A Novel Interactive Network Fuzzer for System Security Assessment

Jaime C. Acosta[1], Christian Murga[2], Alberto Morales[3], Caesar Zapata[4]

[1]*CCDC Army Research Laboratory*
[2]*GreyBox Security*
[3]*University of Texas at El Paso*
[4]*CCDC Data and Analysis Center*

Abstract

Network security testing can be done at different levels of fidelity. This can involve simply scanning a network to identify any open ports for services and versions of services, to uncovering novel vulnerabilities in proprietary or undocumented services. The granularity of such an analysis depends not only on time and cost, but also on the availability of client software that can be used to interact with the different services. Complexity increases when the underlying protocol is undocumented or nontrivial. In this case, testers must first understand the protocols, and then develop software that can be used to interact; past the common handshake or initial connection behavior to uncover vulnerabilities. In this paper, we present an architecture that marries protocol reverse engineering and network fuzzing through a graphical interface. We have developed a proof of concept (PoC) that is capable of intercepting packets between source and destination nodes; allowing analysts to use the interface to interactively or pseudo-interactively (using hooks) observe, modify, drop, and/or forward the traffic during security tests. We designed our experimentation methodology with two perspectives in mind: blue-teaming (cooperative grey/white box) and red-teaming (non-cooperative, black box). We report performance of our PoC with the Transport Control Protocol.

Keywords: *Network Security Testing, Interactive Fuzzing, Network Defense*

1. Introduction

Given the ubiquitous nature of software and the growing complexity of technology and interconnected systems, magnified by adversarial techniques such as social engineering, the idea of keeping systems secure seems a distant goal. Risk assessments are a necessity to ensure that decision makers understand the vulnerabilities on their systems and the potential impacts of an adversarial event. For this reason, it is critical that network security testers have at their disposal tools and techniques that allow for efficient testing; as time and cost are precious resources.

The network technology space is very dynamic; new devices are constantly developed and allowed on networks that access and are accessible across the globe. The Internet of Things (IoT), and in the military world, the Internet of Battlefield Things (IoBT), systems are good examples. While some of these use and rely on vetted and known network protocols; most difficult to test are the

nuances that are specific to the devices. As an example, many times, devices use the traditional Transport Control Protocol/Internet Protocol (TCP/IP) network stack to handle packet fragmentation, reordering, transmission control, and operating system sockets and ports. On transmission, the payload data (or the data that's generated specifically by the device or application), is encapsulated within the other layers. On arrival, each of the layers is removed and processed and eventually the payload data is read and used by the target entity. This payload data may also have additional control logic in place. Since this code is often times specific to the technology documentation related to protocol, details and software source code are not readily available. This makes medium to high-fidelity fuzzing difficult. For a network security analyst, automation and adaptation are critical for testing and uncovering vulnerabilities in these systems. This automation must also take into account the objectives of the particular test. In the case of blue teaming assessments, more emphasis is placed on coverage (e.g., attempting to exhaustively find system weaknesses with granted access to systems). Red-teaming on the other hand is focused on testing systems with a black-box view and finding at least one successful attack vector.

With this as our focus, we present an interactive network fuzzer that derives protocol information from real data from real applications, by leveraging network traffic. Our contributions are the following:

- A novel, and extensible, architecture that consists of a traffic-based model generator (TBMG) and the interactive proxy fuzzer (ProxyFuzz).
- A proof of concept (PoC) system that is capable of modifying traffic on-the-fly either through automated scripts or through manual user interaction.
- A performance study using the PoC on both a laptop and on a Raspberry Pi.

2. Related Work

The concept of traffic interception and modification is not new. Burpsuite (Wear, 2018) and Zap (Bennetts, 2013) are publicly available proxy tools that are used to capture web traffic, and, through a graphical interface allow testers to observe and modify HTTP messages such as POST requests; which may times may hold sensitive data that are not apparent from the browsers. The Non-HTTP Protocol Extension plugin for Burpsuite adds support for raw TCP protocol messages – allowing users to, for example, match and replace strings as they pass through the proxy. These packets are not dissected past the TCP layer (so users are provided with ASCII or raw hex). Additionally, these services intercept packets in user space (after the kernel has processed them).

