Poster: Fuzzing IoT Firmware via Multi-stage Message Generation

Bo Yu, Pengfei Wang, Tai Yue, Yong Tang
College of Computer, National University of Defense Technology
Changsha, China
{yubo0615,pfwang,yuetai17,ytang}@nudt.edu.cn

ABSTRACT

In this work, we present IoTHunter, the first grey-box fuzzer for fuzzing stateful protocols in IoT firmware. IoTHunter addresses the state scheduling problem based on a multi-stage message generation mechanism on runtime monitoring of IoT firmware. We evaluate IoTHunter with a set of real-world programs, and the result shows that IoTHunter outperforms black-box fuzzer boofuzz, which has a 2.2x, 2.0x, and 2.5x increase for function coverage, block coverage, and edge coverage, respectively. IoTHunter also found five new vulnerabilities in the firmware of home router Mikrotik, which have been reported to the vendor.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Grey-box fuzzing; stateful protocol fuzzing; IoT firmware fuzzing

ACM Reference Format:
Bo Yu, Pengfei Wang, Tai Yue, Yong Tang. 2019. Poster: Fuzzing IoT Firmware via Multi-stage Message Generation. In *2019 ACM SIGSAC Conference on Computer& Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3319535.3363247

1 INTRODUCTION

Nowadays, billions of IoT (Internet-of-Thing) devices are connected to the Internet, and security flaws of these endpoints can cause risks to privacy, savings, well-being, or even lives. IoT firmware uses lots of stateful network protocol (e.g., *ssl*, *smb*, *ftp*, and *bgp*) to provide powerful functions such as device management, message exchange, and customized functionalities. However, owing to the complex protocol design, time-to-deliver constraints, and insecure coding practices, the vulnerable network protocol and service applications in IoT firmware are prone to leave unprecedented attack surfaces (e.g., the notorious *Mirai* IoT botnet). Thus, discovering and fixing the vulnerabilities in IoT firmware is of vital significance.

Recent research proposes to detect vulnerability in IoT firmware via fuzzing. However, fuzzing IoT firmware has to overcome three challenges. (1) The complex protocol states. Network protocols in IoT firmware are stateful, which have complex message interactions and state transitions. The input space of a stateful protocol is in the form of a message sequence. Thus, to fuzz a stateful protocol under a certain state, we must set the target system to the desired state in advance. Besides, the protocol state will be lost when sending a mal-formated message. Hence, most black-box fuzzing tools (e.g., boofuzz [2] and IoTFuzzer [4]) could not effectively cover the state trajectory and achieve poor automation. (2) Scalability for different protocols. Current attempts of fuzzing IoT firmware are limited to certain protocols, such as *http* [8], *ssl* [6], and *vpn* [5]. (3) Test case validation. The network messages between clients and servers are well constructed, and each received message has to be checked for format validation, such as message length, protocol identification, and checksum. Current mutation strategies (e.g., bit flipping) in fuzzing are validation-blind, and most of the mutated test inputs fail to pass the format checking and are rejected at an early stage. Thus, discovering deep bugs by fuzzing IoT firmware is largely limited by the current random-mutated message input [3][7][8].

In this work, we present IoTHunter, the first (to the best of our knowledge) grey-box fuzzer for stateful protocols in IoT firmware. During the runtime monitoring of IoT firmware, IoTHunter sequentially switches to protocol states according to the given state sequence to perform a feedback-based state exploration and conduct coverage-guided grey-box fuzzing via multi-stage message generation. In summary, we make the following contributions.

- We propose a novel technique called multi-stage message generation to fuzz the stateful network protocol in multiple process stages fully. Especially, our approach can fuzz known state sufficiently and explore unknown states which are not presented in a given state sequence.
- We implement a prototype tool called IoTHunter, which has a high coverage of the stateful protocol input space, high test case validation rate, and support multiple key protocols (e.g., *snmp*, *ftp*, *ssl*, *bgp*, *smb*) in IoT firmware.
- We evaluate IoTHunter with a set of real-world IoT programs, and the result shows that IoTHunter outperforms black-box fuzzer boofuzz. We also discover five new vulnerabilities, which have been reported to the vendor.

