# A Review of Fuzzing Tools and Methods

James Fell, james.fell@alumni.york.ac.uk

Originally published in PenTest Magazine on 10th March 2017

## 1 Introduction

Identifying vulnerabilities in software has long been an important research problem in the field of information security. Over the last decade, improvements have been made to programming languages, compilers and software engineering methods aimed at reducing the number of vulnerabilities in software [26]. In addition, exploit mitigation features such as Data Execution Prevention (DEP) [65] and Address Space Layout Randomisation (ASLR) [66] have been added to operating systems aimed at making it more difficult to exploit the vulnerabilities that do exist. Nevertheless, it is fair to say that all software applications of any significant size and complexity are still likely to contain undetected vulnerabilities and it is also frequently possible for skilled attackers to bypass any defences that are implemented at the operating system level [6, 7, 12, 66].

There are many classes of vulnerabilities that occur in software [64]. Ultimately they are all caused by mistakes that are made by programmers. Some of the most common vulnerabilities in binaries are stack based and heap based buffer overflows, integer overflows, format string bugs, off-by-one vulnerabilities, double free and use-after-free bugs [5]. These can all lead to an attacker hijacking the path of execution and causing his own arbitrary code to be executed by the victim. In the case of software running with elevated privileges this can lead to the complete compromise of the host system on which it is running.

Techniques for identifying software vulnerabilities can be divided firstly into two different approaches; static analysis and dynamic analysis. Static analysis of software involves ways of examining the source code or compiled binary without executing it. Dynamic analysis involves examining the software at runtime, typically after attaching some kind of debugger. Both of these approaches have their relative advantages and disadvantages [69].

For static analysis a number of automated tools exist that can be combined with manual code review by a skilled analyst. The less sophisticated tools essentially just scan the target source code looking for known dangerous functions such as strcpy() in C programs [31]. The more advanced tools often work using some kind of taint analysis [29]. These tools will identify and 'taint' any variable that has its value set from input that enters the target application from a user. This tainted input and its effect on other data will then be traced as it propagates through the source code. Whenever it is seen that tainted data could reach a 'sink' or potentially dangerous function this will be flagged for further investigation. Static analysis, although useful, often produces many false positives that cannot be exploited in practice and requires a lot of manual verification work to identify which issues are genuine vulnerabilities [69]. It does, however, allow for complete code coverage with the entire application being inspected.

For dynamic analysis the most common automated technique for finding vulnerabilities is the process of fuzz testing or fuzzing. In essence this consists of repeatedly giving an application invalid input and monitoring for any sign of this triggering a bug, such as the application crashing or hanging [67]. There are many advantages to this approach such as the ease of automation and the ability to test even very large applications where code review would be too time consuming. It is also the case that each bug discovered by fuzzing automatically comes with its own 'proof of concept' test case proving that the bug can definitely be triggered

by a user. In practice many serious and high profile memory corruption vulnerabilities today are discovered through fuzzing.

In order to fully test an application using fuzzing, it is important to somehow ensure that the combination of test cases being used execute as large a proportion of the application as possible. That is to say, it is necessary to maximise code coverage [71]. If large sections of code are not executed during the fuzzing process, it is certain that any vulnerabilities that do exist in those areas will not be found.

It is possible to fuzz any application that accepts input from the user, whether this is in the form of opening a file, reading from a network socket, reading environment variables or any other form of input that can be modified.

The motive for fuzzing related work, and indeed any research involving methods for discovering vulnerabilities, can be either defensive or offensive. Software companies use these techniques to identify vulnerabilities in their own products before release as part of the development lifecycle [26]. Whitehat security researchers also use these techniques to find vulnerabilities in both open source and proprietary software and then typically follow the established practice of responsible disclosure to publish the details after first notifying the vendor and giving them time to release a patch. On the other hand, various intelligence agencies, military units, defence contractors, and even organised crime groups also use the very same techniques to gain an advantage for their work compromising the networks of their targets. This type of organisation keeps the details of any vulnerabilities that they discover private rather than disclosing them to the affected vendors. Exclusive knowledge of vulnerabilities for which no patch exists (so called zero day or 0day vulnerabilities) is extremely useful when carrying out offensive work such as CNE (Computer Network Exploitation) [16].

The rest of this paper reviews fuzzing and its context within the field of information security research. We firstly examine how vulnerabilities come to exist in software and how security researchers find them. After a brief overview of common vulnerability types and methods of static analysis, we look in more depth at the field of fuzzing. Competing approaches to fuzzing are examined, from simple random inputs all the way to using genetic algorithms and taint analysis. The importance of measuring code coverage to evaluate the completeness of a fuzzing campaign is examined. Finally, previous work on fuzz testing of web browsers is reviewed.

# 2 Software Vulnerabilities

Modern software applications are often complex and can consist of hundreds of thousands or even millions of lines of code. Not surprisingly, mistakes are made that can lead to situations where attackers can cause software to behave in a way that was not intended by the developers.

For a very simple example, consider the following piece of C code:

```
#include <stdio.h>

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

        char user_buffer[20];
        printf("Please enter a string : ");
        gets(user_buffer);

        char *buffer2 = malloc(strlen(user_buffer));
        strcpy(buffer2, user_buffer);

}
```

Figure 1: Vulnerable C code example

There are (at least) two things wrong with the small program shown in Figure 1.

The first and most obvious mistake is that we have allocated 20 bytes on the stack for user_buffer but we do not check how much input the user has entered before writing it to this buffer. The string will be null terminated and so if the user enters any more than 19 characters we will write more than 20 bytes to the buffer. The stack can thus be corrupted by the user simply typing a longer string than we expected, a typical example of a stack based buffer overflow [62].

If we compile this program with gcc using the -fsanitize=address option to activate AddressSanitizer [17] and then run it and enter 20 characters we see the following.

```
user@laptop03:~/temp$ ./overflows
Please enter a string : abcdefghijklmnopqrst
=================================================================
==735==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffe86807b84 at
READ of size 21 at 0x7ffe86807b84 thread T0
    #0 0x7f78c2bdb567 in strlen (/usr/lib/x86_64-linux-gnu/libasan.so.1+0x31567)
    #1 0x400a51 in main (/home/user/temp/overflows+0x400a51)
```

Figure 2: Stack based buffer overflow being triggered

It can be seen that AddressSanitizer has identified a stack based buffer overflow as our input in this case causes data to be written out of bounds past the end of user_buffer.

The second problem is a little more subtle. Because strlen() returns the length of the user's string excluding the null terminator at the end, we are always allocating one byte too little for buffer2 on the heap with malloc(). This means that whatever string is entered by the user it will always result in a heap overflow [63].

Running the program a second time and this time entering “hello” gives the following result.

```
user@laptop03:~/temp$ ./overflows
Please enter a string : hello
=================================================================
==697==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60200000eff5 at p
WRITE of size 6 at 0x60200000eff5 thread T0
    #0 0x7faac1a3a01f in __interceptor_strcpy (/usr/lib/x86_64-linux-gnu/libasan.so
    #1 0x400ae8 in main (/home/user/temp/overflows+0x400ae8)
    #2 0x7faac1681b44 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21b44
```

Figure 3: Heap based buffer overflow being triggered

This time AddressSanitizer has identified a heap based buffer overflow as our 6 byte string (“hello\0”) is written to our 5 byte heap buffer.

Heap-based buffer overflows occur in a similar way to stack-based, except this affects buffers that have been dynamically allocated at runtime using heap memory. Rather than aiming to overwrite a function's return address, the aim when exploiting a heap overflow is to overwrite the metadata of other heap structures.

