Black-Box Fuzzing for Security in Managed Networks: An Outline

Leon Fernandez and Gunnar Karlsson, *Senior Member, IEEE*

***Abstract*—Service providers are adopting open-source technology and open standards in their next-generation networks. This gives them great flexibility and spurs innovation. But it also means that they must ensure proper interoperability between components; otherwise, vulnerabilities might get introduced in their networks. Unfortunately, state-of-the-art vulnerability scanning tools are unable to handle the complexity of service provider networks. In this letter we show how interoperability issues between seemingly reliable components introduce an injection vulnerability that allows us to control a firewall-protected network management system. We also extend the state-of-the-art in black-box fuzzing to give service providers a tool for combating similar issues.**

***Index Terms*—Cyber security, network management, protocol testing.**

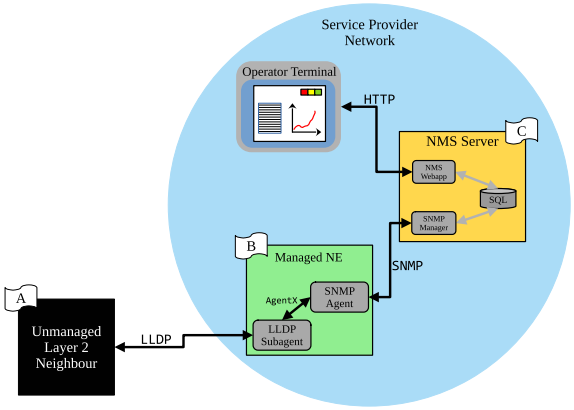


Fig. 1. A diagram showing some of the entities involved when an operator views a network element with a Web-based network management system.

I. Introduction

NETWORK management systems (NMS) are a core component of many service provider (SP) networks. Their purpose is to aid the operator in managing and monitoring a network by providing a single point-of-access to all the nodes that make up the network. NMSs are often expected to be compatible with a wide range of networking technologies. Consequently, NMSs become a nexus for many different communication protocols and several components and protocol implementations may partake in the delivery of the management data from the nodes to the NMS. Due to the crucial role a NMS plays in its network, correct interoperability is of essence. Figure 1 gives an example of some of the protocols involved when viewing the network topology in a Web-based NMS. In the example, a managed network element (NE) picks up a Link Layer Discovery Protocol (LLDP) frame from a neighbour in some other network and a Simple Network Management Protocol (SNMP) agent on the same NE stores the LLDP information. An NMS server queries the NE using SNMP and stores the response in a SQL database that is accessed by the NMS Web application when an operator views the NE status with a Web browser. In next-generation open networks, the implementations that partake in this delivery may well come from different vendors. Ensuring secure and robust interoperation thus falls on the SP. With the NMS being a central function in the network, it becomes a suitable entry point for a high-level vulnerability analysis. Hence, there is a need for new vulnerability scanning tools in order to properly assess the security of Web-based NMS deployments, which is where our research fits in. More specifically, we will focus on injection-type vulnerabilities [1], where an attacker can inject snippets of code (often HTML, JavaScript or SQL) that get executed by the target system. While the consequences of injection vulnerabilities are situational and very difficult to predict, ranging from being a mere nuisance to full compromise of the target system, they are never tolerated in production-grade systems. Therefore, we dedicate this letter to investigating where these vulnerabilities may arise in modern open networks and what research is needed to give SPs better tools to assess the security of their deployments.

The main contributions of this letter are:

- Highlight and examplify how injection-type Web vulnerabilities might manifest themselves in open service provider networks.
- Present a testing framework for finding such vulnerabilities.

II. Related Work

Web application vulnerability scanning is an active research topic. To deal with the complexity of modern Web applications, a black-box approach is typically employed. Enemy of the State [2] and Black Widow [3] are both able to navigate and to find vulnerabilities in production-grade Web applications. However, they are both limited to using the HTTP protocol, which is what our framework intends to extend.

