Fuzz Compatibility Testing Results & Analysis

Charger Active Defense – G12

Noah Sickels, Adam Brannon, William Lochte

**Background**

Our senior design group is the second team working on the Charger Active Defense project. This project aims to develop a fuzzing workflow that effectively tests the networking aspects of the selected target applications, Medusa and Masscan. We strive to identify any hangs or crashes that may occur, which can then be sent back to the host machine to potentially disrupt or halt the adversary’s tool.

After background research and a screening process, we selected two attack tools to target and six fuzzing tools to use. A compiled list of tools is included in Table 1 below. When using each tool against our targets, two main methods are utilized: network protocol fuzzing and pcap traffic generation.

Table 1: Relevant Tools & Targets

|  |  |  |
| --- | --- | --- |
| Tool | Version | Use case |
| AFLnet (AFLplusplus) | 4.21c | Network Protocol Fuzzing |
| Fuzzowski | 0.8.2 | Network Protocol Fuzzing |
| Scapy | 2.6.0 | Pcap Traffic Generation |
| Radamsa | 0.7 | Network Protocol Fuzzing |
| Randpkt | 4.4.1 | Pcap Traffic Generation |
| Peach Fuzzer | 4.3 | Network Protocol Fuzzing |
| Medusa | 2.2 | Attack Tool Target 1 |
| Masscan | 1.3.2 | Attack Tool Target 2 |

The first method involves live fuzzing of the protocol or service stack, usually through a “wrapper,” which acts as an intermediary between systems. It enables the fuzzing tool to incorporate random data into the responses sent from the victim back to the host or vice versa to elicit a crash or hang in the application. This method offers greater accuracy by providing a more thorough representation of how the attack tool will behave in different network states. However, it demands more resources, requiring two virtual machines to run simultaneously—one for the attacking machine and one for the victim machine. Additionally, the wrapper is much more CPU-intensive, so faster hardware is needed to ensure better efficiency and timely performance.

The second method uses complementary tools to create packet capture files (commonly known as “pcap”) with randomly fuzzed traffic. Since both Medusa and Masscan support input from pcap files, this approach allowed us to test various aspects of the attacking tools while requiring fewer resources on the host machines. This method can be more challenging when managing overhead because it lacks an intermediary to organize test cases and record any crashes or hangs that occur. To address this, scripts would be necessary to simplify the testing process and compensate for these limitations.

This semester comprised three reporting periods, each dedicated to testing specific tools. The first period focused on AFLnet/AFLplusplus, the second on Fuzzowski, and the third on Scapy, Randpkt, and Peach Fuzzer.

Testing was conducted using two virtual machines (VMs), Kali Linux (Attacker) and Metasploitable2 (Victim), configured to communicate over an internal network. A detailed list of configuration settings and additional information can be found in the Appendix.

**AFLnet (AFLplusplus)**

AFLnet is a fork of Google’s American Fuzzy Lop plus plus (AFL++), designed as a greybox fuzzing tool for network protocol implementations. It takes a mutational approach and uses state feedback and code-coverage feedback to guide the fuzzing process [5]. Since our sponsor highlighted this tool in our project proposal slide, it was our first choice for initial compatibility testing.

To begin testing with Medusa and Masscan, AFLnet has a few essential requirements. Since AFLnet is a wrapper, it must attach itself to the desired target application through a process called “instrumentation.” Instrumentation of an application requires recompiling its source code using AFLnet’s custom compiler, afl-gcc. This modified version of the standard GCC compiler includes specific optimizations designed for use with AFLnet and AFL++. If you try to run the application without first instrumenting the binary, AFLnet will raise an exception and prevent it from executing. It is important to note that this process does not work with applications not primarily written in C or C++. Fortunately, this was not a concern for us, as Medusa and Masscan comprise 70.7% and 99.8% C/C++ code, respectively.

