Title: Comparative Analysis of current fuzzing methods for Buffer Overflows in C++ Software

Research question: How do AFL++ (American Fuzzy Lop), and Radamsa differ in their effectiveness for detecting buffer overflow induced crashes in software applications using C++, considering factors such as the accuracy in detecting buffer overflow vulnerabilities and the performance in terms of execution speed and resource utilisation?

Table of Contents

0. Reflection / Summary of Findings

The following is a reflection written following the research, which acts as a summary of the key findings and results, as well as suggestions looking back at the process.

One key takeaway was the necessity of designing realistic testing environments in order to achieve results applicable in the real world. Therefore, whilst my experiments still provided valuable insights, I recognize the limitations encountered, particularly the simplifications made for feasibility, whereas real-world scenarios would involve greater complexity than the test cases I designed. Time management emerged as a significant challenge, perhaps by using tools such as the cloud-based ‘droplets’ offered by digital ocean the toll fuzzing takes on my computer, and the sheer scale of time and resources required, would be alleviated and I could thus have minimized delays. Another major obstacle was creating C++ vulnerabilities complex enough to differentiate the fuzzers’ performances yet simple enough to allow repeated testing within my timeline. However, the results did successfully highlight significant differences between AFL++ and Radamsa, with AFL demonstrating much higher accuracy in detecting the vulnerabilities, yet requiring significantly more resources. This corresponded with secondary research conducted and solidifies why most people recommend AFL as a ‘go-to’ fuzzer, as well as why other tools such as Radamsa are still used. If I could revisit this project, I would incorporate additional tools like ASAN (Address Sanitizer) to capture vulnerabilities that don’t cause visible crashes, thus refining my setup for more realistic applications. Overall, the process solidified my interest in cybersecurity, and has allowed me to hone my research, critical thinking, and self-management skills.

1. Introduction

Software programs are at a constant risk of suffering damages, either accidentally or as a result of malicious individuals exploiting vulnerabilities. Some services, such as medical software, are constantly being depended on and cannot afford downtime as a result of even just a simple crash. Overall, these vulnerabilities can end up extorting companies and individuals of millions.

In an attempt to minimise these vulnerabilities, one may employ similar techniques to real attackers where you try to exploit a software with a goal of informing the client of vulnerabilities found during the assessment. Finding these vulnerabilities before attackers allows the software provider to patch these issues before they can be exploited by a malicious user. In CyberEdge's 10th annual Cyberthreat Defense Report, it was noted that approximately 85% of organisations had suffered at least one successful cyberattack in the past year[1], not accounting for the millions of unsuccessful attacks. This astonishing statistic puts into perspective the importance of understanding and finding these vulnerabilities in software being deployed publicly.

To facilitate the process of finding vulnerabilities there exist many tools that use differing methods of identifying faults in programs ranging from static analysis to dynamic testing. So many of these tools have progressed and been developed over the years, to identify different vulnerability types, that there are now often a range of tools which can be used for the same purpose, making it hard to know which tool you

[1] CyberEdge Group, LLC. "Cyberthreat Defense Report 2023." CyberEdge Group, 2023, https://cyber-edge.com/cdr/. Accessed 4 April 2024.

should use. Therefore, comparing the differences between these existing tools can help understand which performs better and what makes them worth using.

1.1 Buffer overflows

Buffer overflows are, and have been for years, one of the most common vulnerabilities facing software applications[2]. In fact, the 14 most common vulnerabilities in C++ software are memory accessing vulnerabilities relating to buffer overflows[3].

Buffer Overflows are a relatively broad umbrella term falling under the even broader category of ‘Overflows’, which contain other vulnerabilities such as ‘Stack Overflows’ which may vary in cause and consequences, affecting different parts of memory such as the stack or heap.

Generally, most Overflows occur when you attempt to store more data to memory than there was space allocated[4]. This leads to data ‘overflowing’ to adjacent memory locations, leading to unexpected behaviour such as altering adjacent data or simply crashing the application. If an attacker manages to leverage these vulnerabilities, it could be a real danger to a software application. In fact, the first ever major internet

---

[2] Synopsys Editorial Team. “Top 10 software vulnerability list for 2019.” synopsys, Synopsys Editorial Team, 15 Jan 2019, https://www.synopsys.com/blogs/software-security/top-10-software-vulnerability-list-2019.html. Accessed 4 April 2024.

[3] “CWE - CWE-658: Weaknesses in Software Written in C (4.14).” Common Weakness Enumeration, 11 April 2008, https://cwe.mitre.org/data/definitions/658.html. Accessed 4 April 2024.

[4] Goedegebure, Coen. “Buffer overflow attacks explained.” Coen Goedegebure, 15 November 2020, https://www.coengoedegebure.com/buffer-overflow-attacks-explained/. Accessed 4 April 2024.

attack which occurred in 1988, the morris worm, was assisted by a buffer overflow vulnerability[5].

The specificities of different types of Overflows is outside of the scope of this study, we will be looking at all kinds of Buffer Overflows as they are the easiest to cause and record.

1.2 Vulnerable Languages

Buffer Overflows rely on inappropriate memory access, and are therefore found in low-level languages where the programmer has direct access with manual memory management. This includes languages such as Assembly, C & C++, which give more control to the programmer and maximise performance, whilst higher level languages such as Java, Rust, Python, and C# have built-in memory management and are therefore more protected against such vulnerabilities.

In this paper we will be analysing buffer overflows found in C++ software, as C++ is a modern language with widespread use in application development, notorious for its buffer overflow risks.

[5] Federal Bureau of Investigation. “The Morris Worm — FBI.” FBI, 2 November 2018, https://www.fbi.gov/news/stories/morris-worm-30-years-since-first-major-attack-on-internet-110218. Accessed 4 April 2024.

1.3 Fuzzing

As prefaced earlier, due to the large issue caused by Buffer Overflows and other similar vulnerabilities it is important to find them and their cause in software before and after it is deployed. Many different techniques and tools can be used for this purpose, with one such technique being fuzzing with the subset of tools used for this purpose being called fuzzers. Fuzzing involves repeatedly throwing large amounts of malformed input at a program to see how it behaves, with a hope that one of the inputs cause an error within the program[6]. For instance uploading a .HEIC image to an application expecting only JPG images allows us to check if there is correct format checking.

