Towards Automated Detection of Higher-Order Memory Corruption Vulnerabilities in Embedded Devices

Lei Yu*†, Linyu Li*†, Haoyu Wang*†, Xiaoyu Wang*†, Houhua He*†, and Xiaorui Gong*†
*School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
†Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
{yulei,lilinyu,wanghaoyu,wangxiaoyu,hehouhua,gongxiaorui}@iie.ac.cn

***Abstract*—The rapid growth and limited security protection of the networked embedded devices put the threat of remote code execution related memory corruption attacks front and center among security concerns. Current detection approaches can detect single-step and single-process memory corruption vulnerabilities well by fuzzing tests, and often assume that data stored in the current embedded device or even the embedded device connected to it is safe. However, an adversary might corrupt memory via multi-step exploits if she manages first to abuse the embedded application to store the attack payload and later use this payload in a security-critical operation on memory. These exploits usually lead to persistent code execution attacks and complete control of the device in practice but are rarely covered in state-of-the-art dynamic testing techniques. To address these stealthy yet harmful threats, we identify a large class of such multi-step memory corruption attacks and define them as higher-order memory corruption vulnerabilities (HOMCVs). We can abstract the detailed multi-step exploit models for these vulnerabilities and expose various attacker-controllable data stores (ACDS) that contribute to memory corruption. Aided by the abstract models, a dynamic data flow tracking (DDFA) based solution is developed to detect data stores that would be transferred to memory and then identify HOMCVs. Our proposed method is validated on an experimental embedded system injected with different variants of higher-order memory corruption vulnerabilities and two real-world embedded devices. We demonstrate that successful detection can be accomplished with an automatic system named Higher-Order Fuzzing Framework (HOFF) which realizes the DDFA-based solution.**

Keywords—vulnerability, memory corruption, embedded device

I. Introduction

Embedded devices are the driving force behind modern cyberspace since they enable daily services with which users interact. Such embedded devices often handle large amounts of data, such as service data, configuration, or user uploaded files. All of this data can potentially be abused by an attacker to cause harm. Many different kinds of memory corruption attacks against embedded devices, such as stack and heap buffer overflow attacks, are known and well understood. Such attacks can be prevented by sanitizing user input, and many approaches to address this problem were presented([1]-[4]).

One common assumption underlying many detection and prevention methods is that the data stored in the current embedded device or even the embedded device connected to it is safe. However, an adversary might bypass the defenses via so-called higher-order memory corruption vulnerabilities (HOMCVs) if she manages first to store the payload in the target embedded device or other device connected to it, and later use this payload in a memory corruption operation of the target device. The adversary could store the payload through a service interface (such as file upload or SNMP OID set interface) or a vulnerability exploit (such as SQL injection or unauthorized file upload exploit).

A typical example is that the attacker first uses an SNMP set request to store the payload in an OID and then uses an HTTP get request to trigger a memory corruption attack by reading the OID. Such vulnerabilities are often overlooked, but they can have severe impacts in practice. For example, stack buffer overflow attacks gain persistence if the payload can be stored in a persistent location when the executable directories are not writable. Thus, detecting HOMCVs is crucial to improve the security of embedded devices.

Detecting such vulnerabilities can be done via either static or dynamic analysis. Generally, static code analysis has no access to the external resources used by the application and does not know the data that is stored in these, while applications in embedded devices could process untrusted data from external resources generated by other applications and even other devices in various ways. There are several dynamic approaches to detect memory corruption attacks in embedded devices via fuzzing [1]-[4]. However, these approaches send different requests randomly and have high false-negative rates since detecting HOMCVs needs to send different requests across different processes or even different devices in specific orders.

To the best of our knowledge, we are not aware of any plain implementation detecting HOMCVs using dynamic analysis. The main problem is to find where the tainted data is stored and how it is used. Assuming all data to be tainted and tracing all data flows would lead to a high number of false positives, while a conservative analysis might miss vulnerabilities.