However, while trying to instrument the binary for Medusa, we encountered several problems. To test a live executable with AFLnet and AFL++, you have to use one of two virtualization modules provided: QEMU or LLVM. Since our attacking virtual machine was running Kali, QEMU was the optimal choice. Due to dependency errors, we struggled to compile the afl-qemu-trace module on the Kali machine, but we successfully built it after many attempts by exporting the PATH variables for QEMU. Nonetheless, even after building the QEMU module, we could not correctly instrument Medusa. After several hours of troubleshooting, we discovered that our issues stemmed from a dependency error from a specific OpenSSL header file from a deprecated version of the OpenSSL library that was unavailable on the Kali version we were using.

We used an Ubuntu 20.04 virtual machine to address this issue and successfully instrumented Medusa with AFLnet against our victim VM, as shown in Figure 1 below. While we could use Medusa during our screening process for attack tools, the binary we initially used was the compiled version provided by Kali’s default installation. Still, it could present more problems in the future.

**A computer screen shot of a program

Description automatically generated**

Figure 1: AFLnet Wrapper Running on Medusa

In contrast to Medusa, Masscan was successfully instrumented without any major issues; however, we still encountered a few complications. For network-based testing, AFLnet requires the server IP address and port you wish to test against. However, Masscan uses Internet Control Message Protocol (ICMP) by default, as it scans the given host or subnet through ICMP echo probes. Because ICMP is a separate protocol and does not designate a port number, providing the correct server information to AFLnet is problematic. Our solution was to specify a specific port on the host and use Masscan’s banner-grabbing capability to fuzz the particular port and protocol, as shown in Figure 2 below.

A screenshot of a computer screen

Description automatically generated

Figure 2: AFLnet Wrapper Running on Masscan

Both Medusa and Masscan are compatible with AFLnet; however, they share a common issue. As illustrated in the screenshots for both tools, the execution speed is extremely slow. This is indicated in red by “zzzzz...” in the Medusa screenshot and “slow!” in the Masscan screenshot. Unfortunately, due to time constraints, we could not determine how this would impact the fuzzing for both tools, and it will need to be explored further.

Moreover, AFLnet is limited in the protocols that it supports. During static analysis in our screening process, we noticed that memory leaks varied immensely depending on the Medusa module used. The primary example is Medusa’s SSH module, with 3,256 bytes of memory lost, against the PostgreSQL module, with 38,324 bytes of memory lost. From these results, we wanted to use the PostgreSQL module as it seemed most favorable, but AFLnet lacks direct support for the PostgreSQL protocol. However, since AFLnet is a community-driven, open-source project that provides extensive support for custom libraries across various protocols, it would be feasible to modify one of the existing modules or create our own for future testing.

Overall, AFLnet is a promising fuzzing tool. It is compatible with our attack tools, offers extensive support for custom modules across various protocols, and features a robust wrapper method for network fuzz testing. Since it was the initial candidate recommended by our project sponsor, we have decided to use AFLnet as one of our primary fuzzing tools moving forward.

**Fuzzowski**

Fuzzowski is a network-enabled fork of BooFuzz, which itself is a successor to the Sulley Fuzzing Framework. Fuzzowski has garnered attention for several reasons. As a fork of a well-known fuzzing framework, it has numerous predefined modules that make creating custom scripts for the desired protocols easy. Additionally, it natively supports several lower-level protocols, including the Line Printing Daemon and Modbus protocols [6]. These are similar to the ICMP protocols used in Masscan’s default scans, showcasing Fuzzowski's capability to interact with various TCP/IP stack layers beyond just layer 3.

Fuzzowski supports several protocols by default but does not natively support File Transfer Protocol (FTP), PostgreSQL, or Secure Shell (SSH). These Medusa modules were favored for testing, as identified in our initial screening. However, it does support Telnet, and we were able to confirm this functionality with Medusa, as shown in Figure 3 below. Additionally, utilizing the existing functions offered by Fuzzowski, we created a proof-of-concept FTP module with a snippet provided in the Appendix.

