Identifying Privilege Separation Vulnerabilities in IoT Firmware with Symbolic Execution

Yao Yao[1,2], Wei Zhou[2], Yan Jia[1,2], Lipeng Zhu[1,2], Peng Liu[3], and Yuqing Zhang[1,2(✉)]

[1] School of Cyber Engineering, Xidian University, Shaanxi, China
[2] National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences, Beijing, China
zhangyq@nipc.org.cn
[3] College of Information Sciences and Technology, Pennsylvania State University, State College, PA, USA

Abstract. With the rapid proliferation of IoT devices, we have witnessed increasing security breaches targeting IoT devices. To address this, considerable attention has been drawn to the vulnerability discovery of IoT firmware. However, in contrast to the traditional firmware bugs/vulnerabilities (e.g. memory corruption), the privilege separation model in IoT firmware has not yet been systematically investigated. In this paper, we conducted an in-depth security analysis of the privilege separation model of IoT firmware and identified a previously unknown vulnerability called *privilege separation* vulnerability. By combining loading information extraction, library function recognition and symbolic execution, we developed `Gerbil`, a firmware-analysis-specific extension of the `Angr` framework for analyzing binaries to effectively identify privilege separation vulnerabilities in IoT firmware. So far, we have evaluated `Gerbil` on 106 real-world IoT firmware images (100 of which are bare-metal and RTOS-based device firmware. Gerbil have successfully detected privilege separation vulnerabilities in 69 of them. We have also verified and exploited the privilege separation vulnerabilities in several popular smart devices including Xiaomi smart gateway, Changdi smart oven and TP-Link smart WiFi plug. Our research demonstrates that an attacker can leverage the privilege separation vulnerability to launch a border spectrum of attacks such as malicious firmware replacement and denial of service.

Keywords: Internet of Things · Firmware analysis · Privilege separation

1 Introduction

According to latest report [6], the IoT device has eclipsed the mobile phone as the most common connected device by 2018, which means we have been living

K. Sako et al. (Eds.): ESORICS 2019, LNCS 11735, pp. 638–657, 2019.
https://doi.org/10.1007/978-3-030-29959-0_31

in a world surrounding by IoT devices. Through interacting with IoT cloud, mobile app, and other entities, IoT devices allow users to monitor and control their living spaces from anywhere at any time. When the user is at home, he can directly send a command to the devices to control it through his mobile app. If he is not in the same LAN with the IoT devices, he still can monitor and control the devices via the IoT cloud. The cloud will forward the command to the devices. Meanwhile, we observe that some operations are performed only when the IoT device is interacting with the IoT cloud, while some other operations are performed only when the device is physically touched (e.g., pushing a button) by a human user. Hence, whether an operation is legitimate (i.e., legal) depends on whom the IoT device is interacting with. One goal of the attacker could be maliciously "confusing" the firmware running on the device in such a way that illegal operations get performed. For example, if a user wants to rebind a smart cleaning robot to another account, the user has to physically press a button on the robot to reset it into the initial state. However, if the rebind operation could accidentally be carried out through commands sent by mobile app or cloud, this would give an opportunity for attackers to bind the device with the attacker's account without physical access.

To understand why an operation triggered by physically pushing a button could be accidentally carried out through commands sent by a mobile app or cloud, we have conducted in-depth root cause analysis and found that the main root cause is as follows. (a) We found that when an operation triggered by physically pushing a button is being performed on an IoT device, one set of functions in the firmware binaries will be executed. We denote this set as set A. (b) We found that when a command sent by a mobile app, cloud or other entities is being executed on the IoT device, other sets of functions will be executed. According to sender entity of the command, we denote the set as set B, C and so on. (c) We found that when the intersection of set A and set B or C is not empty, the attacker could be provided with the above-mentioned attacking opportunity. Since the IoT devices are interacting with various entities such as mobile apps, IoT cloud, gateway, etc., if any two sets are overlapped, it may cause potential risks. The root cause we found is essentially a *privilege separation* vulnerability.

Although researchers have made great efforts in IoT security, we found they still focus on the classic security issues in IoT research such as privacy leak [9,24], authentication bypass [14,18] and memory corruption flaws [2]. To our knowledge, few studies have been systematically conducted on privilege separation vulnerabilities involved in IoT firmware. Furthermore, state-of-the-art dynamic and static firmware analysis approaches [7,20,23,25] have limited ability to analyze the lightweight IoT firmware (i.e., RTOS-based or bare-metal firmware) in large-scale, let alone identify logic privilege separation vulnerabilities due to following challenges. To begin with, converting a lightweight firmware image into an object that can be statically or dynamically analyzed is an open problem [18]. It is not clearly known how to identify the necessary loading information, e.g., load base address architecture and segmentation information, due to the unknown executable and linkable format of the lightweight firmware, which puts a

barrier to take advantage of current binary code analysis tools (e.g., IDA Pro [15] and Angr [19]). Also, existing solutions for dynamical analysis of IoT firmware are far from mature. They are usually designed only for a Linux-based operating system [23], or must be tightly coupled with real hardware environment [25]. However, a large number of real-world IoT devices run RTOSs or bare-metal systems. How to test a variety of lightweight firmware images without real devices remains challenging. Furthermore, even if symbolically executing only a path of IoT firmware, it might also get stuck with path explosion caused by the infinite loops or complex calculation functions such as AES encryption.

To systematically detect the privilege separation vulnerabilities in a variety of IoT firmware, we first collected and analyzed the popular IoT binary formats, and implemented a tool to automatically extract the loading information from IoT firmware. Since symbolically executing entire binary IoT firmware without full-system emulation is not feasible, we developed an assistant tool to slice the part of IoT firmware where most likely exist privilege separation vulnerabilities and slice this portion of code for the symbolic execution. Next, we designed and implemented a path exploration scheme on the top of symbolic execution. It can skip complex library functions via library function recognition to mitigate path explosion and is also able to restore indirect call to explore deeper paths. Finally, we combined the above approaches together as a novel dynamic analysis framework called Gerbil for detecting privilege separation vulnerabilities in large-scale IoT firmware. According to Gerbil output, we successfully identify privilege separation vulnerabilities in 60 real-world IoT firmware. Through further verification, we found the most of them can be exploited.