When the first crash, the stack based buffer overflow, is examined in GNU Debugger (gdb) using the exploitable plugin [39] from CERT we see it is classed as exploitable. So it is not just a bug, but an exploitable security vulnerability that can lead to execution of arbitrary code by an attacker.

```
Program received signal SIGSEGV, Segmentation fault.
0x000000000040067c in main ()
(gdb) exploitable
Description: Access violation during return instruction
Short description: ReturnAv (1/22)
Hash: 543ba29c28d3a06481f08f267946198c.543ba29c28d3a06481f08f267946198c
Exploitability Classification: EXPLOITABLE
Explanation: The target crashed on a return instruction, which likely indicates stack corruption.
Other tags: AccessViolation (21/22)
(gdb)
```

Figure 4: Stack buffer overflow classified as exploitable

These two very simple examples serve to illustrate how potentially exploitable bugs can arise in software through oversights or misunderstanding on the part of the programmer. It is these kinds of mistakes that we are interested in detecting when fuzzing.

As well as the stack based and heap based buffer overflows that we just saw, there are many other types of vulnerabilities that are found in software [64]. We now briefly describe four more of the most common types that are encountered during binary analysis and that are likely to be encountered during fuzzing.

Use-after-free (UAF) vulnerabilities occur when a buffer on the heap is accessed by mistake after it has already been released by the free() command [20]. This can happen when a program has multiple pointers to the same heap buffer, or to different offsets within it. One pointer may get used for a free() operation and if all the other pointers are not then set to NULL then one of them may later be mistakenly used to write to the memory and thus cause corruption. An example of this vulnerability in the real world is CVE-2014-0322 which is a UAF vulnerability in Microsoft Internet Explorer 9 and 10 [21]. This allows attackers to get arbitrary code execution via malicious JavaScript.

Double-free vulnerabilities occur when a buffer that has already been freed is mistakenly freed a second time, after that piece of memory has in fact already been reallocated to something else [23]. This results in an area of memory being marked as unused and hence available to be allocated, even though it is still in use by some other data structure. An example of this is CVE-2015-0058 which is a double free vulnerability in win32k.sys on Microsoft Windows 8.1 and Windows Server 2012 R2 that can be exploited to perform privilege escalation and execute commands as SYSTEM [22].

Integer overflows happen whenever an integer is increased or decreased past the range of values that it can hold [19]. For example, an unsigned 16 bit integer cannot hold a value greater than 65,535. If it has a value of 65,535 and then 1 is added, the value becomes 0 rather than 65,536. Similar issues can occur when converting between integers of different sizes. For example, converting from a 32 bit integer to a 16 bit integer results in the first 16 bits being truncated. Sometimes this behaviour is an expected part of how a program functions and does not cause any problem. In other cases though, it is an oversight on the part of the developer and it does cause problems. An attacker exploiting such an issue could for example cause a buffer to be allocated with less space than it actually needs and thus lead on to a buffer overflow. An example of a real world integer overflow is CVE-2010-2753 which is a vulnerability in the Mozilla Firefox web browser that leads to attackers being able to execute arbitrary code [18].

Off-by-one vulnerabilities occur when a miscalculation has been made that results in some value being one more or one less than it should be [25]. For example this could result in a buffer being one byte smaller than it should be or a loop being executed one more time than it should. This is actually quite a common mistake. The heap buffer overflow example presented earlier was also a good example of an off-by-one bug. The memory allocated on the heap to buffer2 was always one byte too small due to forgetting about the null at the end of the input string. This always resulted in the writing of a null byte out of bounds. A real world example of an off-by-one vulnerability is CVE-2012-5144 which allowed remote attackers to cause a denial of service condition against users of the Google Chrome web browser [24].

All of the vulnerabilities described so far can potentially lead to arbitrary code execution when exploited. They are also all typical vulnerabilities that are good candidates to be discovered by fuzzing as we shall see later.

A range of efforts have been made to reduce the frequency of programmers making mistakes such as these. Modern compilers check for many issues. For example, when compiling the buffer overflows C program example given in Figure 1 earlier, gcc 4.9.2 actually warned that “the gets function is dangerous and should not be used”.

Choice of programming language also has an impact on the likelihood of vulnerabilities being present as newer programming languages often make a considerable effort to prevent developers accidentally introducing vulnerabilities. For example, compiled C programs do not automatically perform bounds checking at runtime whereas Python, Ruby and C# programs all do. For applications programmed in the latter group of languages it is not possible to cause the software to write past the end of an array and cause a buffer overflow as an exception will automatically be raised. However, C or C++ is often chosen when speed and efficiency are important and so many key applications, including operating system internals and also web browsers, are written in that language. This means that plenty of the vulnerabilities that were described earlier are still present and waiting to be discovered in modern software.

Frameworks of best practices have been developed for programmers and software engineers to help make security a core requirement throughout the entire software creation process. One of the more popular examples of this is the Microsoft SDL (Security Development Lifecycle) [26]. This guides the creation of a set of processes from providing programmers with training to attack surface reduction to creating an incident response plan. Fuzzing is also included as part of the SDL.

# 3 Detection of Vulnerabilities

Now that we have set the scene by providing a brief introduction to how vulnerabilities arise we can shift our attention to how to discover them.

Techniques for detecting software vulnerabilities can be divided into two main categories; static analysis and dynamic analysis [69].

## 3.1 Static Analysis

Although the focus of this paper is fuzzing, a form of dynamic analysis, it is worthwhile briefly examining static analysis techniques first. The reason for this is that some fuzzing research has already made advances by taking techniques that are usually part of static analysis and integrating them into the fuzzing process [3, 4]. It is therefore useful to have some knowledge of static analysis while working on fuzzing.

Static Code Analysis (SCA) techniques usually involve examination of the source code of an application or the disassembly of the compiled binary. The important point is that the software under examination is not executed [72]. Analysing the source code or binary statically allows complete code coverage. It is possible to examine absolutely all of the program. The downside of this is that issues are often flagged up that cannot be reached by the user at runtime. From a security point of view these bugs are of no interest because they cannot be exploited. There is therefore the problem of triage; it is frequently necessary to sort through a large number of bugs to identify the small percentage that are of interest to a security researcher [69]. It is also the case that some vulnerabilities cannot be detected or understood without knowing about the runtime environment and thus are unlikely to be detected during static analysis. Another potential issue with static analysis is that vulnerabilities in third party libraries that are imported by the target program will be missed if these libraries are not also analysed. None of these problems exist when fuzzing as we shall examine later.

Some of the more popular free Static Code Analysis tools are Cppcheck [33] and FlawFinder [31] for finding vulnerabilities in C/C++ source code, RIPS [29] for PHP, and FindBugs [13] for Java. Popular commercial solutions, which can deal with software in multiple programming languages, include VeraCode [14] and Fortify [15]. Some tools are stand-alone while others (for example FindBugs) can integrate into common development environments such as Eclipse or Visual Studio.

SCA tools rely on a variety of methods to identify potential vulnerabilities. At the simpler end of the spectrum are tools that essentially just search for specific function names or other occurrences from a blacklist. The FlawFinder tool [31] works in this way. The tool has a database of C/C++ functions that are associated with buffer overflows, format string problems, race conditions and other known issues. The presence of any of these blacklisted functions in source code is flagged by the tool and presented to the user with an explanation of the likely vulnerability in each case. The Jslint tool, described by the developers in [30], detects vulnerabilities in Java source code by checking for compliance with a list of 12 best practices of secure programming. Such tools are useful for finding the 'low hanging fruit' but are unlikely to detect more complex issues.

A more sophisticated technique is taint analysis which is used by RIPS [29] and is also discussed in [34] in the context of testing Java applications. This process involves first labelling every piece of input to the program that can be modified by a user as a 'source' and marking any such input data as 'tainted'. Any data that is then derived from such tainted input is also tainted. The flow of tainted data can then be traced statically through the program and whenever it is seen to reach a dangerous function, labelled a 'sink', this fact is flagged. Such occurrences can then be checked to make sure that adequate sanitisation of the data is being carried out prior to it reaching the sink. Taint analysis has also been applied to fuzzing as we will examine later.

## 3.2 Dynamic Analysis

In contrast to the static analysis techniques that we have examined, dynamic analysis involves examining the application at runtime. This should be seen as complimentary to static analysis rather than something that should be done instead. While dynamic analysis does not have the drawbacks that we described for static analysis, it does have some of its own. The main shortcomings are the difficulty of getting full code coverage, the length of time required to perform a comprehensive test, and also the fact that we can only ever examine the application states resulting from a finite set of inputs rather than all possible inputs [73].

