Journal of Computer Virology and Hacking Techniques (2023) 19:33–49
https://doi.org/10.1007/s11416-022-00430-5

ORIGINAL PAPER

Reversing Kia Motors Head Unit to discover and exploit software vulnerabilities

Gianpiero Costantino[1] · Ilaria Matteucci[1]

Received: 22 December 2021 / Accepted: 17 April 2022 / Published online: 25 May 2022

Abstract
Modern vehicles resemble four-wheels computers connected to the Internet via their In-Vehicle Infotainment (IVI) systems. As with PCs in the past, cars, being connected to the Internet can be potentially vulnerable. The IVI system of a car is part of the intra-vehicle network and can be the entry-point of offensive cybersecurity attacks. The intra-vehicle network, based on the CAN protocol, is vulnerable *by design*: messages are exchanged in clear. Thus, the uncontrolled access to the CAN bus may have serious impact on the vehicle itself and its passengers. In this paper, we present a vulnerability assessment, through a reverse engineering process, of Kia vehicles IVI system. In particular, we focused on reverse engineer the Kia IVI system to discover vulnerabilities that may allow an attacker to compromise the IVI functionalities and inject CAN frames into the CAN bus to alter the behaviour of (part of) the vehicle. By reverse engineering the IVI, we identified four important vulnerabilities that affect all Kia vehicles that embed the studied IVI. Finally, we show how an attacker can easily control the IVI and inject CAN bus frames by means of a Metasploit module that we wrote.

Keywords Automotive · Vulnerability Assessment · Reverse Engineering · IVI Exploit

1 Introduction

Nowadays vehicles have so much technologies that cannot be considered simple mechanical devices. The possibility to connect a vehicle to the Internet by means of its In-Vehicle Infotainment (IVI) system or through the Telematic Unit turns it into a vulnerable device similar to PCs, smartphones and IoT devices. An IVI system is usually connected to the intra-vehicle network as well as all the other Electronic Control Units (ECUs) [1] that, by communicating one another, manage all the vehicle functionalities.

The most used in-vehicle communication protocol is the "Controller Area Network", also known as *CAN bus* [2], which dates back to 1983 with Bosch and is standardised in ISO 11898-1:2015 as a simple protocol based on "CAN-H" and "CAN-L" lines. However the CAN protocol is not secure-by-design: it lacks security measures entirely [3–5]

As a node of the in-vehicle network, an IVI system communicates with other ECUs by using the CAN protocol. It aims to improve the driver user-experience by providing apps for navigation and control vehicle functionalities, such as the HVAC. IVI systems are often based on well-known Operating Systems, such as Android or Linux-based [6]. Even though Linux provides several advantages, Android OS is going to impose its supremacy also in the automotive market [7]. This is mainly caused by the advantages that such OS provides in terms of features in the connected-car scenario. Several car manufacturer (OEM) already mount on their cars IVI with Android OS and others are going to do it soon, e.g., General Motors will embed Android Automotive OS starting in 2021 [8].

As any devices controlled by a software component, also an IVI system can be vulnerable and the vulnerability impact may increase if the IVI is connected to the Internet. This is the case of the most popular attack performed back to 2015 by Miller and Valasek to the Jeep Cherokee [9] where the two researchers were able to find a running process that was listening for incoming connections on a specific port. Exploiting the listening process, they were able to remotely

Gianpiero Costantino and Ilaria Matteucci have contributed equally to this work.

✉ Gianpiero Costantino
gianpiero.costantino@iit.cnr.it

Ilaria Matteucci
ilaria.matteucci@iit.cnr.it

1 Istituto di Informatica e Telematica, CNR Pisa, Pisa, Italy

interact with the IVI system and inject CAN [10] frames into the in-vehicle network by modifying the IVI firmware.

1.1 Motivations and contribution

Beside the above considerations and to the best of our knowledge the interest to propose cybersecurity solutions within the automotive domain is getting more and more attention. On August 2021 the ISO/SAE FDIS 21434 [11] was released to draw the guidelines for the cybersecurity of onboard communication. Other security solutions are proposed by the AUTOSAR SecOC specification [12] in which the AUTOSAR consortium has described a possible solution to introduce integrity and authenticity into on board communication.

Thus, the existing gap between the necessity of introducing standardised cybersecurity solutions into the in-vehicle networks and the lack of existing mechanisms deployed on the vehicles, bring us to investigate the vulnerabilities and their impact on the security into our cars. To this aim, the main goal of this paper is to provide an answer to the following Research Questions: **RQ1**: Which kind of vulnerabilities may affect a vehicle? **RQ2**: Which is the attack impact on the vehicle functionalities? And on the driver's security/safety/privacy? **RQ3**: How strong are the current mechanisms to prevent cybersecurity attacks?

To contribute answering the research questions listed above, we targeted our test vehicle, a Kia CEED. In particular, we reverse engineered its IVI system, hereafter also referred as Head Unit (HU), that is also installed into all Kia vehicles that support the 8-inches IVI system version. In our study, we considered it as the entry point for cyberattacks especially when the vehicle is connected to the Internet.

The Head Unit available in the Kia CEED is the *GEN 5.0*, aka *iAVN* and its OS is based on Android version 4.2.2. This is one of the main reason that leads us to chose this target. In fact, we already had experience on security issues affecting Android devices. Finally, because and the use of Android as the underlying operating system for IVI system interfaces is becoming increasingly popular among major car manufacturers [7].

The reverse engineering activity we carried on allowed us to discover relevant vulnerabilities on the HU software version:
CD.EUR.SOP.003.30.180703.STD_M released to vehicles at the end of 2018. However, the found vulnerabilities still exist in further releases of the HU software, such as CD.EUR.SOP.005.7.181019.STD_M and CD.EUR.SOP.007.1.190212.STD_M, and they impact all Kia Motors vehicles' head units that run those software versions. The found vulnerabilities are listed in Table 1.

A particular attention will be given to the Arbitrary Code Execution (ACE), vulnerability id "2" in Table 1, to trigger unexpected HU functionalities. We exploit such vulnerability to inject CAN bus frames only into the M-CAN (Multimedia) bus of the car, the partition of the network on which the HU is connected. As per vehicle design, the M-CAN partition is separated by other partitions, like power-train and chassis, and CAN bus frames are filtered by the gateway ECU, which acts a physical partitions separator.

Finally, by reverse engineering the HU Operating System (OS), all system apps running on the HU and libraries, we were able also to identify the other three vulnerabilities, id "1", "3" and "4". We noticed that IVI system apps are not obfuscated making the reverse activity feasible. Then, to find vulnerabilities id "3" and "4", we developed an app, called *Kia OFFEnsivE* (KOFFEE), that runs a brute force attack on the HU to find "ids" and "payloads" to make our attack exhaustive.