Fuzz testing [4] is often employed for more general network protocol vulnerability scanning. The basic idea behind fuzz testing is to feed "fuzzed", i.e., randomized or otherwise corrupted, data into the system under test (SUT). State-of-the-art fuzzers employ what is called grey-box fuzzing, where

Manuscript received 14 February 2023; revised 12 April 2023; accepted 3 June 2023. Date of publication 15 June 2023; date of current version 2 January 2024. The associate editor coordinating the review of this article and approving it for publication was M. Aloqaily. *(Corresponding author: Leon Fernandez.)*

The authors are with the Centre for Cyber Defence and Information Security (CDIS), KTH Royal Institute of Technology, 100 44 Stockholm, Sweden (e-mail: leonfe@kth.se; gk@kth.se).

Digital Object Identifier 10.1109/LNET.2023.3286443

specialized compilers and execution environments are used to effectivize the fuzzing process. AFL [5] and its descendant AFLNET [6] belong to this category. Due to being dependant on special compilers and execution environments, grey-box fuzzers are unsuitable for distributed, multi-language SUTs such as an open SP network. *Black-box* fuzzing, abstracts away the internals of the SUT and is therefore more suitable if the system is large and complex. Unfortunately, state-of-the-art black-box fuzzers such as Boofuzz [7] and Snipuzz [8] make simplistic assumptions about how to generate the fuzzy data. This is done with simple byte-level manipulations such as randomly flipping bits and is therefore inefficient for generating injectable code.

The framework we present in Section V addresses this by using a grammar for greater control of how the input is generated, thereby allowing us to efficiently generate injection-prone data. We are also able to detect injection vulnerabilities by monitoring the NMS whereas other black-box fuzzers typically only detect failures that manifest themselves as outright crashes.

III. Web Application Security

The Open Web Application Security Project ranks code injection as the third most serious vulnerability for Web applications [9]. A common type of injection vulnerability is cross-site scripting (XSS), the most severe of which is called stored XSS or persistent XSS. Stored XSS typically occurs in the input fields of a Web page. Figure 2 illustrate a basic example of an XSS attack against Alice and Bob, two users of a hypothetical Web application that supports user input via a chat feature. The evil Mallory enters into the comment field a string that is not properly sanitized and therefore gets "stored" as part of the page's HTML source. Alice or Bob, or any other visitor who can view the comments, will upon visiting that comment field be subjected to Mallory's attack where her JavaScript code will execute in their browsers.

As mentioned in the introduction, the consequences of XSS attacks can vary widely depending on circumstances such as what browser the visitor is using, what other tabs are open in the victim's browser, or the settings in the Web application's backend.

Web-based NMSs or NMSs with Web consoles naturally expose themselves to these types of issues as well. The risk is arguably mitigated by the fact that the NMS server is often not exposed to the public Internet, but is only reachable from within the company firewall. Considering the complexity of Web security, this could be a dual-edged sword where the operators become too reliant on the firewall, thereby making the security configuration of the NMS more prone to human errors. Furthermore, the findings we present in the next section indicate that there may be scenarios where injection attacks can circumvent the firewall entirely.

IV. A Vulnerable Architecture

Referring back to Figure 1, we think of it as a small part of a SP network. A NE is monitored via SNMP, for which a dedicated agent is in charge. The SNMP agent communicates

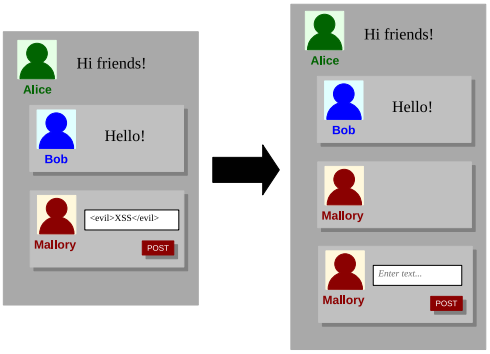


Fig. 2. A XSS attack where the attacker posts a comment whose contents get misinterpreted as HTML and thus becomes part of the website instead.

with another process on the same NE, namely a Link Layer Discovery Protocol (LLDP) subagent using the AgentX protocol. The LLDP subagent listens to all layer 2 ports on the NE for LLDP packets in order to discover its layer 2 neighbours. This is how topology discovery is done in many networks. When the SNMP agent is queried by the NMS, it collects what the LLDP subagent has discovered in its response to the NMS. Later, this information can then be viewed by an operator in the NMSs Web console.

