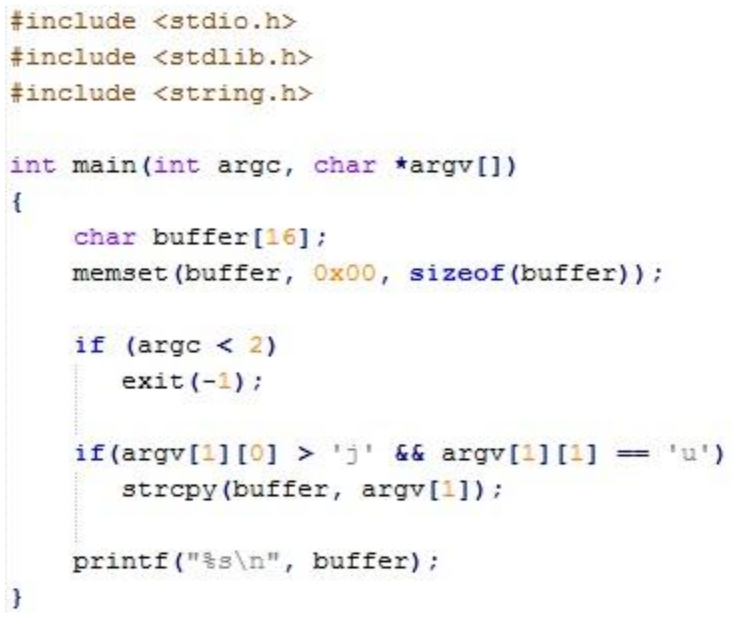
[43] Adobe Secure Software Engineering Team, *Fuzzing Reader – Lessons Learned*, 2009
http://blogs.adobe.com/asset/2009/12/fuzzing_reader_-_lessons_learned.html (last accessed Aug. 31 2011)

APPENDIX A



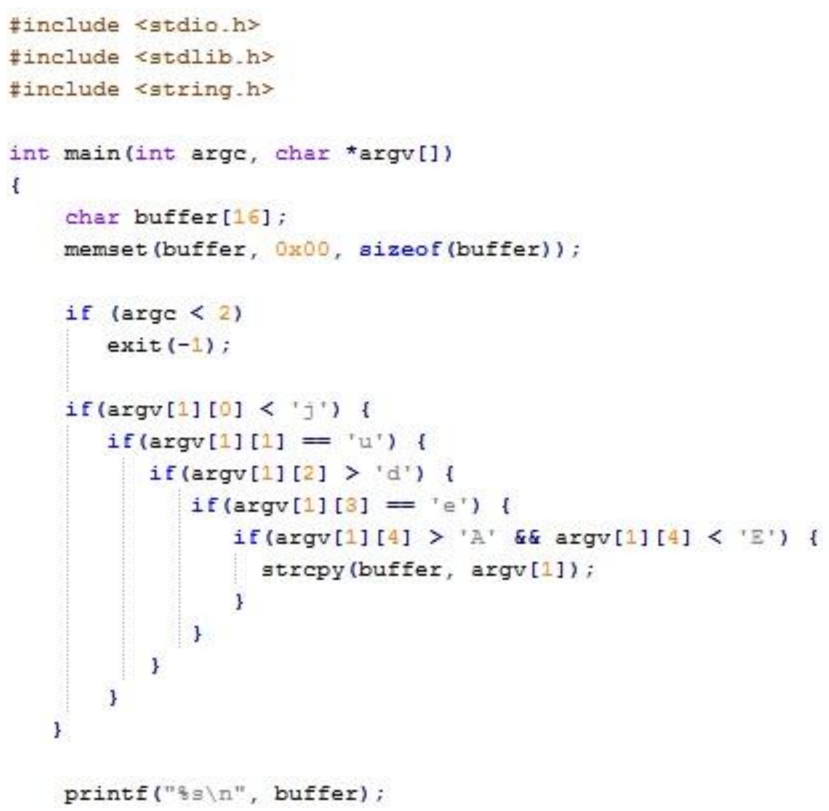
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char buffer[16];
    memset(buffer, 0x00, sizeof(buffer));

    if (argc < 2)
        exit(-1);

    if(argv[1][0] == '\\') {
        strcpy(buffer, argv[1]);
        printf(buffer);
    }

    printf("%s\n", buffer);
}
```

example1.c



```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char buffer[16];
    memset(buffer, 0x00, sizeof(buffer));

