Mercedes-Benz MBUX Security Research Report
# Contents

1 Introduction ................................................................. 2

2 Architecture overview .................................................. 5

2.1 Hardware ........................................................................... 5

2.1.1 Head Unit ................................................................. 5

2.1.2 T-Box ............................................................................ 10

2.1.3 Electronic Ignition Switch ........................................... 11

2.1.4 Instrument Cluster ...................................................... 12

2.2 Software ............................................................................ 12

2.2.1 Head Unit ................................................................. 12

2.2.2 T-Box ............................................................................ 12

2.3 CAN Network Overview ................................................. 13

3 Research Environment Setup ........................................... 14

3.1 Connecting ECUs ........................................................... 14

3.2 Wake Up Test Bench ....................................................... 15

3.3 Anti-Theft ......................................................................... 16

4 Attack Surfaces Analysis .................................................. 17

4.1 Head Unit ......................................................................... 17

4.1.1 Attack Through Browser ........................................... 17

4.1.2 Wi-Fi ............................................................................. 17

4.1.3 Kernel ........................................................................... 18

4.1.4 Ports on MMB ............................................................ 18
# Contents

4.1.5 Bluetooth .......................................................................................................................... 19  
4.1.6 USB .................................................................................................................................. 19  
4.1.7 App .................................................................................................................................. 19  
4.2 T-Box ...................................................................................................................................... 19  
4.2.1 Attack Through Wi-Fi Chip .............................................................................................. 19  
4.2.2 Attack Through GNSS ...................................................................................................... 20  
4.2.3 CAN ..................................................................................................................................... 20  
4.2.4 Baseband ........................................................................................................................... 20  
4.2.5 GSM hijack ....................................................................................................................... 21  

5 Compromise Head Unit ........................................................................................................... 24  
5.1 Access to the Intranet of Head Unit ......................................................................................... 24  
5.1.1 Connect to Head Unit as T-Box ........................................................................................ 24  
5.1.2 Connect to MMB as CSB .................................................................................................. 25  
5.2 Remote Code Execution on Head Unit .................................................................................... 26  
5.2.1 Implementation of HiQnet Protocol .................................................................................. 28  
5.2.2 Vulnerabilities in HiQnet Protocol .................................................................................. 30  
5.2.3 Exploit HiQnet Protocol Vulnerability .......................................................................... 36  
5.3 Exploit the Browser ................................................................................................................ 40  
5.3.1 QtWebEngine ................................................................................................................... 41  
5.3.2 Exploit the QtWebEngine .............................................................................................. 41  
5.4 Local Privilege Escalation ..................................................................................................... 42  
5.4.1 Kernel LPE with A perf Bug .......................................................................................... 42
# Contents

5.4.2 CVE-2017-6786,6001 .......................................................... 43
5.4.3 Bypass Cgroups Restriction ............................................. 43

6 Post Attack in Head Unit ....................................................... 46
6.1 Anti-Theft Unlock ............................................................. 46
6.2 Unlocking Vehicle Functions ............................................. 48
6.3 Engineering Mode ............................................................ 49
6.4 Persistent Backdoor .......................................................... 51
6.5 Display Screen Tampering ................................................ 51
6.6 RH850 Denial of Service .................................................. 53
6.7 Perform Vehicle Control Actions ....................................... 53

7 Compromise T-Box .............................................................. 57
7.1 Compromise Host from Wi-Fi chip ..................................... 57
7.2 Trigger Memory Corruption From SH2A Chip .................. 58
   7.2.1 Message Format between SH2A MCU and Host .......... 59
   7.2.2 Out-of-bound Vulnerability in RemoteDiagnosis .......... 59

8 Post Attack in T-Box ............................................................ 61
8.1 Sending Arbitrary CAN message from T-Box ..................... 61
   8.1.1 CAN Bus Message Transmit Logic ........................... 61
   8.1.2 Vulnerability in SH2A Firmware .............................. 62
   8.1.3 Transmit Arbitrary CAN Message to CAN Bus .......... 63
8.2 Flashing Custom Firmware on SH2A MCU ...................... 63
PART 1
Overview
1 Introduction

In the past years, we have analyzed the security of connected vehicles from top brands worldwide, such as BMW\textsuperscript{[1]}, Lexus\textsuperscript{[2]}, and Tesla\textsuperscript{[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\textsuperscript{[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\textsuperscript{[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.
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 flash 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.