From the defense perspective, related technologies: honeypots, have been around for several decades. In the late 90s, the international nonprofit Honeynet project (Watson & Riden, 2008) was launched, and with it, documentation and tools to allow analysts to deceive adversaries and collect relevant information. In 2003, the Honeyd software (Provo 2003) was made available to the public. It is now capable of claiming over 65000 network addresses and 1000 service personalities (derived using nmap (Lyon 2009) and xprobe2 (Arkin, Yarochkin & Kydyraliev, 2003) scan fingerprinting rules). Researchers were able to automate the generation of service personalities for honeyd using protocol characteristics

and traffic flows (Leita, Dacier, & Massicotte, 2006). However, from a tester's perspective, and as part of live-network fuzzing, it is critical to analyze and modify packets as they traverse the network.

While these technologies are very successful in their respective domains, the work presented here, focuses on live-network assessment; in the cases where protocol communication software is not available to the analyst and fuzzing is critical to uncovering vulnerabilities.

3. Interactive Fuzzer

3.1. System Architecture

The goal of the architecture presented here is to generate skeleton code that incorporates values and behaviors of a real network service, as observed through network traffic. Afterwards, the security tester can refine and improve the fidelity for aspects of the protocol that are not represented in the network traffic. There are many benefits to using this approach. First, even with a short network capture, the generated skeleton will exhibit values that are specific to the technology. For example, an analyst can easily recreate what appears to be a ping originating from different operating systems by modifying the initial time to live fields (TTL) in the messages and default payload data and size (Subin, 2000). In addition, an analyst can choose one of these fields and then test boundary values either manually, using scripts, or by using external fuzzers (e.g., American Fuzzy Lop) to test ranges of values.

The architecture is composed of two main components (see **Figure 1. System Architecture**). The traffic-based model generator is used to extract network packet structures and values from packets. ProxyFuzz is used to capture and modify packets using automated, rule-based scripts as well as through manual user interactions through a graphical interface.

We built a PoC to demonstrate the key components of this architecture including its feasibility and potential application.

3.2 Traffic-Based Model Generator

The traffic-based model generator is responsible for reading traffic and parsing out structures and field values from packet data.

This module works by using Wireshark dissectors to parse packet data into a structured form: packet data markup language (PDML) (Wagener, Popp, Hahn, & Brück, 2002). A vocabulary extractor stores packet field values and a state machine extractor infers sequencing patterns associate with packet types. Afterwards, this data is used to generate Python source code files that use Scapy for networking components such as packet construction, checksum calculation, and receive/transmit functionality.

This module can be run independently from the others. Additionally, this module can also be used for security testing (e.g., for fuzz testing proprietary protocols). A more complete description of this component as well as a case study is described in (Acosta & Estrada, 2017).