1.2 Attacker model

We model the attacker as an active one who can exploit vulnerabilities of the HU to gain digital access to the car, either locally or remotely and to compromise the Head Unit. The attacker does not need to have a total impairment but only a partial one to compromise the HU. She attempts to carry out the following activities:

- The collection of information to acquire exchanged data by:
  - *sniffing* the frames in transit with the aim of learning the data they carry.
  - *information gathering* of frames in transit to identify and interpret their payloads.
- The injection of CAN frames from the HU into in-vehicle CAN bus network to trigger specific functionalities. This activity includes:
  - *fuzzing* as the manipulation of frames to reverse engineer the behaviour of target ECUs.
  - *forging*, the generation of a valid frame with the aim of generating a valid signal and activating a specific ECU functionality.
  - *replaying*, the reuse of valid frames with the aim of repeating the generation of a valid signal and reactivating a specific ECU functionality.
- The distribution of not genuine apps that may contain malicious software, through:
  - *social engineering* techniques to gain remotes access.
  - *untrusted websites* that allow the download of apps with backdoor to gain IVI system remote access.

Table 1 List of discovered Head Unit vulnerabilities

ID	Vulnerability	Description
1	List of "ids" and type of "payloads"	Ease the creation of MICOM messages and CAN frames by the corresponding IVIMessage file
2	Arbitrary Code Execution	Execute MICOM message exploiting the binary file or Framework library
3	Controlled MICOM message	Injection of unexpected messages into HU software
4	Semi-Controlled CAN frames	Injection of unexpected CAN frames into the M.bus

1.3 Outline

The remainder of this paper is structured as follows: the next section reports some cyber-security attacks performed into the automotive domain in the last decade. Section 3 presents the target of our reverse engineering activity described in Section 4. Sections 5 describes the KOFFEE Module we developed to exploit the discovered vulnerabilities to inject CAN frames into the M-CAN bus of the vehicle. Section 6 presents an example of how the KOFFEE Module can be exploit to remotely perform an end-to-end attack. Section 7 discusses the lessons learned providing an answer for each research question at the basis of our work. Section 8 concludes this paper.

2 Related work

The first experimental studies and analysis of vehicle's vulnerability date back to 2010 [13]. The authors demonstrate the fragility of the underlying intra-vehicle network. They proved that, at a certain point in time, an attacker, able to inject messages into an ECU, can leverage this ability to completely circumvent a broad array of safety-critical systems and control a wide range of automotive functions, including disabling the brakes, stopping the engine, and so on. Checkoway *et al.* [14] analyse the external attack surface of a connected vehicle by considering the I/O channels of the car, i.e., indirect physical access (ODBII port, entertainment system), short-range wireless access (Bluetooth, remote keyless entry, tire pressure, RFID car keys, and emerging short-range channels such as hotspots for WiFi access), and long-range wireless access (broadcast channels and addressable channels, i.e., the telematic system).

The first successful attack performed back to 2014 and reported in literature in 2015 by Miller and Valasek [9] is the one to the Jeep Cherokee. This proved that modern vehicles can be hacked like traditional PCs or smartphones. In 2015, also the AUDI TT airbag system has been hijacked [15]: researchers demonstrated how disable the airbags on a Audi TT (and others models) and other functions by exploiting a zero-day flaw in third-party software. Still in 2015, the Tesla Model S, which is the world's most connected car, after two years of in depth hacking, was hacked by M. Rogers and K, Mahaffey [16]. The same model was remotely hacked [17] in 2016. Chinese researchers were able to interfere with the car's brakes, door locks and other electronic features, demonstrating an attack that could cause havoc. In 2015, also General Motors (GM) was target of a carjacking [18]. A 29-year-old software developer figured out a way to attack GM cars by using a personally revised version of OnStar, which is a system built into many GM cars that lets owners do things like remotely unlock or start their cars from an app or a phone service. The tampered version, called OwnStar, allowed the hacker to locate, unlock, and start a GM vehicle by simply attaching a device somewhere on the targeted car. Whenever the car owner opens the OnStar mobile app within WiFi range of the vehicle, the OwnStar gadget placed on the car discloses all kinds of valuable information to the hacker. The official BMW web domain and ConnectedDrive portal was attacked [19]: a research discovered two zero-day vulnerabilities in both of them.

In 2016, Mitshubishi Outlander hybrid car was hacked [20]: the alarm can be turned off via security bugs in its on-board Wi-Fi. In 2016 also Nissan Leaf was hacked via Mobile app and Web Browser [21]: once connected, the vehicle's engine and brakes could be controlled remotely.

Studies of offensive security on vehicles went ahead also in 2017. Among the others, we recall the remote attack on the Bosch Drivelog Connector OBD-II Dongle pointed out by the Argus Research Team [22]. The vulnerabilities allowed the attackers to stop the engine of a moving vehicle using the drivelog platform. In [23], a social engineering attack was developed in order to grab information circulating on the CAN bus of a vehicle by using an Android based after market infotainment system installed on a VW Golf 6. Also in 2017, Tesla model S was remotely hacked by the Keen Security Lab [24] that discovered a vulnerability allowing a full attack chain to implement arbitrary CAN BUS and ECUs remote controls on Tesla motors with latest firmware is possible.

Again, the Keen Security Lab, in 2018, presented a set of vulnerabilities of BMW cars [25] that make them prone to remote access. In particular, an attacker can exploit such vulnerability to inject Unified Diagnostic Services (UDS) frames into the CAN network bypassing the central gateway.

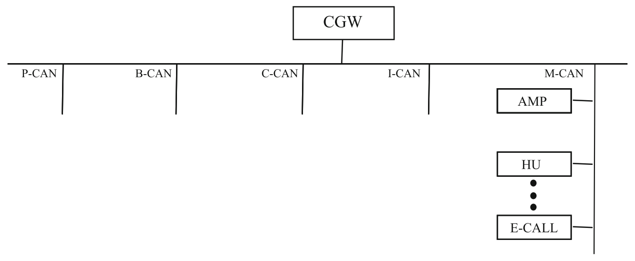


Fig. 1 Kia CEED In-Vehicle Network

In fact, as reported in [26], fault can been injected on automotive diagnostic protocols. Also, Subaru [27], Volkswagen [28] and Tesla [29] again were affected by remote attacks. Note that the attack to Tesla Model S of 2018 is different from the previous one. In 2018, the attackers pointed out a vulnerability that may allow a thief to steal a Tesla Model S in seconds by cloning its key fob. Interestingly enough, recently, November 2020, Belgian researchers demonstrate a third attack on the car manufacturer's keyless entry system [30]. This time breaking into a Model X in matter of minutes. In 2019, the attack CANDY CREAM [31] exploited first a remote vulnerability discovered into an after-market infotainment system. In addition, the researcher were able to inject CAN bus frames when the infotainment system was connected to the CAN bus. In March 2020, the Keen Security Lab has been performed an experimental security assessment on Lexus Cars [32] pointed out a new vulnerability on the infotainment system of the Toyota Lexus car. Recently, spring 2021, Weinmann and Schmotzle [33] remotely exploited the Tesla Model 3. In this attack, the authors were able to disable the car's firewall and send messages through the car's gateway to open its doors.