![Architecture overview diagram](image)

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.
CHAPTER 2: ARCHITECTURE OVERVIEW

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

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
CHAPTER 2: ARCHITECTURE OVERVIEW

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**

![Base Board Top View](image)

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.
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

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.
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.
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.

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

Table 3.1: Wake-up CAN signals

<table>
<thead>
<tr>
<th>ID</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x25E</td>
<td>64 64 64 00 03 00 00 00</td>
</tr>
<tr>
<td>0x2F7</td>
<td>C2 50 10 57 12 5D 5F 53</td>
</tr>
<tr>
<td>0x020</td>
<td>39 C9 41 1C C0 00 00 C0</td>
</tr>
</tbody>
</table>

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.

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

Figure 4.1: Attack surfaces
CHAPTER 4: ATTACK SURFACES ANALYSIS

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.
CHAPTER 4: ATTACK SURFACES ANALYSIS

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.

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.
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](image)

We found 4 Ethernet testing points on **BB**. They are **CSB**’s Ethernet testing points.
We removed CSB from head unit and soldered these testing points with an RJ45 cable.

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.

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
CHAPTER 5: COMPROMISE HEAD UNIT

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.

```
<table>
<thead>
<tr>
<th>Version</th>
<th>Header Length</th>
<th>Header Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Source Device Address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Destination Device Address</td>
</tr>
<tr>
<td></td>
<td>Flags</td>
<td>Hop Count</td>
</tr>
</tbody>
</table>
```

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>
<thead>
<tr>
<th>ATTRIBUTE ID</th>
<th>ATTRIBUTE NAME</th>
<th>ATTRIBUTE TYPE</th>
<th>CATEGORY</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Data Type</td>
<td>STRING</td>
<td>Static</td>
</tr>
<tr>
<td>1</td>
<td>Name String</td>
<td>STRING</td>
<td>Instance+Dynamic</td>
</tr>
<tr>
<td>2</td>
<td>Minimum Value</td>
<td>Data Type</td>
<td>Instance</td>
</tr>
<tr>
<td>3</td>
<td>Maximum Value</td>
<td>Data Type</td>
<td>Instance</td>
</tr>
<tr>
<td>4</td>
<td>Control Law</td>
<td></td>
<td>Static</td>
</tr>
<tr>
<td>5</td>
<td>Flags</td>
<td>UWORD</td>
<td>Static</td>
</tr>
</tbody>
</table>

The Figure 5.7 shows the relationship between these abstract objects.
CHAPTER 5: COMPROMISE HEAD UNIT

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.

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 libplugincontrolinterfacetg6.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

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

**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.
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.

![Payload for Message Type MultiSvGet](image)

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.

![Code snippet in function CHiQnetMsg::CHiQnetMsg](image)

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.

![Figure 5.12: Vulnerability in CHiQnetPayloadMultiSvGet::CHiQnetPayloadMultiSvGet()](image)

**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](image)

During the *analyzing stage*, the function *CHiQnetPayloadGetAttributes::CHiQnetPayloadGetAttributes* 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.
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>
<thead>
<tr>
<th>OFFSET</th>
<th>TYPE</th>
<th>COUNT</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0~0x3FF</td>
<td>USHORT</td>
<td>0x200</td>
<td>Param_ID</td>
</tr>
<tr>
<td>0x400~0x413</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x414~0x415</td>
<td>USHORT</td>
<td>1</td>
<td>count</td>
</tr>
<tr>
<td>0x416~0x417</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x418~0x41F</td>
<td>struct Sv *</td>
<td>0x200</td>
<td>p_Sv</td>
</tr>
</tbody>
</table>

During the processing stage, the function *CHiQnetPayloadMultiSvSet::SetSV*s 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
CHAPTER 5: COMPROMISE HEAD UNIT

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.