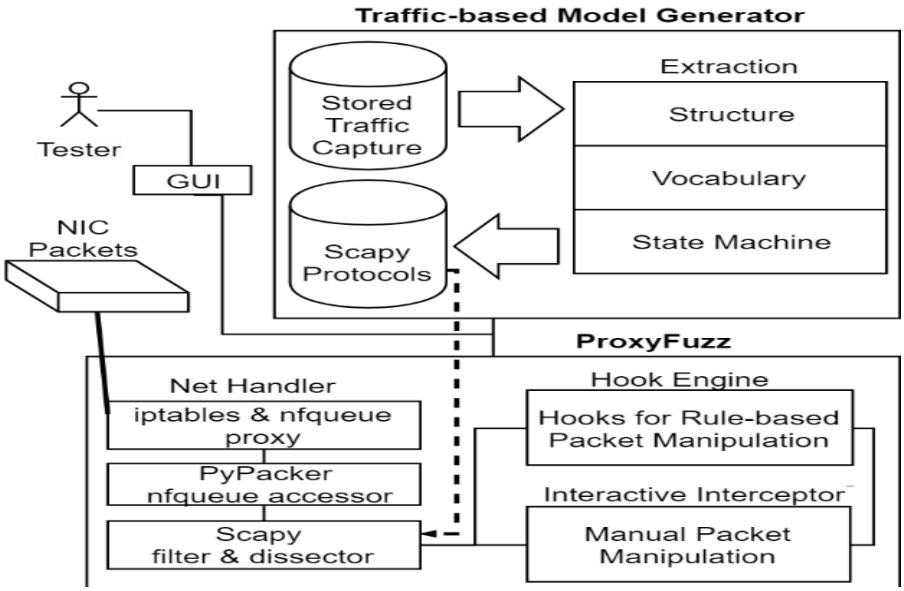

Figure 1. System Architecture

3.3. ProxyFuzz

ProxyFuzz is composed of the net handler, the hook engine, and the interactive interceptor. The net handler primarily uses iptables and nfqueue, as demonstrated in (Nunn, 2018) to capture all IP packets that arrive or will be sent out on the network interface card. Backend PyPacker (Stahn, 2018) scripts are invoked to interact with the packets in the nfqueue (including modification and deciding whether to forward or drop packets) and Scapy is used to dissect packet data and apply capture filters. Unfiltered packets are processed by the hook engine component. Using this approach, incoming packets can be modified or discarded before they are processed by other applications in the operating system. Similarly, the packets may be modified or discarded just before they are sent through the network interface card.

In the hook engine, small code snippets, written in Python, that may use Scapy or other packet manipulation application programming interfaces (APIs), are used to automatically apply some logic to packets based on a set of rules. As an example, a user may specify a rule that matches packets with a particular destination IP Address and then will change the TTL field to make it appear that the packet traversed a set number of gateway devices. The hook engine is built using a plugin architecture, allowing users to develop hooks without needing to know the internals of ProxyFuzz. The hook plugins can be grouped and applied collectively (which is useful when attempting to mimic services that are commonly attributed to operating systems and devices – such as standard services that are enabled with a fresh installation of Windows, Linux, MacOS, or specific IoT and embedded devices). On startup, ProxyFuzz scans for these hook plugins and will apply those that are tagged for use. After hooks are applied, the packets may be forwarded to the Operating System, or if the interceptor component is enabled, will be placed on a queue and will appear on the graphical interface.

The graphical interface is primarily used to manage the interactive interceptor, although it can also be used to enable and configure all of the other components. With respect to the interceptor, users can select packets and view field values, as well as choose to drop, edit, and forward; all in real time. A tabular interface separates features into three categories: incoming packets, outgoing/forwarded packets, and settings.

The settings tab allows users to specify interfaces that should be used to intercept packets. The incoming and outgoing/forwarded packets tabs show a queue of selectable packets for those arriving and those being sent out respectively. When a packet is selected, the labeled field details, based on the dissected structures, are displayed in editable text entries. If field values are modified, then associated length and checksum fields are automatically recalculated. When a packet is forwarded from the PoC, any other packets that are ahead in the queue will also be forwarded. When a packet is selected to be dropped, other packets in the queue will be unaffected.

To determine how well the system would work in a realistic scenario, we analyzed its performance against a widely used network scanner.

4. Experimental Design and Setup

In order to generate a meaningful and practical experiment, we informally consulted several cybersecurity analysts in order to understand gaps and critical needs. Based on these interactions, we continually developed and refined our experimentation setup. The end result was a flexible yet realistic study that reveals the practicality of the the PoC during red and blue-teaming events.

When devices communicate using the TCP/IP network stack, the operating system creates a socket and listens for communication on a port. To test the performance of our PoC, we used the popular network mapper (nmap) tool to generate connection requests.