3 Our target

The target is the Kia Motors Head Unit with software version *CD.EUR.SOP.003.30.180703.STD_M*. The Head Unit we reverse engineered is mounted in all Kia vehicle that support the 8-inches model. In particular, the Head Unit we considered in this paper is mounted in a Kia CEED 1.6 CRDi 136 CV DCT. The in-vehicle network of the Kia CEED is shown in Fig. 1.

The Kia CEED has an internal network separation that aims to divide untrusted zones, such as the multimedia one, named M-CAN bus, from the trusted ones, such as B-CAN *(Body-CAN)* and others. Partitions are connected through a Central Gateway (CGW) that forwards allowed CAN frames from a partition to another (Fig. 1).

3.1 Head unit

The HU is connected to the Multimedia CAN-bus partitions of the vehicle. It is connected to the Internet through a smartphone, as hotspot mode, or 3G, 4G dongle that generates a Wi-Fi network in which the Head Unit is connected.

Physical buttons are present in the HU to quickly activate other functionalities, such as, *maps* and *settings*. The OS is based on Android version 4.2.2. Users cannot access the fully OS and only a set of pre-installed apps are available to provide several built-in functionalities for a good driver experience. Other apps related to OS settings, HU managements and other functionalities are not accessible by the users. We distinguish these two spaces as *UI Front-end* and *UI Back-end*.

3.1.1 UI front-end

It is the space in which the user is allowed to move on within the HU. It contains, for instance, the navigation app and the radio app. In addition, there are also other apps needed for the HU configuration or just to have information about the software version as well as the possibility to upgrade the HU software.

3.1.2 UI back-end

It is the space in which the user is prevented to access. However, in the Internet several guides show on how users can access the full Android OS exploiting a hidden menu [34,35]. The most interesting hidden menu is the *Engineering Menu* and it is referred as "eng menu" throughout this paper. When exploring the UI Back-end through the "eng menu", we observed that other apps are installed in the HU, such as the Browser app, and are not available in the *UI Front-end*.

3.1.3 Third-party applications

The HU by default does not allow users to install third-party applications. Users are limited to use only those apps that are in the UI Front-end and UI Back-end in case of access[1] through the "eng menu". However, googling online it is possible to find several guides that allow users to access the full Android operating system from a hidden menu are available. Access to the eng menu may change from vehicle to vehicle model and it also changes from HU software version. The access to older version is quite simple and become more difficult up to the last available software version. Thus, by accessing the "eng menu", a user is able to install third-party

[1] Accessing the UI Back-end is unauthorised action and only dealers or authorised users should access it. Note that any unauthorised activity may invalid the vehicle's warranty.

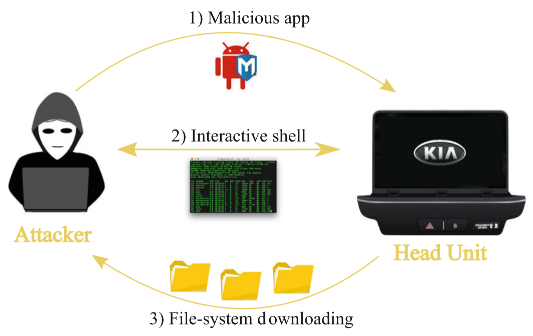


Fig. 2 Flow to extract the Firmware from the HU

applications also from unknown sources. For instance, using the Browser app, we discovered that apps can be installed via a http server web inserting a string similar to:

```
server-ip/foo.apk
```

Where the “foo.apk” is the installation bundle file.

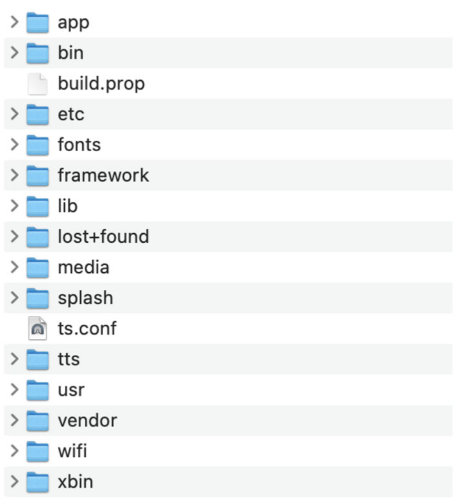
4 Head unit reverse engineering

The methodology we followed to identify all vulnerabilities consists of three steps, one for each element of the HU that we studied. First, we worked on the HU file system to understand how it is organized and we extrapolated the Android Framework files from the HU. Second, we reverse engineered the HU apps, both the ones belonging to the UI Front-end and UI Back-end to understand how the HU software works and communicates on the CAN bus. Third, we developed a supporting app to run a brute force attack on the HU to find valid “id” and “payload” to control the HU and send CAN bus frames into the M-bus of the vehicle.

4.1 Step 1: HU’s file system, framework and system Apps

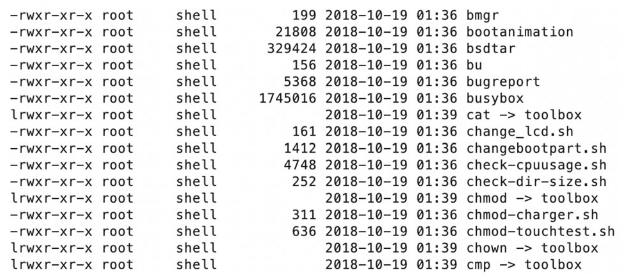
To access the HU file system, we need a system shell to navigate among system files and folders. To achieve this step, we created an Android app[2] that embeds a malicious payload formed by a reverse shell that is spawned once the app is installed, see Fig. 2. After installing this app on the HU through the “eng menu”, we were able to remotely access the HU and navigate the file-system using a Meterpreter [36] reverse-shell.