![Figure 5.15: Vulnerability in CHiQnetPayloadMultiSvSet::SetSVs()](image)

### 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.

![Figure 5.16: Code snippet of CHiQnetPayloadMultiSvSetAttributes::SetSVsAttributes()](image)

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.
CHAPTER 5: COMPROMISE HEAD UNIT

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.

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:

```c
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>
<thead>
<tr>
<th>OFFSET</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>v_pointer</td>
</tr>
<tr>
<td>0x8</td>
<td>CHBString::StringData*p_chbstring</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

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.

```cpp
    case 2u:
        v13 = CSvAttribute::GetMinMaxDataType(v8);
        CHiQnetPayload::Set(v3, v13);
        v14 = *(void *)__fastcall__(int64, CHiQnetPayloadMultiSvGetAttributes *)((int32)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 \( p_{\text{chbstring}} \) 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 \( p_{\text{chbstring}} \) in this CStateVariable. For the second case, the pointer \( p_{\text{chbstring}} \) 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:

```
0000000004A5000 ; `table for `am::TAmShTimerCallBack<am::CAmCommonAPIWrapper>`
0000000004A5000 ; |ZTVNzam18TAmShTimerCallbackINS_19CAmCommonAPIWrapperEEDCQ 0
0000000004A5000 ; | ; DATA XREF: LOAD:000000000462A470
0000000004A5000 ; | ; .got_ZTVNzam18TAmShTimerCallbackINS_19CAmCommonAPIWrapperEEE_ptr+4
0000000004A5000 ; | ; offset to this
0000000004A5000 ; | DCO _ZTVNzam18TAmShTimerCallbackINS_19CAmCommonAPIWrapperEEE : `typeinfo for `am::TAmShTimerCallBack`
0000000004A5010 ; DCO _ZTVNzam18TAmShTimerCallbackINS_19CAmCommonAPIWrapperEEDCQ : `am::TAmShTimerCallBack`
0000000004A5010 ; | ; DATA XREF: LOAD:000000000462A470
0000000004A5010 ; | ; .got_ZTVNzam18TAmShTimerCallbackINS_19CAmCommonAPIWrapperEEDCQ
0000000004A5020 ; DCO _ZTVNzam18TAmShTimerCallbackINS_19CAmCommonAPIWrapperEEDCQ : `am::TAmShTimerCallBack`
0000000004A5020 ; | ; DATA XREF: LOAD:000000000462A470
0000000004A5020 ; | ; .got_ZTVNzam18TAmShTimerCallbackINS_19CAmCommonAPIWrapperEEDCQ
0000000004A5020 ; | WEAK _ZTVN5TCLAPI2ArgExceptionHandler
```

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 \( p_{\text{chbstring}} \) 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 4th QWORD to 0x11 to bypass the header of CHBString::data.

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

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.

![Figure 5.23: Libraries used by DevCtrlBrowser](image)

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.

![Figure 5.24: Reversed shell](image)
5.4 Local Privilege Escalation

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

In the audio/video 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 -n 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 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\textsuperscript{[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 \textit{upstream}\textsuperscript{[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 \textsuperscript{[13]}. This fix has been applied in this kernel. But there’s a second unapplied fix CVE-2017-6001 \textsuperscript{[14]}.

Without the second fix, the bug is still exploitable.

\textbf{5.4.2 CVE-2017-6786,6001}

KeenLab published the bug analysis and exploit method in \textit{PACSEC}\textsuperscript{[15]}. Exploit steps in \textit{PACSEC} are:

- Trigger race condition in \textit{move_group} to cause UAF.
- Freeze with \textit{futex\_wait\_queue\_me()} to avoid kernel Oops.
- Spray heap with \textit{ret2dir}. Filling malformed \textit{perf\_event\_context\_object}.
- Wake frozen task with \textit{futex\_wake()} and hijack control flow.