Fuzzing itself also consists of a range of different approaches, each attempting to maximise the code coverage (the percentage of the total code which was run as a result of that input), whilst minimising factors such as performance overhead and time taken. These techniques range from ‘Dumb’ fuzzing techniques such as the most primitive of swarming the program with completely random values, all the way to more intricate approaches such as guided / behavioural, mutation, and coverage-based fuzzing amongst others. The most important distinction to make is between ‘Black Box’ and ‘White Box’ fuzzing, ‘Black Box’ being when the fuzzer does not have access to the program source code. When the fuzzer has access to the source code (‘White Box’ fuzzing) it may use a technique known as ‘instrumentation’, where it prepares and compiled the code in such a way to maximise code coverage when being fuzzed.

[6] Bowden, Andy. “Coalfire Blog - Fuzzing: Common Tools and Techniques.” Coalfire, https://coalfire.com/the-coalfire-blog/fuzzing-common-tools-and-techniques. Accessed 4 April 2024.

To ensure the maximum amount of available software is free of vulnerabilities, tools such as OSS-FUZZ[7] continuously fuzz open-source code to check for easy to find vulnerabilities. This means that as an individual, you have little chance of finding memory errors such as buffer overflows on popular tools using only primitive fuzzing. However, this does not mean it is impossible, as you can specify the fuzzing environment using existing knowledge of the software and code to optimise fuzzing and uncover vulnerabilities more efficiently, allowing you to find vulnerabilities other tools may have missed.

1.4 AFL & Radamsa

In this paper we will be comparing two specific fuzzers, AFL++ (American Fuzzy Lop) and Radamsa. The following secondary research aims to best point the key differences between these fuzzers to then understand how we could expect them to react to our test cases.

AFL++ is an advanced fork of Google's AFL, designed to be faster & more efficient with improved mutations, instrumentation, and custom module support. AFL leverages an instrumentation-guided genetic algorithm to maximise code coverage. It is widely regarded as an industry standard and is often the default fuzzer used in educational and professional settings due to its ease of use and ability to uncover a wide variety of real-world bugs[8].

[7] google. “OSS-Fuzz | Documentation for OSS-Fuzz.” Google, https://google.github.io/oss-fuzz/. Accessed 3 July 2024.
[8]Fioraldi, Andrea, et al. “AFL++ : Combining Incremental Steps of Fuzzing Research.” USENIX, https://www.usenix.org/conference/woot20/presentation/fioraldi. Accessed 5 July 2024.

Radamsa, on the other hand, is another popular fuzzer known for its “black box” approach.[9] Unlike AFL, Radamsa generates test cases without any knowledge of the internal structure of the target application, it acts as the most basic form of a fuzzer with its brute-force approach supposedly making it an ideal tool for testing software where source code is not available. Its simplicity and effectiveness have made it a favoured choice for discovering unexpected vulnerabilities in a wide range of applications especially surrounding a wide range of file formats other than basic text.[10]

I have chosen these two fuzzers due to their widespread use and differing approaches to fuzzing. AFL++’s white-box fuzzing technique and Radamsa’s black-box approach provide a comprehensive view of current buffer overflow detection methods in C++ software. By analysing both, we aim to understand the strengths and weaknesses of each method in detecting buffer overflow vulnerabilities and although there is no pre-existing direct comparison between these two fuzzers we can see if our results align with what we may expect considering the pre-existing research into each of these individual fuzzers discussed above.

[9]Helin, Aki. “Aki Helin / radamsa · GitLab.” GitLab, 4 June 2013, https://gitlab.com/akihe/radamsa. Accessed 5 July 2024.
[10]Syn Cubes Community. “Securing Your App: Radamsa Fuzzing for User Input Validation Issues.” syn cubes, 28 February 2023, https://www.syncubes.com/using_radamsa_to_fuzz_for_improper_user_input_validation. Accessed 24 October 2024.

2. Methodology / Scope of Study

2.1 Objective

This paper aims to provide a detailed analysis of the differences between AFL++ and Radmsa in detecting buffer overflow vulnerabilities in both pre-compiled and uncompiled test-cases of C++ software. By recording and comparing the time taken to detect a vulnerability and the cpu usage of the computer while running the fuzzer, the findings should hopefully better help in choosing the appropriate fuzzing tool based on specific requirements such as efficient resource usage, detection accuracy, or the type of software (compiled or uncompiled) being tested.

2.2 Experimental Setup

A set of 3 C++ applications written by me with known buffer overflow vulnerabilities will be used as test subjects for the fuzzers. The applications take a single line of input from the terminal using the standard input library, the fuzzer will be tasked with altering this input for malformed input to cause a crash.

The 3 applications have increasingly complicated buffer overflow vulnerabilities, with the third one requiring a condition being met for the overflow to occur: the input data should start with “BFEE”. The 3 complete C++ applications can be found in Appendix 3, Appendix 4, and Appendix 5.

The fuzzers effectiveness will be tested when they both have access to the source code, or when only the final compiled application is available, meaning both fuzzers will have to adopt a black-box approach. Both fuzzers require an ‘input case’ which tells the fuzzer which input the application expects. In our case we will leave the input case as “1234” for all test cases unless specified otherwise.

To best compare their differing techniques in the simplest environment, we will keep both fuzzers on their default settings and we will run them without the use of any external tools, meaning they may miss some vulnerabilities but they can still detect common crashes such as stack smashing or heap metadata corruptions that are usually checked by libc. Since Radamsa has no instrumentation of its own it will be run with AFL’s instrumented binary when testing source versions, which will use the same setup as for AFL’s tests.

The tests will be performed on a virtual system created using VMWare[11] to best isolate the experiment and cpu usage from other noise, with the same hardware and software configurations to ensure consistency.

2.3 Data Collection

I developed a short bash script to automate the process of compiling and instrumenting the code, as well as to run the fuzzers and measure how long they took to detect a crash whilst monitoring average CPU usage.