In summary, our contributions are as follows:

1. We performed the first in-depth analysis of the privilege separation vulnerability associated with IoT firmware to fill gaps in previous research.
2. We developed an extension of the Angr framework for IoT firmware analysis including loading information extraction, library function recognition and indirect control flow recovery.
3. We designed and implemented a path exploration scheme on the top of symbolic execution to explore more paths, mitigate the path explosion and output more meaning path constrains at same time.
4. We successfully discovered privilege separation vulnerabilities in 69 out of 106 real-world IoT firmware images and evaluated the hazards of the privilege separation vulnerability with several real smart devices.

We are releasing Gerbil as an open-source tool in Github repo[1] in the hope that it will be used for further IoT firmware analysis research.

2 Background

In this section, we first introduce the general privilege separation model involved in real-world IoT firmware. Then, we demonstrate the potential

[1] https://github.com/daumbrella/Gerbil.

privilege separation vulnerability behind this model. Note that to clarify the remainder of the presentation, we highlight the key terminologies in bold.

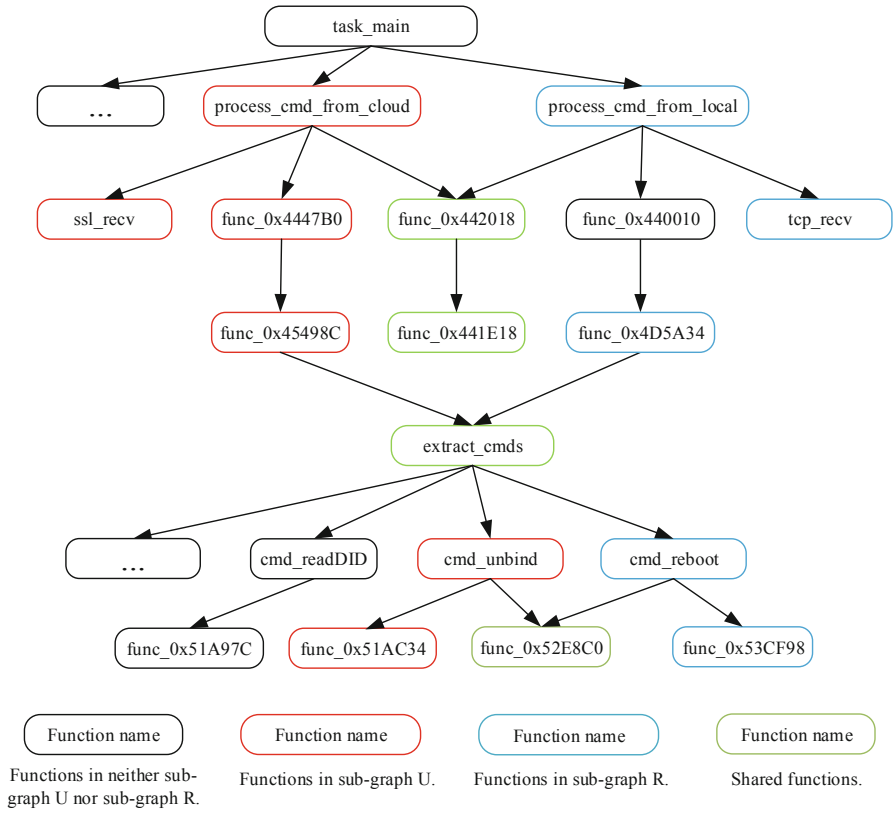


Fig. 1. A part of call graph of an IoT firmware image. (Color figure online)

2.1 Privilege Separation Model Involved in IoT Firmware

IoT devices are designed to interact with human beings via mobile app, cloud or physical access (e.g., pushing a button), to perform a variety of operations. Thus, the main logic of IoT firmware is to perform specific operations corresponding to different commands sent by its interactive entities (e.g., the mobile app, the IoT cloud, and the gateway). To be specific, after receiving messages from various interactive entities, the device deciphers and parses the messages and performs the specific tasks corresponding to the commands extracted from the messages.

Figure 1 shows the call graph of the major functions of *task_main* (generated by disassembling the firmware) which is responsible for processing all receiving network data. Some library functions have been renamed according to their original semantic. Section 3.2 details how we recognize library functions. Each node

denotes a function and each edge represents the calling relationship between two functions. To finally finish the tasks corresponding to a specific command in the interactive message, a sequence of functions will be invoked. Thus, the collection of functions in the execution path from receiving a message to executing a specific command can be considered as an individual call **sub-graph** of the whole call graph in IoT firmware. For instance, as shown in Fig. 1, to perform a remote "unbind" command received from *ssl_recv* function, the sub-graph U, which is indicated in red will be invoked, while the sub-graph R, which is used to handle a local "reboot" command from *tcp_recv* function is indicated in blue. The functions shared by multiple sub-graph U and R are in green. We refer to a function shared by two or more call sub-graphs as **shared function**.

There are two kinds of key function in a call sub-graph. The **caller function** represents the highest node (i.e, start point) of a call sub-graph. For example, function *process_cmd_from_cloud* function and *process_cmd_from_local* in Fig. 1 are the caller functions of sub-graph U and R respectively. We refer to the caller function used to process commands from local interactive entities such as a mobile app and a gateway as a local caller function. The caller function used to process commands from a remote interactive entity (i.e., the IoT cloud) is a remote caller function. The caller function used to process commands corresponding to physical access is a physical caller function. The **command function** represents the nearest node to the last shared functions in a call sub-graph, which is always the first function used to perform a specific command (e.g., *cmd_reboot* and the *cmd_unbind* in sub-graph U and R in Fig. 1). Similar to caller function, generally the three most common kinds of command functions in IoT firmware are local, remote and physical.