2 DESIGN AND IMPLEMENTATION

2.1 Message-state transition model

Considering a stateful protocol P with a state set $S =< s_0, s_1, s_2, ..., s_n >$ and message sequence $M = \{M_{i,j}, 0 <= i, j <= n\}$, the state transition relations can be expressed with a directed graph $G = (S, M)$, where s_0 is the initial state of a service request. In a graph G, each message type $M_{i,j}$ denotes the message that can migrate from state s_i to state s_j. Especially, if $j = 0$, we call $M_{i,0}$ an invalid message (i.e.,

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11.
https://doi.org/10.1145/3319535.3363247

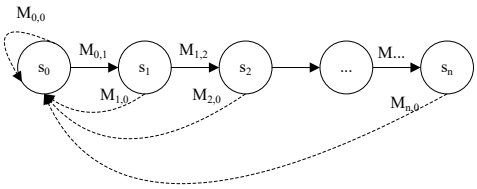

Figure 1: Protocol state model driven by input message.

wrong format), which will cause the system to lose the current state s_i and migrate to the initial state s_0. Figure 1 shows a message-state transition model that expresses the state transition relationship driven by message sequence M. The solid and dotted arrows denote the valid and invalid type of messages, respectively. In the stateful fuzzing process, the test cases are new messages generated by the fuzzer. Whether a message is valid or invalid is determined by the runtime state monitoring of IoT firmware. Hence, in nature, the stateful fuzzing process is to find the boundary between the valid and invalid regions of the input space.

2.2 Multi-stage Message Generation

Input. As Figure 2 shows, the input space consists of two-part: (1) a state sequence S that describes the known states from protocol specification, and a seed set M that include valid message in each state; (2) the message format requirements comprising the basic message constraints, such as message type, length field, concrete version number, checksum field.

Workflow. First, we use a protocol-specific parser to extract metadata by inspecting each message against format requirements. Then, we schedule protocol states according to state sequence S, and then call the message mutation scheduling module to generate new test cases based on message seeds. After that, we check test cases against message format requirements and updates the corresponding field if any violation happens. Finally, we send the message to a running target of IoT firmware to collect the execution trace and message response.

Protocol state scheduling process. By considering the known states in message sequence and unknown states which are not presented, the protocol state scheduling process consists of two parts: (1) **The known state fuzzing**: We start from the initial state s_0, which directly generates new messages based on message seeds. Then we update the message format when any violation against the format requirements occurs. After sending the new message to target IoT firmware, the next state s_1 will be achieved if the new message is valid, and we collect the valid message $M_{0,1}$. Otherwise, the current state will be lost and reset to s_0. After a period of fuzzing, we proceed to the next state if no new path found. For the fuzzing of state s_i ($i > 0$) in each step, we first need to check the current state, if it is not si-1, additional messages $M_i = M_{0,1} + M_{1,2} + ... + M_{i-2,i-1}$ is selected from the valid message set and sent for scheduling current state to s_i-1. (2) **The unknown state exploration**: Since most of the existing stateful protocols contain a type field, we use an integer mutation policy to change the message type value randomly and packs the message type and in the generated message as a new test case. However, if we find a

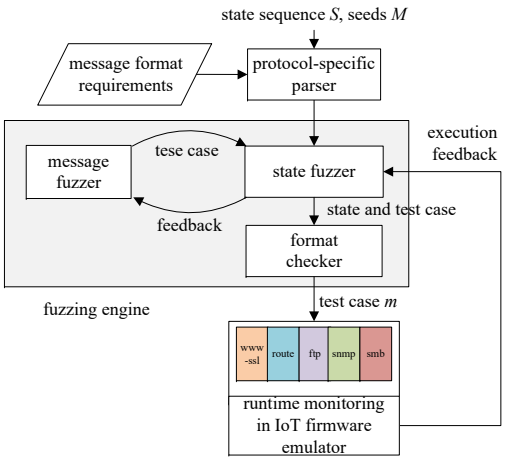

Figure 2: Workflow of multi-stage message generation.

new state that not presented in state sequence S, it will be added to the state sequence. After the whole fuzz process finishes, a valid message set $M = M_{0,1} + M_{1,2} + ... + M_{n-1,n}$ is got, which is the whole input space of a stateful protocol.

