International Journal of Digital Crime and Forensics
Volume 13 • Issue 6

Towards Automated Detection of Higher-Order Command Injection Vulnerabilities in IoT Devices:
Fuzzing With Dynamic Data Flow Analysis

Lei Yu, University of the Chinese Academy of Sciences, China
Haoyu Wang, University of the Chinese Academy of Sciences, China
Linyu Li, University of the Chinese Academy of Sciences, China
Houhua He, University of the Chinese Academy of Sciences, China

ABSTRACT

Command injection vulnerabilities are among the most common and dangerous attack vectors in IoT devices. Current detection approaches can detect single-step injection vulnerabilities well by fuzzing tests. However, an attacker could inject malicious commands in an IoT device via a multi-step exploit if he first abuses an interface to store the injection payload and later uses it in a command interpreter through another interface. The authors identify a large class of such multi-step injection attacks to address these stealthy and harmful threats and define them as higher-order command injection vulnerabilities (HOCIVs). They develop an automatic system named Request Linking (ReLink) to detect data stores that would be transferred to command interpreters and then identify HOCIVs. ReLink is validated on an experimental embedded system injected with 150 HOCIVs. According to the experimental results, ReLink is significantly better than existing command injection detection tools in terms of detection rate, test space, and time.

KEYWORDS

Command Injection Vulnerability, Dynamic Data Flow Tracking, IoT Devices

INTRODUCTION

IoT devices provide daily services interacting with users and often handle large amounts of user-provided data. All of this data can potentially be abused by an attacker to cause harm. Many different kinds of command injection attacks against IoT devices, such as OS command injection attacks and SQL injection attacks, are well understood. Such attacks can be prevented by sanitizing user input, and many approaches to address this problem were presented in the last few years.

One common assumption underlying many detection and prevention methods is that the data stored in the current IoT device or even devices connected to it is safe. However, an attacker might bypass the defenses via so-called higher-order injection vulnerabilities (HOCIVs) if he first stores the payload through a request and later uses this payload in a command interpreter through another request.

Consider a web service interface that saves the user's setting parameter (e.g., syslog server's address) to a config file. Another interface read the config file's setting parameter to a command

DOI: 10.4018/IJDCF.286755

shell. In this example, testing the former or latter interface in isolation could result in false negatives because they cannot trigger the vulnerability alone. Existing testing methods (Stasinopoulos, 2019) (Tool, 2020) need to enumerate and test all interface combinations to find this vulnerability, and it's not realistic in IoT devices with many interfaces.

Such vulnerabilities are often overlooked, but they could have severe impacts in practice. For example, an OS command injection attack would gain persistence if the injection payload could be stored in a configuration file while executable directories are not writable. Thus, detecting HOCIVs is crucial to improve the security of IoT devices.

Detecting such vulnerabilities can be done via either static or dynamic approaches. For static approaches, Dahse et.al. (Dahse, 2014) proposed RIPS to detect second-order vulnerabilities and related multi-step exploits in web applications by analyzing reads and writes to the web server's data stores. Redini et.al. (Redini, 2020) proposed Karonte to detect insecure multi-binary interactions in embedded firmware using a set of inter-process communication (IPC) data stores, it could find HOCIVs across multiple binary files. However, it is challenging to construct vulnerability PoCs (Proof of Concept) based on static analysis approaches, and a lot of manual analysis by source code or binary analysis expert is required. It is not realistic for the security testing of ever-changing IoT devices.

There are several dynamic approaches to detect command injection attacks in IoT devices via fuzzing (Stasinopoulos, 2019) (Tool, 2020), which do not require expert experience when testing. Such approaches are focused on fuzzing a single request and try to inject command injection payloads to all possible inputs. The analysis tools determine if the injected commands are executed. However, these approaches have high false-negative rates when detecting HOCIVs since triggering HOCIVs need to send different requests in specific orders.

In this paper, we present ReLink, a novel detection method for HOCIVs in IoT devices combining fuzzing and dynamic data flow tracking. First, we explore all network interfaces of the target IoT device by sending all possible requests on them and collect all data stores that are written to or read from by these requests. Second, we can get the data dependency graph between these requests based on the collected data stores and identify all possible request chains that may trigger HOCIVs based on the graph. Third, all the possible request chains are sent in order. A detailed data flow analysis is then performed to determine if the request chain could trigger commands' execution with user's input. Finally, we fuzz these selected request sequences to detect HOCIVs. With ReLink, we can detect HOCIVs with linear test space and low memory consumption, while most of the other methods require exponential test space and high memory consumption.