Nmap is used to identify services running on systems (e.g., by attempting to establish a connection to the system on a specific port) and it is capable of inferring information about services to include version numbers (by, e.g., banner grabbing, or analyzing the first message after a successful connection and mapping to a known service), operating systems (e.g., by matching known characteristics to include timing and packet field values), and other system information. Basic scanning, which is our focus here, is the default scanning configuration: nmap will simply run a port scan on a specified range of systems.

In our PoC, we developed a hook that would identify TCP packets and respond with valid packets associated with the handshake (correct sequence number, ports, connection flags, etc.). We wanted nmap to report that all scanned ports were open and listening for a connections.

We envision this tool being useful in several environments ranging from large enterprise networks, to performance constrained systems, such as IoT, embedded and/or those commonly used in tactical networks. For this reason, we tested performance using modern laptops (Dell Precision 7720 with a 7th generation i7 processor and 16GB of RAM) and on a Raspberry Pi device (Raspi2 Model B with 900 Mhz ARM Cortex-A7 processor and 1GB RAM). As shown in **Figure 2. Experimental Setup**, the client machine used the nmap tool to port scan a server machine with 5 different intensity configuration (using the -T option and values

from 1-5). We incrementally probed the top 500 most commonly used port. In addition, we had 4 different configurations. In configurations A and B, the client targeted a device that was not running the PoC; a laptop and a Raspberry Pi respectively. In configurations C and D, we had a device (laptop or the Raspberry Pi, respectively) running the PoC. During our study, we observed differences in nmap completion times and scan results.

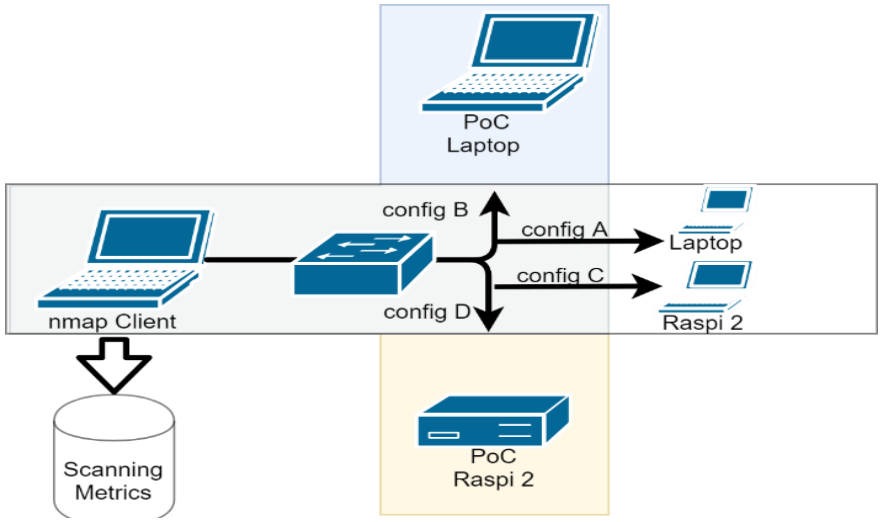

Figure 2. Experimental Setup

5. Results

Our results show that the proof of concept system is practical on both modern laptop systems and, in certain circumstances, also on Raspberry Pi hardware. Table 1. Port Scan Completion Times (in seconds) shows the time taken for an nmap scan to complete under different configurations.

Using the Laptop configuration, when using a hook to intercept and modify values of up to 300 ports, only a small delay is incurred by the POC. This is a good indication that a tester could use the system to introduce fuzzing and packet modifications (to, e.g., test how devices behave with boundary or invalid field values) without perturbing the timing between legitimate end points. A small delay of roughly 12-20 seconds is evident when using the software to intercept traffic on 500 ports.

Use of the proof of concept is more limited, but still useful with the Raspberry Pi. Even though with most profiles, noticeable delays are introduced, in all cases except T5, the nmap scan results show all ports as open, meaning that a connection could be established; with the proof-of-concept in place. When using T5, the scan results were inaccurate.