In addition, commands sent by different interactive entities usually serve different purposes. For example, remote commands sent by the cloud are usually responsible for device management services such as unbinding the device with an owner and updating the firmware, while the device control commands (e.g., turn on/off the device) are usually sent by a mobile app locally. To this end, the developer should implement a strict **privilege separation model** to divides a firmware into parts and grant each part with different privileges to finish specific tasks through letting the IoT device perform certain operations.

2.2 Privilege Separation Vulnerability

Ideally, if an IoT firmware image strictly implements the privilege separation model, its call graph should have the following property: the two sub-graphs of any two different caller functions should not have any common nodes unless the common nodes have identical set of descendant (callee) nodes in the two sub-graphs. For instance, the shared function *func_0x442018* only calls shared function *func_0x441E18* function in sub-graph U or R. However, due to time-to-market pressure and the limited storage space of IoT devices, we found that developers usually implement some **over-privileged shared functions** which can be reached from different caller functions but can also call different command functions in real-world IoT firmware. Due to the over-privileged shared function,

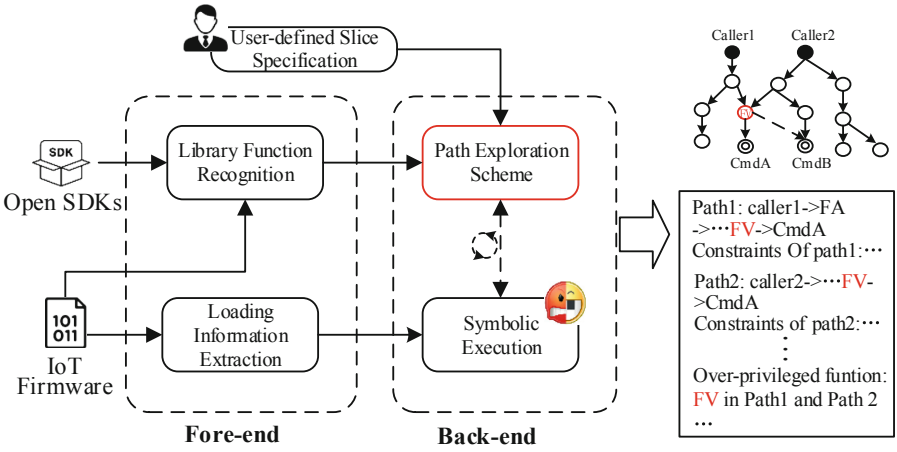


Fig. 2. Gerbil overview.

there will be an execution path from the caller function of one sub-graph to the command function which belongs to another sub-graph with different caller function. We call such an unexpected execution path as a privilege separation vulnerability.

As an example, the *extract_cmds* function in Fig. 1 can be reached from caller function *process_cmd_from_cloud* in sub-graph U, but it can also be reached from *process_cmd_from_local* in sub-graph R. Thus, *extract_cmds* function is an over-privileged shared function. Consequently, an unexpected execution path from the *process_cmd_from_local* function to the *func_unbind* function can be triggered. In this case, a local attacker has a chance to send a remote "unbind" command to the device to unbind user's device unconsciously. Other severe consequences can happen when IoT firmware opens up other unexpected execution paths. We discuss further attack effects through exploiting privilege separation vulnerability with real-world IoT firmware in Sect. 4.4.

3 Gerbil Framework Design and Implementation

In this section, we detail the design and implementation of core components of Gerbil as shown in Fig. 2. Gerbil is a firmware-analysis-specific extension of Angr [19], which is a python framework for analyzing binaries. Angr combines both static and dynamic symbolic analysis; accordingly, the fore-end modules of Gerbil are used to extract loading information and restore library function semantic of IoT firmware. In the back-end, we propose a novel path exploration scheme on top of the symbolic execution engine of Angr to mitigate the path explosion and explore more promising paths for IoT firmware. Furthermore, to effectively identify privilege separation vulnerabilities, the user-defined slice specification could help the analyst to slice the firmware which parts is most likely to contain privilege separation vulnerabilities for symbolic execution.

3.1 Loading Information Extraction

Before analyzing firmware, the analysis tools have to know the basic loading information, including architecture, base address, entry point, segmentation information, etc. For Unix-based (e.g., Linux) firmware they usually adopt the common standard executable and linkable format (ELF). Thus, their loading information can be easily extracted from the ELF header of the firmware. However, many lightweight firmware images (i.e., bare-metal and RTOS-based firmware) do not have a common fixed binary format and it is generally unknown for state-of-the-art binary analysis tools to properly initialize the loading environment of these firmware.

```
0       1       2       3       0       1       2       3
|-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+|
| Magic = 'MRVL'                | SDK version                   |
|-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+|
| Creation time                 | Number of segments            |
|-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+|
| ELF version                   | Segment_i type=0x2            |
|-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+|
| Segment_i Offset              | Segment_i  Size               |
|-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+|
| Segment_i Load_address        | Segment_i CRC32-checksum      |
|-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+|
|                      Segment Data                           |
|-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+|
```

Listing 1. Binary Format of Marvell MW300/302 MCU (Byte Width)

We found that the lightweight firmware running on the same series of microcontrollers (MCU) adopts a similar binary format, which can be easily found in the corresponding public MCU datasheet. In addition, identifying the MCU model of IoT firmware is also effortless, because it is common practice that developers hard-code the corresponding MCU model in the firmware. For instance, we find "MRVL" and "MW300" string in XiaoMi plug firmware, which indicates it runs on Marvell MW300 MCU. As an example, we show the binary format of the firmware which runs on Marvell MW300/302 MCU in Listing 1. As indicated by each field of the binary format, we can easily extract the corresponding loading information such as load address and segmentation information.

Therefore, we maintained an up-to-date binary format database of popular IoT MCUs[2]. Then we implemented a Python script to automatically search the strings referring to the MCU model in the IoT firmware. If its MCU model matches one MCU record in the database, the script extracts the loading information according to the corresponding binary format. Otherwise, we will try to find the binary format of this unknown MCU model and add it to our database.