We implemented ReLink in a prototype for Linux-based IoT devices. We evaluated our approach by testing an experimental embedded system injected with 150 variants of HOCIVs compared our system with four well-known command injection fuzzing tools. Overall, ReLink detected 87.3% HOCIVs while reference tools missed more than 90% HOCIVs, indicating that these approaches do not correctly handle such vulnerabilities. ReLink cut more than 50% test space and time than the reference tools. In summary, we make the following contributions:

We introduce novel combinations of dynamic analysis techniques to perform cross-request taint analysis. To do so, we design a novel approach to precisely apply and propagate taint information across multiple requests.

We propose ReLink, a novel dynamic analysis approach to identify HOCIVs triggered by request sequences. ReLink radically reduces the number of test cases, making real-world IoT devices analysis practical.

We implement and evaluate our prototype of ReLink on an experimental IoT device with 150 HOCIVs, showing that our tool can successfully propagate taint information across multiple requests, resulting in the discovery of 87.3% preset HOCIVs.

ATTACK MODEL AND COMMAND INJECTION VULNERABILITIES

Attack Model

IoT devices process data through the network. This data can come directly from the user through the various interface (e.g., through a web interface). In this paper, we assume that the attacker could access the target IoT device's various services by sending network requests either through a local network or the Internet.

This attack scenario is common since IoT devices are designed to provide users (including attackers) with various network interfaces. It is also easy to extend. For example, combining with a CSRF attack, the attacker could perform the attack even without interacting with the IoT devices directly.

Requests to IoT Devices

IoT devices usually provide a large number of interfaces. Users can send requests to these interfaces to achieve interaction with the device. Different requests sent to an IoT device are often handled by different threads or processes. These requests usually share data through a finite set of inter-thread or inter-process communication channels, especially data stores that can be persisted for stable communications. For example, a request first writes a configuration file, then another request reads the file and processes the configuration.

This paper distinguishes different types of requests as follows:

Inputting and writing request: A request that accepts the user's input and writes it to a data store.

Reading and writing request: A request that reads a data store and writes the data store's content to another data store.

Reading and processing request: A request that reads a data store and processes the data store's content.

However, we could also identify the inputting and processing request that could lead to common command injection vulnerabilities, but we ignore them since this paper focuses on HOCIVs.

Data stores are important links between different requests. They are mainly divided into the following types: file, database, socket, pipe, network location, shared memory, environment variable, HTTP Session, hostname, directory and symbolic link.

Command Injection Vulnerabilities in IoT Devices

Command injection is an attack in which the goal is to execute arbitrary commands on the host operating system via a vulnerable application. Command injection attacks are possible when an application passes unsafe user-supplied data (forms, cookies, HTTP headers, etc.) to a command interpreter (generally refers to the OS command shell, including SQL, NoSQL, SMTP, and LDAP, etc. in this paper).

It is easier to exploit command injection vulnerabilities because it just uses command scripts to complete attacks. Many IoT malware exploits this type of vulnerability, such as Mirai (CVE-2020-5902, 2020). Therefore, such vulnerabilities bring significant security risks to IoT devices.

HIGHER-ORDER COMMAND INJECTION VULNERABILITIES

Motivation Example

CVE-2019-7297 in D-Link DIR 823G with firmware version no larger than 1.02B03 is a typical higher-order command vulnerability. The vulnerability triggering process is divided into two steps:

1) Sending payload through *SetNetworkTomographySettings* request.
2) Trigger payload through *GetNetworkTomographyResult* request.

First, the payload was stored in the data store through the first request, and then the payload in the data store was read and processed through another request, which finally triggered the vulnerability.

Definition

Unlike traditional command injection vulnerabilities triggered directly within a one-step request for a single process (we call them first-order command injection vulnerabilities, FOCIVs), higher-order command injection vulnerabilities (HOCIVs) are triggered through multi-step requests across different threads, processes, or devices. To exploit HOCIVs, attackers need to save the payload through the inputting and writing request and trigger the payload through the reading and processing request.