Suppose that the managed NE is on the edge of the SP network. The LLDP packet it receives will be coming from some untrusted device, perhaps from another SP or from an end-user. This provides a way for untrusted strings to travel all the way to the NMS and its Web console. Despite the NMS server being only accessible from within enterprise firewalls, it can still suffer injection attacks similar to those that public sites may suffer, albeit not with HTTP requests as the carrying protocol.

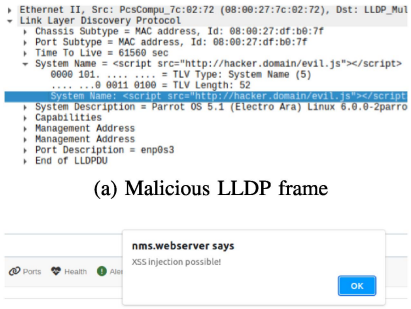
A. Our Findings

During an investigative audit of LibreNMS [10], a popular Web-based NMS (≈3k stars on GitHub), we have come across at least one verified injection vulnerability in a setup illustrated by Figure 1. Using the System Name field in an LLDP frame, we were able to inject malicious HTML code into the NMS without having IP connectivity to the NMS server, as depicted in Figure 3. This was done using the official server VM image[1] with default settings.

Our attack injects an HTML `<script>` element into one of the pages of the NMS. Upon visiting that page, the victim's browser will fetch and run arbitrary JavaScript code that we control. This way, we were able to interact with the NMS on the victim's behalf, meaning we could bring up or down services or change the victim's password. We were also able to exfiltrate sensitive data such as network topology and billing information. As a proof-of-concept of a working attack, the fetched script raises an alert dialog, as seen in Figure 3(a).

The issue was disclosed in private and patched in a subsequent version.[2] Figure 3(a) shows a dissected view of the malicious LLDP frame. Note the System Name field contains

[1] Available at https://github.com/librenms/packer-builds/releases/tag/21.2.0.
[2] Available at https://github.com/librenms/librenms/pull/14658.



(a) Malicious LLDP frame

(b) Proof-of-concept alert dialog

Fig. 3. A malicious LLDP frame that will cause the victim's browser to fetch a script from an attacker-controlled computer. The script raises an alert dialog as a proof-of-concept of a working injection attack.

illegal characters according to the LLDP standard but it was forwarded nonetheless.

B. Who Is the Culprit?

Our testbed was built using various open-source programs in an attempt to emulate and illustrate how open SP networks may be built today. Table I lists the core components. A natural question to ask regarding integration vulnerabilities is "Who is the culprit?". Unfortunately, there is no clear answer to this question. LibreNMS does not sanitize the incoming LLDP data, which poses a risk. But according to the LLDP standard, the System Name field should only contain alphanumeric characters. Thus, the LLDP and SNMP agents are also introducing a risk by forwarding frames that do not adhere to the protocol. When used together, these components introduce an exploitable vulnerability, as we have shown here. It is possible that the vulnerability would not have been present if other components than those in Table I were used. This uncertainty is of concern to service providers because they are the only ones who can thoroughly test *their* selection of components. SPs can not solely rely on the testing practices of third-party vendors because it is unfeasible for them to test all the possible scenarios where their component might be deployed. Black-box fuzzing is a viable strategy that SPs can adopt to secure their systems. In the next section we present our framework, which addresses the shortcomings of the state-of-the-art.

V. Proposed Testing Framework

To combat this issue of "malicious strings" propagating through the network, we propose a framework based on the concept of grammar-based fuzzing [11]. Fuzzing, is a popular methodology for security testing and vulnerability discovery. The core concept is to input data that has been randomized in some (more or less) systematic way while monitoring for unexpected behavior in the system under test. We take a black-box approach to fuzzing, meaning we make minimal assumptions about the system internals. Most importantly, we do not make any assumptions about the source code of the software in the testbed, meaning our framework can handle multi-platform and multi-language systems with a mix of proprietary and open-source components.

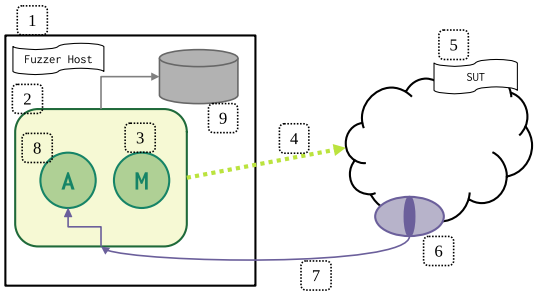


Fig. 4. An overview of a test setup based on our fuzzer. Details are found in Section V-A.