Figure 3, we show an excerpt of the fine system that we are able to explore with the remote shell. Exploiting the shell, we got into the HU file system and we noticed that it resembles the file system of an Android-based device. The access we got did not have root privileges so we were able to open only



```
> app
> bin
  build.prop
> etc
> fonts
> framework
> lib
> lost+found
> media
> splash
  ts.conf
> tts
> usr
> vendor
> wifi
> xbin
```

Fig. 3 File system excerpt



```
-rwxr-xr-x root     shell        199 2018-10-19 01:36 bmgr
-rwxr-xr-x root     shell      21808 2018-10-19 01:36 bootanimation
-rwxr-xr-x root     shell     329424 2018-10-19 01:36 bsdtar
-rwxr-xr-x root     shell        156 2018-10-19 01:36 bu
-rwxr-xr-x root     shell       5368 2018-10-19 01:36 bugreport
-rwxr-xr-x root     shell    1745016 2018-10-19 01:36 busybox
lrwxr-xr-x root     shell            2018-10-19 01:39 cat -> toolbox
-rwxr-xr-x root     shell        161 2018-10-19 01:36 change_lcd.sh
-rwxr-xr-x root     shell       1412 2018-10-19 01:36 changebootpart.sh
-rwxr-xr-x root     shell       4748 2018-10-19 01:36 check-cpuusage.sh
-rwxr-xr-x root     shell        252 2018-10-19 01:36 check-dir-size.sh
lrwxr-xr-x root     shell            2018-10-19 01:39 chmod -> toolbox
-rwxr-xr-x root     shell        311 2018-10-19 01:36 chmod-charger.sh
-rwxr-xr-x root     shell        636 2018-10-19 01:36 chmod-touchtest.sh
lrwxr-xr-x root     shell            2018-10-19 01:39 chown -> toolbox
lrwxr-xr-x root     shell            2018-10-19 01:39 cmp -> toolbox
```

Fig. 4 Excerpt of /system/bin folder

granted folders and read/write files that do not require root privileges.

Under the root “/” there is the “system” folder in which Android OS files are present. This folder usually contains pre-installed apps and the framework files. In addition, the “bin” folder contains the majority of executable files that can be run into the system. Some files in this folder are already in a binary format, i.e., they can be run as system application, while others are “sh” script files in which we can observe the source code.

Figure 4 shows part of files contained in the “bin” folder. The majority of files available in “/system/bin” requires privileged access to be run. For instance, the applets contained in the *busybox* [37] executable cannot be run without privileged access. This means that a not root user cannot execute commands like chmod, chown and other commands available into busybox. On the other sice, there exist files can be executed without privileged access, this because they have “read” and “execute” permission for the *others* scope users.

The Framework contains the Android software libraries that allows the HU to properly work. It is located under the folder /system/framework and we needed it to reserve engineering system apps and develop an app that uses system libraries to work. So, to download the Framework files and

[2] Appendix B illustrates how we created the app.

the system apps from the HU, we leveraged the Meterpreter commands:

```
download -R /system/framework
download -R /system/bin
```

In particular, each system app is composed by two files, one with .apk extension and one ending as .odex file [38]. This indicates that installed apps are odex optimized executable files to allow the Android OS to quickly load the apps when the HU boots up. However, apps split into .apk and .odex files make the reverse engineering phase more complicated since each app needs to be first *deodexed*, i.e., we need to get a single .apk file with all application files inside. To perform apps "deodexing", we exploited the tool [39] that in combination with the framework files allowed us to get the full .apk files.

4.2 Step 2: Digging into system apps

Working on the HU version SOP.003.30.18.0703, we decompiled[3] 98 system apps ranging from core apps to command the HU to apps managing, for instance, the navigation, climate, and so on. Then, we dug into the source code to understand its main functionalities and any kind of detail related to CAN bus access. All decompiled system apps generated 59093 files, whose 22209 are JAVA files and are 7979 XML files. The total amount of decompiled code lines[4] is 2654557.

Seen the huge amount of decompiled files, we started a *fuzzy search* by looking for any string reference to *canbus*, *send*, *message* and so on. By means of the fuzzy search, we were able to go more in depth and obtain important findings and understand some details of the system apps working method.

4.2.1 Inter-process communication

The first thing we discovered is that apps were able to communicate among them using the so-called mechanism *Inter-Process Communication* (IPC). When an app is executed, it runs in a sort of sandbox and the IPC mechanism is exploited to exchange data with other apps. Without going into details on this, Android has its own IPC working method [40] that allows apps to exchange data. This is done through JAVA classes that behave as *Interfaces* and expose methods that can be called. These methods, when invoked, provide information related to the HU. For instance, a non-system app may be written and developed to invoke an interface by

[3] For the decompiling operation we used the http://www.javadecompilers.com/apk third-party online tool.

[4] All these values are calculated using the "cloc" command line tool. https://github.com/AlDanial/cloc

```
import com.lge.ivi.message.AskAudioBeepInfo;
import com.lge.ivi.message.AskAudioBeepOffset;
import com.lge.ivi.message.AskAudioInfo;
import com.lge.ivi.message.AskAudioLastMemory;
import com.lge.ivi.message.AskAudioMuteInfo;
import com.lge.ivi.message.AskAudioSourceScaleInfo;
import com.lge.ivi.message.AskBackLightValue;
import com.lge.ivi.message.AskBootInfo1;
import com.lge.ivi.message.AskBootInfo2;
import com.lge.ivi.message.AskBootingType;
import com.lge.ivi.message.AskCanData;
import com.lge.ivi.message.AskCanDbVersion;
```

Fig. 5 Import classes of eIVIMessage

exploiting a system library extracted as explained in Section 4.1.

Then, we discovered that the Framework library *com.lge.ivi.jar* contains several interesting interfaces that can be invoked to obtain vehicle's information. For instance, it is possible to get vehicle details by invoking methods of the *ConfigurationManager* class: we were able to get the car type by means of the method .getCarType(). In addition, we can get the car buyer by invoking .getBuyer(). Other details can be found also in the *CarInfoManager* class to retrieve the vehicle speed with the method .getCarSpeed().

4.2.2 IVIMessages

When digging into apps working mechanism, we discovered the vulnerability id "1" listed in Table 1. The vulnerability is represented by the JAVA class called *eIVIMessage* and it shows the full list of messages that apps can use to control HU functionalities and send/receives CAN bus frames.

By exploring the source code, we saw that the *eIVIMessage* class imports other classes categorized under a "message" topology. An extract of these classes is illustrated in Fig. 5.

All these classes represent objects that can be shared among apps when IPC communications are used. By opening each single message, we noticed a common structure, see Table 2, that resembles an interface written with the *Android Interface Definition Language* (AIDL). Taking as an example the "AskCanData" message class, we focused on the *parse* method showed in Listing 1.

Listing 1 Parse method comparison

```
// Parse method of AskCanData class
public boolean parse(byte[] b, int len) {
if (len != 2) { Log.e(TAG, "Invalid data size. Expects 2, but " + b.length + " is
        given");
return false;   }
this.canId = (((b[0] >>> 0) & 255) << 8) | (((b[1] >>> 0) & 255) << 0);
return true;}
// Parse method of CanCluHuE00 class
public boolean parse(byte[] bArr, int i) {
if (i != 8) {Log.e("CanCluHuE00", "Invalid data size. Expects 8, but " +
        bArr.length + " is given");
return false;}
```

Table 2 Most relevant methods of the IVIMessage class

Methods			
Return Type	Name	Input parameters	Description
int	id()	//	It specifies a message "id"
boolean	isCan()	//	If true, the message is a CAN message otherwise not
boolean	parse()	byte[] b, int len	It parses a message from its raw hex format to a high-level format, such as Integer. If the parsing is correct, then a true values it outputted, otherwise false
byte[]	compose()	//	It takes high-level format values and converts them into hex values.
String	toString()	//	It displays as textual value the hex value of each variable contained in the class

```
this.sysCluVer = (((bArr[0] >>> 0) & 255) << 8) | (((bArr[1] >>> 0) & 255) <<
    0);
this.cluDateInfoReq = (bArr[2] >>> 2) & 3;
this.cluClockInfoReq = (bArr[2] >>> 0) & 3;
return true;}
```

The *parse* method of "AskCanData" message class works on a byte array of two elements, while that one in "CanCluHuE00" class expects a byte array of 8 elements.