Among HOCIVs, 2-step request triggered is called second-order command injection vulnerabilities (SOCIVs), 3-step request triggered is called third-order command injection vulnerabilities (TOCIVs), and so on.

Modeling

The general form of HOCIV can be described as state transitions triggered by multiple requests:

1) Inputting and writing request r_1 reads the payload p from attacker u and writes the payload into data store l_1.
2) Reading and writing requests (r_2-r_{i-1}) would lead copying or transferring of the payload p from one data store to another.
3) Reading and processing request r_i triggers the command injection with reading payload p from one of all the possible locations storing p.

HOCIV's triggering process is shown in the following expression:

$$\text{HOCIV}=\{\text{r1(read(u,p),write(l1, p)), r2(read(l1,p),write(l2, p)),…,ri(read(l1||…||li-1, p),triggered(p))}\} \quad (1)$$

where read(v_1,v_2) / write(v_1,v_2) stands for reading or writing the data of v_2 from or to the location v_1, i=2 for the SOCIVs, i=3 for the TOCIVS, and so on.

Challenges

HOCIVs are inevitable and difficult to detect in modern IoT devices. There are challenges for both static and dynamic analysis approaches.

For static analysis, IoT devices' compositions are becoming more complex and interconnected, including many third-party components composed of many code libraries and million lines of code. So, it isn't easy to analyze the entire IoT system efficiently, and lots of security expert experience is also required.

For dynamic analysis, black-box fuzzing approaches are challenging to detect HOCIVs because HOCIVs require precise request sequences to be triggered, while these approaches focus on fuzzing one single request. Grey-box fuzzing approaches need to inject codes into all the tested processes, which is an unacceptable cost for the IoT devices with limited memory and computing resources, and also some online systems cannot modify their executable files for protection while simulating these systems is also challenging due to the reliance on a lot of hardware input.

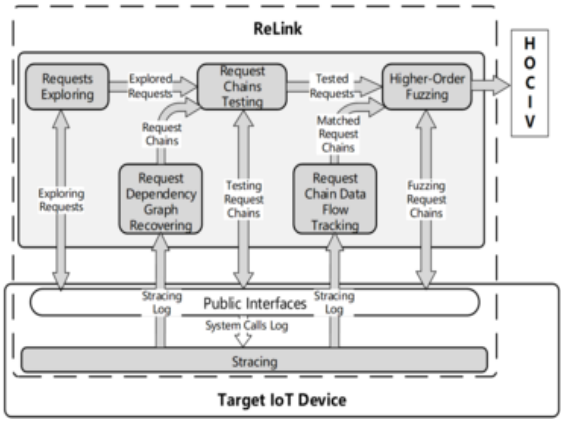
Approach Overview

To detect HOCIVs, we designed and implemented a lightweight and efficient automated detection system named Request Linking (ReLink) combining fuzzing with dynamic data flow tracking. ReLink is for Linux-based IoT devices. Fig.1. shows the system block diagram. ReLink mainly includes six

modules: stracing, requests exploring, request dependency graph recovery, request chains testing, request chain data flow tracking, and higher-order fuzzing.

As is shown in Fig.1., the work process of ReLink is as follow:

Figure 1. Framework of Relink



1) Stracing module traces all the system calls of all the network service processes in target IoT device and provides log file to other modules of ReLink.
2) Requests exploring module explores and records all the legal requests on the public interfaces of target IoT device.
3) Request dependency graph recovering module recovers dependency graph for the explored requests based on system call logs traced by strace and generates request chains based on the graph.
4) Request chains testing module tests the generated request chains based on the explored requests recorded by requests exploring.
5) Request chain data flow tracking module tracks data flows of these tested request chains based on strace logs and matches request chains that might trigger HOCIVs.
6) Higher-order fuzzing module fuzzes the inputting and writing request of matched request chains, replays the other requests in chains, and detects HOCIVs if the injection payload is excuted.

DESIGN AND IMPLEMENTATION

This section will detail the design and implementation of ReLink.

Stracing

Stracing module installs on the target IoT devices and monitors the selected system calls (Page, 2020) of all the network service processes using the command-line tool strace (Strace, 2020).