In addition, some functions use absolutely-addressed memory accesses to call the function pointers stored sequentially in device memory. In most cases, such

[2] https://www.postscapes.com/iot-chips-modules/.

memory pointers are hard-coded in the data segment of firmware and loaded to the memory during booting. Thus, after identifying the data segment of firmware, we also copy it to the memory map used beforehand by the symbolic execution.

3.2 Library Function Recognition

Typically, to protect the intellectual property of the company, IoT manufacturers have stripped the symbols and most of debug strings. However, to reduce the time to market and to be compatible with the central IoT cloud interfaces, manufacturers usually implement the same system and communication libraries and even common peripheral functions (e.g., FreeRTOS, lwIP and WiFi interfaces) in all their firmware. Since these library functions usually take charge of the core functionality of IoT devices, restoring the original function context of libraries can greatly minimize the manual work involved in the firmware analysis, particularly, for security analysis. For example, if we can follow the data and control flow of specific encryption functions and we are able to locate the cryptographic keys or the derived key for data encryption and decryption.

To address this challenge, we implemented a function matching algorithm used by FLIRT [17] to recognize the library functions of IoT firmware. To be specific, we collected widely used IoT libraries from official GitHub repos of popular MCU manufacturers and IoT platform providers. Then we compiled them to conduct a library function comparison with the tested firmware. If a matching was found, we directly got its semantic and are able to restore it during static and dynamic analysis.

3.3 Path Exploration Scheme

The original symbolic execution engine of Angr has some drawbacks for exploring paths of IoT firmware, including overlooking indirect call and path explosion. In the following, we detail how to solve them in practice. The overview of our path exploration scheme is shown in Fig. 3.

Adding High-Level Constraints. The original path constraints generated by Angr are directly based on the byte value of registers and memory, which is obscure for analyst and inconvenient for further manual analysis. Thus, we add library function context to the symbolic execution path constraints to provide more useful and meaning information for analyst. Specifically, if current jump address is a library function address (line 4 in Fig. 3) we add library function name and parameters corresponding to the current register values of *SimState* (which is used to synchronize execution context during symbolic execution including register value, memory data, symbol variable constraints, etc.) as high-level constraints to current path constraints.

Skipping Selected Library Function. Library functions are usually irrelevant to application-specific logic and most logic vulnerabilities such as our identified privilege separation vulnerability are associated with application scenarios. Thus, to mitigate the well-known path explosion problem, Gerbil is able

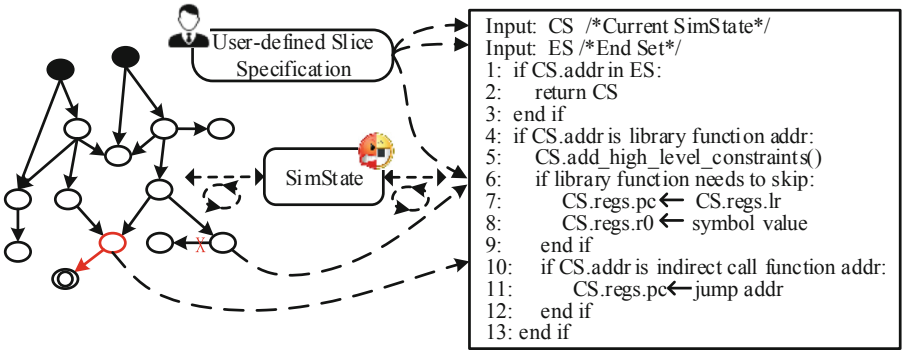


Fig. 3. Path exploration scheme (Color figure online).

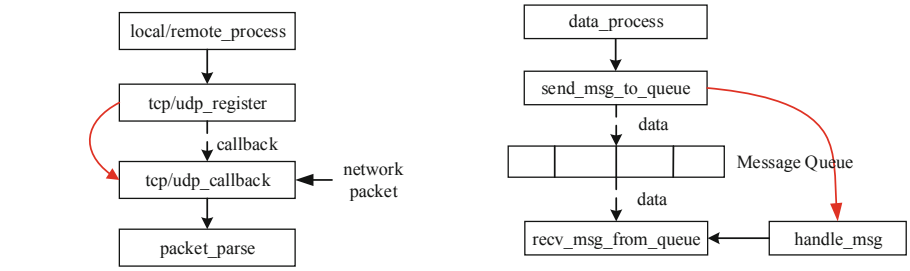
to exclude the complicated library functions selected by the analyst from symbolic execution and assign a new symbol value as the return value for skipped functions. Lines 6–9 in Fig. 3 clearly demonstrate this process.

Indirect Call Restoration. Through manual analysis, we found there are two kinds of typical indirect call (i.e., callback and message queue) that cannot be identified by Angr, but have been frequently used in IoT firmware.

A callback, also known as a "call-after" function, is any executable code that is passed as an argument to other functions that are expected to call execute the argument under a certain conditions. For instance, as shown in Fig. 4a, the *local/remote_process* function passes a callback function pointer *tcp/udp_calback* to *tcp/udp_register* function to parse a network packet. When the *tcp/udp_register* function receives a network packet, it will automatically call the callback function. Thus, we consider the function which registers a callback function to have a call relationship with this callback function.

Message queues are frequently used to send data between functions, which also implies a indirect call relationship. For example, *handle_msg* function waits for the data from message queue through *recv_msg_from_queue* function and *data_process* function writes processed data to this queue as shown in Fig. 4b. Therefore, the functions which send and receive the data from the same message queue have an indirect call relationship.

To add the above indirect call paths to the original execution path, we first record the address of all functions which have an indirect call relationship with other functions. Then during symbolic execution, we check whether the current jump address of the Angr *SimState* matches these addresses as shown in line 10 in Fig. 3. If a match is found, we replace the jump address with the indirect function address in line 11. For instance, when a symbolic execution engine complete the right red circle function in Fig. 3 which has an indirect call relationship with the left one, it will not stop exploring this path but continue to execute the left red function.