We understood that the parse and compose methods recall the working fashion of two well-known functions run by *Electronic Control Units* (ECU) when sending and receiving CAN frames. These two functions are the *encode* and the *decode* ones. The encode function is in charge of taking high-level values, such as integer, decimal, and covert them into hex values. The decode function makes the opposite task, namely it takes hex values, which represent the CAN format, and turns those values into a high-level format easy to be read and understood.

Going ahead to analyse the *eIVIMessage* class, we achieved the core of the class formed by the definition of a group of constants by means of an *enum* JAVA class. This one recalls the "imports" seen before. Appendix A shows an extracts of the constants.

For each element of this structure, we associate the following meaning: the first value represents the "id" bound to that message, in Appendix A, "id" equal to 273 identifies the *AUDIO_SUB_CH_CONTROL* message. The second and third field represent the message name and the corresponding JAVA class filename. Last value identifies the type of message among the following:

Listing 2 Message Types

```
MICOM = new eMessageType("MICOM", 0);
CAN = new eMessageType("CAN", 1);
MPDT = new eMessageType("MPDT", 2);
TYPE_ERROR = new eMessageType("TYPE_ERROR", 3);
```

The message type in Listing 2 shows the different protocols that can be managed inside the Head Unit. From our study, we saw that only MICOM and CAN messages are used.

By analysing the eIVIMessage class, we found 566 messages. Among these ones, 302 are CAN type while the others are MICOM type. From our reverse engineering process, we observed that each IVIMessage class defines the message semantic: the set of signals composing the payload and the corresponding variable to store the parsed values, e.g., *sysCluVer*, *cluDateInfoReq*. *cluClockInfoReq* variables of the "CanCluHuE00" class.

Note that from automotive security perspective, vulnerability id "1" represents a relevant leak of sensitive data since messages semantic are secrets information known only by the car manufacturers and ECUs providers.

4.2.3 Micomd binary file

By digging into the reversed apps, we came across the file named *AutoTestService.java*. This JAVA class contains more than 3000 lines of code and, in particular, we focused on the method *doCmdMuteToggle()*. The method has an interesting line of code that we found (Listing 3).

Listing 3 doCmdMuteToggle sendMicomMsg.

```
private void doCmdMuteToggle() {
...
sendMicomMsg("8351 04"); }
```

The same *AutoTestService.java* JAVA file contains the method *sendMicomMsg* (Listing 4).

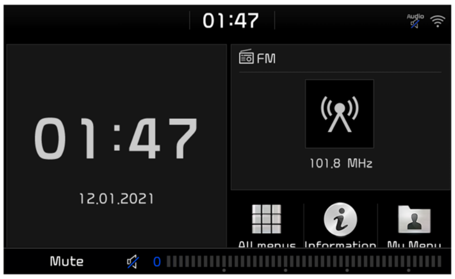
Listing 4 sendMicomMsg method content

```
private boolean sendMicomMsg(String msg) {
try {Process process = Runtime.getRuntime().exec("micomd –c inject " + msg);
.    ... }
```

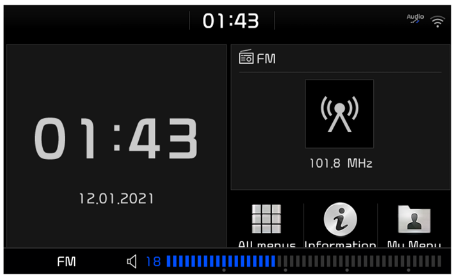
This method invokes an executable binary called `micomd` with option `-c` and parameter `inject`. Then, to the command the string `msg` is passed. So, if we consider the method "doCmdMuteToggle()", the resulting system command is:

```
micomd -c inject 8351 04
```

Fig. 6 Micomd command result



(a) Muting the Head Unit



(b) Unmuting the Head Unit

The above command indicates a send operation whose specific content is "8351 04". Next to the *sendMicomMsg* method, we found also the method in Listing 5.

Listing 5 sendMicomMsgOutgoing method content

```
private boolean sendMicomMsgOutgoing(String msg) {
try {Process process = Runtime.getRuntime().exec("micomd –c
        inject–outgoing " + msg);
...}
```

This method is very similar to the *sendMicomMsg* and again the binary "micomd" is considered but with the parameter "inject-outgoing". This indicates that the binary invoked with the option "inject-outgoing" may send a value somewhere outside the unit.

What we observed is that, other methods invoke the "sendMicomMsgOutgoing" one and this is the case, for instance, of the method listed in Listing 6.

Listing 6 doCmdCarAccOn method content

```
private void doCmdCarAccOn() {
Log.d(AutoTestService.LOG_AUTO_TOOL, "doCmdCarAccOn");
sendMicomMsgOutgoing("0170 01");}
```

At this phase, we discovered vulnerability id "2" represented by the "micomd" binary file under the /system/bin folder. To make this vulnerability concrete is the possibility to arbitrary exploit the binary even not being root. So, by executing the command line "micomd -c inject 8351 04" in the HU shell, we were able to mute and unmute the HU volume, see Fig. 6a and b.

4.2.4 MICOM messages semantic

Our goal is to choose an "id" among the ones listed in the eIVIMessage JAVA file, such as 8351, and payload, like 04 that is valid: it triggers unexpected functionalities into the HU. We noticed that the "id" 8351 is represented as hexadecimal format and to obtain the proper JAVA object listed within the eIVIMessage file, we had to convert it into the decimal format and we get:

$$(8351)_{16} = (33617)_{10} \quad (1)$$

```
public static final class RemoteKey {
    public static final int BT_CALL = 9;
    public static final int BT_CALL_LONG = 25;
    public static final int BT_END = 8;
    public static final int BT_END_LONG = 24;
    public static final int LONG_KEY_DISARM = 32;
    public static final int MODE = 3;
    public static final int MODE_LONG = 19;
    public static final int MUTE = 4;
    public static final int MUTE_LONG = 20;
    public static final int SEEK_DN_LONG = 18;
    public static final int SEEK_DN_LONG_3_8S = 34;
    public static final int SEEK_DOWN = 2;
    public static final int SEEK_UP = 1;
    public static final int SEEK_UP_LONG = 17;
    public static final int SEEK_UP_LONG_3_8S = 33;
    public static final int VOICE_RECOG = 7;
    public static final int VR_LONG = 23;
```

Fig. 7 RemoteKey class definition

The decimal value of 8351 is 33617 and it corresponds to the *ETC_SW_REMOTE_KEY_EVENT* JAVA object (Appendix A line 27).