A. Fuzzer Overview

Figure 4 shows an overview of a setup using our fuzzer [12]. The numbers mentioned in this paragraph refer to the numbers in Figure 4. The core part of our fuzzer runs on a generic host system (1) that is physically connected to the SUT. The fuzzer program (2) uses a grammar-based model (3) for generating the traffic (4), in this case consisting of malicious LLDP frames. The frames enter the SUT (5), making the fuzzer look like a discovered neighbour. To check whether the LLDP frame has been handled correctly, a probe (6) is monitoring relevant points in the SUT. This is where some assumption about the SUT internals has to be made, depending on the desired detail. For testing a system such as the one in Figure 1, a probe acting as the end user's Web browser might suffice, i.e., a small program making HTTP GET requests after each transmitted LLDP frame. If more detail is needed, more probes can be deployed, for instance, to make SNMP requests to intermediate points inside the SUT. The fuzzer itself makes no assumptions about where the probes are deployed as long as it can establish a network connection to them. The data gathered by the probes are fed back (7) to the fuzzer where they are processed by its arbiter subcomponent (8). The arbiter, like the probes, is written by the tester and its main responsibility is determining if an LLDP frame has been handled correctly based on the probe data. Each transmitted frame makes up a test case and the fuzzer stores the data from each test case in a database (9) that can then be explored by the tester.

B. Design Considerations

Black-box fuzzing has the advantage of being able to fuzz many different types of targets. To build upon that flexibility, many components in the fuzzer are implemented as plug-ins, which is why we call it a testing *framework*. Most importantly, the grammar-based model can be easily replaced by the tester's own model simply by pointing it out in a configuration file. Moreover, the probes use a well-defined interface for communication with the fuzzer, giving the tester freedom when implementing the probes, such as the choice of programming language and where and how to deploy the probes. The arbiter component is also implemented as a plug-in. Depending on the probe configuration, different pass/fail criteria might be needed which is why the arbiter should be easily replaceable as well. Lastly, the delivery mechanism for the fuzzed traffic is also replaceable. Different test targets require different

TABLE I
COMPONENTS IN THE TESTBED. VALUES IN RIGHTMOST COLUMN REFER TO THE LABELS IN FIGURE 1

Component name	Function	Deployed on
Fuzzer [12]	Fuzzer core	Node A/Fuzzer Host
HTTP Probe	For probing the NMS	Node A/Fuzzer Host
lldpd [16]	LLDP implementation	Node B
Net-SNMP [17]	SNMP implementation	Node B
LibreNMS [18]	Web-based network manager	Node C

connection methods: TCP sockets, UDP sockets, raw Ethernet sockets and so on. Furthermore, different steps, such as authentication, might be required in order to test a certain type of frame or packet.

Our fuzzer is open-source software.[3] This is arguably the most important design choice as it allows for greater re-use of plug-ins and probes between users in different organizations.

C. Proof-of-Concept Implementation

Our framework [12] has been implemented at a proof-of-concept level using the Python programming language. It uses Python's native dictionary type to implement the protocol grammars in a manner largely inspired by Zeller et al. [11]. Plug-in components are simply Python modules that expose certain functionality as expected by the fuzzer. The probe interface is specified using the Google Protobuf interface description language [13]. gRPC [14] can then be used to generate skeleton code for the probes in a wide variety of programming languages.

We used our proof-of-concept implementation to test the network in Figure 1, with core components listed in Table I. Using the expressiveness of grammar-based fuzzing, we incorporated into our LLDP frame representation some heuristics on problematic strings as gathered by OWASP [15]. In more traditional Web-based injection attacks, such strings would typically have been carried in an HTTP POST request originating from a Web browser. But through grammar-based fuzzing we are able to incorporate them, and more, in an arbitrary protocol data unit and automatically generate a large amount of tests from that grammar. With our tool, SPs can discover and patch similar integration vulnerabilities before they pose any risk.