In the head unit, exploit steps need to be adjusted because of \textit{Cgroups} restriction.

\textbf{5.4.3 Bypass Cgroups Restriction}

After running our exploit inside the spawned shell from \textit{AudioManager}, the exploit was killed by \textit{OOM killer} in \textit{ret2dir} heap spray stage.
CHAPTER 5: COMPROMISE HEAD UNIT

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 obtaining 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.

![Figure 6.1: Anti-Theft DLT log](image)

By searching string literals in file SysAct, we found a relevant function.
CHAPTER 6: POST ATTACK IN HEAD UNIT

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

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.
CHAPTER 6: POST ATTACK IN HEAD UNIT

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.

![DLT injection dialog](image)

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.

```c
    case 2u:
        v7 = 0;
        std::string::string(&s8, *(QWORD *)&s2 + 17LL, v13);
        v8 = *(QWORD *)(s1 + 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
                    ss (int)dlt_user_log_write_start(snk_7D2A70, v15, 4LL) > 0 )
                    {dlt_user_log_write_string(v15, "Activating all subsystems after DLT injection verification!");
                     dlt_user_log_write_finish(v15);
                    }
                CBsysActActionHandler::enableAllSubsystems(s1);
            }
            else if ( (unsigned int)((__int64 *)__fastcall *)__int64, __int64)sub_4B26DC(3LL, v10) == 1
                    { ss (int)dlt_user_log_write_start(snk_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.

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.

Figure 6.7: Dealer Mode menu
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’.

---

**Figure 6.8: part of README.md file**

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>Like a DRAG WEST gesture. Opens function list menu.</td>
</tr>
<tr>
<td>Right</td>
<td>Like a DRAG EAST gesture. Closes function list menu.</td>
</tr>
<tr>
<td>Up</td>
<td>Like a DRAG NORTH gesture. Often used to navigate lists.</td>
</tr>
<tr>
<td>Down</td>
<td>Like a DRAG SOUTH gesture. Often used to navigate lists.</td>
</tr>
<tr>
<td>Return</td>
<td>Often used for selection.</td>
</tr>
<tr>
<td>Escape</td>
<td>Like pressing the CCE Back button. Used to traverse back up a menu hierarchy.</td>
</tr>
<tr>
<td>Backspace</td>
<td>&quot;send&quot; button</td>
</tr>
<tr>
<td>F1-F9</td>
<td>Open/close Home overlay.</td>
</tr>
<tr>
<td>F10-F12</td>
<td>Open/close Drive/Agility overlay.</td>
</tr>
<tr>
<td>F13-F15</td>
<td>Open/close Favorites overlay.</td>
</tr>
<tr>
<td>F16-F18</td>
<td>Open/close Audio overlay.</td>
</tr>
<tr>
<td>F19-F21</td>
<td>Open Navigation application.</td>
</tr>
<tr>
<td>F22-F24</td>
<td>Open Media Player application.</td>
</tr>
<tr>
<td>F25-F27</td>
<td>Open Phone application.</td>
</tr>
<tr>
<td>F28-F30</td>