Strace exists on most real-world Linux-based IoT devices though many Linux tools have been cut according to our findings. Strace is a lightweight, diagnostic, debugging, and instructional userspace utility for Linux. It is used to monitor and tamper with interactions between processes and the Linux kernel, including system calls, signal deliveries, and process state changes.

The memory and CPU resources occupied by strace are meager (usually less than 1% on CPU and memory usage). Therefore strace is suitable for online testing of resource-constrained IoT devices.

Stracing focuses on system calls with reading/writing on data stores (Table I) and writing to command interpreters (Table II) triggered by network requests.

Stracing records these logs into a file and provides the file to request dependency graph recovering and request chain data flow tracking of ReLink.

Table 1. Observable readings/writings on data stores in strace log

Data Store Type	Observable System Calls	
	Read	Write
File	read/readv/pread/preadv/preadv2/readahead(file fd,…) mmap(…, file fd,…)	write/writev/pwrite/pwritev/pwritev2(file fd,…)
Database	read/readv/pread/preadv/preadv2(database server socket fd,…)	write/writev/pwrite/pwritev/ pwritev2(database server socket fd,…)
Socket	read/readv/pread/preadv/preadv2(socket fd,…)	write/writev/pwrite/pwritev/ pwritev2(socket fd,…)
Pipe	read/readv/pread/preadv/preadv2(pipe fd,…)	write/writev/pwrite/pwritev/ pwritev2(pipe fd,…)
Network Location	read/readv/pread/preadv/preadv2(network server socket fd,…)	write/writev/pwrite/pwritev/ pwritev2(network server socket fd,…)
Shared Memory	shmat(shmid,…)	shmat(shmid,…)
Environment Variable	execve(“/bin/echo”,…)	write(etc/profile fd,…)
HTTP Session	read(session file fd/session server socket fd,…)	write(session file fd/session server socket fd,…)
Hostname	gethostname()/getdomainname()/hostname/uname	sethostname()/setdomainname()
Directory	open(pathname,…)/openat(dirfd,…)/ creat(pathname,…)	mkdir(pathname,…)/mkdirat(dirfd,…)
Symbolic Link	readlink(pathname,…)/readlinkat(dirfd,…)	symlink(pathname,…)/ symlinkat(dirfd,…)

Table 2. Observable writings to command interpreter in strace log

Interpreter Type	Observable System Calls
OS Shell	execve(pathname,argv,…)/execveat(…, pathname,argv,…)
SQL	write(SQL server_socket fd,…)
NoSQL	write(NoSQL server socket fd,…)
IMAP/SMTP	write(IMAP/SMTP sever socket fd,…)
LDAP	write(LDAP server socket fd,…)

Algorithm 1 Request Dependency Graph Recovery Algorithm

function RDG_recovery(R)
E¬{}
for ra Î R:
for rb Î {R-ra}:
if ra.postcondition.type=reading
if rb.precondition.type=writing
if ra.postcondition.value=rb.precondtion.value
E.add(ra,rb)
end if
end if
end for
end for
return {R,E}
end function

Requests Exploring

Requests exploring module explores all the network requests that could be sent to the target IoT device such as requests of Web, SNMP, UPNP requests. It consists of two stages, service identification and request exploration.

Service identification identifies open service through the network scanner tool Nmap (nmap, 2021).

Request exploration explores the available requests that could be sent to the open service through exploration tools based on service type. Such as we can use Web crawler tool to explore request that could be sent to the web service, SNMP walking tool to explore all the availbale SNMP requests.

Request Dependency Graph Recovering

Given the strace log triggered by the exploring requests, request dependency graph recovering module identifies different types of requests as mentioned in section II.B, builds a Request Dependency Graph (RDG), and generates request chains based on RDG.

Request type identifying: Request type is identified by system calls of the request handler thread in target IoT device according to the strace log.

For an inputting and writing request, the thread reads user's input and writes full or part of the input to a data store from the strace log.

For a reading and writing request, the thread reads a data store and write full or part of the data store's content to another data store.

For a reading and processing request, the thread reads a data store and processing full or part of the data store's content in a command interpreter (as shown in Table II).

It is recorded with the following keys for every identified request:

{*id, time, source address, target address, content, type, precondition, postcondition*}.