VI. CONCLUSION

We have performed a security audit of a Web-based network management system. During the audit we found one verified HTML injection vulnerability that arose from the choice of network components where no single component could be held responsible. This created something of a security blind spot, which we exploited to control the network manager. For service providers to combat issues like this, we have proposed a grammar-based black-box fuzzing framework with which we were able to discover the issue.

Currently, our tool is limited to discovering XSS vulnerabilities with LLDP as the carrying component. In our future work we aim to extend our tool to encompass other injection vulnerabilities, most importantly SQL injection. We also intend to use other protocols for carrying the injection strings such as DHCP, SNMP or DNS.

[3]Available at gitlab.com/zluudg/qwilfish.

REFERENCES

[1] "A03 injection-OWASP top 10:2021, OWASP." 2021. Accessed: Mar. 9, 2023. [Online]. Available: https://owasp.org/Top10/A03_2021-Injection/
[2] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A state-aware black-box Web vulnerability scanner," in *Proc. 21st USENIX Secur. Symp.*, Aug. 2012, pp. 523–538. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/doupe
[3] B. Eriksson, G. Pellegrino, and A. Sabelfeld, "Black widow: Blackbox data-driven Web scanning," in *Proc. IEEE Symp. Secur. Privacy (SP)*, 2021, pp. 1125–1142.
[4] V. J. Manès et al., "The art, science, and engineering of fuzzing: A survey," *IEEE Trans. Softw. Eng.*, vol. 47, no. 11, pp. 2312–2331, Nov. 2021.
[5] M. Zalewski. "American fuzzy lop." 2017. Accessed: Nov. 16, 2022. [Online]. Available: https://lcamtuf.coredump.cx/afl/
[6] V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNET: A Greybox Fuzzer for network protocols," in *Proc. IEEE 13th Int. Conf. Softw. Test. Validat. Verificat. (ICST)*, 2020, pp. 460–465.
[7] J. Pereyda. "Fuzzing for humans: Real fuzzing in the real world, DEF CON 24." 2016. Accessed: Jan. 25, 2023. [Online]. Available: https://www.youtube.com/watch?v=N3Z4C2D15JM
[8] X. Feng et al., "Snipuzz: Black-box fuzzing of IoT firmware via message snippet inference," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, New York, NY, USA, 2021, pp. 337–350. [Online]. Available: https://doi.org/10.1145/3460120.3484543
[9] "OWASP top 10:2021, OWASP." 2021. Accessed: Jan. 25, 2023. [Online]. Available: https://owasp.org/www-project-top-ten/
[10] LibreNMS. "Home-LibreNMS docs." 2023. Accessed: Jun. 12, 2023. [Online]. Available: https://docs.librenms.org
[11] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler. "The fuzzing book, CISPA Helmholtz Center for Information Security." 2023. Accessed: Jan. 7, 2023. [Online]. Available: https://www.fuzzingbook.org/
[12] L. Fernandez, G. Karlsson, and D. Hübinette, "A framework for feedback-enabled Blackbox fuzzing using context-free grammars," Netw. Syst. Eng., KTH, Stockholm, Sweden, Rep. TRITA-EECS-RP-2023:2, 2023.
[13] Google. "Protocol buffers." 2023. Accessed: Jan. 12, 2023. [Online]. Available: https://developers.google.com/protocol-buffers/
[14] Google. "DocumentationlgRPC." 2023. Accessed: Jun. 12, 2023. [Online]. Available: https://grpc.io/docs/
[15] OWASP. "Fuzz vectors." 2021. Accessed: Mar. 9, 2023. [Online]. Available: https://owasp.org/www-project-web-security-testing-guide/stable/6-Appendix/C-Fuzz_Vectors
[16] P.-Y. Ritschard and V. Bernat. "lldpd." 2023. Accessed: Jun. 12, 2023. [Online]. Available: https://lldpd.github.io/
[17] "Net-SNMP." 2019. Accessed: Jun. 12, 2023. [Online]. Available: https://www.net-snmp.org/
[18] "LibreNMS v22.9.0–3-g06c361c2b VM image." Feb. 2021. Accessed: Jan. 25, 2023. [Online]. Available: https://github.com/librenms/packer-builds/releases/tag/21.2.0