<td>Open Radio application.</td>
</tr>
<tr>
<td>F31-F32</td>
<td>Open Developer application (for internal use only).</td>
</tr>
<tr>
<td>Ctrl + Up</td>
<td>Toggle on-screen logging overlay (debug view only)</td>
</tr>
<tr>
<td>Ctrl + Down</td>
<td>Open Audio overlay or close Home overlay.</td>
</tr>
<tr>
<td>Ctrl + Left</td>
<td>Open Home overlay or close Audio overlay.</td>
</tr>
<tr>
<td>Ctrl + Right</td>
<td>Open Favorites overlay or close Drive/Agility overlay.</td>
</tr>
<tr>
<td>Ctrl + Z</td>
<td>Open Drive/Agility overlay or close Favorites overlay.</td>
</tr>
<tr>
<td>Alt + L</td>
<td>Toggle styles.</td>
</tr>
<tr>
<td>Shift + L</td>
<td>Toggle themes.</td>
</tr>
<tr>
<td>Control</td>
<td>Toggle between text and translation 10s.</td>
</tr>
<tr>
<td>(Command</td>
<td>Toggle between text and text fill mode.</td>
</tr>
<tr>
<td>on OSX)</td>
<td>Toggle between text and grid text mode.</td>
</tr>
<tr>
<td>Control</td>
<td>Save current station to Favorites station list.</td>
</tr>
<tr>
<td>(Command</td>
<td>Activate full automatic mode for Fan Speed and Air Distribution.</td>
</tr>
<tr>
<td>on OSX)</td>
<td>Scale window up by 2x. (limited to 100%)</td>
</tr>
<tr>
<td>Control</td>
<td>Scale window down by 2x. (limited to 10%)</td>
</tr>
<tr>
<td>(Command</td>
<td>Toggle window scale between 10.5&quot; and 12&quot;. Additional values may be</td>
</tr>
<tr>
<td>on OSX)</td>
<td>added to Global.conf</td>
</tr>
<tr>
<td>Control</td>
<td>Triggers &quot;Home Menu&quot;</td>
</tr>
<tr>
<td>(Command</td>
<td>Triggers &quot;Dealer Mode&quot;</td>
</tr>
<tr>
<td>on OSX)</td>
<td>Activate &quot;Climate menu overlay&quot; in Touch UI or &quot;Climate Control Popup&quot; in PQR UI</td>
</tr>
<tr>
<td>Control</td>
<td>Perform a jump to Shortcuts or Assistance applied in SY.</td>
</tr>
<tr>
<td>(Command</td>
<td>Toggles 'Control Center (BGA) overlay' in TOUCH UI</td>
</tr>
<tr>
<td>on OSX)</td>
<td>Toggles additional couple details for the Bluetooth device manager</td>
</tr>
<tr>
<td>Control</td>
<td>Toggle Defrost Front</td>
</tr>
<tr>
<td>(Command</td>
<td>Toggle Defrost Rear (Backlite)</td>
</tr>
<tr>
<td>on OSX)</td>
<td>Toggle between PQR UI and Touch UI</td>
</tr>
<tr>
<td>Control</td>
<td>Toggle air condition system.</td>
</tr>
<tr>
<td>(Command</td>
<td>Change air flow mode</td>
</tr>
<tr>
<td>on OSX)</td>
<td>Activate Residual Heat mode</td>
</tr>
<tr>
<td>Control</td>
<td>Toggle Auxiliary Heating</td>
</tr>
<tr>
<td>(Command</td>
<td>Toggle Preconditioning</td>
</tr>
<tr>
<td>on OSX)</td>
<td>Toggle Proximity mode</td>
</tr>
<tr>
<td>Control</td>
<td>Driving Program Popup</td>
</tr>
<tr>
<td>(Command</td>
<td>Toggle air condition SYNC mode for Passenger</td>
</tr>
<tr>
<td>on OSX)</td>
<td>Show handwriting recognition overlay (Poc)</td>
</tr>
<tr>
<td>Control</td>
<td>Switch display Off (if DISPLAYSWMITCHAVLY)</td>
</tr>
<tr>
<td>(Command</td>
<td>Active Body Control hardware</td>
</tr>
<tr>
<td>on OSX)</td>