(a) Indirect Call via a Callback Function (b) Indirect Call via a Message Queue

Fig. 4. Two typical modes of indirect call in IoT firmware

3.4 User-Defined Slice Specification

Since Gerbil is not a full system emulation, we have to let analysts set the start and end points of symbolic execution. In this work, we carry out symbolic execution on the parts of the firmware may occur privilege separation vulnerabilities. As we mentioned in Sect. 2.2, the privilege separation vulnerability is caused by over-privileged shared functions involved in the paths from caller functions to command functions. Thus, we need to set caller functions as start points and command functions as end points. However, it is difficult to recognize the call and command functions through manual analysis. We provide an assistant tool, *User-defined Slice Specification*, based on the call graph to help analysts locate the caller and command functions. To provide a more complete call graph beforehand, we first improve the control flow graph generation tool CFGFast[3] used by Angr. To be specific, we restore the indirect control flows as we did in path exploration scheme implementation. In addition, we also found the function prologues used by CFGFast to identify the start point of functions were incomplete, so we also added the missing function prologues for it.

Command Function. Although identifying all interactive commands by manual analysis is impossible, it is simple to find several specific commands. For example, we can easily find some strings such as "unbind" and "reset" which refer to the names of specific commands in the firmware, then further identify which functions use these strings. These functions are the command functions in most cases. Next, we search the nearest shared parent node of any two command functions, which is most likely to deal with other commands in the call graph. Then we can use these functions to find more command functions. For instance, as shown in Fig. 1, the *cmd_reboot* and the *cmd_unbind* functions use "unbind" and "reset" strings respectively. Their nearest common parent node is *extact_cmds* and it is also used to call other command functions like *cmd_readDID*. Therefore, we only ask the analyst to input at least two command functions and our tool is able to list most command functions in the firmware.

[3] https://docs.angr.io/built-in-analyses/cfg.

Caller Function. In comparison to caller functions, it is much easier to find functions which are used for receiving the network data at first, because IoT firmware usually uses common library functions such as *ssl_recv* and *tcp_recv* to receive network data. Therefore, instead of inputting all the caller function address, we only ask analyst to input the name of these library functions. After determining receiving network data functions and commands functions, we can find all the paths between them. Then we can list all the highest nodes in every path, which are the caller functions as mentioned in Sect. 2.1.

Note that in case we miss or choose the wrong caller or command functions, we also provide an interface for analysts to override the output of caller and command function sets.

3.5 Result Generation

After determining the caller and command function collection, we first input the caller functions as the start point collection and command functions as the end point collection to the symbolic execution engine. Then the symbolic execution engine explores all possible paths based on the path exploration scheme and outputs possible routes from the caller functions to command functions and corresponding path constraints. Finally, we mark all the functions (except for command functions) which can be reached from more than one caller function as over-privileged shared functions according to our definition in Sect. 2.2. For example, where the right side of Fig. 3 shows the output of Gerbil, the function FV can be passed from different caller functions in path 1 and path 2, which is an over-privileged function.

4 Evaluation

We first evaluate the performance of components of Gerbil and how their capabilities of them benefit IoT firmware analysis. Then we show the result of using Gerbil to identify privilege separation vulnerabilities with real-world IoT firmware. Finally, we elaborate three cases to show how to verify and exploit privilege separation vulnerabilities based on the Gerbil output results (i.e., identified over-privileged shared functions).

Gerbil comprises over 1,000 lines Python in total. More specifically, the loading information extraction module has 93 lines, the library function recognition module (excluding FLIRT library functions) has 288 lines, the path exploration scheme implement has 460 lines (excluding Angr SDK functions), the assistant tool used to identify caller and command function collection for used-define policy has 110 lines and the result generation module has 70 lines. We run Gerbil on a machine running GNU/Linux Ubuntu 16.04 LTS with a dual-core 3.6 GHz CPU and 16 GB memory.

4.1 Lightweight Firmware Collection

To protect intellectual property (IP), most IoT device manufacturers do not make their firmware public especially for the devices running RTOS or bare-metal systems. Thus, to test how Gerbil deals with diverse lightweight IoT firmware used in real-world, we first leveraged the *phantom* devices introduced by research [22] to download and collect a total of 173 firmware images used by different kinds of IoT devices (e.g., gateways, cameras, air conditioners, etc.) from cloud service of five popular IoT device vendors including Alibaba, JD, XiaoMi, TP-Link, and iRobot. In this paper, we focus on the ARM-based and MIPS microcontrollers which are most widely used in IoT device, thus the firmware run on other architecture like Xtensa cannot support by most analysis tools such as Angr are out of our evaluation. Note that the design of Gerbil is applicable to other architectures as well. We also leave the encrypted firmware out of our scope. Finally, we evaluated Gerbil on the rest 106 IoT firmware including 100 lightweight firmware images and 6 linux-based firmware images.

4.2 Performance Analysis of Gerbil

Loading Information Extraction. Except for six Linux-based firmware which can be automatically loaded by Angr, we successfully identified the different kinds of binary format of all tested lightweight firmware as shown in Table 1. Then, we extracted all the loading information used by symbolic execution with 100% accuracy. We have shared our collection of all binary formats of lightweight firmware at GitHub repo[4].

Table 1. Accuracy of the loading information extraction

MCU model	MW300	RTL8711B	RTL8159A	RTL8159A	HF-MC101	STM32F4
# Firmware	43	26	15	11	3	2
Accuracy rate	100%	100%	100%	100%	100%	100%

#: the number of

Library Function Recognition. The IoT library database we collected mostly from two sources. One is *MCU-related SDKs* on the official GitHub repos of popular MCU manufacturers (e.g., Marvell and STMicroelectronics) which are usually implemented in the firmware running on corresponding MCUs, including RTOS and common cryptographic functions and peripheral interfaces. The another is *platform-related* SDKs implemented by device vendors to support devices communicating with popular IoT platform such as Joylink, Alink, MIJIA, and AWS IoT. Our database contains over 20,000 library functions including 2,893 functions of platform-related SDKs and 17,538 functions in MCU-related SDKs.

[4] https://github.com/daumbrella/LoadLightweightFirmware.