By observing the corresponding *ETC_SW_REMOTE_KEY_EVENT* JAVA source file, we noticed that the method *isCan()* gives as output false, that the method *parse()* expects an input of 1 byte length, and that the payload is built by the method *compose* as:

```
this.remoteKey & 255
```

From the above line of code, we understood that the payload takes the value of *remoteKey* and makes an and logical operation with the value 255. *remoteKey* is defined as static class within the *ETC_SW_REMOTE_KEY_EVENT* JAVA file.

We noticed that *remoteKey* may assume different values among those in Fig. 7. For instance, the value "4" corresponds to *mute* and, indeed, it is the value that triggers the "mute" volume action on the HU. So, we were able to rebuild the following command:

```
micomd -c inject 8351 04
```

where both "id" and "payload' are converted from decimal to hexadecimal values.

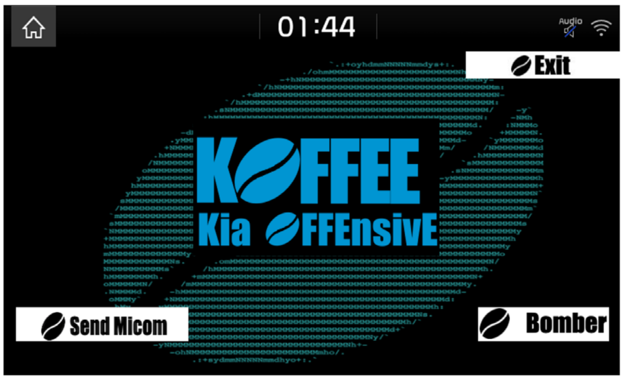


Fig. 8 KOFFEE main window

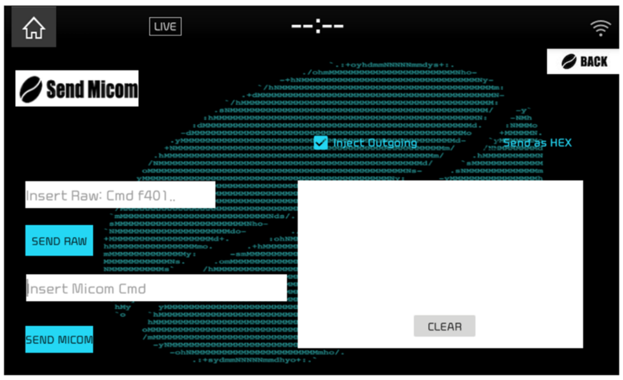


Fig. 9 KOFFEE Send Micom window

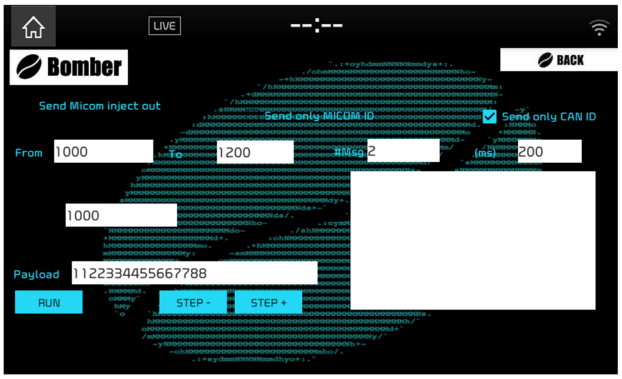


Fig. 10 KOFFEE Bomber window

To summarise, it is possible to control the HU by exploiting the "micomd" binary with an hex "id" and a "payload". While the "id" is composed by a single parameter, the "payload" may have a different length greater than 1, expressing at least the presence of one byte. Then, the "id" can be obtained converting the integer "id" listed in the eIVIMessage JAVA file. Instead, the "payload" can be obtained observing the corresponding source code of the MICOM message. From this file, we also understood the "payload" length and how to *compose* and *parse* a message payload.

In the following, we show how vulnerability "id 2" can be exploited to find chosen "id" and "payload" to control the HU functionalities (vuln. "id 3") and inject crafted CAN bus frames into the M-bus (vuln. "id 4").

4.3 Step 3: KOFFEE

Our next goal is to discover the semantic message available into the *eIVIMessage* JAVA file, in particular, those "ids" and "payloads" that regulate the HU functionalities and modify the behaviour of ECUs in the M-CAN bus.

Brute-forcing the "micomd" binary is an onerous task to be manually performed. Thus, we wrote an Android app, named *Kia OFFensivE* (KOFFEE), that iterates among all MICOM message "ids". When executing the KOFFEE app into the HU, the main window appears (Fig. 8) giving the opportunity to choose between two options: *Send Micom* and *Bomber*. The first option allows us to manually trigger the "micomd" binary, while the second one performs the brute-force attack exploiting the "micomd" binary, i.e., vulnerability id "2".

4.3.1 Send MICOM message

In Fig. 9 we illustrate the *Send Micom* window of the KOFFEE app. In the following we describe its main functionalities:

- *Send RAW* button: it arbitrary sends a MICOM message invoking the *sendRaw* method of the *iIVIMessageService* class contained in the com.lge.ivi.jar Framework class file. The *sendRaw* function takes as input only an "id" (as integer) and "payload" (as byte array).
- *Send MICOM* button: it sends a MICOM message written in the text box.
- *Inject Outgoing* check-box: if ticked, the message is sent as *micomd -c inject-outgoing*. Otherwise, *micomd -c inject*.
- *Send as HEX*: it takes the input "payload" as hex format when *Send RAW* is clicked. Otherwise, the "payload" inserted in the edit-box must be as byte format. In both case the "payload" is then converted to byte array.
- *Insert Raw* edit-box: the raw command to send.
- *Insert Micom* edit-box: the MICOM command to send.