<td>Toggle Air Circulation</td>
</tr>
<tr>
<td>Control</td>
<td>Toggle Air Compressor (A/C)</td>
</tr>
<tr>
<td>(Command</td>
<td>Toggle Headunit Variant From High to Entry, Entry to High</td>
</tr>
<tr>
<td>on OSX)</td>
<td>Toggle Headunit Layout Direction From Left-to-Right to Right-to-Left</td>
</tr>
</tbody>
</table>
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](image)

### 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
CHAPTER 6: POST ATTACK IN HEAD UNIT

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=4500000
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_MMM_OVERALLOCATION=50
export QV4_MMM_MAXBLOCK_SHIFT=1
export QV4_MMM_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

![Custom images](image_url)
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](image)

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.
CHAPTER 6: POST ATTACK IN HEAD UNIT

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

<table>
<thead>
<tr>
<th>ACTION</th>
<th>PACKET IN HEXADEcimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>open ambient light</td>
<td>00 00 00 1f</td>
</tr>
<tr>
<td></td>
<td>3f 3f</td>
</tr>
<tr>
<td></td>
<td>00 00 00 1f</td>
</tr>
<tr>
<td></td>
<td>3f 3f</td>
</tr>
<tr>
<td></td>
<td>00 00 00 1f</td>
</tr>
<tr>
<td></td>
<td>3f 3f</td>
</tr>
<tr>
<td>close ambient light</td>
<td>00 00 00 1f</td>
</tr>
<tr>
<td></td>
<td>3f 3f</td>
</tr>
<tr>
<td></td>
<td>00 00 00 1f</td>
</tr>
<tr>
<td></td>
<td>3f 3f</td>
</tr>
<tr>
<td></td>
<td>00 00 00 1f</td>
</tr>
<tr>
<td></td>
<td>3f 3f</td>
</tr>
<tr>
<td>open driver reading light</td>
<td>00 00 00 17</td>
</tr>
<tr>
<td></td>
<td>3f 00</td>
</tr>
<tr>
<td>close driver reading light</td>
<td>00 00 00 17</td>
</tr>
<tr>
<td></td>
<td>3f 00</td>
</tr>
<tr>
<td>open passenger reading light</td>
<td>00 00 00 17</td>
</tr>
<tr>
<td></td>
<td>7f 00</td>
</tr>
<tr>
<td>close passenger reading light</td>
<td>00 00 00 17</td>
</tr>
<tr>
<td></td>
<td>3f 00</td>
</tr>
<tr>
<td>open sunshade cover</td>
<td>00 00 00 15</td>
</tr>
<tr>
<td></td>
<td>3f 3f</td>
</tr>
<tr>
<td>open back-seat passenger light</td>
<td>00 00 00 17</td>
</tr>
<tr>
<td></td>
<td>3f 00</td>
</tr>
<tr>
<td>close back-seat passenger light</td>
<td>00 00 00 17</td>
</tr>
<tr>
<td></td>
<td>3f 00</td>
</tr>
</tbody>
</table>
PART 3
T-BOX
CHAPTER 7: COMPROMISE T-BOX

CHAPTER 7: COMPROMISE 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:0000E000  LDR    R3, [LR]
.text:0000E004  ORR    R3, R3, #1
.text:0000E008  STR    R3, [LR]
.text:0000E00C  LDR    R3, [R4, #0x24]
.text:0000E010  LDR    R6, [R5, #0x20]
.text:0000E014  UBFX   R3, R3, #0x6, #1
.text:0000E018  BL     vfs_create
.text:0000E01C  SUBS   R5, R0, #0
.text:0000E020  BEQ    loc_C0086CC

.text:0000E024  loc_C0086CC
.text:0000E024  LDR    R0, [SP, #0x78+var_60]
.text:0000E028  BL     gprintf
.text:0000E02C  LDR    R12, [SP, #0x78+var_50]

Figure 7.1: Original code of kernel
```