[11]Broadcom. “What is a Virtual Machine?” VMware, https://www.vmware.com/topics/virtual-machine. Accessed 27 August 2024.

Since instrumentation removes some detected buffer overflow vulnerabilities we have to use the ‘-O0’ & ‘fno-stack-protector’ flags. This gives a full command used for instrumentation as seen in Figure 1.

```
C/C++
AFL_INST_RATIO=10 afl-g++ -O0 -fno-stack-protector -o

vulnerable_1-instr vulnerable_1-source.cpp
```

Figure 1, AFL instrumentation setup

The tool will be recurrently running the system monitoring tool ‘top’, for resource utilisation, to then calculate the CPU and memory usage during the fuzzing process as can be seen in Figure 2.
This helps ensure the best consistency and accuracy in the data, as well as automating the process to simplify data collection.

```
C/C++
# function to get current CPU usage percentage
get_cpu_usage() {
    # calculate CPU usage using `top`
    top -bn1 | grep "Cpu(s)" | sed "s/.*, *\([0-9.]*\)%* id.*/\1/" | awk
'{print 100 - $1}'
}
```

Figure 2, get_cpu_usage function which calculates the current cpu usage

The script takes the raw C++ (.cpp) file and runs the tests on first the pre-compiled then instrumented test cases as can be seen in Figure 3. The script will be run and recorded 3 times and an average of the results will be taken.

```
C/C++
g++ -o vulnerable-nonInstr_binary $input_file

file=./vulnerable-nonInstr_binary


echo "--------------------------"

echo "running NON-Instrumented"

echo "--------------------------"

run_main


# instrument with AFL and then run again with new input file

AFL_INST_RATIO=10 afl-g++ -O0 -fno-stack-protector -o vulnerable-instr

$input_file > /dev/null 2>&1

file=./vulnerable-instr


echo ""

echo "--------------------------"

echo "running Instrumented"

echo "--------------------------"

run_main
```

Figure 3, section of bash script used to automate compiling and instrumentation for fuzzing and data collection process

For the full bash program see Appendix 1.

2.4 Theory

In theory, according to secondary research as discussed in the Introduction 1.4: “AFL & Radamsa”, AFL++ should demonstrate the highest accuracy in detecting the buffer overflow vulnerabilities in the non-compiled programs due to its instrumentation and genetic-algorithms, and will therefore be quicker at finding harder to find vulnerabilities in these cases.

Radamsa may perform better in terms of resource utilisation, as well as time taken in simpler test cases, given its simpler black-box approach. Radamsa relies heavily on chance for success, as due to its lack of persistence it could repeat the same/similar output several times causing a waste of time and may never test unexpected new combinations which stray far from the input cases. This may be noticeable considering the use of “1234” as an input case where the hardest vulnerable program would require text to crash.

Since instrumentation aids fuzzing engines to measure code coverage, identify executed paths, and provide feedback to guide the fuzzing process, an instrumented binary is more likely to result in detecting a memory corruption vulnerability if the fuzzer input triggers it. This means we should expect a lower average time to detect a crash for the instrumented binaries, where we have access to source code, especially for AFL who’s instrumentation we are using. I am unsure if the instrumentation will impact Radamsa’s results.

3. Processing & Interpretation of Results

The following Figures are extracts from the full tables of results, to see the complete tables for all test cases see Appendix 2. To see a sample of the raw results for each test case see Appendix 6, Appendix 7 and Appendix 8.

Running the bash script on the simple C++ program that has the most basic form of buffer overflow Radamsa causes a near instant crash in all cases. All of the crashes fell under a second, with some being recorded as zero to 3 decimal places as can be seen in Figure 4, a section of the output from running the script on the simplest C++ example.

```
Unset
--[Radamsa] Crash detected! | Time: 00:00:00 (mins [0]) | Average CPU Usage:
0%
```

Figure 4, Section of sample output from the bash script fuzzing the simplest C++ application

AFL++ records similar results. There is no notable difference between the pre-compiled and source versions as can be seen in Figure 5.

	Time Taken (seconds)			
	Try 1	**Try 2**	**Try 3**	**Average**
Radamsa	0.60	0.00	0.00	0.20
AFL	0.60	0.00	0.00	0.20

Figure 5, Section of table showing time taken for fuzzing the simplest program when pre-compiled

However, what stands out is the noticeable difference in CPU usage between Radamsa and AFL++. There is again no significant difference between the pre-compiled and source versions, however in both AFL++ uses significantly more processing power than Radamsa, in fact over 100 times as much, as can be seen below in Figure 6.

	CPU Usage (%)			
	Try 1	**Try 2**	**Try 3**	**Average**
Radamsa	0.20	0.00	0.00	0.07
AFL	6.40	6.40	9.90	7.57

Figure 6, Section of table showing CPU usage while fuzzing the simplest program when pre-compiled

Throughout all the test cases AFL++ and Radamsa each maintained the same average CPU usages, with AFL consistently having a notably higher usage. I took an average of Radamsa and AFL's CPU usage across all the tests (Pre-Compiled & Source) which can be seen below in Figure 7.

Average	**CPU (%)**
Radamsa	0.34
AFL	7.99

Figure 7, Table showing average CPU usage across all test cases for both fuzzers

However as the C++ applications being fuzzed became increasingly complex a clear separation starts to form between the times taken to detect a crash in AFL and

Radamsa, with AFL boasting significantly lower times. In fact the difficult example program containing the logical condition that the fuzzing input begins with “BFEE” was taking too long to fuzz, resulting in no crashes found by Radamsa after over 4 hours when using the default input case of “1234”, remaining frozen on the text output which can be seen below in Figure 8.

```
Unset
                    --------------------------

                     running NON-Instrumented

                    --------------------------

                    Starting Radamsa fuzzing...
```

Figure 8, Output from the bash script fuzzing the hardest C++ application with no altered input case after over 4 hours

Therefore to accelerate the fuzzing process and ensure the fuzzer can at least detect the crash I altered the input case several times until Radamsa could detect the crash in under an hour. Testing several input cases that became increasingly closer to the required prefix such as “ABCD”, “BFAB”, “BFAE” did not result in a crash until “BFEA”, which finally resulted in crashes in under 5 minutes, with all previous input cases resulting in over an hour of waiting. Radamsa then takes an average of 110 seconds to find the crash compared to an average of under 7 seconds from AFL++ as can be seen in Figure 9, an extract of the full data which can be found in Appendix 2.