Key *precondition* and *postcondition* have sub keys as following:

{ *type*(*reading / writing / inputting / processing*), *value*(*data store location description / inputting content / processing command*)}

RDG building: RDG is a directed graph that models communications among exploring requests. The RDG is iteratively recovered by leveraging data store linking based on the identified requests, which can reason about the different inter-request communication paradigms. RDG recovery algorithm is as follows:

Request chain generating: A request chain is a path in the RDG, starts from an inputting and writing request, ends with a reading and processing request, and connected by sharing data stores. It represents a request sequence that may trigger an HOCIV.

However, these generated request chains may have false positives, mainly because many requests read or write the same data store that does not contain user input. Therefore we need to test these request chains further.

Request Chains Testing and Data Flow Tracking

Given request chains generated from RDG, request chains testing module sends these request chains based on the recorded requests sent and tacking their data flow using strace again.

Given the strace log triggered by request chains testing, request chain data flow tracking module tracks input data flows of these tested request chains to detect whether there is an input data flow. Without input data flow, a request chain could not trigger HOCIVs since the user's input is not injected into the executed command.

In detail, this module checks whether the user's input content of the inputting and writing request appears in the processing command of the reading and processing request. If yes, filter out this request chain for further testing.

This module outputs the matched request chains that have input data flows.

Higher-Order Fuzzing

Given the matched request chains, higher-order fuzzing module fuzzes the inputting and writing request and replays the other requests in the chain to detect whether the injection payload is executed.

This module is based on popular command injection detection tools, such as Commix (Github, Commixproject/Commix, 2020) and Sqlmap (Tool, 2020). We use these tools to fuzz the inputting and writing request. But we modified their logic of judging vulnerabilities. The logic of determining whether there is an HOCIV is executed after all requests in the request chain are sent.

The original command injection vulnerability detection logic is as follows:

1) Send a request with an injection payload.
2) Detect whether the payload is executed.

The modified detection logic is as follows:

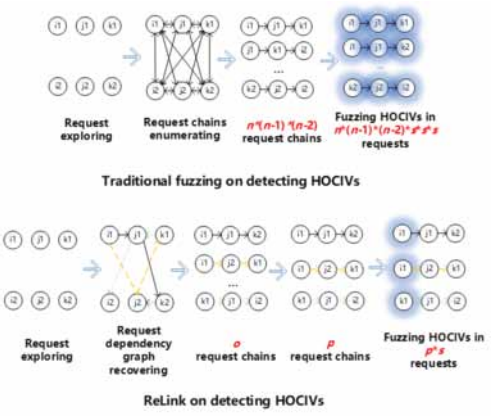
1) Send an inputting and writing request with injection payloads.
2) Send reading and writing requests in order(if there are).
3) Send a reading and processing request.
4) Detect whether the payload is executed.

Comparison with Existing Fuzzing Approaches

A good fuzz-based detection method aims to detect all vulnerabilities with a small number of test cases. Existing fuzz methods need to enumerate all possible combinations of requests to detect all HOCIVs.

Fig.2. shows the comparison between ReLink(lower part) and existing fuzzing approaches (upper part) when detecting HOCIVs triggered by three steps.

Figure 2. ReLink vs traditional fuzzing on detecting HOCIVs



The number of existing fuzzing approaches' test cases is: $n*(n-1)*(n-2)*s*s*s$, where n represents the number of explored requests, s represents the maximum number of fuzz requests for each explored request.

The number of ReLink's test cases is: $p*s$, where p represents a constant number of request chains matched by ReLink.

From this example, we can see that ReLink has greatly reduced the test space through fewer test cases($>n*(n-1)*(n-2)*s$ times smaller).

It can be inferred that the greater the number of explored requests, the longer the HOCIV request chain, and the greater the test space ratio that ReLink can reduce.

EXPERIMENTAL RESULTS

We did a comparative test with top-ranked command injection fuzzing tools, such as Commix, Sqlmap, shelling(works with Burpsuite (Portswigger, 2020)) (Github, Ewilded/Shelling, 2020), and fuzzdb(works with ZAP (Owasp, 2020)) (Github, Fuzzdb-Project/Fuzzdb, 2020).