In this paper, we introduce a lightweight and efficient HOMCVs detection method for embedded devices combining dynamic data flow tracking (DDFT) and fuzzing. We implemented our approach in a prototype named Higher-Order Fuzzing Framework (HOFF) for embedded devices. We evaluated our approach by testing an experimental embedded system injected with nine variants of HOMCVs and two real-world embedded devices with two known HOMCV. Overall, we detected all the 11 HOMCVs without false positives. We compared our system with four well-known network protocol fuzzing tools and found that they missed most HOMCVs, indicating that these approaches do not correctly handle such vulnerabilities. In summary, we make the following contributions:

- We broadly define higher-order memory corruption vulnerabilities (HOMCVs) from a systematic perspective and provide abstractions for nine subsets.
- We are the first to propose an automated approach to detect HOMCVs in embedded devices by analyzing attacker-controllable data stores (ACDS) through a

lightweight and efficient method combining dynamic data flow tracking (DDFT) with fuzzing.

- We built a prototype system named Higher-Order Fuzzing Framework (HOFF) of the proposed approach and evaluated it on an experimental embedded system and two real-world embedded devices. As a result, we detected 11 of 11 HOMCVs while the existing fuzz tools missed 9 of 11.

II. Memory Corruption Vulnerabilities and Attack Model

A. Memory Corruption Vulnerabilities in Embedded Devices

Memory corruption vulnerabilities in embedded devices have attracted increasing attention because their remote code execution abilities and limited memory security protections in most embedded devices, such as Ripple20[6] and Urgent/11[7], affected billions of embedded devices.

While most memory corruption vulnerabilities are triggered by reading an attacker's input through one single request, more hidden forms of often triggered by several requests, among them, such as stack overflow caused by SQL injection in the Netgear router[8], stack overflow caused by configuration file injection in the Huawei router(CVE-2016-5366/CVE-2016-5367).

In this paper, we systematically define a broad class of these memory corruption attacks with multi-step requests as higher-order memory corruption vulnerabilities (HOMCVs) in Section III. We call one-step triggered memory corruption vulnerabilities as first-order memory corruption vulnerabilities (FOMCVs).

B. Attack Model

We assume that the attacker can access the target embedded device's various services through the network, which is a common scenario.

Under this assumption, the attacker can exploit HOMCVs by storing the memory corruption payload through a request and triggering the target application to read the payload in another request.

The attacker can exploit HOMCVs to take full control of vulnerable embedded devices. After gaining control, they could extract sensitive data, install ransomware, install persistent spyware on the device, pivot and attack a connected company network or enroll the device in a global botnet to attack other sites, as happened during the Mirai denial-of-service (DDoS) attacks that attacked Dyn in 2016, briefly taking Amazon, PayPal, Netflix, and Twitter offline.

III. Abstraction of Higher-Order Memory Corruption Vulnerabilities

A. HOMCV Modeling

The general form of HOMCV can be described as state transition triggered by multiple requests. It starts with a request reading the payload p from attacker a and ends with memory corruption triggered by p. Intermediate requests will lead to the copy or transfer of the payload p from location l_{i-1} to location l_i. HOMCV's triggering process is shown in the following expression:

$$HOMCV=r_1(read(p,a),write(p,l_1))+r_2(read(p,l_1),write(p,l_2)) +\ldots+r_i(read(p, l_1||\ldots||l_{i-1}),triggered(p)) \quad (1)$$

where $read(v_1,v_2)$ / $write(v_1,v_2)$ stands for reading or writing the data of v_1 from the location v_2.

We define attacker-controllable data stores (ACDS) as locations that could be controlled by the attackers to store memory corruption payload through network requests. We further divide HOMCVs into nine sub-categories, including files, databases, shared memory, pipes, heaps, environment variables, HTTP sessions, network locations, mixed-carrier, -based HOMCVs. We now introduce two typical HOMCVs.

B. Model of Files-based HOMCVs

One practical example of the files-based HOMCVs is the stack overflow attacks caused by configuration file injection in the Huawei router(CVE-2016-5366). When exploiting this type of vulnerability, the attacker first stores the payload p into a file f_1 in the target embedded device by a request. Then she may send requests to trigger payload p copied or cut to another file. Finally, the target application's memory is corrupted through a request when reading any file f_i containing the payload. It is worth noting that the attacker can store the payload in file content or name. According to the triggering process, Equation (1) can be rewritten as:

$$HOMCV_{files}=r_1(read(p,a),write(p,f_1))+r_2(read(p,f_1),write(p,f_2)) +\ldots+r_i(read(p, f_1||\ldots f_{i-1}||file_dir(f_{i-1})||mlocate.db),triggered(p)) \quad (2)$$

where f_i=($file_path$), $file_dir(f)$ stands for the directory of file f, $mlocate.db$ stands for the database of command $locate$.