One dynamic analysis technique, fault injection, was first used as a hardware testing method and later applied to software. Fault injection is a close predecessor to fuzzing and involves testing a system's tolerance to having faulty states induced at runtime [74]. In a hardware context this could involve varying the voltage being supplied to a circuit beyond the range that it expects or placing it in a magnetic field. In a software context it could involve placing a system on a network between a client and server, and corrupting the traffic between the two to see how well they deal with this. Fault injection was originally aimed more at quality assurance than security research, and this brings us on to the technique of fuzzing.

# 4 Fuzzing

The term fuzzing was first used in 1988 by Professor Barton Miller at the University of Wisconsin. Whilst connected to a University terminal via a dial-up connection during a storm, Miller noticed that line noise was causing extra, random characters to be added to the commands that he was sending [9]. In addition to the obvious usability problems that one would expect from this, it was observed that the corrupted inputs often caused the various UNIX utilities that he was using to crash and 'core dump'.

This experience motivated Miller and other researchers at the University to create a tool for generating random strings and then pipe its output into the input of a range of UNIX utilities. The group tested almost ninety different UNIX utilities this way on seven different UNIX distributions and found that they were able to crash over 24% of them. These experiments were described in detail in [9] and gave birth to a relatively simple way to supplement the formal verification and software testing techniques that already existed.

Since Miller's early work, fuzzing has become steadily both more popular and more sophisticated. One important step forwards for the popularity of fuzzing was the 2001 public release of the SPIKE fuzzing tool by Dave Aitel followed by his presentation of it at the Black Hat USA conference in 2002 [10]. The SPIKE framework was an API and set of tools for network protocol fuzzing. It allowed the user to create a model of any network protocol and then use this model to create and send traffic that almost complied with the specification, but not quite. It proved quite successful at finding vulnerabilities in servers and lowered the barrier to entry for people interested in running their own fuzzing campaigns.

A successful fuzzing framework consists of more than just a way to repeatedly create invalid inputs and pass them to the target application. Some way of instrumenting the target is required in order to monitor what is happening internally as each test case is processed. Crashes and other misbehaviour must be detected and also logged and reported to the user along with as much data as would be necessary for further manual investigation. There must also be a test harness that can automatically restart the application as necessary so that the fuzzing campaign can run unsupervised. Better frameworks will also handle issues such as crash deduplication [97], crash triage based on suspected exploitability [39], and test case minimisation [38].

In [40] and [102] we see two different studies in which a number of popular free fuzzing tools are evaluated and compared to each other. This work illustrates well that there are many different kinds of fuzzers and they are suitable for different situations. There is no single tool that provides the best solution in every situation.

We will now look in more detail at the different aspects of creating a successful fuzzing tool and the different approaches that can be used.

## 4.1 Instrumentation

Fuzzing cannot be successful without some form of instrumentation being applied to the target application. At the very least, there must be some way of detecting when the program crashes and logging which test case caused the crash. However, far more than this can be achieved using more advanced instrumentation. It is possible to detect subtle forms of memory corruption and other errors even when they do not cause the program to crash outright [68]. This allows the detection of vulnerabilities that would otherwise be missed. It is also possible to measure code coverage and therefore know which parts of the target application have been tested so far and which have not [71]. This can be simply for informational purposes for the user or can actually be acted upon by the fuzzer itself when creating new test cases.

Instrumentation can be provided directly by the fuzzing framework or by using a separate third party tool. If a fuzzing framework does not have its own built-in instrumentation, a range of existing tools can be made use of whether the target's source code is available or not.

When no source code is available it is common to use dynamic analysis tools such as Valgrind [36], DynamoRIO [75] or PaiMei [37]. It would also be possible to simply attach a debugger such as IDA Pro [79] to detect when the target crashes. These tools can all attach themselves to an already running process and add instrumentation on the fly.

Valgrind runs on Linux and contains its own memory error detector and two thread error detectors [76]. This allows it to be used during fuzzing to detect memory corruption even when it does not cause the program to crash. DynamoRIO can run on Windows as well as Linux and can be used to extract code coverage

information among many other things [75]. PaiMei is a reverse engineering framework that can also be used for monitoring code coverage of Windows applications through its Pstalker module [37] and Flayer is a tool built on top of Valgrind which allows taint analysis to be performed at runtime [77]. All of these tools can be made use of for instrumentation needs when fuzzing a target for which no source code is available.

If source code is available then instrumentation can be added at compile time. For example, on Linux systems AddressSanitizer allows instrumentation for detecting memory errors such as buffer overflows and use-after-free to be added as the application is compiled [68]. If at any point during execution of an Asan instrumented program there is any memory corruption detected, the program will be terminated and a crash trace will be logged along with an automatic diagnosis of what occurred. This has two benefits. Vulnerabilities will be detected that may not ordinarily have crashed the program, and also further investigation for the purpose of either fixing the problem or developing an exploit will be aided by the data provided by Asan.

Some fuzzers, notably AFL (American Fuzzy Lop), add their own custom instrumentation at compile time [41]. This allows the fuzzer to get precise feedback from the target application during the execution of each test case. The creation of future test cases can then be based on this feedback. Such evolutionary based fuzzing is explained in more depth in section 4.3.

Finally, it should be noted that all forms of instrumentation slow the target program down, often significantly [78]. It is therefore important to consider what is needed for each specific fuzzing scenario as there is a trade-off between more speed and better instrumentation.

### 4.1.1 Code Coverage

As already mentioned, code coverage refers to tracking which parts of a program have been executed. This is a very important type of instrumentation to have during a fuzzing session as it makes it possible to see how much of the target has actually been tested. Vulnerabilities cannot be found in a section of a program if that section does not get executed by any of the test cases.

In [94] research was carried out showing that measuring the code coverage of an initial corpus and adding new test cases to it until no further coverage can be gained, then minimising the corpus before a fuzzing campaign starts is highly beneficial and increases the number of vulnerabilities discovered. This was confirmed in [98]. Further, specific to browser fuzzing it was documented in both [61] and [100] that code coverage for an initial web browser test corpus can be increased by including existing conformance test suites from the browser vendors as these are designed to test as many parts of the browser as possible.

There are three common approaches to measuring code coverage; line, branch and path coverage [71].

Line coverage keeps track of which individual lines of source code have been executed. This is the most basic kind of coverage information that can be collected. The shortcoming is that for example a conditional statement can be marked as executed as soon as the condition is tested, whether or not it evaluated to true, if it is all on one line. To give a pseudo code example, take the following line.

If (x < 10) then y=8 else z=9

This line will be marked as executed whatever the value of x is, and therefore we will not know whether the bit of code that assigns a value to y or to z was actually executed and thus tested.

Branch coverage deals with this by keeping track of which branches have been taken for each conditional jump in the program. In the pseudo code example above, two different test cases would be required to get full branch coverage; one where x is less than 10 and one where it is not. If only one branch had been executed during our testing we would know this fact if using branch coverage and we would know which

branch had not been tested.

Path coverage is even more detailed and keeps track of which different paths have been taken through the program, meaning which sequences of lines and branches have been executed and in which order.

It obviously takes more test cases to achieve full path coverage than full branch coverage, and more test cases to achieve full branch coverage than full line coverage. “In general, a program with n reachable branches will require 2n test cases for branch coverage and 2^n test cases for path coverage” [71].

The gcov tool on Linux is a popular tool for adding code coverage instrumentation when the target's source code is available. It is now built in to the gcc compiler and can be activated by using the “-fprofile-arcs -ftest-coverage” option when compiling [107]. Additionally there is a tool called lcov that takes the output of gcov and produces user friendly HTML reports from it including annotated source code. Using gcov provides line, function and branch coverage data but not path coverage. The report from lcov will show how many times each line, function and branch was executed and will flag up code that did not get executed.

As mentioned in section 4.1 various tools such as Valgrind can be used to get coverage data when the source code is not available. In that case we would be reasoning about lines of assembly language from a disassembled binary rather than lines of source code.