    if (argc < 2)
       exit(-1);

    if(argv[1][0] > 'j' && argv[1][1] == 'u')
       strcpy(buffer, argv[1]);

    printf("%s\n", buffer);
}
```

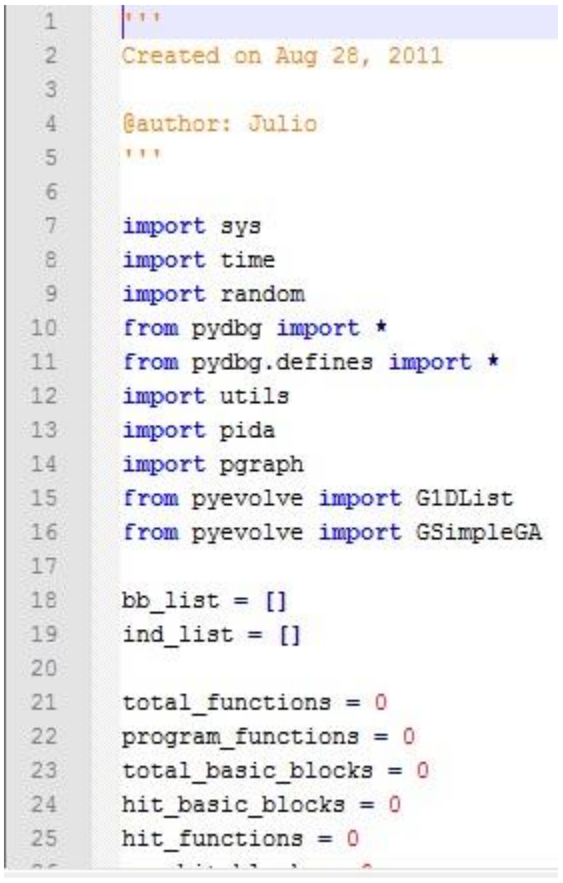
example2.c



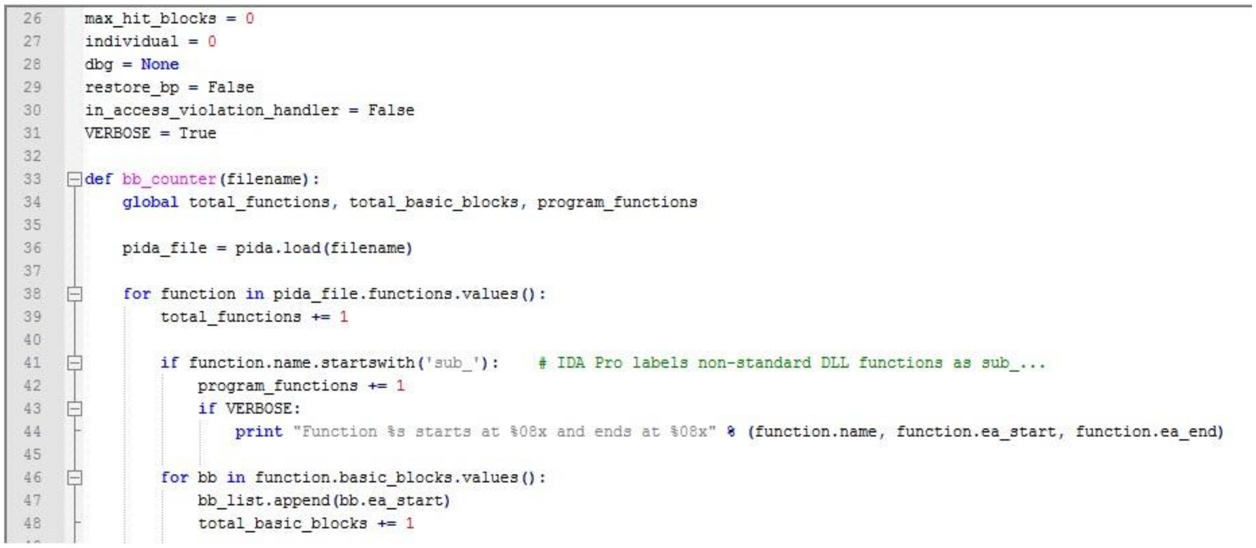
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char buffer[16];
    memset(buffer, 0x00, sizeof(buffer));

    if (argc < 2)
        exit(-1);

    if(argv[1][0] < 'j') {
        if(argv[1][1] == 'u') {
            if(argv[1][2] > 'd') {
                if(argv[1][3] == 'e') {
                    if(argv[1][4] > 'A' && argv[1][4] < 'E') {
                      strcpy(buffer, argv[1]);
                    }
                }
            }
        }
   }

    printf("%s\n", buffer);
```