We can find all the possible request sequences that could lead to files-based HOMCVs according to (2). However, some situations may cause (2) to capture some false-positive request sequences. For example, the payload in the file f_i's content was cut off, or the embedded application did read the file f_i but did not process the payload. But we could fuzz these request sequences to reduce false-positives.

C. Model of Network-Location-Based HOMCVs

Network-Location-Based HOMCVs refers to the subset of HOMCVs where memory corruption is caused by reading network location controlled by attackers. When exploiting this type of vulnerability, the attacker first stores the payload into a network location n through a request (such as setting an SNMP OID), then crashes the target embedded application through a request reading the network location n. According to the triggering process of this type of vulnerability, Equation (1) can be rewritten as:

$$HOMCV_{network_location}=r_1(read(p,a),write(p,n))+ r_2(read(p,n),triggered(p)) \quad (3)$$

where r_1 and r_2 could be sent to two different devices.

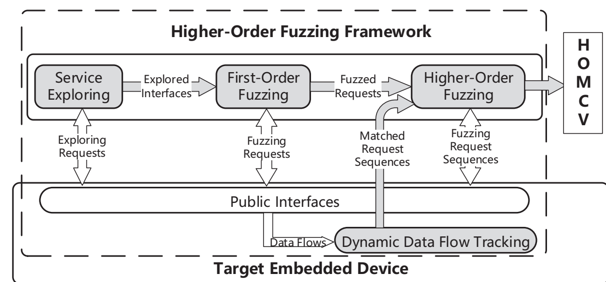

Fig. 1. Framework of HOFF

IV. Higher-Order Fuzzing Framework

To detect HOMCVs in embedded devices, we designed and implemented a lightweight and efficient automated detection system named Higher-Order Fuzzing Framework (HOFF) based on dynamic data flow tracking (DDFT). HOFF mainly includes four modules: *service exploring*, *first-order fuzzing*, *DDFT*, and *higher-order fuzzing*. Fig.1. shows the system block diagram. This section will introduce the design and implementation of each module separately.

A. Service Exploring

Service exploring module explores the open services on an embedded device and provides test objects for *first-order fuzzing*. *Service exploring* consists of two stages, *service identification* and *service interface exploration*.

Service identification identifies open service through *Nmap*. *Service interface exploration* explores open interfaces of the open service through exploration tools based on its protocol, such as web crawler tool, SNMP walk tool.

B. First-Order Fuzzing

First-order fuzzing module fuzzes the test objects generated in *service exploring* using traditional methods to test all the possible reading and writing on ACDS.

C. DDFT

DDFT module tracks and analyzes all the reading and writing on ACDS triggered by *first-order fuzzing* requests. *DDFT* installs on the target embedded devices and can track the target processes with low memory usage.

TABLE I. Observable ACDS Read/Write Operation in Strace

ACDS Type	Observable Keyword in Strace Log	
	Read	***Write***
File	fread(fopen(…), …))	fwrite(fopen(…), …)
Database	read(fd_db[a],…)	write(fd_db…)
Heap	read(fd_in[b], heap…)	write(fd_in, heap…)
Shared Memory	read(fd_in,) shmat(shmid…))	write(fd_in,) shmat(shmid…)
Environment Variable	setenv(…)	execve(…, getenv(…))
HTTP Session	read(fd_s[c],…)	read(fd_s,…)
PIPE	read(pipe[0], …)	write(pipe[1], …)
Network Location	SNMPpp[d]::get, etc.	SNMPpp::send, etc.

a. File descriptor of database. b. File descriptor of user input. c. File descriptor of HTTP session . d. SNMPpp is used to communicate with SNMP.

Reading and writing operations on ACDS that can be observed through *strace*[9] are shown in Table I. *Strace* is a diagnostic, debugging, and instructional userspace utility for Linux. Although many command-line tools on embedded devices have been cut, *strace* exists on many real-world embedded devices.

After tracking all the reading and writing operations on ACDS triggered by fuzzing requests, we start dynamic data flow analysis as follows.

1) *Taint source:* If a request writes data to an ACDS, mark the written ACDS as a taint source point, and mark the request as a writing request.