We will return to the subject of code coverage when we examine evolutionary fuzzing in section 4.3 but it is clear that we can only find vulnerabilities in the segments of code that we actually execute during our fuzzing and this depends entirely on the test cases that we make use of.

## 4.2 File Format and Protocol Fuzzing

It is possible to apply fuzzing to any kind of input that the application being tested can consume. For example, environment variables, strings passed in on the command line, or even file metadata [104] can be fuzzed. Typically though, most fuzzers are either file format fuzzers or protocol fuzzers.

File format fuzzers are used for fuzzing any application that receives a file as an input. Image viewers, movie players, PDF readers, word processors and audio file players are common examples of applications that are fuzzed in this way. A series of slightly malformed files generated by the fuzzer would be opened in the target application one after the other, while monitoring for error states. An example of a file format fuzzer is American Fuzzy Lop [43].

Protocol fuzzers are used to fuzz clients and servers that communicate over a network, typically using TCP/IP although any protocol stack can be fuzzed. Malformed packets are repeatedly sent to the target and its state is monitored. Mail servers, web servers and FTP clients are all examples of targets that would be fuzzed in this way. An example of a protocol fuzzer is SPIKE [10].

Some targets actually lend themselves to being fuzzed by either type of fuzzer. For example, web browsers can receive input over the network as a response to a HTTP request, or as a file opened from the filesystem. In the former case the HTTP headers could be included as part of the testing as well as the actual web page content.

## 4.3 Synthesis of Test Cases

Although issues such as instrumentation and efficiency are important, probably the most crucial aspect of a successful fuzzing campaign is creating good test cases that will interact with the target in a way likely to expose any defects. There are three main approaches to creating the test cases or inputs for use in a fuzzing campaign; mutation, generation and evolution [71, 99].

Mutation fuzzing works by starting with a corpus of one or more valid input samples. For example, when fuzzing an image viewer one might start with a number of valid JPEG files. These are then repeatedly modified or corrupted to produce new test cases that are each given as input to the software being tested. This is sometimes referred to as 'dumb fuzzing' as the fuzzer has no specific knowledge of the program being fuzzed or the correct format or syntax to use for inputs.

There are many ways in which the test cases can be mutated. Walking bit flipping and byte flipping involves inverting sequences of bits at different places in the test case [81]. Deleting segments of the test case and also splicing in sections from other test cases are common mutation approaches [80]. Another approach is to select sections of the test case and repeat them by inserting copies at random positions [100]. A recent example of a mutation fuzzer is Zulu from NCC Group [81].

Generation fuzzing works by creating test cases based only on some kind of model that describes a valid input, such as a grammar of a programming language or a specification of a binary format. Continuing with the image viewer example, with a specification for the PNG image format it would be possible to repeatedly generate slightly invalid PNG files to use as test cases without needing to use any samples of existing files. This is sometimes referred to as 'smart fuzzing' as the fuzzer is applying detailed knowledge of the format or protocol being fuzzed. This generally results in better code coverage and deeper testing of the target program [2]. An example of a generation fuzzing framework is Sulley [84].

Evolutionary fuzzing creates new test cases based on the response of the target program to previous test cases [103, 105]. This can be an extension of either mutation or generation fuzzing, as either technique can be guided to create the new test cases. In either case, the difference with evolutionary fuzzing is the way in which test cases are created based on feedback from the instrumentation so that they evolve towards a specific goal [99]. Based on that goal and also the precise way in which it is achieved, evolutionary fuzzing can itself be subdivided into three common approaches; coverage-based , taint-based and symbolic-assisted fuzzing.

Coverage-based evolutionary fuzzing attempts to evolve test cases that result in new lines, branches or paths being executed in the target. Test cases that exercise new code are favoured over those that do not. An example of a popular coverage-based evolutionary fuzzer is AFL (American Fuzzy Lop) [43]. AFL carries out file format mutation fuzzing and so is initially seeded with examples of valid input files. It then measures code coverage for each test case and uses Genetic Algorithms [108] to evolve a population of test cases that cover the maximum amount of code. This approach has proved extremely successful in real world fuzzing campaigns with AFL discovering many serious vulnerabilities in many popular applications [41]. In addition, an experiment was carried out that involved fuzzing the djpeg utility on Linux with AFL using only a simple text file containing the word “hello” as its initial corpus. This text file was seen to evolve into a valid JPEG file over a couple of days, simply by AFL observing how different mutations to the original test case resulted in different code paths being executed in the target [42]. In effect, the fuzzer was able to learn the JPEG file format from scratch.

Taint-based evolutionary fuzzing attempts to evolve test cases that exercise specific known dangerous code paths. Instead of simply aiming for maximum code coverage, the aim is to evolve test cases that will propagate user input into specific target functions within the program [96]. The taint analysis can either be performed beforehand as a static technique or can be performed dynamically during the fuzzing. Lanzi et al created a prototype system that first uses static analysis to identify specific paths of execution that lead to dangerous functionality and hence potential vulnerabilities [3]. The fuzzing stage is then guided by this information in an attempt to reduce the test space and choose inputs that will exercise those specific code paths thus “driving the program into corner states suitable to memory attacks” [3].

Similar to this, Sofia Bekrar et al worked on a system that uses dynamic taint analysis to assist fuzzing [4,

85]. The idea behind this is to allow a mutation fuzzer to focus its mutations on the areas of the test cases that are most likely to expose a vulnerability. These areas can be identified by tainting all user input and then tracing which inputs reach dangerous or potentially vulnerable sinks. The work carried out in [3] and [4] is therefore quite similar in concept, the main difference being that the former applied the taint analysis to the binary without executing it whilst the latter carried out the analysis on the binary at runtime.

Symbolic-assisted evolutionary fuzzing combines symbolic execution [82] with fuzzing in an attempt to enjoy the strengths of both. As with coverage-based fuzzing, its goal while generating new test cases is to increase code coverage. However, while coverage-based fuzzing can monitor code coverage and increase it through some form of trial and error (even using genetic algorithms falls into this category), symbolic-assisted fuzzing goes further. Test cases are modified using a dynamic symbolic execution engine to make changes that are known to result in the execution of specific segments of previously uncovered code [83]. An example of a symbolic-assisted fuzzer is SAGE (Scalable Automated Guided Execution) which is a proprietary tool that Microsoft uses in-house to search for vulnerabilities in their own software [1].

There are strengths and weaknesses to be considered for all of the approaches that we have examined. Creating a mutation fuzzer is generally quicker and easier than creating a generation fuzzer as far less knowledge is needed about the target application. Additionally, this means that one mutation fuzzer can often be used to fuzz a range of different target applications without modification [99]. A generation fuzzer however will either need modifying for each target or at least a new input model or grammar will need to be specified each time. It is also the case that as long as a sample of valid input can be obtained, even if the specification is proprietary and unavailable it is possible to carry out mutation fuzzing, whereas it may be impossible (or at least require a lot of reverse engineering skills) to carry out generation fuzzing.

The mutation approach however can run into problems with things like checksums where invalid inputs can be quickly rejected by the program under test without getting processed fully. In these cases a common approach is to comment out the sanity checks in the target if the source code is available. This will make it accept and process inputs even when items like checksums or the file's 'magic number' are invalid. If the source code of the target is not available then a specific test harness may need to be written to process the output of the fuzzer and fix checksums before forwarding it to the target. This is not a problem when using generation fuzzing as of course the fuzzer itself will be able to ensure that each test case is at least valid enough to pass any such tests.

Mutation fuzzers are also heavily dependent on the quality of the initial corpus of test cases used to seed them. Without an excellent corpus to start from they are unlikely to test as deeply as generation fuzzers. For example, if a protocol being fuzzed consists of 100 different commands, but the sample we give to the mutation fuzzer only contains 5 of them, we will not have a very thorough fuzzing campaign. There may be certain parts of a file format or a protocol that are valid but are rarely seen in practice and so may be absent from a typical corpus. With a generation fuzzer we would supply a detailed specification of the protocol and hence the full set of commands would be included in test cases. Developing this model of the file format or network protocol can be very time consuming though [40] and it is also possible to miss out undocumented features if the model is not created using reverse engineering. In [94] undocumented HTTP responses were discovered in a corpus that was created by web crawling showing that sometimes mutation fuzzing can provide coverage that would actually be missed by generation fuzzing.