- Commix: a top-ranked open-source OS command injection detection tool.
- Sqlmap: a top-ranked open-source SQL injection detection tool.
- Shelling: a popular open-source OS command injection detection tool, usual-ly works with Burpsuite.
- Fuzzdb: a popular open-source injection (OS command injection, NoSQL injection, LDAP injection, SMTP injection, etc.) detection tool, usually works with ZAP.

Although there are some research related to high-order vulnerabilities (Dahse, 2014) (Pellegrino G. &., 2014) (Olivo, 2015) (Alhuzali A. E., 2016) (Alhuzali A. G., 2018) (Zhang, 2019) (De Meo, 2020) (Redini, 2020) (Xiao, 2020), we could not find any open source tool that can be used to detect HOCIVs in IoT devices without major modification, so we will compare them with them in the related work section.

Target Device

We implemented an experimental embedded system based on ARM Vexpress OS simulated in QEMU with 150 HOCIVs implanted for evaluation. We have implemented 150 applications with HOCIVs written in C/C++/Bash/Lua, all of which have network interfaces.

An example of vulnerable applications is shown in Listing 1-3.

Table 3. HOCIVs implanted in vulnerable applications

Command Injection Type	Number of HOCIVs		
	SOCIV	TOCIV	FOCIV
OS Shell	10	10	10
SQL	10	10	10
NoSQL	10	10	10
IMAP/SMTP	10	10	10
LDAP	10	10	10

```
1 http_args = ngx.req.get_uri_args();
2 config = http_args[“config”];
3 file = io.open(“test.config”, “w”);
4 assert(file);
5 file:write(config);
6 file:close();
```

Listing 1: Inputting and writing request handler

```
1 reading_file = io.open(“test.config”, “r”);
2 assert(reading_file);
3 local config = reading_file:read(“*a”);
4 reading_file:close();
5 writing_file = io.open(“ip.config”, “w”);
6 assert(writing_file);
7 writing_file:write(config);
8 writing_file:close();
9 file:close();
```

Listing 2: Reading and writing request handler

```
1 file = io.open(“ip.config”, “r”);
2 assert(file);
3 local config = file:read(“*a”);
4 file:close();
5 local r = io.popen(‘ifconfig ‘ .. config);
```

Listing 3: Inputting and processing request handler

Table III shows a summary of developed vulnerable applications. Each type of HOCIVs' triggering process covers 10 data stores, and the corresponding applications cover three programming languages (C/Bash/Lua). Application types include Web, SNMP, analog private protocols, etc. Application's concurrent methods include multi-process and multi-thread.

The applications with SOCIVs provides two external interfaces, the applications with TOCIVs provides three external interfaces, and the applications with FOCIVs provides four external interfaces. A user can send a request containing a parameter to these interfaces.

Table 4. Experimental result

Type	Tools	Detection Results		
		Detection Rate	Requests Sent	Time Spent
SOCIV	ReLink	45/50	5200	<10min
	Commix	4/50	10000	>20min
	SQLmap	5/50	10000	>20min
	Shelling	3/50	10000	>20min
	Fuzzdb	2/50	10000	>20min
TOCIV	ReLink	44/50	5300	<10min
	Commix	1/50	15000	>30min
	SQLmap	2/50	15000	>30min
	Shelling	0/50	15000	>30min
	Fuzzdb	0/50	15000	>30min
FOCIV	ReLink	42/50	5400	<10min
	Commix	0/50	20000	>40min
	SQLmap	0/50	20000	>40min
	Shelling	0/50	20000	>40min
	Fuzzdb	0/50	20000	>40min

Testing with ReLink and Reference Tools

We first installed the stracing script in the experimental embedded system for ReLink. Stracing was activated when ReLink started testing.

To facilitate comparison, we set the number of payloads of each tool for each type of injection attack to the same since each tool comes with a different number of payloads.

We directly offer the explored requests data as input for each tool since 3/4 of the reference tools don't support service exploring.

Results and Analysis

For detection rate: As shown in Table IV, 87.3% of all the vulnerable embedded applications with HOCIVs had been detected by ReLink.

After detailed analysis, the false negatives of ReLink are HOCIVs using the hostname as the data store. The main reason is that the system restricts the value of hostname and cannot contain most special characters. At present, it is impossible to construct a valid payload through hostname, although this is still a potential injection vector in some situations.