4.3.2 Bomber MICOM messages

The *Bomber* window appears as illustrated in Fig. 10 and allows us to iterate among all MICOM and CAN message types available in the eIVIMessage file. This window is made by the following main functionalities:

Table 3 Subset of MICOM messages discovered using the KOFFEE app

Micomd command			
Parameter	Id	Payload	Description
inject	8350	07 01	It shows the radio app as foreground. This action can be reproduced also with 07 03 payload
inject	8350	0C 01	It shows the navigation app as foreground. This action can be reproduced also with 0C 03 payload
inject	8350	0E 01	It shows the settings app as foreground. This action can be reproduced also with 0E 03 payload
inject	8350	0D 03	It shows the navigation app as foreground. This action can be reproduced also with 0C 03 payload
inject	8350	20 01	It changes the radio station to the next available
inject	8350	21 01	It changes the radio station to the previous available
inject	8351	04	It mutes/unmutes the Head Unit volume
inject	8351	07	It enables vocals command
inject	8353	03 01	It enables the parking camera on the Head Unit
inject-outgoing	112	F4	It increase the volume of the Head Unit
inject-outgoing	112	F0	It increases the Head Unit volume at maximum

- *From* edit-box: it indicates the starting "id" to brute-force. In Fig. 10, it is indicated as 1000.
- *To* edit-box: it indicates the last "id" to brute-force. In Fig. 10, it is indicated as 1200.
- *#Msg* edit-box: it indicates the number of times to send the same message. For instance, the message 1000 01 01 01 01 is sent twice if the value in this edit-box is 2. We put this option to set how many times a message can be sent to be sure that a message is sent and also processed by the HU.
- *(ms)* edit-box: how much time passes from one message to another, e.g., 200ms. If this option is not set or the value is to low, e.g., < 50, the HU may be overloaded and may not properly respond.
- *Current* edit-box: it indicates the sent "id" during the brute-force action. The value ranges between "From" and "To".
- *Payload* edit-box: it contains the payload that is sent when the "micomd" binary executable is triggered.
- *Send MICOM inject out* check-box: if ticked, the message is sent as *micomd -c inject-outgoing*. Otherwise, as *micomd -c inject*.
- *Send only MICOM id* check-box: when the brute-force is active, the "ids" sent in the "From"-"To" window are only MICOM type. This is a feature that we developed to accelerate the brute-force operation by avoiding CAN "ids" and other not used "ids".
- *Send only CAN id* check-box: when the brute-force is active, the "ids" sent in the "From"-"To" window are only CAN type. This is a feature that we developed to accelerate the brute-force operation by avoiding MICOM "ids" and other not used "ids".

4.3.3 Injecting Micomd commands

Table 3 shows a subset of "micomd" commands that we found exploiting our KOFFEE app. In particular, we identified the "id" and the "payload" needed to trigger an action.

Some "ids" were first identified using a simple chosen payload, such as 01 01 01 01. Then, once we observed that an "id" triggers an action we decided to explore that "id" by testing other "payloads". However, a more accurate search of payloads can be done by analysing the source code of a specific "id" message. For instance, if we consider "id" 8350, it converted to integer is equal to 33616. This "id" corresponds to *MICOM_KEY_EVENT* message on the eIVIMessage source file. Then, if we see the related *MicomKeyEvent* JAVA file, we know more about this message. In this specific case, this message has length two bytes (from *parse()* method), and two are the values that can be set through this message, and they are: *KEY* and *INFO*.

During the brute-forcing attack, we noticed two relevant behaviours. In the first case, even manually forging a payload, we were not always able to trigger the expected functionalities. This behaviour may be due to the car model that we tested. In fact, the Head Unit software, and in particular, its MICOM software version, may differently behave from vehicle to vehicle: not all observed "payloads" may trigger an action on our HU. This may represent a limitation but also opens an important result of our research: the impact of our findings may be larger and cover other vehicles that we have not tested yet. To this purpose, we know that the firmware we analysed is shared with other Kia Motors vehicles and with other cars of another car manufacturer, which is part of the same South Korean industrial group, that all together represent the 8.2%, 8.6% and 10.3% percentage of the best

Table 4 CAN ids generated from the HU and identities through CAN hacker as MITM

CAN Frame		
ID	DLC (byte)	Description
03E	8	Speed limit signals
0FB	8	GPS timestamp
03F	8	Navigation info e.g., Country and Motor
1E7	8	Navigation route
115	8	Navigation info
122	8	FM radio frequency
123	8	Route signals
173	8	Language info, navigation
197	8	As id 173
1E5	8	Speed limit signal

European selling car manufacturers in 2019, 2020 and 2021 years.

The second behaviour that we noticed is related to the "payload" length. If we use smaller or bigger payload compared to what the parse() method says, for instance two bytes, the "micomd" command is processed without any check. Our opinion is that when we execute the "micomd" command, this is not processed by the corresponding JAVA message class but "id" and "payload" are directly injected into a system socket located at the "/dev/socket/micomd" position as stated in Listing 7.

Listing 7 Micomd socket connection

```
public synchronized void connect() { ...
this.f769D.connect(new LocalSocketAddress("micomd",
        Namespace.RESERVED));
 ... }
```

This missing check allowed us to discover payloads although they were not properly forged: if the correct "payload" is 01 01 and we sent the command with a longer "payload", e.g., 01 01 01 01, it is not discarded, but is processed since the first two bytes are correct.

4.3.4 Head Unit CAN bus frames

As we said in Section 3, the HU is a node that belongs to the M-CAN bus. The HU is able to send CAN frames into the M-bus as well as receive frames from other nodes. To know which are the CAN bus frames sent and received by the HU, we decided to sniff the HU CAN bus wires performing a man-in-the-middle (MITM) attack as we show in Fig. 11. We were able to connect a sniffer device, we used *CAN hacker* [41] together with its *Car BUS analyzer* software [42] to sniff all CAN traffic from that bus. We observed that the M-CAN bus works at 100kbps, and the sniffed frames are reported in Table 4. In particular, for each CAN frame we report the "ID', ' "DLC", i.e., the payload length, and its "description".

4.3.5 Injecting CAN frames

We leveraged our KOFFEE app to discover CAN frames from those ones available in the *eIVIMessage* JAVA file that can be injected exploiting the ACE vulnerability ("id 2"). The brute-forcing attack only on CAN message type can be done by flagging the box *Send only CAN ID* in the *Bomber* window (Fig. 10).

We found 9 messages that can be injected as CAN frames from the HU using the "micomd" binary file (Table 5). In addition, we noticed that CAN frame were generated only if the "micomd" is executed with the "inject-outgoing" parameter.

Through the injection of these 9 CAN "ids", we are able to control some functionalities of the ECUs that reside on the M-CAN bus. For instance, some CAN "ids" allow us to alter the GPS and Navigation information collected by the GPS component and provided to the driver by the User Interface of the Instrument Cluster.

Note that, we do not directly compromise the GPS component of the HU, but we act on the GPS information integrity.

Semi-controlled CAN frames Table 5 shows the CAN "ids" that we were able to reverse and inject into the M-CAN bus. Since the CAN bus protocol foresees that for a single "id" it is possible to carry out more signals by working on its "payload", our next step is to forge a valid payload for each chosen "id". In an ideal situation if we send a 8 bytes payload in a command like micomd -c inject-outgoing 4D1 11 22 33 44 55 66 77 88, we expect the same 11 22 33 44 55 66 77 88 payload is transmitted on the M-CAN bus. However, this matching does not always occur. We believe that this behaviour is controlled by the MICOM firmware that is in charge of converting MICOM message to CAN frames. So, not having access to the firmware, we decided to discover through the KOFFEE app which are the payload that we were able to create.