A computer screen shot of a computer program

Description automatically generated

Figure 3: Fuzzowski Fuzzing Output

Although Fuzzowski does have a lot of potential, this method does have several drawbacks. Firstly, this method sends the fuzzed traffic to the specific port and protocol provided. Since it does not have a wrapper or any connector to the targeted application, measuring its impact on the application becomes challenging, and any results found will have a higher false positive rate and potentially questionable accuracy. Secondly, the absence of a wrapper means the targeted application must run in a loop to generate traffic to the desired service. We ran Medusa continuously using a bash script to repeat the command until the user stopped the script, as shown in Figure 4 below. This solution is not ideal, highlighting one of the many challenges we faced with network-based fuzzing this semester.

A screenshot of a computer program

Description automatically generated

Figure 4: Medusa Repetition Bash Script

While Fuzzowski shows promise in terms of capability and functionality and is compatible with our attacking tools, we ultimately decided to prioritize other tools for protocol fuzzing. We believe using a wrapper for handling test cases is a more reliable method, but we may utilize Fuzzowski's built-in library for custom fuzzing cases in the second semester.

**Scapy & Radamsa**

Scapy and Radamsa are two well-established programs that we used together to generate random, fuzzed pcap files. Radamsa generates random data, while Scapy creates and packages the packets into pcap format. One of the significant challenges we faced this semester was using Masscan, as it has a custom, ad-hoc TCP/IP stack that is separate from the host machine. While Masscan’s banner-grabbing functionality enables it to perform the three-way handshake with any desired protocol or service, its default protocol is the Internet Control Message Protocol (ICMP). This protocol poses a challenge because it fundamentally differs from traditional protocols and does not use a port number. By utilizing Radamsa and Scapy, we can thoroughly test Masscan’s ICMP capabilities and any of Medusa’s modules.

To verify this functionality, we wrote a Python script that uses Radamsa and Scapy for this purpose. A snippet from the script is shown in Figure 5 below. The full Python script is available in the Appendix.

A screen shot of a computer program

Description automatically generated

Figure 5: Scapy & Radamsa Python Script Snippet

At first, while working with Scapy, we struggled to achieve satisfactory results when crafting the pcap file. To address this, we utilized the Python `randbytes` library as a temporary solution until we could obtain better results with Scapy and Radamsa. The FTP server script and the packet generation script are provided in the Appendix.

As we enter the second semester, our primary goal will be to concentrate on live protocol fuzzing. Given that Scapy necessitates extra scripting for automation and provides a limited view of ongoing tests, we have decided to use Scapy and Radamsa as an alternative solution for generating pcap files.

**Radamsa**

While Radamsa is a helpful complement to other tools like Scapy, it can also be used to fuzz network services by functioning as a TCP client or server [8]. However, it varies significantly between different services. For demonstration purposes, we set up a simple PHP HTTP web server on the local host, operating on TCP port 8080. After installing the necessary dependencies, including php-cli and curl, we created a directory named 'www' to host the web server. Inside this directory, we placed a basic index.html file containing a single HTML header:

<h1>Radamsa Network Service Test</h1>

We started the PHP server in the www directory with the php -S 127.0.0.1:8080 command. Then, we used the curl command from a separate terminal session to retrieve the index.html file, as demonstrated in Figure 6 below. This output confirmed that the server was functioning correctly. After setting up the web server, we created an HTTP request file named http-request.txt to serve as input data for Radamsa, as illustrated in Figure 7 below.

**A computer screen with text and numbers

Description automatically generated**

Figure 6: PHP Curl Output

A screen shot of a computer

Description automatically generated

Figure 7: HTTP Request File ‘http-request.txt’

Next, we utilized Radamsa to send the sample HTTP request to the PHP web server. In this scenario, Radamsa functions as a TCP client, connecting through a specified IP address and port. The command used for this process and the output is displayed in Figure 8 below.