example3.c

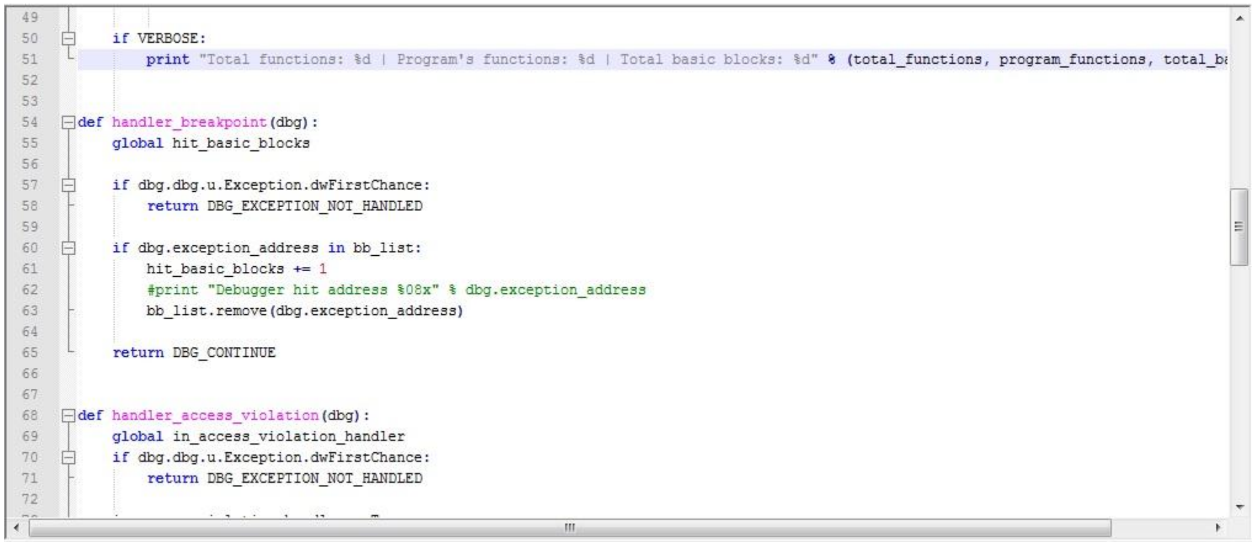
APPENDIX B



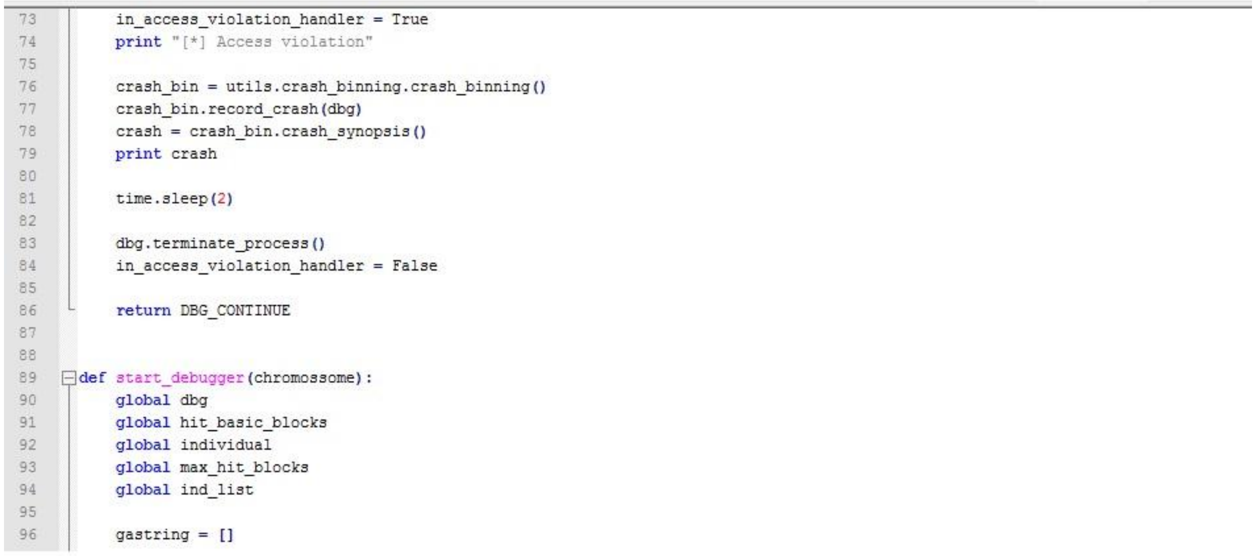
```
'''
Created on Aug 28, 2011

@author: Julio
'''

import sys
import time
import random
from pydbg import *
from pydbg.defines import *
import utils
import pida
import pgraph
from pyevolve import G1DList
from pyevolve import GSimpleGA

bb_list = []
ind_list = []

total_functions = 0
program_functions = 0
total_basic_blocks = 0
hit_basic_blocks = 0
hit_functions = 0
```