However, the reference tools could not find more than 90% of the HOCIVs, mainly because they are most suitable for single-process applications. Anyway, these reference tools did detect some HOCIVs mainly because they happened to send requests in the right orders. But they can't reproduce the injection attack because they cannot know which requests triggered the attack.

For test cases and time spent: ReLink sent 48% less test requests than other reference tools when detecting SOCIVs with two explored requests, 64.7% less when detecting TOCIVs with three explored requests, 73% less when detecting FOCIVs with for explored requests. The more explored requests of the test target, the more obvious the advantage of ReLink to reduce the test space. The reduction in the number of test requests directly reduces the time spent on testing, as shown in Table IV.

Comparisons to Related Work

Research on multi-step triggering vulnerabilities using is relatively few. Existing work can be divided into static analysis, dynamic analysis, and dynamic-static mixed analysis.

For static analysis: Karonte (Redini, 2020) detects malicious interactions between firmware modules through dynamic taint tracking using IPC channel identification. It can successfully track and constrain multi-binary interactions including security bugs, and could find HOCIVs across multiple binary files. SVHunter (Xiao, 2020) could effectively identify data dependency creation and chaining attacks on SDN controllers. It combines data flow backtracking, an event reasoning language used to formally specify the preconditions and postconditions of data dependency chaining events, and automated causality reasoning. ChainSAW (Alhuzali A. E., 2016) tackled the problem of automated exploit generation for web applications. It is based on precise models of application workflows, database schemas, and native functions to achieve exploit generation. It could analyze open source Web applications and generate first- and second-order injection exploits. RIPS (Dahse, 2014)could detect second-order vulnerabilities and related multi-step exploits in web applications using an automated static code analysis approach. However, it relies on modeling and analyzing PHP and other source code, which does not apply to IoT devices composed of multiple components. Our work is not limited to any programming language that can run on Linux. Torpedo (Olivo, 2015) is a static analysis approach for detecting second-order DoS vulnerabilities in web applications. It can successfully detect second-order DoS vulnerabilities in widely used web applications written in PHP.

However, these static approaches rely on modelling and analyzing the source code or binary code, which does not apply to multi-interaction systems composed of complex components. They also produce many false positives, and the problem would be more serious when applied to large-scale multi-interaction systems involved in this paper. In contrast, our work is not limited to any specific programming language and has few false positives due to our adopted dynamic analysis method.

For dynamic analysis: Navex (Alhuzali, 2018) is the follow-up work of chainsaw, it combines dynamic analysis that is guided by static analysis techniques in order to automatically identify vulnerabilities and build working exploits. It can scale the process of automatic vulnerability analysis and exploit generation to large applications and to multiple classes of vulnerabilities. SrFuzzer (Zhang, 2019) is an automated fuzzing framework for testing physical SOHO devices. It continuously and effectively generates test cases by leveraging two input semantic models, i.e., KEY-VALUE data model and CONF-READ communication model, and automatically recovers testing environment with power management. It can trigger multi-type vulnerabilities. Deemon (Pellegrino G. J., 2017) is an automated security testing framework to discover CSRF vulnerabilities. It is based on based on a new modeling paradigm which captures multiple aspects of web applications, including execution traces, data flows, and architecture tiers in a unified, comprehensive property graph. WAFEx (De Meo, 2020) is a formal and automated approach that allows one to (i) reason about vulnerabilities of web applications and (ii) combine multiple vulnerabilities for the identification of complex, multi-stage attacks. Giancarlo Pellegrino and Davide Balzarotti (Pellegrino, 2014) presents a black-box testing technique to detect logic vulnerabilities in web applications. It is based on the automatic identification of a number of behavioral patterns starting from few network traces in which users interact with a certain application's functionality.

We also use the dynamic analysis method, the above methods are more closely related to our work. Navex is based on source code while it is hard to extend to IoT devices usually without source code. SrFuzzer is based on black-box testing and specific vulnerability patterns and can not find vulnerabilities more than three steps. Deemon is also based on specific vulnerability triggering patterns and is hard to extend to the injection vulnerabilities addressed in our work. WAFEx is heavily on modelling existing vulnerabilities. While our work is based on data flow tracking inside the system, we can find vulnerabilities triggered in multiple steps. For taint-style vulnerability, our work is more general and efficient.