A screenshot of a computer program

Description automatically generated

Figure 8: Radamsa Server Fuzzing Output

In the first terminal session, the fuzzed responses generated by Radamsa are sent to the server and recorded in the logs, as illustrated below. It is important to note that when Radamsa operates as a TCP client, it can potentially transmit malformed or malicious data over the network. While these error messages may appear problematic, they are expected behavior, as the web server views them as invalid, similar to packets created with Randpkt, as demonstrated in the next section.

To achieve better results, we intend to write automation scripts for this process using Radamsa to save any detected crashes or hangs. Unfortunately, we could not complete this due to time constraints during the reporting period.

A screen shot of a computer

Description automatically generated

Figure 9: PHP Server Logs from Radamsa Fuzzing

When using Radamsa as a TCP server, we successfully intercepted the responses from Medusa, which acts as our network service, allowing us to fuzz the responses before the attacking VM receives them. This method uses Radamsa to provide fuzzed versions of known valid outputs from the desired tool. In Figure 10 below, we use Medusa's PostgreSQL module on TCP port 5432 with a cycle count 4. Similarly to Randpkt, the packets are labeled as malformed or invalid, but this behavior is expected.

A screenshot of a computer

Description automatically generated

Figure 10: Radamsa Fuzzing with Medusa’s PostgreSQL Output

A screenshot of a computer

Description automatically generated

Figure 11: Wireshark Network Packet Capture of Medusa’s PostgreSQL Fuzzing

Given Radamsa's versatility in mimicking client and server interactions, its effectiveness as a complementary tool for generating fuzzed pcap files, and its straightforward installation and usability, we have decided to adopt Radamsa as our primary tool moving forward.

**Randpkt**

Randpkt is a small utility built into Wireshark designed to create trace files filled with random packets. This tool allows users to quickly generate randomized packets of a specific data type and save them in either pcap or pcapng file formats [17]. We initially discovered Randpkt while searching for tools to generate fuzzed pcap files. One of the convenient features of Randpkt is that it does not require scripting to specify the desired packets or protocols, unlike other tools such as Scapy and Radamsa. It also natively supports specialty protocols like ICMP and ARP without additional configuration. For example, as shown in Figure 12 below, you can use the command randpkt -t icmp test to generate ICMP packets.

**A screenshot of a computer

Description automatically generated**

Figure 12: Randpkt ICMP Pcap File

While Randpkt offers an easy method for generating pcap files in various protocols, it does not support modifying the contents of each packet. Therefore, we decided to use Randpkt as an alternative tool, but we believe that Scapy and Radamsa would provide a more customizable approach to creating the desired files. It is worth noting that the source code for Randpkt and its parent application, FuzzShark, is available under the GNU General Public License v3 (GPLv3). This licensing means that it is freely accessible for public use and can be utilized to develop a custom fuzzing tool in the future.

**Peach Fuzzer**

Peach Fuzzer is considered one of the first fuzzing tools ever popularized, with its release in 2004. Its strong network fuzzing capabilities made it a preferred choice for many users [12]. While exploring potential fuzzing tools for our project, we discovered a Peach Fuzzer XML "model" file for PostgreSQL and FTP, two of the Medusa modules we aimed to focus on, as illustrated in Figure 13 below. However, we faced numerous challenges during the installation and configuration of Peach Fuzzer.

A screen shot of a computer program

Description automatically generated

Figure 13: Peach Fuzzer PostgreSQL XML Model File

On June 11, 2020, GitLab acquired the official Peach Fuzzer project and all its repositories [14]. Following this acquisition, Peach Fuzzer transitioned from the GitLab repository "peach-fuzzer-community-edition" to "protocol-fuzzer-ce." The objective for Peach Fuzzer during this period was to refactor and modernize the project by gradually removing large files and slowly translating and committing them back over time [12].