After the attack, the crash log on serial is shown in Figure 7.2.
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.
CHAPTER 7: COMPROMISE T-BOX

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:

| OxFFFF | OxFFA | 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.

```c
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.

```c
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 blocklpcServer. 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>
<thead>
<tr>
<th>PARTS</th>
<th>VERSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Part Number</td>
<td>2479026602</td>
</tr>
<tr>
<td>TCU Core</td>
<td>E334.2</td>
</tr>
<tr>
<td>SH2</td>
<td>18232C</td>
</tr>
</tbody>
</table>

### 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 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.

<table>
<thead>
<tr>
<th>序号</th>
<th>传输方向</th>
<th>时间标识</th>
<th>标识符</th>
<th>帧格式</th>
<th>帧类型</th>
<th>数据长度</th>
<th>数据(0xH)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00791513</td>
<td>接收</td>
<td>17:13:18 096.0</td>
<td>0x00000000</td>
<td>封装帧</td>
<td>标准帧</td>
<td>0x08</td>
<td>00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>00791514</td>
<td>接收</td>
<td>17:13:18 103.0</td>
<td>0x00000000</td>
<td>封装帧</td>
<td>标准帧</td>
<td>0x08</td>
<td>11 22 33 44 55 66 77 88</td>
</tr>
<tr>
<td>00791515</td>
<td>接收</td>
<td>17:13:19 107.0</td>
<td>0x00000000</td>
<td>封装帧</td>
<td>标准帧</td>
<td>0x08</td>
<td>39 40 41 42 00 00 00 00 00</td>
</tr>
<tr>
<td>00791516</td>
<td>接收</td>
<td>17:13:19 113.0</td>
<td>0x00000000</td>
<td>封装帧</td>
<td>标准帧</td>
<td>0x08</td>
<td>11 22 33 44 55 66 77 88</td>
</tr>
<tr>
<td>00791517</td>
<td>接收</td>
<td>17:13:19 118.0</td>
<td>0x00000000</td>
<td>封装帧</td>
<td>标准帧</td>
<td>0x08</td>
<td>39 40 41 42 00 00 00 00 00</td>
</tr>
<tr>
<td>00791518</td>
<td>接收</td>
<td>17:13:19 123.0</td>
<td>0x00000000</td>
<td>封装帧</td>
<td>标准帧</td>
<td>0x08</td>
<td>11 22 33 44 55 66 77 88</td>
</tr>
<tr>
<td>00791519</td>
<td>接收</td>
<td>17:13:19 142.0</td>
<td>0x00000000</td>
<td>封装帧</td>
<td>标准帧</td>
<td>0x08</td>
<td>11 22 33 44 55 66 77 88</td>
</tr>
<tr>
<td>00791520</td>
<td>接收</td>
<td>17:13:19 151.0</td>
<td>0x00000000</td>
<td>封装帧</td>
<td>标准帧</td>
<td>0x08</td>
<td>11 22 33 44 55 66 77 88</td>
</tr>
<tr>
<td>00791521</td>
<td>接收</td>
<td>17:13:19 155.0</td>
<td>0x00000000</td>
<td>封装帧</td>
<td>标准帧</td>
<td>0x08</td>
<td>11 22 33 44 55 66 77 88</td>
</tr>
<tr>
<td>00791522</td>
<td>接收</td>
<td>17:13:19 158.0</td>
<td>0x00000000</td>
<td>封装帧</td>
<td>标准帧</td>
<td>0x08</td>
<td>11 22 33 44 55 66 77 88</td>
</tr>
<tr>
<td>00791523</td>
<td>接收</td>
<td>17:13:19 158.0</td>
<td>0x00000000</td>
<td>封装帧</td>
<td>标准帧</td>
<td>0x08</td>
<td>11 22 33 44 55 66 77 88</td>
</tr>
<tr>
<td>00791524</td>
<td>接收</td>
<td>17:13:19 158.0</td>
<td>0x00000000</td>
<td>封装帧</td>
<td>标准帧</td>
<td>0x08</td>
<td>11 22 33 44 55 66 77 88</td>
</tr>
<tr>
<td>00791525</td>
<td>接收</td>
<td>17:13:19 158.0</td>
<td>0x00000000</td>
<td>封装帧</td>
<td>标准帧</td>
<td>0x08</td>
<td>11 22 33 44 55 66 77 88</td>
</tr>
</tbody>
</table>

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:
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:

```
<table>
<thead>
<tr>
<th>SH2 VERSION</th>
<th>TCU CORE VERSION</th>
<th>VERSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>18514B</td>
<td>E409.6</td>
<td>2479027703</td>
</tr>
<tr>
<td>19472B</td>
<td>E511.6</td>
<td>2479022604</td>
</tr>
</tbody>
</table>
```