To measure the performance of our library function recognition, we calculated the ratio of recognized library functions to total functions as shown in Table 2. The lower rate is mainly due to our insufficient database rather than technical reasons. For example, we did not gather the MCU-related SDKs running on RTL8711B, RTL8159A and STM32F4XX MCUs. In addition, the platform-related SDKs used by tested firmware running on STM32F4XX are previous versions of those we collected. The technical limitations of our library function recognition are discussed in Sect. 5.1

Table 2. The recognition ratio of library function recognition

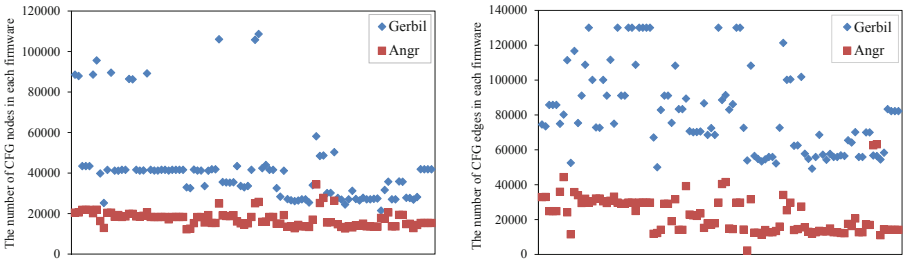
MCU model	Platform	# Firmware	$ Function number	Recognition ratio
MW300/302	MiJia	36	2084	21.81%
	Joylink	1	1748	23.05%
	Alink	2	1707	26.36%
	AWS IoT	4	2246	26.97%
HF-MC3000	Alink	3	4957	30.57%
	Joylink	8	4191	23.59%
HF-MC101	Joylink	3	2648	78.97%
RTL8711B	Joylink	8	4169	2.72%
	Alink	18	4156	2.74%
RTL8159A	Alink	14	4330	2.76%
	MiJia	1	3358	3.25%
STM32F4XX	Alink	2	2247	1.29%

#: the number of $: the average of

Control Flow Graph Restoration. To identify caller and command functions in IoT firmware, we first improve the CFG generation tool used by Angr as we described in Sect. 3.4. Figure 5a and b show a visualized comparison between the number of call graph nodes and edges generated by Gerbil and original Angr with all tested firmware. We can clearly see that Gerbil's restoration of control graphs of most tested firmware was significantly improved compared to original Angr.

4.3 Identifying the Privilege Separation Vulnerability

Gerbil run the 106 test firmware within ten minutes in average (including only symbolic execution time). Since firmware which runs on devices fabricated by the same vendor usually adopts the same privilege separation model, the results of Gerbil's output are categorized according to device vendor as shown in Table 3. The results show that Gerbil identified 69 firmware images have one or more over-privileged shared functions (i.e, privilege separation vulnerabilities). After manual verification, we found that most privilege separation vulnerabilities can



(a) Comparison of the Number of CFG Nodes Generated by Angr and Gerbil

(b) Comparison of the Number of CFG Edges Generated by Angr and Gerbil

Fig. 5. Comparison of CFG restoration between Angr and Gerbil

be exploited. In general, there are two or three caller functions of one firmware corresponding to the three different interactive entities (local mobile app, remote IoT cloud and physical user access). However, the Alibaba devices only support remote commands, which means there is only one caller function in their firmware. Thus, their devices are immune to privilege separation vulnerabilities.

Table 3. Detection results of tested firmware

Detection results	XiaoMi	Alibaba	JD	TP-Link	iRobot
# firmware	37	39	20	6	4
# vulnerable firmware	37	0	14	4	4
# caller functions in each firmware	3	1	2	2 or 3	2
$ command functions in each firmware	36.5	31.3	26.2	21.3	13

#: the number of $: the average of

4.4 Impact Analysis of Privilege Separation Vulnerabilities Exploitation

In this subsection, we use three vulnerable firmware, including the TP-Link smart WiFi plug, Xiaomi Smart Gateway and JD smart oven with model Changdi CRWF321ML, to demonstrate how to exploit our identified privilege separation vulnerabilities and several attack effects.

TP-Link Smart WiFi Plug. We found one over-privileged shared function can be reached by remote and local caller functions in the firmware of the TP-Link smart WiFi plug and 52 command functions can be invoked by this over-privileged shared function. Thus, we can send remote device control commands locally. Next, we use an example of how to achieve an illegal device occupation attack by taking advantage of this over-privileged function.

According to the user manual, we know only one legitimate user is allowed to bind a TP-Link smart home device at a time. To this end, the TP-Link cloud

assigns a unique device ID (i.e, *deviceId*) to one device and binds it with one user account. If other users request to bind the same device again, the cloud will refuse this request unless the device has already been unbound by the original user. In addition, we found the command function *set_device_id* is normally used by the IoT cloud to assign *deviceId* to the device. However, leveraging our identified over-privileged function, this command can also be performed locally. Therefore, a local attacker can send a *set_device_id* command to change the *deviceIdA* of a unsold device to the *deviceIdB* which has been bound with his account. When consumers buy this device, they cannot bind it to their accounts, because the deviceIdB is already bound to the attacker's account. Worse still, the device cannot be unbound, because the victim does not have the attacker's account. Thus, the attacker can illegally occupy this device forever.

Xiaomi Smart Gateway. Similar to the TP-Link smart plug, the XiaoMi smart gateway firmware has one over-privileged function shared by local and remote caller function. Furthermore, all command functions can be directly called by this function. Therefore, all remote commands can be sent by a local interactive entity. In addition to abusing the commands which used to complete simple tasks like setting the device ID in above example, we show how to distribute malicious firmware to the device through a complicated *OTA_update* command function.

Normally, when new version firmware is available in a cloud, the cloud will send the download URL to the corresponding device. Then the device downloads the firmware from the URL. Due to over-privileged shared functions, we can also trigger this command locally. However, simply invoking the *OTA_update* command function cannot successfully complete the whole process of updating firmware. The *OTA_update* function will further call a sequence of functions to download, parse and verify the firmware. If one function cannot be completely finished, the whole process will be fail.