CONCLUSION

In this paper, higher-order command injection vulnerabilities (HOCIVs) are proposed to expose the stealthy yet harmful threats in IoT devices. Using features extracted from the HOCIV models, a dynamic-data-flow-tracking-based fuzzing method is presented to detect HOCIVs, and an automated detection system named ReLink is developed to realize the solution. We will apply the developed approach to more complex real-world IoT devices using static analysis methods in our future work.

REFERENCES

AlhuzaliEshete, B., Gjomemo, R., & Venkatakrishnan, V. N. A. (2016). Chainsaw: Chained automated workflow-based exploit generation. *2016 ACM SIGSAC Conference on Computer and Communications Security*, 641-652.

AlhuzaliGjomemo, R., Eshete, B., & Venkatakrishnan, V. N. A. (2018). NAVEX: Precise and scalable exploit generation for dynamic web applications. USENIX Security, 377-392.

CVE-2020-5902. (2020). *CVE*. Retrieved from https://blog.trendmicro.com/trendlabs-security-intelligence/mirai-botnet-exploit-weaponized-to-attack-iot-devices-via-cve-2020-5902/

Dahse, J. a. (2014). Static detection of second-order vulnerabilities in web applications. *23rd {USENIX} Security Symposium*.

De MeoViganò, L. F. (2020). A formal and automated approach to exploiting multi-stage attacks of web applications. *Journal of Computer Security*.

Github. (2020a). *Commixproject/Commix*. Retrieved from https://github.com/commixproject/commix

Github. (2020b). *Ewilded/Shelling*. Retrieved from https://github.com/ewilded/shelling

Github. (2020c). *Fuzzdb-Project/Fuzzdb*. Retrieved from https://github.com/fuzzdb-project/fuzzdb

nmap. (2021). Retrieved from https://nmap.org/

Obaidat & Sridhari. (2019). Exploiting command injection vulnerabilities in conman for IoT devices. *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 247–255.

OlivoDillig, I., & Lin, C.O. (2015). Detecting and exploiting second order denial-of-service vulnerabilities in web applications. *22nd ACM SIGSAC Conference on Computer and Communications Security*, 616-628.

Owasp. (2020). *OWASP ZAP Zed Attack Proxy*. Retrieved from Owasp.Org: https://owasp.org/www-project-zap/

Page, S.-L. (2020). *Man7.Org*. Retrieved from https://man7.org/linux/man-pages/man2/syscalls.2.html

Pellegrino & Balzarotti. (2014). Toward Black-Box Detection of Logic Flaws in Web Applications. NDSS.

PellegrinoJ, M., Koch, S., Backes, M., & Rossow, C.G., (2017). Deemon: Detecting CSRF with dynamic analysis and property graphs. *2017 ACM SIGSAC Conference on Computer and Communications Security*, 1757-1771.

Portswigger. (2020). *Burp Suite - Application Security Testing Software*. Retrieved from https://portswigger.net/burp

Redini. (2020). Karonte: Detecting insecure multi-binary interactions in embedded firmware. *2020 IEEE Symposium on Security and Privacy (SP)*.

shmget. (2021). Retrieved from https://man7.org/linux/man-pages/man2/ shmget.2.html

Stasinopoulos, A. C. (2019). Commix: Automating evaluation and exploitation of command injection vulnerabilities in Web applications. *International Journal of Information Security*, 49–72.

Strace. (2020). *Strace.Io*. Retrieved from https://strace.io/

Tool, S. A. (2020). *Sqlmap.Org*. Retrieved from https://sqlmap.org/

XiaoZhang, J., Huang, J., Gu, G., Wu, D., & Liu, P.F., (2020). Unexpected data dependency creation and chaining: A new attack to SDN. *2020 IEEE Symposium on Security and Privacy (SP)*, 1512-1526.

ZhangHuo, W., Jian, K., Shi, J., Lu, H., Liu, L., ... Liu, B.Y. (2019). SrFuzzer: An automatic fuzzing framework for physical soho router devices to discover multi-type vulnerabilities. ACSAC, 544-556.