In [2] experiments were performed to compare the code coverage obtained from mutation and generation fuzzing in the context of targeting a PNG image file reader. It was found that several parts of the PNG format's specification were not in common use and therefore not present in any of the PNG files downloaded from the web. This means that significant parts of a typical PNG image viewer would never be executed when loading readily available images. The only way that the missing pieces of the PNG format would

appear in the initial corpus and thus these functions could be tested during fuzzing would therefore be by using a generation fuzzer. The authors concluded that “applications often contain large sections of code that will only execute with uncommon inputs”. In this situation the experiments found that generating test cases resulted in 76% more code coverage than mutation of PNG files downloaded from the Internet [2].

No academic literature was found researching if the experimental results seen in [2] hold true for HTML files with respect to code coverage in web browsers. It may be that the common practice of downloading a selection of random web pages from the Internet to form an initial corpus for browser fuzzing [94] is actually missing out some valid HTML tags and attributes and therefore not getting the best code coverage possible.

Evolutionary fuzzing approaches are generally seen as the future of fuzzing and have been shown in practice to be better at finding vulnerabilities than using mutation or generation alone without incorporating feedback from the instrumentation. Building new test cases intelligently based on the observation of how previous test cases caused the target to behave allows general code coverage to be maximised or specific target functions to be reached. Implementing such a fuzzer can of course often be far more complicated and difficult than creating a less sophisticated tool. One aid of note for this task is LibFuzzer which is a C++ library for producing coverage-based evolutionary fuzzing tools [86].

Comparing subcategories of evolutionary fuzzing, coverage-based and taint-based fuzzing can both run into problems due to not having semantic understanding of the target. The fuzzer may know which code branches it would like to reach, but not necessarily how to modify the test cases so as to reach them [69]. In practice, genetic algorithms have been found to often cope with this issue rather well. The population of test cases can be optimised using a coverage-based fitness function to evolve a corpus with the desired properties [43]. Nevertheless, adding dynamic symbolic execution does allow the fuzzer to reason about how to reach specific code branches more directly. The target is executed in an emulated environment and constraints on variables are tracked. From this it is then possible to craft inputs that drive execution into the desired branches [87]. The TaintScope fuzzing tool documented in [88] combines dynamic taint-analysis with dynamic symbolic execution and is thus able to not only locate dangerous areas of the target but also reason about how to craft inputs so that the fuzzing session can reach them. A similar approach is taken by Confuzzer in [106].

## 4.4 Fuzzing Web Browsers

The web browser is an ubiquitous piece of software used by everyone who has a computer. It is also routinely used to open untrusted input every time it is used to search for information and visit new web sites. Vulnerabilities in web browsers that allow remote code execution are therefore extremely dangerous and can rapidly result in wide scale compromise of users around the world. Plenty of examples from recent years can be found of web browser vulnerabilities being used for everything from stealing banking information to state sponsored espionage campaigns [46].

When fuzzing web browsers to find vulnerabilities it is common to focus on the rendering engine [44, 57]. This part of the browser is responsible for taking the HTML documents, CSS and images that make up a web page and turning them into what is displayed to the user. The rendering engine parses HTML and CSS and then constructs a tree structure from this using the DOM (Document Object Model) API. Scripting languages such as JavaScript can then interact with the rendered document through DOM API calls to make changes to it on the client side without further connections to the web server being necessary. Popular rendering engines in use today include EdgeHTML, Trident, Gecko, WebKit and Blink (itself a fork of WebKit) [47]. The rendering engine is a common location to find exploitable UAF (Use-after-free) vulnerabilities [56].

There are two main approaches to fuzzing a web browser's rendering engine; file format fuzzing and DOM fuzzing. The file format fuzzing approach consists of simply applying the mutation or generation techniques

that we have already examined to create malformed web pages and providing these to the browser as input. This is therefore applying a generic fuzzing approach and using the same methods that one might use to fuzz a PDF reader or a video player.

The DOM fuzzing approach applies techniques that are specific to web browsers [44, 59] and the actions are carried out by the browser itself through JavaScript. There are many variations but a typical approach is to crawl the DOM tree of a test case document and collect element references. The elements' attributes can then be mutated and random DOM nodes can also deleted. It is also possible to perform the mutations on collections rather than individual DOM elements. The DOM tree can be rearranged in various pseudo-random ways. Garbage collection is then triggered somehow and the process is repeated [59, 60]. This can result in the discovery and triggering of various memory corruption bugs.

Creating logical views from a document tree can also be done as part of DOM fuzzing as it forces the rendering engine to attempt to maintain consistency between the logical views and the DOM as the mutations occur. This activity can also result in memory corruption bugs being triggered and revealed [59].

DOM fuzzing tools in general have had a problem with producing test cases to efficiently reproduce crashes when they are discovered. A typical DOM fuzzing tool builds up the DOM tree by randomly selecting HTML tags from an array and calling document.createElement with each. The built-in PRNG in JavaScript (Math.random) is often used which cannot be explicitly seeded with a known value. The script then does some mutations to the DOM tree (again often involving the use of Math.random) which may cause a crash, and if the browser does not crash the script reloads the page and starts over. This continues for perhaps several hours until eventually the browser crashes. When the browser does crash there is not a specific test case available with which to reproduce the crash. This is in contrast to loading a series of mutated HTML files into a web browser, where any crash can immediately be traced back to an exact test case.

Nothing can be written to disk from JavaScript within a web page for obvious security reasons. Most DOM fuzzers instead use the console.log() method from JavaScript to log messages to the browser's console [133]. This information can then be used to try to investigate what happened and reconstruct a test case. This is far from ideal compared to traditional fuzzing where any time a crash occurs the specific test case that caused it is available right away. The consequence of this has been that DOM fuzzers have been good at triggering bugs in browsers but investigating the bug to determine the root cause and fix it is often challenging.

## 4.4.1 Existing Browser Fuzzing Tools

Several tools exist that were created specifically to fuzz web browsers. We now examine some of the more successful ones and where enough detail can be found contrast their differing approaches to the problem.

The earliest browser fuzzing tool of note was Mangleme which was released in 2004 by Michal Zalewski [89]. This tool was a generation fuzzer that produced malformed HTML pages. It was implemented in C and deployed as a compiled CGI program that would run on a web server. By navigating to the CGI program's URL in the web browser to be tested, the page would repeatedly refresh and a different HTML test case would be processed each time. The test case generation worked by referencing a two dimensional array containing all the possible HTML tags and all the attributes of each tag. Malformed HTML pages were constructed by selecting random tags and adding random attributes from the array, while also scattering some random characters and bytes throughout. The whole tool consisted of about 200 lines of C. Despite the simplicity of this tool it found serious vulnerabilities in all the main web browsers of the time.

HD Moore of the Metasploit project was involved in the release of two web browser fuzzers in 2005 and 2006. Hamachi was a tool used to fuzz Dynamic HTML (DHTML) by mutating its elements, properties and attributes [91]. CSSDIE was used to fuzz CSS and was released as part of the July 2006 'Month of Browser

Bugs' [92]. Both tools found vulnerabilities in all major browsers at the time. Unfortunately, precise details of how they worked and their source code both appear to no longer be available.

Another CSS fuzzer simply called 'CSS grammar' was released in 2009 by Jesse Ruderman [55]. This is a generation fuzzer written in JavaScript that includes a context free grammar [93] for Cascading StyleSheets. The grammar can be extended by the user (or even replaced in order to target some other language) and it is also possible to assign weights to different symbols in the grammar to influence how regularly they will appear in test cases. As with Mangleme [89], this generation fuzzer follows the grammar for creating most of the test case but also throws random symbols in and makes other changes to produce slightly malformed input. The tool was used to find several bugs in the Gecko rendering engine.