As of the time of this project, the Peach Fuzzer project is incomplete. However, a community version is still available in its previous repository, though it is no longer maintained. The project uses Python 2.7 and outdated library versions of OpenJDK, Ant, TypeScript, NodeJS, and Intel PIN. Locating these deprecated libraries was challenging but manageable. Once we found all the necessary packages, we attempted to install them on our Kali Linux virtual machine, where we ran into our next hurdle.

Peach Fuzzer includes a built-in utility called "waf" that checks for the necessary libraries and packages to configure, build, and install the tool. While trying to run WAF on our Kali VM, several dependencies were flagged as missing. However, we eventually got the WAF utility to correctly identify the Linux kernel, as shown in Figure 14. From this point, we moved on to building Peach Fuzzer with WAF.

A screen shot of a computer

Description automatically generated

Figure 14: Peach Fuzzer WAF Configuration Output

However, when attempting to build Peach with WAF, we encountered a compilation error with the "ant\_glob" dependency. This library is responsible for accepting or rejecting nodes from NodeJS. After verifying correct configuration of our environment variables and PATH, the NodeJS package should have been visible system-wide, but was still not found by WAF. Further investigation revealed that the WAF kernel check produced a false positive. Although our system was a Debian flavor, it wasn't a proper Debian release. After researching online, we decided to use a Docker container to get Peach Fuzzer to run, as a skeleton of one was already made by a Peach Fuzzer community forum member.

The Docker image effectively resolved the issues with NodeJS and significantly simplified the management of packages and dependencies. However, new dependency issues emerged with the OpenJDK library. On June 30, 2022, Debian 9 discontinued support for their operating system's "Stretch" release branch [15]. Although the "Buster" release was still available, not all packages had been migrated, including the OpenJDK Java Runtime Environment version required by Peach Fuzzer. After trying various online suggestions to fix this, we were ultimately unsuccessful and decided against using Peach Fuzzer for this project. The final version of the Dockerfile is shown in the Appendix.

**Summary & Conclusion**

Based on our compatibility testing conducted during each reporting period, we have decided to concentrate on further testing with AFLnet and Radamsa using live protocol fuzzing. Although most of our selected fuzzing tools are compatible with Medusa and Masscan, we believe AFLnet and Radamsa are better suited for this project. A summary of our findings is presented in Table 2 below.

Table 2: Fuzzing Tools Compatibility Results & Priority

|  |  |  |
| --- | --- | --- |
| Tool | Compatible | Priority (1 - 5) |
| AFLnet (AFLplusplus) | Yes | 1 |
| Radamsa | Yes | 1 |
| Fuzzowski | Yes | 2 |
| Scapy | Yes | 3 |
| Randpkt | Yes | 4 |
| Peach Fuzzer | No | 5 |

In conclusion, network protocol fuzzing is more complex than we initially thought. Although many different fuzzing tools are available, they differ significantly in installation ease, usability, features, and fuzzing approaches. Even though this semester had some issues, we feel more confident going into the second semester.

**References**

[1] Kali, “Kali Tools | Kali Linux Tools,” *Kali Linux*. https://www.kali.org/tools/

‌

[2] “Debian Security Tools Packaging Team / medusa · GitLab,” *GitLab*, 2018. https://salsa.debian.org/pkg-security-team/medusa (accessed Oct. 27, 2024).

‌ [3] R. D. Graham, “robertdavidgraham/masscan,” *GitHub*, Oct. 31, 2020. <https://github.com/robertdavidgraham/masscan>

[4] “Debian Security Tools Packaging Team / reaver · GitLab,” *GitLab*, 2023. https://salsa.debian.org/pkg-security-team/reaver (accessed Oct. 27, 2024).

[5] aflnet, “AFLNet: A Greybox Fuzzer for Network Protocols,” *GitHub*, Oct. 01, 2023. https://github.com/aflnet/aflnet

‌

[6] nccgroup, “GitHub - nccgroup/fuzzowski: the Network Protocol Fuzzer that we will want to use.,” *GitHub*, 2019. https://github.com/nccgroup/fuzzowski/tree/master (accessed Oct. 31, 2024).