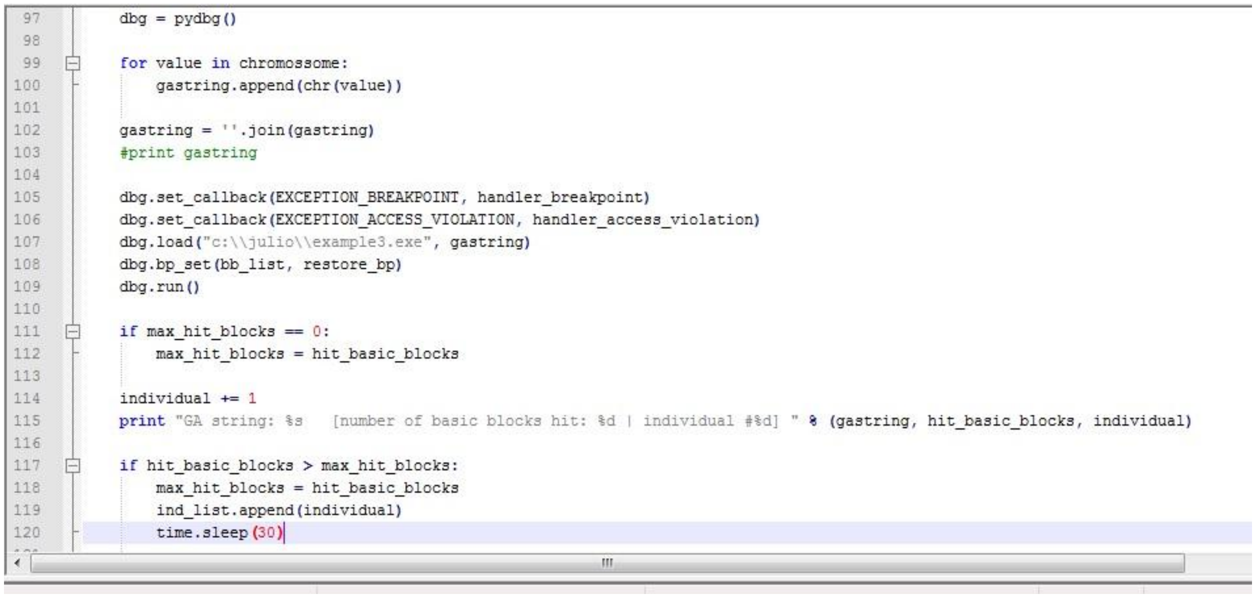
```
max_hit_blocks = 0
individual = 0
dbg = None
restore_bp = False
in_access_violation_handler = False
VERBOSE = True

def bb_counter(filename):
    global total_functions, total_basic_blocks, program_functions

    pida_file = pida.load(filename)

    for function in pida_file.functions.values():
        total_functions += 1

        if function.name.startswith('sub_'):    # IDA Pro labels non-standard DLL functions as sub_...
            program_functions += 1
            if VERBOSE:
                print "Function %s starts at %08x and ends at %08x" % (function.name, function.ea_start, function.ea_end)

        for bb in function.basic_blocks.values():
            bb_list.append(bb.ea_start)
            total_basic_blocks += 1
```