Radamsa is a general purpose mutation fuzzer, developed by researchers at the Oulu University Secure Programming Group (OUSPG). Although it is a general purpose fuzzer, it has been used effectively to target web browsers as documented in [100]. Radamsa uses model-inference assisted fuzzing, that is to say it attempts to build its own model of the input based on the valid inputs in the initial test corpus that it is given. It then uses this model to apply mutations that are likely to be valid enough to be processed properly by the target rather than being immediately rejected as invalid. This allows the fuzzer to increase the depth of testing it is capable of whilst still not requiring the user to provide anything more than some valid input samples.

Radamsa was applied to web browser fuzzing in 2010 and 2011 as described in [100]. It was used to mutate HTML, CSS and JavaScript as well as PDF files, Flash and also image formats such as PNG and JPEG. All of these file formats were used as web browser test cases. The test cases were obtained first by random web crawling and then by downloading various conformance test suites. From this it was assumed that code coverage would be quite high, although nothing was done to actually measure this or try to increase it even further. It is possible that if this had been done the work would have been even more successful. Approximately 60 new bugs were found in a range of web browsers.

Crossfuzz is a DOM fuzzer released in 2011 by Michal Zalewski [48]. The tool works by “dynamically generating extremely long-winding sequences of DOM operations across multiple documents, inspecting returned objects, recursing into them, and creating circular node references that stress-test garbage collection mechanisms”. The tool is written in JavaScript and can be loaded into the web browser either from the filesystem or by hosting it on a web server, the same as with the CSS Grammar tool. In either case, the JavaScript executes in the browser to carry out the sequence of DOM operations. This tool found many vulnerabilities in all the major web browsers during 2010 and 2011 [90].

LangFuzz is a 2012 tool for fuzzing interpreters that takes a context free grammar [93] as input along with a test suite of valid sample progams and an additional corpus of programs that were seen to previously cause errors in the target application. The tool then combines aspects of mutation and generation fuzzing. It uses the grammar to parse the test suite and previous crashes and then recombines fragments of crash code into the test suite to create new programs in an attempt to find new issues [58]. For example, the tool was used to fuzz the Mozilla JavaScript engine by providing it with a context free grammar describing JavaScript, a JavaScript test suite and also a collection of JavaScript programs that had previously caused errors in the Mozilla engine. The newly generated JavaScript programs that LangFuzz created from this resulted in 105 new vulnerabilities being discovered in 3 months [58]. LangFuzz can be applied to any compiler or interpreter and is not specifically aimed at fuzzing web browser related technology. Its successful use to find serious vulnerabilities in the Firefox browser via its JavaScript interpreter makes it worth including here though.

The success of the LangFuzz approach in [58] suggests that when forming an initial corpus for general browser fuzzing it may help to include previous crashing test cases that are already known as well as public

browser exploits and proof of concepts. By doing this it may be that new browser vulnerabilities which happen to be similar to old, fixed vulnerabilities can be revealed. No academic literature was found to suggest that this idea has already been evaluated.

Grinder, released in 2012, is a distributed framework to support fuzzing Windows web browsers on a large scale [95]. It consists of a web application running on a centralised Grinder Server and any number of Grinder Nodes that carry out the actual fuzzing. As a framework, the aim is to provide an easy solution to problems such as documenting crashes, providing instrumentation, crash deduplication, restarting the browser as needed and related features rather than the fuzzing itself. The user must provide their own fuzzing program that creates the test cases. The value of such a framework is that it lets the researcher focus on how to create good test cases without worrying about all the other issues.

GramFuzz was developed in 2013 by researchers in China [8]. Their work examined combining techniques from generation and mutation fuzzing while using HTML, CSS and JavaScript test cases to fuzz web browsers. The tool obtains a corpus of web pages from the Internet using web crawling and then builds a grammar tree from them. The grammar is then used to create mutations to the corpus that are likely to be correct enough to be accepted by the browser's JavaScript interpreter rather than being rejected immediately as may be the case with a pure mutation fuzzing approach. This is quite similar to the approach taken by Radamsa that we saw earlier. During testing GramFuzz found 36 new severe vulnerabilities in Firefox and Internet Explorer.

DOMfuzz [51] and Jsfunfuzz [52] are two fuzzers released by Mozilla at the end of 2013 and beginning of 2014. DOMfuzz employs a mix of mutation and generation techniques to carry out DOM fuzzing. It takes an initial corpus of pages and then applies techniques such as rearranging the DOM tree by moving nodes around. Jsfunfuzz is a generation fuzzer for testing JavaScript engines. It works by generating and executing random JavaScript function bodies. Unlike the JavaScript fuzzing that was performed with LangFuzz it therefore does not take an initial corpus but rather relies entirely on generation.

NodeFuzz is a modular cross platform browser fuzzing harness released in 2014 [53, 56]. It solves a lot of the same problems as Grinder but it is written in Node.js and is platform independent. The user is expected to supply their own test case creation modules and also their own instrumentation modules. These can therefore be specific to the exact browser being tested if necessary. A popular test case creation module written to be used with NodeFuzz is Wadi [50]. It was released at Defcon 23 in 2015 and provides functionality to create JavaScript test cases that fuzz the DOM. A grammar is used to generate syntactically correct random JavaScript routines that act on the DOM when executed and a HTML test case is generated each time with the script inside it. The module is aimed at testing both Firefox and Chromium using Asan as the instrumentation to detect memory corruption.

Finally, ShakeIt is a mutation fuzzer published in 2015 for use within a browser fuzzing framework [57]. The program takes a corpus of valid web pages and extracts tokens from them. It then swaps random tokens with each other within these pages to create new test cases. Memory corruption bugs can be exposed in this way as the rendering engine attempts to make sense of seeing valid tokens in invalid locations or in an invalid order.

This is by no means a comprehensive list of all existing browser fuzzing tools, but it does serve to highlight the typical functionality and approaches that have been tried over the years. It can be seen that while fuzzing research in general has given considerable attention to measuring and increasing code coverage, the work that has been done on browser specific fuzzing tools does not appear to have made this a priority. It can also be seen that there has been a trend to focus on generation fuzzing when attacking web browsers, one notable exception being the work done applying Radamsa to browsers [100]. The work in [100] proved that mutation of an existing corpus of web pages can be a successful approach to browser fuzzing, and had work been done

to ensure the maximum code coverage of the initial corpus it could perhaps have been even more successful.

# 5 Conclusion

This paper reviewed some of the most noteworthy academic literature and practical work that has been produced in the field of fuzzing.

We first examined how vulnerabilities come to exist in software and how security researchers find them. After a brief overview of common vulnerability types and methods of static analysis, we looked in depth at the field of fuzzing. Competing approaches to fuzzing were examined, from simple random inputs all the way to using genetic algorithms and taint analysis. The importance of measuring code coverage to evaluate the completeness of a fuzzing campaign was examined. Finally, the focus was placed on the fuzz testing of web browsers and the specific tools and techniques related to that.

# 6 References


[1] – Patrice Godefroid et al. “SAGE: Whitebox Fuzzing for Security Testing” in Communications of the ACM, Vol 55, No 3. March 2012.

[2] – Charlie Miller and Zachory Peterson. “Analysis of Mutation and Generation-Based Fuzzing”, 1st March 2007.

[3] – Andrea Lanzi et al. “A Smart Fuzzer for x86 Executables” in 29th International Conference on Software Engineering Workshops, IEEE. 2007.

[4] – Sofia Bekrar et al. “A Taint Based Approach for Smart Fuzzing” in 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation.

[5] – Chris Anley, John Heasman, Felix Linder and Gerardo Richarte. “The Shellcoder's Handbook: Discovering and Exploiting Security Holes”, Second Edition, Wiley Publishing, 2007.

[6] - Ryan Roemer, Erik Buchanan, Hovav Shacham and Stefan Savage. “Return-Oriented Programming: Systems, Languages, and Applications”, University of California, San Diego.