2) *Taint propagation:* If a request reads the taint source point and generates a writing operation on another ACDS, mark the new written ACDS as a new taint source point and mark the request as a propagating request.

3) *Taint target:* If a request reads the taint source points and does not generate a writing operation on any ACDS, the request is marked as a reading request. Based on the HOMCV model, a matched request sequence is then generated and could be traced back to the source writing request marking the taint source point.

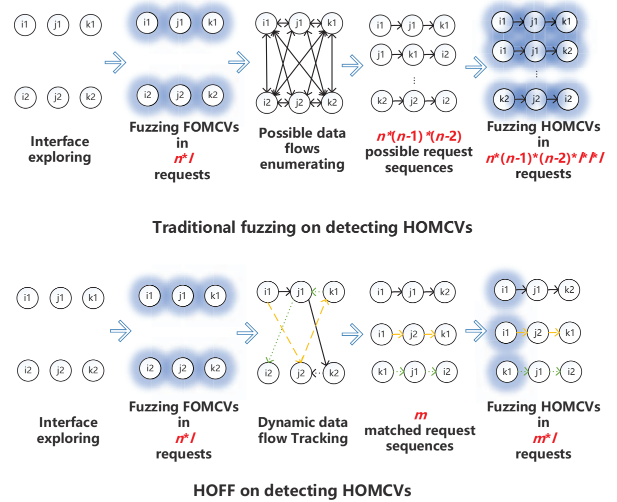

Fig. 2. HOFF vs traditional fuzzing on detecting HOMCVs

D. Higher-Order Fuzzing

After the *DDFT* module provided matched request sequences, the *higher-order fuzzing* module executes the request sequence to fuzz the writing request and replay the propagating/reading request. Under the precise guidance of the request sequence provided by *DDFT*, *high-order fuzzing* could find HOMCVs triggered by multi-step requests.

Fig.2. shows the comparison between HOFF((lower part)) and traditional fuzzing technology(upper part) when detecting HOMCVs. HOFF can achieve an exponential reduction in the number of test requests compared to traditional fuzzing.

As is shown in Fig.2., for the detection of HOMCVs triggered by three steps, the existing fuzzing method's test space is:

$$O(n*(n-1)*(n-2)*l*l*l)$$

where l represents the maximum number of requests for each interface.

While HOFF's test space is:

$$O(m*l)$$

where m represents a constant number of request sequences detected by *DDFT*.

V. Experimental Results

We evaluated HOFF on one experimental embedded system with HOMCVs implanted and two real-world embedded devices with known HOMCVs. We did a comparative test with network protocol fuzzing tools which are often used in embedded devices testing, such as *peach fuzzer*[10], *boofuzz*[11], *Sulley*[12], and *fuzzowski*[13].

A. Target Devices

For evaluation, we implemented an experimental embedded system based on ARM Vexpress OS simulated in QEMU with nine HOMCVs implanted and selected two real-world embedded devices under evaluation.

For the experimental embedded system: We have planted nine developed applications with HOMCVs shown in Table II written in C/C++, all of which have network interfaces.

TABLE II. DEVELOPED VULNERABLE APPLICATIONS

Vulnerable Apps	Property	
	Concurrent Type	***Memory Corruption Type***
File-based	SP&ST	Heap buffer overflow
Database-based	SP&ST	Format string
Heap-based	SP&ST	Heap buffer overflow
Shared-memory-based	MP	Stack buffer overflow
Environment-variable-based	MP	Stack buffer overflow
HTTP-session-based	MT	Stack buffer overflow
PIPE-based	MP	Heap buffer overflow
Network-location-based	MD	Stack buffer overflow
Mixed-based	MP	Stack buffer overflow

SP:Single Process,ST:Single Thread,MP:Multiple Processes,MT:Multiple Threads,MD: Mulitple Devices

For the real-world embedded devices: We selected Netgear router WNDR3700(version 3) and Huawei router WS851(version 1.1.20) with known HOMCVs.

B. Results and Analysis

As shown in Table Ⅲ, our experimental results showed that HOFF detected 100% of all the HOMCVs. The existing fuzzing tools detected 18.2% of them, mainly because they are more suitable for single-process applications. They could detect the heap and session-based HOMCVs but couldn't reproduce the crashes stably because they sent requests in random sequences and accidentally crashed the target.