To ensure the completion of all necessary functions, we can use Gerbil again to identify the path constraints. Through manual analysis, we know if firmware has been successfully updated, the device will reply with a finalization message. Thus, we input the command function as the start point and functions which send the finalization message as end point to the Gerbil. According to path constraints of Gerbil, we can construct a firmware contrived to meet all the constraints. For example, we found the device uses the MD5 message-digest algorithm to verify the firmware, so we can calculate the MD5 value matching our manipulated firmware.

JD Smart Oven with Model Changdi CRWF321ML. There are three over-privileged shared functions in Changdi smart oven firmware. One of them can be reached from all three caller functions and invoke most command functions. We identify 22 command functions called by this shared function including setting worktime and temperature etc. In contrast to vulnerabilities that can be successfully exploited, We show some commands cannot be abused by over-privileged function and explain the reason in this case.

Command function *unbind* and *reset* can only be invoked when user physically pushing the corresponding button on the oven in normal use. Due to the

over-privileged shared function, these functions can also be called by local caller function. However, we found these two command functions are not successfully completed if we invoke them from a local or remote caller function. After manual reverse-engineering, we found the button peripheral value will be checked after these two command functions invoked. Since the value of peripheral register can only be changed through physically touching (e.g., pressing or releasing the button), the oven cannot perform these two commands sent by mobile app or cloud.

5 Discussion

In this section, we discuss the how to prevent privilege separation vulnerabilities. At the same time, we also discuss the limitations of Gerbil and how to mitigate them in our plan for future work.

5.1 Mitigation

Above all, developers should deploy a strict privilege separation model in IoT firmware. To be specific, operations carried out by the device should be clearly divided into several mutually independent sets based on interactive entity, e.g., cloud-set and local-set. Therefore, depending on which set a command belongs to, each caller function should be granted appropriate privileges. More importantly, the control flow and data flow from the cloud caller function, local caller function and physical caller function should be strictly separated. In addition, if the developer has to use a shared function to handle commands from different interactive entities, they should require an additional verification of the identity of the caller function in the shared function.

As an alternative, the manufacturer can eliminate local interface to IoT devices, as with Alibaba's devices. In other words, every command must be routed to the cloud and IoT devices only accept commands from the cloud. In this case, even if the user is at home, the commands must go through the cloud, resulting in a longer latency. However, we argue that latency is not a critical metric in the smart home scenario, and sacrificing some performance for security is worthwhile. In addition, smart devices should enhance authentication of interactive entities. In the interaction scenarios, the lack of local authentication makes the privilege separation vulnerability easier to exploit.

5.2 Limitation

First, the function recognition method used by Gerbil is based on the FLIRT algorithm. However, FLIRT cannot handle the problem of signature conflicts. Thus, if multiple functions generate the same code signature, only one of them can be selected for identification. In order to solve the conflict problem, we plan to integrate other function recognition methods such as the control flow based method [16] and function semantic based method [12].

Second, using Gerbil to perform symbolic execution requires human intervention. In this work, we have developed an assistant tool to help analyst easily slice the portion of firmware which is most likely to have privilege separation vulnerabilities for symbolic execution. However, Gerbil cannot perform symbolic execution on entire binary firmware images if analysts want to use Gerbil to identify other kinds of vulnerabilities, they have to rely on manual analysis for the slice specification. We will integrate other technology like taint track optimize the Gerbil to minimize manual work.

Third, not all privilege separation vulnerabilities can be successfully exploited and need to be further verified. Since it is hard to find the entire execution path for successfully and completely performing one command in firmware, we use command function which is the first individual function to perform a specific command as the end point for symbolic execution to identify privilege separation vulnerabilities. In most cases, if the command functions are invoked, they will automatically call all necessary program related to finishing the tasks indicated by the corresponding command. However, for some commands associated with complicated processes like updating the firmware, command function will carry out some further additional checks to the parameters, as we mentioned in Sect. 4.4. Thus, the analyst has to invoke target command functions to verify whether the corresponding commands have actually carried out or not. If not, the analyst need to do further manual analysis or reuse Gerbil to figure it out.

6 Related Work

We review related research on IoT security from two aspect: privilege management and firmware analysis.

Privilege Management. Fernandes et al. [8] revealed that over 55% of SmartApps in Samsung's store are over-privileged because the privilege management of capabilities implemented in the programming frameworks are too coarse-grained. On the other hand, many IoT platforms support trigger-action services such as IFTTT. Fernandes et al. [10] also found that the OAuth tokens for the IFTTT services are over-privileged, which can be misused by attacker to invoke API calls that are outside the capabilities of the trigger-action service itself. Some corresponding mitigation [11,13,21] also has been proposed. Our work focuses on the privilege separation model of involved in IoT firmware implementation, instead of the privilege management problem in IoT cloud services

Firmware Analysis. Several approaches are proposed for detecting the vulnerabilities in IoT firmware, including static analysis [4], dynamic analysis [1,3,25], and fuzzing [5,23]. Costin et al. [4] carried out a large-scale analysis of IoT firmware by coarse-grained comparison of files and modules. Chen et al. [1] proposed and implemented a robust software-based full system emulation, FIRMADYNE, based on kernel instrumentation. However, their approaches only work for Linux-based embedded firmware, whereas a large number of real-world IoT devices run RTOS or bare-metal systems and have limit ability to find

logic vulnerabilities. Avatar [25] enables dynamic program analysis for embedded firmware by access to the physical hardware, either through a debugging interface, or by installing a custom proxy in the target environment. However, such hardware requirements are usually unrealistic for real-world devices (e.g., in the presence of locked hardware), and not suitable for testing large-scale firmware.

For combining static and dynamic analysis, and closest to our work, Firmalice [18], an IoT binary analysis framework, utilizes symbolic execution on the part of firmware binary to identify the authentication vulnerabilities. Compared to Firmalice, Gerbil greatly enhances the capabilities for symbolic execution to deal with unknown lightweight IoT firmware. For example, Gerbil can restore the library function semantic information in IoT firmware thus it can output function-level path constraints and skip complicated library functions to mitigate path exploration.