[7] – Toby Reynolds. “Bypassing Address Space Layout Randomization”, Null Security, April 2012.

[8] - Tao Guo et al. “GramFuzz: Fuzzing Testing of Web Browsers Based on Grammar Analysis and Structural Mutation” in 2013 Second International Conference on Informatics & Applications (ICIA 2013), Lodz, Poland, 23 – 25 September 2013.

[9] – Miller, Fredriksen and So. “An Empirical Study of the Reliability of UNIX Tools” in Communications of the ACM, Volume 33 Issue 12, Dec. 1990, pp 32-44.

[10] – Dave Aitel. “Black Hat USA 2002 – An Introduction to SPIKE”. [Online] Available: https://www.youtube.com/watch?v=oGJKdGZUGOk [Accessed: 3rd February 2017]

[12] – Shacham et al. "On the Effectiveness of Address-Space Randomization" in Proceedings of the 11th ACM conference on Computer and Communications Security, October 2004.

[13] – FindBugs: Find Bugs in Java Programs. [Online]. Available: http://findbugs.sourceforge.net/ [Accessed: 3rd February 2017]

[14] – VeraCode: Static Analysis Tools and Platforms. [Online]. Available: https://www.veracode.com/products/static-analysis-sast/static-analysis-tool [Accessed: 3rd February 2017]

[15] – Fortify Static Code Analyzer. [Online]. Available: http://www8.hp.com/uk/en/software-

solutions/static-code-analysis-sast/ [Accessed: 3rd February 2017]

[16] – Bruce Schneier. “Computer Network Exploitation vs. Computer Network Attack.“ [Online]. Available: https://www.schneier.com/blog/archives/2014/03/computer_networ.html [Accessed: 3rd February 2017]

[17] – AddressSanitizer. [Online]. Available: http://clang.llvm.org/docs/AddressSanitizer.html [Accessed: 3rd February 2017]

[18] - “CVE-2010-2753: Integer overflow in Mozilla Firefox, Thunderbird and SeaMonkey,” 2010. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2753 [Accessed: 3rd February 2017]

[19] - Will Dietz, Peng Li, John Regehr, and Vikram Adve. “Understanding Integer Overflow in C/C++” in Proceedings of the 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, June 2012.

[20] – Common Weakness Enumeration, “CWE-416: Use After Free”. [Online]. Available: https://cwe.mitre.org/data/definitions/416.html [Accessed: 3rd February 2017]

[21] – CVE Details, "CVE-2014-0322: Use-after-free vulnerability in Microsoft Internet Explorer 9 and 10". [Online]. Available: https://www.cvedetails.com/cve/CVE-2014-0322/ [Accessed: 3rd February 2017]

[22] - CVE Details, "CVE-2015-0058: Double free vulnerability in win32k.sys in the kernel-mode drivers in Microsoft Windows". [Online]. Available: https://www.cvedetails.com/cve/CVE-2015-0058/ [Accessed: 3rd February 2017]

[23] - Chris Anley, John Heasman, Felix Linder and Gerardo Richarte. “The Shellcoder's Handbook: Discovering and Exploiting Security Holes”, Second Edition, Wiley Publishing, 2007. Page 498.

[24] – NIST National Vulnerability Database, “CVE-2012-5144: Google Chrome off-by-one vulnerability”. [Online]. Available: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-5144 [Accessed: 3rd February 2017]

[25] – Common Weakness Enumeration, “CWE-193: Off-by-one Error”. [Online]. Available: https://cwe.mitre.org/data/definitions/193.html [Accessed: 3rd February 2017]

[26] – Microsoft SDL (Security Development Lifecycle). [Online]. Available: https://www.microsoft.com/en-us/sdl/default.aspx [Accessed: 3rd February 2017]

[29] - Johannes Dahse and Thorsten Holz. “Simulation of Built-in PHP Features for Precise Static Code Analysis” in NDSS ’14, 23-26 February 2014, San Diego, CA, USA.

[30] - John Viega, Tom Mutdosch, Gary McGraw and Edward W. Felten. “Statically Scanning Java Code for Security Vulnerabilities” in IEEE Software, Volume 17 Issue 5, September 2000, pp 68-74.

[31] – Flawfinder. [Online]. Available: http://www.dwheeler.com/flawfinder/ [Accessed: 3rd February 2017]

[33] – Cppcheck. [Online]. Available: http://cppcheck.sourceforge.net/ [Accessed: 3rd February 2017]

[34] - V. Benjamin Livshits and Monica S. Lam. “Finding Security Vulnerabilities in Java Applications with Static Analysis” in Proceedings of the 14th conference on USENIX Security Symposium - Volume 14, 2005.

[36] – Valgrind. [Online]. Available: http://valgrind.org/ [Accessed: 3rd February 2017]

[37] – PaiMei. [Online]. Available: https://github.com/OpenRCE/paimei [Accessed: 3rd February 2017]

[38] – Art Manion and Michael Orlando, CERT. “Fuzz Testing for Dummies” at Industrial Control Systems Joint Working Group (ICSJWG), May 2011.

[39] – GDB exploitable plugin. [Online]. Available: https://github.com/jfoote/exploitable [Accessed: 3rd

February 2017]

[40] – Matthew Franz. “A Maze of Twisty Passages all Alike: A Bottom-Up Exploration of Open Source Fuzzing Tools and Frameworks” at CERT Vulnerability Discovery Workshop, February 2010.

[41] - Tobias Ospelt. “American Fuzzy Lop, A short Introduction” at Silicon Valley Fuzzers meeting, 9th March 2015.

[42] – Michal Zalewski. “Pulling JPEGs out of thin air”. [Online]. Available: https://lcamtuf.blogspot.co.uk/2014/11/pulling-jpegs-out-of-thin-air.html [Accessed: 3rd February 2017]

[43] – Michal Zalewski. “Technical whitepaper for afl-fuzz”. [Online]. Available: http://lcamtuf.coredump.cx/afl/technical_details.txt [Accessed: 3rd February 2017]

[44] – Rosario Valotta. “Browser Fuzzing in 2014: David vs Goliath”. [Online]. Available: https://www.syscan360.org/slides/2014_EN_BrowserFuzzing_RosarioValotta.pdf [Accessed: 3rd February 2017]

[46] – CVE-2010-0249: Operation Aurora Use-after-free vulnerability in Internet Explorer. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0249 [Accessed: 3rd February 2017]

[47] – Wade Alcorn, Christian Frichot and Michele Orrù. “The Browser Hacker's Handbook”, Wiley, 2014. Pp 7-8.

[48] – CrossFuzz. [Online]. Available: http://lcamtuf.coredump.cx/cross_fuzz/ [Accessed: 3rd February 2017]

[50] - Saif El-Sherei. “Wadi Fuzzer”. [Online]. Available: https://www.sensepost.com/blog/2015/wadi-fuzzer/ [Accessed: 3rd February 2017]

[51] – Mozilla Security. “DOMFuzz”. [Online]. Available: https://github.com/MozillaSecurity/funfuzz/tree/master/dom [Accessed: 3rd February 2017]

[52] – Mozilla Security. “jsfunfuzz”. [Online]. Available: https://github.com/MozillaSecurity/funfuzz/tree/master/js/jsfunfuzz [Accessed: 3rd February 2017]

[53] – Atte Kettunen, University of OULU. “Test Harness for Web Browser Test Fuzzing”. Master’s Thesis, Degree Programme in Computer Science and Engineering, September 2014.

[55] – Jesse Ruderman. “CSS grammar fuzzer”. [Online]. Available: https://www.squarefree.com/2009/03/16/css-grammar-fuzzer/ [Accessed: 3rd February 2017]

[56] – Atte Kettunen. “Browser Bug Hunting: Memoirs of a Last Man Standing” at 44CON, 2013.

[57] – Jeremy Brown. “Browser Fuzzing with a Twist (and a Shake)” at Zero Nights, 2015.

[58] - Christian Holler, Kim Herzig and Andreas Zeller. “Fuzzing with Code Fragments”. Presented as part of the 21st USENIX Security Symposium (USENIX Security 2012).