	Time Taken (seconds)			
Tool	**Realistic Simple**		**Difficult conditional**	
	Compiled	**Source**	**Compiled**	**Source**
Radamsa	10.4	2.00	44.2	109

AFL	1.20	0.60	8.00	6.80

Figure 9, Table showing average time taken across both harder test cases for both fuzzers for pre-compiled and source versions of the programs

Following the impressive results from AFL, I tested to see if AFL could also detect the crash in under an hour with the previous input cases which Radamsa failed with, such as “ABCD”. AFL’s basic Instrumented mode found the crash using input case “BABC” in a total time of: 5.01 minutes as can be seen in Figure 10. The full script I made to simplify the testing process for this can be found in Appendix 9.

```
Unset
        Starting AFL fuzzing...
        -- [AFL] Crash Detected! | Time: 00:05:01 (mins [5.01]) | Average CPU
        Usage: 9.14%
```

Figure 10, Output of bash script made to test if AFL could detect the crash with worse input cases, which Radamsa failed at.

The most surprising part was that both Radamsa and AFL++ showed little difference in average times between the pre-compiled and source versions, as can be seen in Figure 11, below.

Avg. Times	Compiled	Source
Radamsa	18.27	37.0
AFL	3.13	2.6

Figure 11, Table showing average time taken across all test cases for both fuzzers for pre-compiled and source versions of the programs

Although there was a small improvement for AFL, which aligns with what we expected, Radamsa actually performed a lot worse.

4. Conclusion

4.1 Evaluation

We only noticed a marginal improvement for AFL using the instrumented source versions which stands as a surprise for AFL considering our secondary research which stated AFL should perform considerably better on an instrumented program. However, we did use such crude setups in the sake of simplicity that better results could be expected using more sophisticated methods mentioned in section 4.3: “Considerations”. Radamsa actually performed worse, but considering it is not made to work with AFL’s instrumentation it is no surprise that it experienced no benefit; the worse results may have been a product of pure chance.

What we did see, however, was a significant difference between the two fuzzers, AFL++ & Radamsa, which align with our secondary research and hypothesis, stating AFL should provide results in a smaller amount of time, especially for more complicated cases, but at the cost of additional processing resources.

Directly answering our research question: “How do AFL++ (American Fuzzy Lop), and Radamsa differ in their effectiveness for detecting buffer overflow induced crashes in software applications using C++, considering factors such as the accuracy in detecting buffer overflow vulnerabilities and the performance in terms of execution speed and resource utilisation?”, AFL displayed significantly higher accuracy finding some vulnerabilities Radamsa could not without changing the input cases. AFL also had an average lower time to detect the crashes, especially on the

more complex programs where multiple conditions were required to trigger these crashes. Radamsa on the other hand showed stellar performances considering resource utilisation, using an average of below 1% of the CPU during fuzzing.

These differences in CPU usage and performance stem from how AFL++ holds a persistent fuzzing session on the target program, logging the programs reactions to input and using this to expand code coverage, whilst Radamsa simply generates a mutated output of the user input which we then pipe to the program ourselves, meaning Radamsa has no vision of the program being fuzzed itself. This makes Radamsa require negligible resources to run but means it relies on a much more brute force ‘luck’ approach, which although could get lucky on the first try, leads to significantly more inconsistent results. This also explains how Radamsa had the most variance in times, as further discussed in Section 4.2: “Uncertainties”.

Overall, AFL++ seems better suited for corporations or individuals looking for the best, most efficient fuzzing there can be, with no resource limitations. Considering its impressive results and easy customizability explain its widespread use and why it's considered industry standard across most businesses and academic institutions as mentioned in our secondary research and hypothesis.

4.2 Uncertainties

The unpredictable nature of fuzzing means individual runs may yield widely varying times for crash detection as a result of pure luck rather than systematic reasoning.

As briefly mentioned in Section 4.1: “Evaluation”, Radamsa showed the greatest variance in times, which can be seen below in Figure 12. However, both fuzzers have extremely high uncertainties, which questions the conclusions we can draw from our results.

Largest Uncertainty		
Program	**Δ Time**	**Precision (%)**
Radamsa	142.20	151
AFL	6.00	100

Figure 12, Table showing precision for Radasa and AFL on test case 3 - source

The error in time is taken using the following equation, where t is the time taken

$$\Delta\, Time \;= t_{maximum} - \; t_{minimum}$$

The precision error as a percentage is obtained using the following equation

$$precision \;= \frac{\Delta\, Time}{t_{average}} \times 100$$

These large uncertainties are to be expected considering the previously mentioned chance-based nature of fuzzing and therefore the divide between AFL and Radamsa may not be exactly what was obtained, however, considering the rest of the data and disregarding outliers, the results are still significant enough to prove a shorter average time for AFL when compared to Radamsa.

4.3 Considerations

As computational tasks become more accessible, so will fuzzing, underscoring the importance of research into techniques assisting in finding these vulnerabilities before they can be exploited in real world software. However, it is crucial that such research be conducted ethically. In this study, I was only using my own software. If conducted on third-party software, obtaining authorization and following responsible disclosure of any vulnerabilities found is essential.

Fuzzing is also computationally intensive, which on top of raising environmental considerations due to the resources required have also posed limitations for our research considering we had to stop the fuzzer after detecting nothing for 4 hours on the last test case. Although fuzzing inherently remains an act of bruteforce which relies on intensive computation, time & luck, effectively optimising your fuzzing environment can drastically change the required time. In our final test case we observed that tools such as AFL, but especially Radamsa, rely on good input cases for efficient fuzzing. Using rented cloud machines through a service such as ‘Digital Ocean’, we could also run several machines with more processing power simultaneously, increasing our available overhead for fuzzing.