Table 8.2: Version of T-Box firmware
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:

```
PublicKeg(3889016252714363598206503582631367898651735139178913298264849091761606586473420569619327469216696
8109846372689109173911828658657174615058215080363617897383125718463364803441353223939267111182842978432713421339294
72229688161913564381056421583761681816278182891159517787685740178227997765973144437290180234044492437389739913826827661284
443828359732239058942994696595871890960948993987285979287228406227412117724919549657808553790914992954872518193
108547204457319529829911808748625991412193998184233852534539062727412261991198221080849592255658864317491465632939
2454251814871152176559850871840594329273248252280151522875666168288491691656378275580118204568769291500859122316879
8586934210084732218911644525730972216181578270186974108586648652156527981780985917557951809129595122532418837353936
37775972174252978708538166453828142925183177464268828635845804448012019255818091469758279373, 65537)
```

8.2.2.2 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:

```
Aug 25 22:11:34.826 UpdateManager[1157]: [info]: SH2 image 100% complete
Aug 25 22:11:37.675 UpdateManager[1157]: [info]: SH2 image 100% 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]: Open IPC channel for SWDL
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 22:11:44.880 UpdateManager[1157]: [info]: Send message "start"
Aug 25 22:11:44.883 UpdateManager[1157]: [info]: Send chunk size 1024
Aug 25 22:11:44.883 UpdateManager[1157]: [info]: Send file size 103840
Aug 25 22:11:44.883 UpdateManager[1157]: [info]: Send write address 0x000000B
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 5% complete
Aug 25 22:11:46.677 UpdateManager[1157]: [info]: SH2 image 10% complete
Aug 25 22:11:47.139 UpdateManager[1157]: [info]: SH2 image 15% complete
Aug 25 22:11:47.605 UpdateManager[1157]: [info]: SH2 image 20% complete
Aug 25 22:11:48.067 UpdateManager[1157]: [info]: SH2 image 25% complete
Aug 25 22:11:48.530 UpdateManager[1157]: [info]: SH2 image 30% complete
Aug 25 22:11:49.002 UpdateManager[1157]: [info]: SH2 image 35% complete
Aug 25 22:11:49.463 UpdateManager[1157]: [info]: SH2 image 40% complete
Aug 25 22:11:50.026 UpdateManager[1157]: [info]: SH2 image 45% complete
Aug 25 22:11:50.496 UpdateManager[1157]: [info]: SH2 image 50% complete
Aug 25 22:11:50.969 UpdateManager[1157]: [info]: SH2 image 55% complete
Aug 25 22:11:51.442 UpdateManager[1157]: [info]: SH2 image 60% complete
Aug 25 22:11:51.914 UpdateManager[1157]: [info]: SH2 image 65% complete
Aug 25 22:11:52.386 UpdateManager[1157]: [info]: SH2 image 70% complete
Aug 25 22:11:52.858 UpdateManager[1157]: [info]: SH2 image 75% complete
Aug 25 22:11:53.330 UpdateManager[1157]: [info]: SH2 image 80% complete
Aug 25 22:11:53.802 UpdateManager[1157]: [info]: SH2 image 85% complete
Aug 25 22:11:54.274 UpdateManager[1157]: [info]: SH2 image 90% complete
Aug 25 22:11:54.746 UpdateManager[1157]: [info]: SH2 image 95% complete
Aug 25 22:11:55.218 UpdateManager[1157]: [info]: SH2 image 100% complete
Aug 25 22:11:55.690 UpdateManager[1157]: [info]: ---------------- END SH2 session ----------------
```

```
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.

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.
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.
The log shows that we successfully uploaded \textit{uHERMES.bin} and \textit{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](image)

### 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](image)

**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 is the Airbag Control Module. It controls airbag deployment.

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

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.

Figure 10.1: Verified attack chains on two scenarios
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.
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
CHAPTER 11: TARGET VERSION

11 Target Version

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

Table 11.1: Version list

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

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

* ID=Information Disclosure, RCE=Remote Code Execution, LPE=Local Privilege Escalation, DoS=Denial of Service, OOB=Out of Bound, FD=Firmware Downgrade
* 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


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