Format Checking. During the fuzzing process, each message is checked against protocol format requirements before sending out. Based on the protocol-specific parser, the metadata of each message is extracted and stored. Then, the metadata is checked for preserving message validation. For example, the length of a new message is re-computed and written to the length filed. Meanwhile, the message type must be consistent with the current state.

2.3 Implementation

IoTHunter is implemented on top of AFL [1] and boofuzz [2], and Avatar2 [7]. Two core fuzzing engines are included in IoTHunter, a state fuzzer (implemented on top of boofuzz) that schedules protocol state and a message fuzzer (implemented on top of AFL) that mutates messages smartly. To fuzz both protocol states and messages together, IoTHunter schedules its two core fuzzers in order. Especially, in order to keep the workflow of AFL intact meanwhile allow AFL to fuzz a target program in an IoT firmware, we replace the user-mode with the state fuzzer. Other components of IoTHunter include a firmware emulator based on Avatar2, a python-based parser, and a format checker. IoTHunter initializes an Avatar2 target with a given firmware image, then calls the state fuzzer to schedule the protocol state and notifies message fuzzer to perform a random mutation. Besides, a monitoring module with new control commands is added in emulator Avatar2 to support IoTHunter. The current implementation of IoTHunter supports CPU architectures, including x86, x86-64. Theoretically, it also supports mips and arm.

3 EVALUATION

Configuration. We use eight real-world IoT programs from home router Mikrotik v6.38.4 x86 and NAS Synology v15284 x86-64 to evaluate IoTHunter. The test programs cover key service protocols (e.g., *snmp*, *ftp*, *ssl*, *bgp*, *smb*) that handle network requests and thus are easily-reached targets for remote attacks. Same as the other

Table 1: Vulnerabilities found by IoTHunter.

ID	Protocol	Description
CVE-2018-10070	ftp	A buffer overflow caused by sending a crafted FTP request with many '\0' characters.
CVE-2019-13074	ftp	A denial of service caused by repeated service request to exhaust all available memory.
CVE-2018-7445	smb	A buffer overflow caused by sending a NetBIOS SMB message with request ID=0x14 or 0x81.
CVE-2019-13964	snmp	An assert failure caused by sending an snmp GET message with extra-long pedding.
CVE-2019-13606	snmp	A denial of service caused by sending an snmp GET message with extra-long pedding.

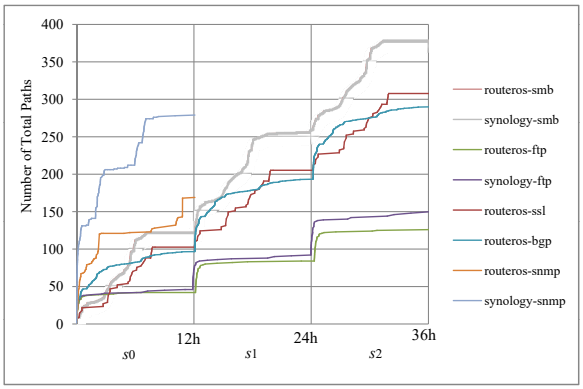

Figure 3: Fuzzing process of the stateful protocols.

researchers [6][5], we only fuzz the states in the handshake process of each protocol. For example, the handshake process of protocol *ssl* and *smb* includes three states s_0, s_1 and s_2. Although *snmp* has only one state, fuzzing *snmp* follows the workflow we proposed. We fuzzed each protocol state for 12 hours (on a single core) and repeated each 12 hours experiment 5 times for each test program to reduce randomness in fuzzing. We ran our experiments on a 64-bit machine with 40 cores (2.8 GHz Intel Xeon E5- 2680 v2), 64GB of RAM, and Ubuntu 16.04 as server OS. The total hours of our experiments are over 30 CPU days.