Overall, fuzzing can be as complicated or simple as you wish it to be, it is also widely recommended to run it in conjunction with external tools such as ASAN (Address Sanitizer) or a debugger such as gdb (the gnu debugger). External tools allow implementing a runtime to detect most memory corruption vulnerabilities the moment they're triggered (and call abort) and catches some vulnerabilities which would otherwise be missed, since not all vulnerabilities cause a crash we can detect (like

OOB read, or use-after-free). So fuzzing an instrumented binary in conjunction with external tools is more likely to result in detecting a memory corruption vulnerability if the fuzzer input triggers it.

Through a continuation to this study, optimising AFL++ for the specific environment it will be fuzzing, with a setup incorporating the improvements mentioned above, could allow for testing harder to detect vulnerabilities from patched real-world software, providing a more realistic assessment of AFL++'s performance with and without instrumentation. This approach could give more conclusive results in a realistic scenario, an area where our current results didn't mirror our expectations due to a lack of program complexity and fuzzer optimization for the environment.

5. Works Cited

Bowden, Andy. “Coalfire Blog - Fuzzing: Common Tools and Techniques.” *Coalfire*, https://coalfire.com/the-coalfire-blog/fuzzing-common-tools-and-techniques. Accessed 4 April 2024.

“CWE - CWE-658: Weaknesses in Software Written in C (4.14).” *Common Weakness Enumeration*, 11 April 2008, https://cwe.mitre.org/data/definitions/658.html. Accessed 4 April 2024.

Syn Cubes Community. “Securing Your App: Radamsa Fuzzing for User Input Validation Issues.” syn cubes, 28 February 2023, https://www.syncubes.com/using_radamsa_to_fuzz_for_improper_user_input_validation. Accessed 24 October 2024.

CyberEdge Group, LLC. “Cyberthreat Defense Report 2023.” *CyberEdge Group*, 2023, https://cyber-edge.com/cdr/. Accessed 4 April 2024.

Federal Bureau of Investigation. “The Morris Worm — FBI.” *FBI*, 2 November 2018, https://www.fbi.gov/news/stories/morris-worm-30-years-since-first-major-attack-on-internet-110218. Accessed 4 April 2024.

Goedegebure, Coen. “Buffer overflow attacks explained.” *Coen Goedegebure*, 15 November 2020, https://www.coengoedegebure.com/buffer-overflow-attacks-explained/. Accessed 4 April 2024.

Synopsys Editorial Team. “Top 10 software vulnerability list for 2019.” *synopsys*, Synopsys Editorial Team, 15 Jan 2019,

https://www.synopsys.com/blogs/software-security/top-10-software-vulnerability-list-2019.html. Accessed 4 April 2024.

google. “OSS-Fuzz | Documentation for OSS-Fuzz.” Google, https://google.github.io/oss-fuzz/. Accessed 3 July 2024.

Fioraldi, Andrea, et al. “AFL++ : Combining Incremental Steps of Fuzzing Research.” USENIX, https://www.usenix.org/conference/woot20/presentation/fioraldi. Accessed 5 July 2024.

Helin, Aki. “Aki Helin / radamsa · GitLab.” GitLab, 4 June 2013, https://gitlab.com/akihe/radamsa. Accessed 5 July 2024.

Broadcom. “What is a Virtual Machine?” VMware, https://www.vmware.com/topics/virtual-machine. Accessed 27 August 2024.

6. Bibliography

Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. “AFL++: Combining incremental steps of fuzzing research”. In 14th USENIX Workshop on Offensive Technologies (WOOT 20). USENIX Association, Aug. 2020.

Böck, Hanno. “Beginner's Guide to Fuzzing Part 2: Find more Bugs with Address Sanitizer.” *The Fuzzing Project*, https://fuzzing-project.org/tutorial2.html. Accessed 30 August 2024.

Bowden, Andy. “Coalfire Blog - Fuzzing: Common Tools and Techniques.” *Coalfire*, https://coalfire.com/the-coalfire-blog/fuzzing-common-tools-and-techniques. Accessed 4 April 2024.

Broadcom. “What is a Virtual Machine?” *VMware*, https://www.vmware.com/topics/virtual-machine. Accessed 27 August 2024.

CentralNic Ltd. “Technical "whitepaper" for afl-fuzz.” *[lcamtuf.coredump.cx]*, 2001, https://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed 30 August 2024.

“CWE - CWE-658: Weaknesses in Software Written in C (4.14).” *Common Weakness Enumeration*, 11 April 2008, https://cwe.mitre.org/data/definitions/658.html. Accessed 4 April 2024.

Syn Cubes Community. “Securing Your App: Radamsa Fuzzing for User Input Validation Issues.” syn cubes, 28 February 2023, https://www.syncubes.com/using_radamsa_to_fuzz_for_improper_user_input_validation. Accessed 24 October 2024.

CyberEdge Group, LLC. “Cyberthreat Defense Report 2023.” *CyberEdge Group*, 2023, https://cyber-edge.com/cdr/. Accessed 4 April 2024.

Federal Bureau of Investigation. “The Morris Worm — FBI.” *FBI*, 2 November 2018, https://www.fbi.gov/news/stories/morris-worm-30-years-since-first-major-attack-on-internet-110218. Accessed 4 April 2024.

Fioraldi, Andrea, et al. “AFL++ : Combining Incremental Steps of Fuzzing Research.” *USENIX*, https://www.usenix.org/conference/woot20/presentation/fioraldi. Accessed 5 July 2024.

Goedegebure, Coen. “Buffer overflow attacks explained.” *Coen Goedegebure*, 15 November 2020, https://www.coengoedegebure.com/buffer-overflow-attacks-explained/. Accessed 4 April 2024.

google. “OSS-Fuzz | Documentation for OSS-Fuzz.” *Google*, https://google.github.io/oss-fuzz/. Accessed 3 July 2024.

Google. “Using ASAN with AFL — AFL 2.53b documentation.” *AFL (american fuzzy lop)*, 2019, https://afl-1.readthedocs.io/en/latest/notes_for_asan.html. Accessed 30 August 2024.