TABLE III. EXPERIMENTAL RESULTS

Test Cases	Detection Methods				
	HOFF	***Peach Fuzzer***	***boofuzz***	***Sulley***	***fuzzo wski***
File-based	√	×	×	×	×
Database-based	√	×	×	×	×
Heap-based	√	√	√	√	√
Shared-memory-based	√	×	×	×	×
Environment-variable-based	√	×	×	×	×
HTTP-session-based	√	√	√	√	√
PIPE-based	√	×	×	×	×
Network-location-based	√	×	×	×	×
Mixed-based	√	×	×	×	×
Huawei Router	√	×	×	×	×
Netgear Router	√	×	×	×	×

VI. COMPARISONS TO EXISTING SOLUTIONS

FIRM-AFL[1] enhances AFL to perform gray box testing on firmware. IoTFuzzer[3] performs black-box testing on mobile apps corresponding to IoT devices. FIoT[2] uses symbolic execution and fuzzing to discover memory corruption vulnerabilities in the firmware of lightweight IoT devices. However, these methods and tested *peach fuzzer*[10], *boofuzz*[11], *Sulley*[12], and *fuzzowski*[13] were suitable for detecting single-process memory corruption vulnerabilities, while our work focuses on detecting HOMCVs. Karonte[5] detects malicious interactions between firmware modules through dynamic taint tracking. It detects the IPC inside the firmware. But it's unable to detect various non-IPC communication data storage such as databases and network locations. Compared with our approach, it is more time-consuming and requires more manual participation.[14] focuses on detecting faults when memory is corrupted, while our work focuses on how to corrupt memory.

VII. CONCLUSION

In this paper, higher-order memory corruption vulnerabilities are proposed to expose the stealthy yet harmful threats in embedded systems. Using features extracted from the HOMCV models, a dynamic-data-flow-tracking-based method is presented to detect higher-order memory corruption paths, and an automated higher-order fuzzing framework is developed to realize the solution. In our future work, we will apply the developed method to more complex real-world embedded systems using more advanced technology nodes.

VIII. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments. This work was supported by the National Natural Science Foundation of China under Grant No.62032010, the Key Laboratory of Network Assessment Technology of Chinese Academy of Sciences, and Beijing Key Laboratory of Network Security and Protection Technology.

REFERENCES.

[1] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "Firm-AFL: High-throughput grey-box fuzzing of IoT firmware via augmented process emulation," in 28th USENIX Security Symposium (USENIX Security 19). 2019, pp. 1099–1114.

[2] L. Zhu, X. Fu, Y. Yao, Y. Zhang, and H. Wang, "FIoT: Detecting the memory corruption in lightweight IoT device firmware," in 2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), 2019, pp. 248–255.

[3] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "IoTFUZZER: Discovering memory corruptions in IoT through app-based fuzzing." in the Network and Distributed System Security Symposium, 2018.

[4] K. V. English, I. Obaidat, and M. Sridhar, "Exploiting memory corruption vulnerabilities in conman for IoT devices," in 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2019, pp. 247–255.

[5] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Karonte: Detecting insecure multibinary interactions in embedded firmware," in 2020 IEEE Symposium on Security and Privacy (SP), 2020, pp. 1544–1561.

[6] "IDENTIFYING AND PROTECTING DEVICES VULNERABLE TORIPPLE20."[Online].Available:https://www.forescout.com/company/blog/identifying-and-protecting-devices-vulnerable-to-ripple20/

[7] "URGENT/11 affects additional RTOSs - Highlights Risks on Medical Devices." [Online]. Available: https://www.armis.com/urgent11/

[8] "SQL Injection to MIPS Overflows: Rooting SOHO Routers." [Online]. Available: {https://media.blackhat.com/bh-us-12/Briefings/Cutlip/BH US 12 Cutlip SQL Exploitation WP.pdf}

[9] "strace." [Online]. Available: {https://strace.io/}

[10] "Peach Fuzzer." [Online]. Available: {https://www.peach.tech/}

[11] "boofuzz.[Online]. Available: {https://github.com/jtpereyda/boofuzz}

[12] "Sulley." [Online]. Available: {https://github.com/OpenRCE/sulley}

[13] "fuzzowski."[Online].Available:{https://github.com/nccgroup/fuzzo

[14] Muench, Marius, et al. "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices." NDSS. 2018.