7 Conclusion

In this paper, we approached the vulnerability analysis of IoT firmware from a new angle -the privilege separation model- and identified privilege separation vulnerability caused by over-privileged shared function abuse. Then, we presented Gerbil, a firmware-analysis-specific extension of Angr to detect privilege separation vulnerabilities in IoT firmware with little manual analysis. The high-level idea is to identify over-privileged shared functions based on path constraints of symbolic execution. With the help of Gerbil, we show that privilege separation vulnerabilities widely exist in real-world IoT firmware. We also demonstrated how to verify and exploit privilege separation vulnerabilities with real-world devices. Besides, our evaluation shows that all components of the Gerbil can efficiently help IoT firmware analysis. Finally, we proposed several defensive design suggestions to prevent the generation of privilege separation vulnerabilities in the first place and our plan for Gerbil's future improvement.

Acknowledgments. We would like to thank the anonymous reviewers for their helpful feedback. Wei Zhou and Yuqing Zhang were support by National Key R&D Program China (2016YFB0800700), National Natural Science Foundation of China (No. U1836210, No. 61572460) and in part by CSC scholarship. Peng Liu was supported by NSF CNS-1505664 and NSF CNS-1814679. Note that any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any funding agencies.

References

1. Chen, D.D., Woo, M., Brumley, D., Egele, M.: Towards automated dynamic analysis for linux-based embedded firmware. In: NDSS, pp. 1–16 (2016)
2. Chen, J., Diao, W., Zhao, Q., Zuo, C.: IoTFuzzer: discovering memory corruptions in IoT through app-based fuzzing. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA (2018)

3. Choi, Y.H., Park, M.W., Eom, J.H., Chung, T.M.: Dynamic binary analyzer for scanning vulnerabilities with taint analysis. Multimedia Tools Appl. **74**(7), 2301–2320 (2015)
4. Costin, A., Zaddach, J., Francillon, A., Balzarotti, D.: A large-scale analysis of the security of embedded firmwares. In: 23rd USENIX Security Symposium (USENIX Security 2014), pp. 95–110 (2014)
5. Costin, A., Zarras, A., Francillon, A.: Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, pp. 437–448. ACM (2016)
6. Ericson: The Ericsson Mobility Report (2019). https://www.ericsson.com/en/mobility-report
7. Feng, Q., Zhou, R., Xu, C., Cheng, Y., Testa, B., Yin, H.: Scalable graph-based bug search for firmware images. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 480–491. ACM (2016)
8. Fernandes, E., Jung, J., Prakash, A.: Security analysis of emerging smart home applications. In: 2016 IEEE symposium on security and privacy (SP), pp. 636–654. IEEE (2016)
9. Fernandes, E., Paupore, J., Rahmati, A., Simionato, D., Conti, M., Prakash, A.: FlowFence: practical data protection for emerging IoT application frameworks. In: Proceedings of Usenix Security Symposium, pp. 531–548 (2016)
10. Fernandes, E., Rahmati, A., Jung, J., Prakash, A.: Decentralized action integrity for trigger-action IoT platforms. In: Proceedings of Network and Distributed Systems Symposium (NDSS), pp. 18–21 (2018)
11. He, W., et al.: Rethinking access control and authentication for the home Internet of Things (IoT). In: 27th USENIX Security Symposium (USENIX Security 2018), pp. 255–272 (2018)
12. Jacobson, E.R., Rosenblum, N.E., Miller, B.P.: Labeling library functions in stripped binaries. In: Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools, pp. 1–8. ACM (2011)
13. Jia, Y.J., et al.: ContexIoT: towards providing contextual integrity to appified IoT platforms. In: NDSS (2017)
14. Jiang, Y., Xie, W., Tang, Y.: Detecting authentication-bypass flaws in a large scale of IoT embedded web servers. In: Proceedings of the 8th International Conference on Communication and Network Security, pp. 56–63. ACM (2018)
15. Pro, I.: Fast library identification and recognition technology (2019). https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml
16. Qiu, J., Su, X., Ma, P.: Using reduced execution flow graph to identify library functions in binary code. IEEE Trans. Softw. Eng. **42**(2), 187–202 (2016)
17. Rays, H.: Fast library identification and recognition technology (2015). https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml
18. Shoshitaishvili, Y., Wang, R., Hauser, C., Kruegel, C., Vigna, G.: Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In: NDSS (2015)
19. Shoshitaishvili, Y., et al.: SoK: (State of) the art of war: offensive techniques in binary analysis. In: IEEE Symposium on Security and Privacy (2016)
20. Stephens, N., et al.: Driller: augmenting fuzzing through selective symbolic execution. In: NDSS, pp. 1–16, no. 2016 in 16 (2016)
21. Tian, Y., et al.: Smartauth: user-centered authorization for the Internet of Things. In: 26th USENIX Security Symposium (USENIX Security 2017), pp. 361–378 (2017)

22. Wei, Z., et al.: Discovering and understanding the security hazards in the interactions between IoT devices, mobile apps, and clouds on smart home platforms. In: 28th USENIX Security Symposium (USENIX Security 2019). USENIX Association, Santa Clara (2019). https://www.usenix.org/conference/usenixsecurity19/presentation/zhou
23. Yaowen, Z., Ali, D., Heng, Y., Chengyu, S., Hongsong, Z., Limin, S.: FIRM-AFL: high-throughput greybox fuzzing of IoT firmware via augmented process emulation. In: 28th USENIX Security Symposium (USENIX Security 2019). USENIX Association, Santa Clara (2019). https://www.usenix.org/conference/usenixsecurity19/presentation/zheng
24. Yu, H., Lim, J., Kim, K., Lee, S.B.: Pinto: enabling video privacy for commodity IoT cameras. In: CCS, pp. 1089–1101. ACM (2018)
25. Zaddach, J., Bruno, L., Francillon, A., Balzarotti, D., et al.: AVATAR: a framework to support dynamic security analysis of embedded systems' firmwares. In: 21st Annual Network and Distributed System Security Symposium, NDSS, pp. 1–16 (2014)