Hanno Böck - American Fuzzy Lop and Address Sanitizer - QtCon Akademy 2016. 2017. *Youtube*, https://www.youtube.com/watch?v=mKVHuXQTZu0.

Helin, Aki. “Aki Helin / radamsa · GitLab.” *GitLab*, 4 June 2013, https://gitlab.com/akihe/radamsa. Accessed 5 July 2024.

Heuse, Marc, et al. “AFLPlusPlus documentations.” *AFLplusplus: The AFL++ fuzzing framework*, 2020, https://aflplus.plus/. Accessed 30 August 2024.

Iskhodzhanov, Timur. “Finding races and memory errors with LLVM instrumentation.” *LLVM*, 18 November 2011, https://llvm.org/devmtg/2011-11/Serebryany_FindingRacesMemoryErrors.pdf. Accessed 30 August 2024.

Kratkiewicz, Kendra June. *Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code*. A Thesis in the Field of Information Technology for the Degree of Master of Liberal Arts in Extension Studies. Harvard University, March 2005, https://apps.dtic.mil/sti/pdfs/ADA511392.pdf.

Oftedal, Erlend. “Fuzzing with AFL - Erlend Oftedal.” *YouTube*, 1 October 2018, https://www.youtube.com/watch?v=DFQT1YxvpDo. Accessed 30 August 2024.

Synopsys Editorial Team. “Top 10 software vulnerability list for 2019.” *synopsys*, Synopsys Editorial Team, 15 Jan 2019,

https://www.synopsys.com/blogs/software-security/top-10-software-vulnerability-list-2019.html. Accessed 4 April 2024.

Tavares, Pedro. “Fuzzing introduction: Definition, types and tools for cybersecurity pros.” *Infosec*, 30 September 2020, https://www.infosecinstitute.com/resources/penetration-testing/fuzzing-introduction-definition-types-and-tools-for-cybersecurity-pros/. Accessed 30 August 2024.

7. Appendix

Appendix 1, Complete script used to automate the fuzzing processes and capturing of data.

C/C++

```
#!/bin/bash

# ensure valid file argument is provided
if [ "$#" -ne 1 ]; then
    echo "Usage: $0 <path_to_target_executable>"
    exit 1
fi

input_file="$1"

# ensure Radamsa & AFL installed
command -v radamsa > /dev/null 2>&1 || { echo "Radamsa not found";
exit 1; }
command -v afl-fuzz > /dev/null 2>&1 || { echo "AFL not found"; exit
1; }

# function to get current CPU usage percentage
get_cpu_usage() {
    # calculate CPU usage using `top`
    top -bn1 | grep "Cpu(s)" | sed "s/.*, *\([0-9.]*\)%* id.*/\1/" |
awk '{print 100 - $1}'
}


run_main(){
# start tracking time
start_time=$(date +%s)

# Init average CPU usage variables
total_cpu=0
num_samples=0

# Radamsa fuzzing
echo "Starting Radamsa fuzzing..."

while true; do
    # Generate fuzzed input and pipe it to the target executable
    radamsa input_cases/test_case | "$file" > /dev/null 2>&1

    if [ $? -ne 0 ]; then
```

```
        # get elapsed time and average CPU usage
        elapsed_time=$(( $(date +%s) - start_time ))

        if [ "$num_samples" -eq 0 ]; then
            num_samples=1
        fi

        average_cpu=$(echo "scale=2; $total_cpu / $num_samples" | bc)
        echo ""
        echo ""
        echo "-- [Radamsa] Crash detected! | Time: $(date
-d@$elapsed_time -u +'%H:%M:%S') (mins [$(echo "scale=2; $elapsed_time
/ 60" | bc)]) | Average CPU Usage: $average_cpu%"
        echo ""
        echo ""
        # break if detect crash
        break
    fi

    # update average CPU usage
    current_cpu=$(get_cpu_usage)
    total_cpu=$(echo "$total_cpu + $current_cpu" | bc)
    num_samples=$((num_samples + 1))
done

# AFL fuzzing
echo "Starting AFL fuzzing..."

# remove output if exists
if [ -d "output_cases" ]; then
    rm -r output_cases
fi

# cpu tracking
total_cpu=0
num_samples=0

# start AFL fuzzing
afl_start_time=$(date +%s)
if [ "$file" = "./vulnerable-nonInstr_binary" ]; then
    afl-fuzz -n -i input_cases -o output_cases -- "$file" >
afl_output.log 2>&1 &
    folder=output_cases/crashes
else
    afl-fuzz -i input_cases -o output_cases -- "$file" >
afl_output.log 2>&1 &
    folder=output_cases/default/crashes
fi
```

```
afl_pid=$!

# monitor output for crash detection
while true; do
    if [ -d "output_cases" ] && find $folder -mindepth 1 -print -quit
| grep -q .; then
        kill -TERM "$afl_pid"
        wait "$afl_pid" 2>/dev/null

        # track end time and calculate duration for AFL fuzzing
        afl_end_time=$(date +%s)
        afl_elapsed_time=$(( afl_end_time - afl_start_time ))
        if [ "$num_samples" -eq 0 ]; then
            num_samples=1
        fi

        afl_cpu_usage=$(echo "scale=2; $total_cpu / $num_samples" |
bc)

        echo "-- [AFL] Crash Detected! | Time: $(date
-d@$afl_elapsed_time -u +'%H:%M:%S') (mins [$(echo "scale=2;
$elapsed_time / 60" | bc)]) | Average CPU Usage: $afl_cpu_usage%"
        break
    fi
    # update average CPU usage
    current_cpu=$(get_cpu_usage)
    total_cpu=$(echo "$total_cpu + $current_cpu" | bc)
    num_samples=$((num_samples + 1))
done

}

g++ -o vulnerable-nonInstr_binary $input_file
file=./vulnerable-nonInstr_binary

echo "--------------------------"
echo "running NON-Instrumented"
echo "--------------------------"
run_main

# instrument with AFL and then run again with new input file
AFL_INST_RATIO=10 afl-g++ -O0 -fno-stack-protector -o vulnerable-instr
$input_file > /dev/null 2>&1
file=./vulnerable-instr

echo ""
echo "--------------------------"
```

```
echo "running Instrumented"
echo "--------------------------"
run_main

echo ""
echo ""
echo "Fuzzing Complete!"
```