[59] - Rosario Valotta. “Taking Browsers Fuzzing To The Next (DOM) Level” at DeepSec 2012.

[60] – Chen Zhang. “Smashing the Browser: From Vulnerability Discovery to Exploit” at HITCON, Taiwan, August 2014.

[61] – Renata Hodovan. “Fuzz Testing of Web Browsers” at 3rd User Conference on Advanced Automated Testing, Sophia Antipolis, French Riviera, 20-22 October 2015.

[62] – Aleph One. “Smashing the Stack for Fun and Profit”. Phrack, vol. 7, no. 49, 1996.

[63] – Chris Anley, John Heasman, Felix Linder and Gerardo Richarte. “The Shellcoder's Handbook:

Discovering and Exploiting Security Holes", Second Edition, Wiley Publishing, 2007. "Chapter 5: Introduction to Heap Overflows", pp 89-107.

[64] – Pascal Meunier. "Classes of Vulnerabilities and Attacks". [Online] Available: http://homes.cerias.purdue.edu/~pmeunier/aboutme/classes_vulnerabilities.pdf [Accessed: 3rd February 2017]

[65] – Microsoft. "A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003" [Online]. Available: https://support.microsoft.com/en-us/kb/875352 [Accessed: 3rd February 2017]

[66] – Tilo Muller. "ASLR Smack & Laugh Reference: Seminar on Advanced Exploitation Techniques", February 2008.

[67] – Gadi Evron et al. "Open Source Fuzzing Tools", Syngress Publishing, 2007. "Chapter 2: Fuzzing – What's That?", pp 11-26.

[68] - Konstantin Serebryany et al, Google Inc. "AddressSanitizer: A Fast Address Sanity Checker" in Proceedings of the 2012 USENIX Annual Technical Conference.

[69] - Yan Shoshitaishvili et al, UC Santa Barbara. "(State of) The Art of War: Offensive Techniques in Binary Analysis" in 2016 IEEE Symposium on Security and Privacy.

[71] – Charlie Miller. "Fuzzing with Code Coverage by Example" at ToorCon, October 2007.

[72] - Chris Anley, John Heasman, Felix Linder and Gerardo Richarte. "The Shellcoder's Handbook: Discovering and Exploiting Security Holes", Second Edition, Wiley Publishing, 2007. "Automated Source Code Analysis Tools", page 484.

[73] – Jacob West. "How I Learned to Stop Fuzzing and Find More Bugs" at Defcon 15, August 2007.

[74] – Chris Anley, John Heasman, Felix Linder and Gerardo Richarte. "The Shellcoder's Handbook: Discovering and Exploiting Security Holes", Second Edition, Wiley Publishing, 2007. "Chapter 16: Fault Injection", pp 445-459.

[75] – Derek L. Bruening. "Efficient, Transparent, and Comprehensive Runtime Code Manipulation". PhD Thesis, Massachusetts Institute of Technology, September 2004.

[76] - Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation" in Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2007, pp 89-100.

[77] – Will Drewry and Tavis Ormandy. "Flayer: Exposing Application Internals" in Proceedings of the first USENIX workshop on Offensive Technologies, 2007.

[78] – MWR Infosecurity. "15 Minute Guide to Fuzzing". [Online]. Available: https://www.mwrinfosecurity.com/our-thinking/15-minute-guide-to-fuzzing/ [Accessed: 3rd February 2017]

[79] – IDA Pro. [Online]. Available: https://www.hex-rays.com/products/ida/index.shtml [Accessed: 3rd February 2017]

[80] - Michal Zalewski. "Binary fuzzing strategies: what works, what doesn't". [Online]. Available: https://lcamtuf.blogspot.co.uk/2014/08/binary-fuzzing-strategies-what-works.html [Accessed: 3rd February 2017]

[81] – Andy Davis. "Fuzzing the easy way, using Zulu", NCC Group, 2014.

[82] - J.C. King. "Symbolic execution and program testing" in Communications of the ACM, 1976, pp. 385-

394.

[83] - S. K. Cha, M. Woo, and D. Brumley. “Program-Adaptive Mutational Fuzzing” in Proceedings of IEEE Symposium on Security and Privacy, volume 2015-July, pp 725–741.

[84] – Pedram Amini and Aaron Portnoy. “Sulley: Fuzzing Framework”. [Online]. Available: http://fuzzing.org/wp-content/SulleyManual.pdf [Accessed: 3rd February 2017]

[85] – Sofia Bekrar et al. “Finding Software Vulnerabilities by Smart Fuzzing” in 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation.

[86] - “LibFuzzer: A Library for Coverage-Guided Fuzz Testing” [Online]. Available: http://llvm.org/docs/LibFuzzer.html [Accessed: 3rd February 2017]

[87] – Nick Stephens et al. “Driller: Augmenting Fuzzing Through Selective Symbolic Execution” at The Network and Distributed System Security Symposium 2016, San Diego, California.

[88] - Wang, T., Wei, T., Gu, G., and Zou, W. 2011. “Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution” in ACM Trans. Inf. Syst. Secur. 14, 2, Article 15 (September 2011).

[89] – Michal Zalewski. “Mangleme”. [Online]. Available: http://lcamtuf.coredump.cx/soft/mangleme.tgz [Accessed: 3rd February 2017]

[90] – Bugzilla @ Mozilla. “Bug 581539 - (crossfuzz) Bugs found by Michal Zalewski's cross_fuzz”. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=581539 [Accessed: 3rd February 2017]

[91] – H.D. Moore, Rapid7. “Browser fuzzing for fun and profit”. [Online]. Available: https://community.rapid7.com/community/metasploit/blog/2006/03/30/browser-fuzzing-for-fun-and-profit [Accessed: 3rd February 2017]

[92] - Michael Sutton, Adam Greene, and Pedram Amini. “Fuzzing: brute force vulnerability discovery”, Pearson Education, 2007, p 275.

[93] – John Martin. “Introduction to Languages and the Theory of Computation”, Second Edition, McGraw-Hill, 1997. Pp 163-197.

[94] – Tavis Ormandy, Google Inc. “Making Software Dumber” at HIRBSecConf 2009.

[95] – Stephen Fewer. “Grinder – System to Automate Fuzzing of Web Browsers”. [Online]. Available: https://github.com/stephenfewer/grinder [Accessed: 3rd February 2017]

[96] - Vijay Ganesh, Tim Leek and Martin Rinard. “Taint-based Directed Whitebox Fuzzing” in Proceedings of the 31st International Conference on Software Engineering, 2009, pp 474-484.

[97] – Yang Chen et al. “Taming Compiler Fuzzers” in Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2013, pp 197-208.

[98] - Alexandre Rebert et al. “Optimizing Seed Selection for Fuzzing” in Proceedings of the 23rd USENIX conference on Security Symposium, 2014, pp 861-875.

[99] - Richard McNally, Ken Yiu, Duncan Grove and Damien Gerhardy. “Fuzzing: The State of the Art”. Australian Government Department of Defence, Defence Science and Technology Organisation, 2012.

[100] - Pekka Pietikäinen et al, University of Oulu. “Security Testing of Web Browsers”.

[102] - Mikko Vimpari. “An Evaluation of Free Fuzzing Tools”. Master's Thesis, University of Oulu, May 2015.

[103] - Jared D. DeMott, Richard J. Enbody and William F. Punch. "Revolutionizing the Field of Grey-box

Attack Surface Testing with Evolutionary Fuzzing" at Defcon 2007.

[104] - Alejandro Hernández. “ELF Parsing Bugs by Example with Melkor Fuzzer”. IOActive, 2014.

[105] - Fabien Duchene. “How I Evolved your Fuzzer: Techniques for Black-Box Evolutionary Fuzzing” at SEC-T Conference, 2014.

[107] – GNU. “Gcov – Using the GNU Compiler Collection”. [Online] Available: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html [Accessed: 3rd February 2017]

[108] – Tom Mitchell. “Machine Learning”, McGraw-Hill, 1997. “Chapter 9: Genetic Algorithms”, pp 249-273.