‌

[7] “Scapy,” *GitHub*, Aug. 10, 2022. https://github.com/secdev/scapy

‌

[8] “Aki Helin / radamsa · GitLab,” *GitLab*, 2024. <https://gitlab.com/akihe/radamsa/> (accessed Oct. 27, 2024).

[9] “Wireshark Foundation / Wireshark · GitLab,” *GitLab*, 2024. https://gitlab.com/wireshark/wireshark (accessed Oct. 27, 2024).

‌

[10] “randpktdump(1),” *Wireshark.org*, 2024. <https://www.wireshark.org/docs/man->pages/randpktdump.html (accessed Oct. 27, 2024).

[11] “Files · main · GitLab.org / security-products / protocol-fuzzer-ce · GitLab,” *GitLab*, 2024. https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce/-/tree/main?ref\_type=heads (accessed Oct. 27, 2024).

[12] “Peach Fuzzer,” *peachtech.gitlab.io*. <https://peachtech.gitlab.io/peach-fuzzer-community/>

[13] “DebianReleases - Debian Wiki,” *wiki.debian.org*. https://wiki.debian.org/DebianReleases

‌

[14] “GitLab,” Gitlab.com, 2021. https://about.gitlab.com/press/releases/2020-06-11-gitlab-acquires-peach-tech-and-fuzzit-to-expand-devsecops-offering/ (accessed Nov. 04, 2024).

[15] L. Grimmer, “OS Platform End of Life (EOL) Announcement for Debian Linux 9,” *Percona Database Performance Blog*, Aug. 03, 2022. https://www.percona.com/blog/os-platform-end-of-life-eol-announcement-for-debian-linux-9/ (accessed Nov. 04, 2024).

‌[16] “FuzzTesting,” *wiki.wireshark.org*. https://wiki.wireshark.org/FuzzTesting

‌ [17] “randpkt(1),” *Wireshark.org*, 2024. https://wireshark.org/docs/man-pages/randpkt.html (accessed Nov. 06, 2024).

[18] “Carbon,” *carbon.now.sh*. https://carbon.now.sh/

‌

‌

**Appendix A**

Table 1: Testbed Configuration

|  |  |  |
| --- | --- | --- |
| Component | Configuration |  |
| Hypervisor | VirtualBox | 7.1.0 |
| Virtual Machine 1 (Host) | OS | Kali Linux 2024.3 |
| Kernel Version | Linux kali 6.10.11-amd64 |
| GCC Version | 14.2.0 (Debian 14.2.0-3) |
| Network Adapter 1 | NAT |
| Network Adapter 2 | Internal Network *(intent)* |
| IP Address | 192.168.1.99 /24 |
| Miscellaneous | 16,384 MB RAM, 2 CPU cores, 80 GB HDD |
| Virtual Machine 2 (Target) | OS | Metasploitable2 |
| Kernel Version | Linux Metasploitable 2.6.24-16-server |
| GCC Version | 4.2.4 (Ubuntu 4.2.4-lubuntu4) |
| Network Adapter 1 | Internal Network *(intent)* |
| Network Adapter 2 | *Optional* |
| IP Address | 192.168.1.100 /24 |
| Miscellaneous | 2,048 MB RAM, 1 CPU core, 8 GB HDD |

A screen shot of a computer program

Description automatically generated

Figure 1: AFLnet Instrumented Masscan Binary

A screen shot of a computer program

Description automatically generated

Figure 2: Fuzzowski Proof of Concept FTP Module

A screen shot of a computer screen

Description automatically generated

Figure 3: randbytes Library Python Script

A computer screen shot of a program

Description automatically generated

Figure 4: Scapy Packet Crafting Script

A screen shot of a computer program

Description automatically generated

Figure 5: Peach Fuzzer Dockerfile