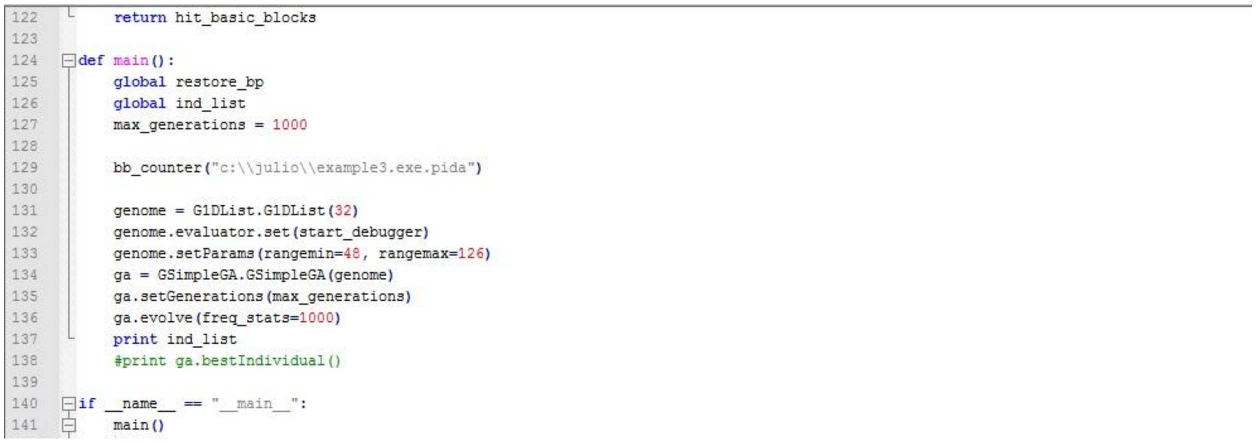
```python

    if VERBOSE:
        print "Total functions: %d | Program's functions: %d | Total basic blocks: %d" % (total_functions, program_functions, total_b


def handler_breakpoint(dbg):
    global hit_basic_blocks

    if dbg.dbg.u.Exception.dwFirstChance:
        return DBG_EXCEPTION_NOT_HANDLED

    if dbg.exception_address in bb_list:
        hit_basic_blocks += 1
        #print "Debugger hit address %08x" % dbg.exception_address
        bb_list.remove(dbg.exception_address)

    return DBG_CONTINUE


def handler_access_violation(dbg):
    global in_access_violation_handler
    if dbg.dbg.u.Exception.dwFirstChance:
        return DBG_EXCEPTION_NOT_HANDLED

```



```python
    in_access_violation_handler = True
    print "[*] Access violation"

    crash_bin = utils.crash_binning.crash_binning()
    crash_bin.record_crash(dbg)
    crash = crash_bin.crash_synopsis()
    print crash

    time.sleep(2)

    dbg.terminate_process()
    in_access_violation_handler = False

    return DBG_CONTINUE


def start_debugger(chromossome):
    global dbg
    global hit_basic_blocks
    global individual
    global max_hit_blocks
    global ind_list

    gastring = []
```



```
    dbg = pydbg()

    for value in chromossome:
        gastring.append(chr(value))

    gastring = ''.join(gastring)
    #print gastring

    dbg.set_callback(EXCEPTION_BREAKPOINT, handler_breakpoint)
    dbg.set_callback(EXCEPTION_ACCESS_VIOLATION, handler_access_violation)
    dbg.load("c:\\julio\\example3.exe", gastring)
    dbg.bp_set(bb_list, restore_bp)
    dbg.run()

    if max_hit_blocks == 0:
        max_hit_blocks = hit_basic_blocks

    individual += 1
    print "GA string: %s   [number of basic blocks hit: %d | individual #%d] " % (gastring, hit_basic_blocks, individual)

    if hit_basic_blocks > max_hit_blocks:
        max_hit_blocks = hit_basic_blocks
        ind_list.append(individual)
        time.sleep(30)
```



```
    return hit_basic_blocks

def main():
    global restore_bp
    global ind_list
    max_generations = 1000

    bb_counter("c:\\julio\\example3.exe.pida")

    genome = G1DList.G1DList(32)
    genome.evaluator.set(start_debugger)
    genome.setParams(rangemin=48, rangemax=126)
    ga = GSimpleGA.GSimpleGA(genome)
    ga.setGenerations(max_generations)
    ga.evolve(freq_stats=1000)
    print ind_list
    #print ga.bestIndividual()

if __name__ == "__main__":
    main()
```