Findings. During the test, we found five new vulnerabilities (listed in Table 1) from the home router Mikrotik v6.38.4 x86. When we report these vulnerabilities to the vendor, the first three were also discovered by other researchers, even though, the PoC we provided is different from the other reporters (e.g., the request ID 0x81 for CVE-2018-7445).

Stateful fuzzing. Figure 3 illustrates the stateful fuzzing process. The horizontal axis represents the fuzzing time, while the vertical axis represents the number of paths explored. Each line shows the fuzzing process of one protocol. We can see that the total paths increase dramatically at the beginning of each fuzzing state and remain stable at the end of each fuzzing state, which implies that each protocol is fuzzed sufficiently by state scheduling mechanism.

Comparison. We compare IoTHunter with black-box fuzzer boofuzz [2] with three protocols. The comparison is conducted

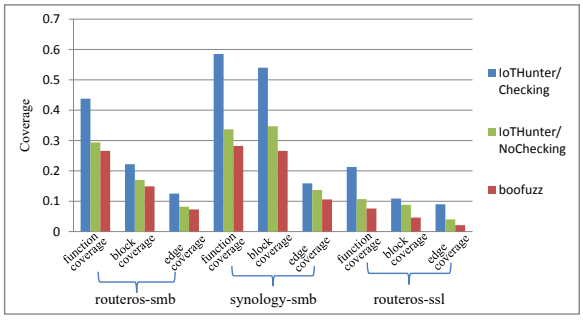

Figure 4: Comparison result with boofuzz.

from three aspects: function coverage, block coverage, and edge coverage. As Figure 4 shows, when fuzzing stateful protocols, IoTHunter outperforms boofuzz, which has a 2.2x, 2.0x, 2.5x increase for function coverage, block coverage, and edge coverage, respectively. We also compare IoTHunter with a version without format checking. The result shows that the format checking mechanism in IoTHunter can averagely increase 1.7x, 1.4x, 1.6x for function coverage, block coverage, and edge coverage, respectively. The results imply that coverage-guided grey-box fuzzing performs better than black-box fuzzing when fuzzing stateful network protocols.

4 CONCLUSION

In this poster, we present the first grey-box fuzzer, IoTHunter, to fuzz stateful protocols in IoT firmware. Based on a state transition model, we propose a novel technique called multi-stage message generation to fuzz the stateful network protocol in multiple process stages fully. The experiment shows that IoTHunter outperforms black-box fuzzer boofuzz in function coverage, block coverage, and edge coverage. We also found five new vulnerabilities in the firmware of Mikrotik, which have been reported to the vendor.

ACKNOWLEDGEMENT

The work was supported by the National Natural Science Foundation of China (61902412, 61902416) and the Natural Science Foundation of Hunan Province (2019JJ50729).

REFERENCES

[1] 2019. American fuzzy lop. [Online]. http://lcamtuf.coredump.cx/afl/.
[2] 2019. Boofuzz. [Online]. https://boofuzz.readthedocs.io/en/latest/.
[3] Andrea Biondo. 2018. Coverage-guided fuzzing of embedded firmware with avatar2. [Online]. https://siagas.math.unipd.it/siagas/getTesi.php?id=2030.
[4] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *NDSS*.
[5] Lesly-Ann Daniel, Erik Poll, and Joeri de Ruiter. 2018. Inferring OpenVPN State Machines Using Protocol State Fuzzing. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 11–19.
[6] Joeri De Ruiter and Erik Poll. 2015. Protocol State Fuzzing of TLS Implementations. In *24th USENIX Security Symposium (USENIX Security 15)*. 193–206.
[7] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar 2: A multi-target orchestration platform. In *Workshop on Binary Analysis Research (colocated with NDSS Symposium)(February 2018), BAR*, Vol. 18.
[8] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *28th USENIX Security Symposium (USENIX Security 19)*. 1099–1114.