Appendix 2, Complete tables of collected data

Simple program | Pre - Compiled								
Tool	**Time Taken (seconds)**				**CPU Usage (%)**			
	Try 1	**Try 2**	**Try 3**	**Average**	**Try 1**	**Try 2**	**Try 3**	**Average**
Radamsa	0.60	0.00	0.00	0.20	0.20	0.00	0.00	0.07
AFL	0.60	0.00	0.00	0.20	6.40	6.40	9.90	7.57

Simple program | Source - Version								
Tool	**Time Taken (seconds)**				**CPU Usage (%)**			
	Try 1	**Try 2**	**Try 3**	**Average**	**Try 1**	**Try 2**	**Try 3**	**Average**
Radamsa	0.60	0.60	0.00	0.40	0.00	1.10	0.00	0.37
AFL	0.60	0.60	0.00	0.40	8.80	8.90	7.30	8.33

Realistic simple buffer overflow | Pre - Compiled								
Tool	**Time Taken (seconds)**				**CPU Usage (%)**			
	Try 1	**Try 2**	**Try 3**	**Average**	**Try 1**	**Try 2**	**Try 3**	**Average**
Radamsa	22.8	4.80	3.60	10.4	0.40	0.28	0.40	0.36
AFL	0.00	1.80	1.80	1.20	6.40	7.11	7.68	7.06

Realistic simple buffer overflow | Source - Version								
Tool	**Time Taken (seconds)**				**CPU Usage (%)**			
	Try 1	**Try 2**	**Try 3**	**Average**	**Try 1**	**Try 2**	**Try 3**	**Average**
Radamsa	1.80	3.60	0.60	2.00	0.60	0.33	1.02	0.65
AFL	0.60	0.60	0.60	0.60	7.45	8.97	9.90	8.77

Difficult example with conditional check | Pre - Compiled								
Tool	**Time Taken (seconds)**				**CPU Usage (%)**			
	Try 1	**Try 2**	**Try 3**	**Average**	**Try 1**	**Try 2**	**Try 3**	**Average**
Radamsa	36.0	91.8	48.00	58.6	0.35	0.39	0.18	0.31
AFL	6.60	10.8	6.60	8.00	7.55	7.81	9.28	8.21

Difficult example with conditional check | Source - Version								
Tool	**Time Taken (seconds)**				**CPU Usage (%)**			
	Try 1	**Try 2**	**Try 3**	**Average**	**Try 1**	**Try 2**	**Try 3**	**Average**
Radamsa	131	4.8	147	94	0.31	0.26	0.29	0.29
AFL	0.60	10.8	9.00	6.80	8.15	7.85	7.90	7.97

Appendix 3, Simple C++ program written by me with most basic form of buffer overflow

```C/C++
#include <iostream>
#include <cstring>

int main() {
    char buffer[10];
    std::cout << "Enter some text: ";
    std::cin >> buffer; // potential overflow if input longer than 9
chars
    std::cout << "You entered: " << buffer << std::endl;
    return 0;
}
```

Appendix 4, C++ program containing a realistic simple buffer overflow

```C/C++
#include <iostream>
#include <cstring>
#include <cstdlib>
#include <unistd.h>

#define BUFFER_SIZE 64
#define EXTRA_SIZE 128
```

```cpp
void performComplexOperation(const char* input) {
    char* buffer = new char[BUFFER_SIZE];
    char* extra = new char[EXTRA_SIZE];

    // initialize both buffers with known values
    memset(buffer, 'A', BUFFER_SIZE);
    memset(extra, 'B', EXTRA_SIZE);

    std::cout << "Buffer content before overflow: " << buffer <<
std::endl;

    // vulnerability: unsafe strcpy causing buffer overflow
    strcpy(buffer, input);

    // display content after overflow to observe effect
    std::cout << "Buffer content after overflow: " << buffer <<
std::endl;

    // 'extra' buffer to simulate critical memory corruption
    std::cout << "Extra buffer content: " << extra << std::endl;

    delete[] buffer;
    delete[] extra;
}

// intermediate function for complexity
void intermediateFunction(const char* input) {
    performComplexOperation(input);
}

int main() {
    std::cout << "Enter input: ";
    std::string userInput;
    std::getline(std::cin, userInput);

    // call func
    intermediateFunction(userInput.c_str());

    std::cout << "Program finished executing.\n";
    return 0;
}
```

Appendix 5, C++ program containing a difficult to find buffer overflow with a conditional check

C/C++

```
#include <iostream>
#include <cstring>
#include <cstdlib>
#include <ctime>

struct AuxiliaryData {
    int extra_value;
    char extra_buffer[8];

    AuxiliaryData() : extra_value(0x12345678) {
        std::memset(extra_buffer, 0, sizeof(extra_buffer));
    }
};

struct Data {
    int important_data;
    char buffer[16];
    int checksum;
    AuxiliaryData aux_data;
    bool valid;

    Data() : important_data(0xDEADBEEF), checksum(0), valid(true) {
        std::memset(buffer, 0, sizeof(buffer));
    }

    void update_checksum() {
        checksum = 0;
        for (size_t i = 0; i < sizeof(buffer); ++i) {
            checksum += buffer[i];
        }
        checksum ^= important_data;
    }

    void validate() {
        // complex validation logic
        valid = (checksum % 17 != 0) && (important_data % 11 == 0);
        valid &= (aux_data.extra_value % 13 == 0);
    }
};

void process_input(const char* input) {
    Data data;
```

```
    // check if input starts with specific prefix
    const char* prefix = "BFEE";
    if (strncmp(input, prefix, strlen(prefix)) != 0) {
        std::cout << "Input does not start with the required
prefix.\n";
        return;
    }

    // copy input to buffer without size check for overflow
    std::strcpy(data.buffer, input + strlen(prefix)); // Skip the
prefix

    data.update_checksum();
    data.validate();

    // conditions for crash
    if (!data.valid && (data.checksum % 19 == 0)) {
        std::cout << "Validation failed, checksum divisible by 19.\n";
        // cause real crash by accessing out-of-bounds memory
        int* crash = nullptr;
        *crash = 0; // cause segmentation fault
    }

    std::cout << "Buffer: " << data.buffer << "\n";
    std::cout << "Checksum: " << std::hex << data.checksum << "\n";
    std::cout << "Auxiliary Data Extra Value: " << std::hex <<
data.aux_data.extra_value << "\n";
}

int main() {
    char input[128];
    std::cout << "Enter input: ";
    std::cin.getline(input, sizeof(input));

    process_input(input);

    std::cout << "Program finished.\n";
    return 0;
}
```