While, there is still room for improvement regarding performance, it is worthwhile to note the same program instance (including the hooks) works on both platforms (x86-64 and ARM architectures) without modification.

Table 1. Port Scan Completion Times (in seconds)

# Ports	Profile	Config A L-NoProxy	Config B L-Proxy	Config C R-NoProxy	Config D R-Proxy
100	T2	40.96	40.88	40.88	41.28
	T3	0.02	7.88	0.04	109.43
	T4	0.02	6.46	0.04	214.05
	T5	0.02	3.41	0.04	N/A
300	T2	121.03	120.96	40.88	41.28
	T3	0.03	16.6	0.09	261.91
	T4	0.03	16.61	0.12	469.03
	T5	0.03	5.95	0.04	N/A
500	T2	201.16	201.04	201.5	201.22
	T3	0.03	15.77	0.14	326.13
	T4	0.03	19.7	0.15	549.4
	T5	0.03	12.62	0.04	N/A

6. Conclusions

We have presented an architecture and a proof of concept for an interactive fuzzer. The fuzzer allows analysts to capture packets and make modifications through manual or pseudo-automated means. Against basic TCP probes from the nmap tool, our system introduces latencies, but in most cases, scan results are still accurate even when using different scanning configurations.

In future work, we will improve the system performance by experimenting with different nfqueue parameters and revisiting the system's internal design. We will also expand the capabilities of the PoC to incorporate complex behaviors of protocols and incorporate elements from honeyd, to include the use of nmap and xprobe2 fingerprint rules, to mimic network service behavior. For protocols that have no specifications, we will introduce a dissector generator module that will allow analysts to use a graphical interface to develop lua dissectors, leveraging Netzob and other related work.

References

Acosta, J. C., & Estrada, P. (2017, May). A preliminary architecture for building communication software from traffic captures. In Disruptive Technologies in Sensors and Sensor Systems (Vol. 10206, p. 102060T). International Society for Optics and Photonics.

Arkin, O., Yarochkin, F., & Kydyraliev, M. (2003). The present and future of xprobe2: The next generation of active operating system fingerprinting. sys-security group.

Bennetts, S. (2013). Owasp zed attack proxy. AppSec USA.

Leita, C., Dacier, M., & Massicotte, F. (2006, September). Automatic handling of protocol dependencies and reaction to 0-day attacks with ScriptGen based honeypots. In International Workshop on Recent Advances in Intrusion Detection (pp. 185-205). Springer, Berlin, Heidelberg.

Lyon, G. F. (2009). Nmap network scanning: The official Nmap project guide to network discovery and security scanning. Insecure.

Nunn, X. (2018) Scapy_bridge.py. Retrieved from https://gist.github.com/eXenon/85a3eab09fefbb3bee5d; Accessed December 20, 2018.

Provos, N. (2003, February). Honeyd-a virtual honeypot daemon. In 10th DFN-CERT Workshop, Hamburg, Germany (Vol. 2, p. 4).

Stahn, M. (2018) Pypacker. Retrieved from https://github.com/mike01/pypacker

Subin, Siby (2000) Default TTL (Time to Live) Values of Different OS. **(**April 2014). Retrieved from https://subinsb.com/default-device-ttl-values/; Accessed December 20, 2018.

Wagener, A., Popp, J., Hahn, K., & Brück, R. (2002). PDML-A XML-based process description language. In Proceedings of the 9th European Concurrent Engineering Conference, Modena.

Watson, D., & Riden, J. (2008, April). The honeynet project: Data collection tools, infrastructure, archives and analysis. In 2008 WOMBAT Workshop on Information Security Threats Data Collection and Sharing (pp. 24-30). IEEE.

Wear, S. (2018). *Burp Suite Cookbook: Practical recipes to help you master web penetration testing with Burp Suite*. Packt Publishing Ltd.