Appendix 6, bash script output for most basic C++ program

```
Unset

--------------------------
```

```
running NON-Instrumented
-------------------------
Starting Radamsa fuzzing...
./../../test_app.sh: line 23: 172729 Done                    radamsa
input_cases/test_case
     172730 Segmentation fault      | "$file" > /dev/null 2>&1


-- [Radamsa] Crash detected! | Time: 00:00:00 (mins [0]) | Average CPU
Usage: 0%


Starting AFL fuzzing...
-- [AFL] Crash Detected! | Time: 00:00:01 (mins [0]) | Average CPU
Usage: 9.90%

-------------------------
running Instrumented
-------------------------
Starting Radamsa fuzzing...
./../../test_app.sh: line 23: 172980 Done                    radamsa
input_cases/test_case
     172981 Segmentation fault      | "$file" > /dev/null 2>&1


-- [Radamsa] Crash detected! | Time: 00:00:00 (mins [0]) | Average CPU
Usage: 0%


Starting AFL fuzzing...
-- [AFL] Crash Detected! | Time: 00:00:00 (mins [0]) | Average CPU
Usage: 7.30%


Fuzzing Complete!
```

Appendix 7, bash script output for realistic but simple C++ program

Unset

```
-------------------------
running NON-Instrumented
-------------------------
```

```
Starting Radamsa fuzzing...
./../../test_app.sh: line 23: 174051 Done                    radamsa
input_cases/test_case
     174052 Aborted                 | "$file" > /dev/null 2>&1


-- [Radamsa] Crash detected! | Time: 00:00:02 (mins [.03]) | Average
CPU Usage: .43%


Starting AFL fuzzing...
-- [AFL] Crash Detected! | Time: 00:00:01 (mins [.03]) | Average CPU
Usage: 10.80%

--------------------------
running Instrumented
--------------------------
Starting Radamsa fuzzing...
./../../test_app.sh: line 23: 174413 Done                    radamsa
input_cases/test_case
     174414 Segmentation fault      | "$file" > /dev/null 2>&1


-- [Radamsa] Crash detected! | Time: 00:00:02 (mins [.03]) | Average
CPU Usage: .10%


Starting AFL fuzzing...
-- [AFL] Crash Detected! | Time: 00:00:01 (mins [.03]) | Average CPU
Usage: 8.67%


Fuzzing Complete!
```

Appendix 8, bash script output for most difficult C++ program

Unset

```
--------------------------
running NON-Instrumented
--------------------------
```

```
Starting Radamsa fuzzing...
./../../test_app.sh: line 23: 237430 Done                    radamsa
input_cases/test_case
     237431 Segmentation fault      | "$file" > /dev/null 2>&1


-- [Radamsa] Crash detected! | Time: 00:00:36 (mins [.60]) | Average
CPU Usage: .35%


Starting AFL fuzzing...
-- [AFL] Crash Detected! | Time: 00:00:07 (mins [.11]) | Average CPU
Usage: 7.55%

--------------------------
running Instrumented
--------------------------
Starting Radamsa fuzzing...
./../../test_app.sh: line 23: 251117 Broken pipe             radamsa
input_cases/test_case
     251118 Segmentation fault      | "$file" > /dev/null 2>&1


-- [Radamsa] Crash detected! | Time: 00:02:11 (mins [2.18]) | Average
CPU Usage: .31%


Starting AFL fuzzing...
-- [AFL] Crash Detected! | Time: 00:00:01 (mins [.01]) | Average CPU
Usage: 8.15%


Fuzzing Complete!
```

Appendix 9, bash script for AFL fuzzing the most difficult C++ program

C/C++

```
#!/bin/bash

get_cpu_usage() {
    # get CPU usage using `top`
```

```
    top -bn1 | grep "Cpu(s)" | sed "s/.*, *\([0-9.]*\)%* id.*/\1/" |
awk '{print 100 - $1}'
}

echo "Starting AFL fuzzing..."

# remove output if exists
if [ -d "output_cases" ]; then
    rm -r output_cases
fi

# cpu tracking
total_cpu=0
num_samples=0

file=./vulnerable-instr

# AFL fuzzing
afl_start_time=$(date +%s)
afl-fuzz -i input_cases -o output_cases -- "$file" > afl_output.log
2>&1 &
folder=output_cases/default/crashes

afl_pid=$!

# monitor output for crash detection
while true; do
    if [ -d "output_cases" ] && find $folder -mindepth 1 -print -quit
| grep -q .; then
        kill -TERM "$afl_pid"
        wait "$afl_pid" 2>/dev/null

        # get time and calculate duration
        afl_end_time=$(date +%s)
        afl_elapsed_time=$(( afl_end_time - afl_start_time ))
        if [ "$num_samples" -eq 0 ]; then
            num_samples=1
        fi

        afl_cpu_usage=$(echo "scale=2; $total_cpu / $num_samples" |
bc)

        echo "-- [AFL] Crash Detected! | Time: $(date
-d@$afl_elapsed_time -u +'%H:%M:%S') (mins [$(echo "scale=2;
$afl_elapsed_time / 60" | bc)]) | Average CPU Usage: $afl_cpu_usage%"
        break
    fi
    # update CPU usage
```

```
        current_cpu=$(get_cpu_usage)
        total_cpu=$(echo "$total_cpu + $current_cpu" | bc)
        num_samples=$((num_samples + 1))
    done
```