For each MICOM message, Table 6 reports the bytes that we are able to control through the "micomd" binary: last column shows the "✓" symbol to the payload byte position that we are able to control, the "✗" symbol otherwise. For instance, this means that if we send the MICOM message 4D1 11 22 33 44 55 66 77 88 from the HU, the CAN frame 115 11 22 33 44 55 66 77 88 is injected into M-CAN bus as. On the other side, if we send the MICOM message 4D3 11 22 33 44 55 66 77 88 from the HU, an example of the resulting CAN frame is 197 11 AA BB CC 55 DD EE FF meaning that we can control only the first and the fifth byte. Moreover, we noticed that the "ids" that we are able to arbitrary inject in the M-CAN bus are only received by ECUs that belong to

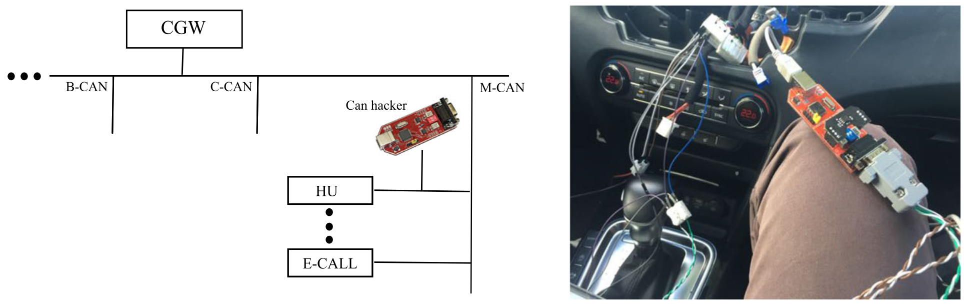


Fig. 11 CAN hacker connected as node into the M-CAN bus

Table 5 eIVIMessages related to CAN ids discovered through brute-force

CAN Frame				
eIVIMessage Name	MICOM (int)	MICOM (HEX)	CAN ID (HEX)	Description
HU_TMU_E_02	1026	402	0FB	GPS timestamp
CAN_HU_NAVI_E_01_EVT	1053	41D	03F	Navigation info, e.g. country, town and others
CAN_CLU_P_00_EVT	1232	4D0	1EF	Navigation trajectory.
CAN_CLU_PE_02_EVT	1233	4D1	115	Navigation data displayed into the Instrument Cluster
CAN_CLU_PE_05_EVT	1235	4D3	197	Instrument Cluster language
CAN_CLU_PE_11_EVT	1236	4D4	1E5	Navigation speed limit
CAN_CLU_PE_06_EVT	1238	4D6	122	Radio frequency
CAN_CLU_PE_07_EVT	1242	4DA	123	Route signals displayed into the Instrument Cluster
CAN_CLU_NAVI_E_00_EVT	1243	4DB	03E	Speed limits displayed into the Instrument Cluster and HU

Table 6 Byte payload that we can control with the "micomd" binary when CAN frames are sent

CAN frames			
MICOM (int)	MICOM (hex)	CAN ID (hex)	Payload Position (8 byte)
1026	402	0FB	✓✓ ✓✓ ✓✓ ✗✗ ✗✗ ✗✗ ✗✗ ✗✗
1053	41D	03F	✓✓ ✓✓ ✓✓ ✗✗ ✗✗ ✗✗ ✗✗ ✗✗
1232	4D0	1E7	✓✓ ✓✓ ✓✓ ✓✓ ✓✓ ✓✓ ✓✓ ✓✓
1233	4D1	115	✓✓ ✓✓ ✓✓ ✓✓ ✓✓ ✓✓ ✓✓ ✗✓
1235	4D3	173	✓✓ ✗✗ ✗✗ ✗✗ ✗✗ ✗✗ ✗✗ ✗✗
		197	✓✓ ✓✓ ✓✓ ✓✓ ✓✓ ✗✗ ✗✗ ✗✗
1236	4D4	122	✓✓ ✓✓ ✓✓ ✓✓ ✓✓ ✓✓ ✓✓ ✓✓
1242	4DA	123	✗✗ ✗✗ ✗✗ ✓✓ ✗✗ ✗✗ ✗✗ ✓✓
1243	4DB	03E	✓✓ ✓✓ ✓✓ ✓✓ ✓✓ ✓✓ ✗✗ ✗✗

this partition. In fact, we noticed that the Central Gateway (CGW) properly isolates this partition from the others, such as P-CAN, C-CAN and so on.

5 Weaponizing KOFFEE

In this section, we introduce the KOFFEE Module [43,44] as the exploit implementation of the local *Arbitrary Code Execution* vulnerability on the "micomd" binary. To ease the attack, we developed the KOFFEE Module for Metasploit [36] that with a set of actions that allow us to select the commands to execute into the HU.

Over an active Metasploit session with the HU, we can load our Module with the command use post/android/local/Koffee, then set the session with set session "session_number" and, finally, execute one of the available actions on the KOFFEE Module, i.e., camera_reverse_on.

Hereafter, we illustrate a subset of actions available in the Module that affect the Head Unit and the vehicle instrument cluster:

- CAMERA_REVERSE_OFF: It hides the parking camera video stream.
- CAMERA_REVERSE_ON: It shows the parking camera video stream.

- CLUSTER_CHANGE_LANGUAGE: It changes the cluster language.
- CLUSTER_RADIO_INFO: It shows radio info in the instrument cluster.
- CLUSTER_RANDOM_NAVIGATION: It shows navigation signals in the instrument cluster.
- CLUSTER_ROUNDABOUT_FARAWAY: It shows a round about signal with variable distance in the instrument cluster.
- CLUSTER_SPEED_LIMIT: It changes the speed limit shown in the instrument cluster.
- INJECT_CUSTOM: It injects custom micom payloads.
- LOW_FUEL_WARNING: It pops up a low fuel message on the head .
- MAX_RADIO_VOLUME: It sets the radio volume to the max.

In particular, with the action “INJECT_CUSTOM”, we can input a custom MICOM message to send with an “id” and “payload”.

6 Showcasing an end-to-end attack: vehicle’s remote exploitation

This section illustrates how an attacker can take i) control of the HU and ii) use the KOFFEE Module to control the HU and inject CAN frames into the M-CAN bus, see Fig. 12.

To take control of the HU, we designed and developed an Android app that acts as a Trojan-horse in the HU. To use an appealing Trojan-horse app for the victim, we downloaded from the Market Place a genuine app that shows the gas stations close to vehicles. Then, we generated a malicious payload using the “msfvenom” (Appendix B) utility of Meterpreter and backdoored the genuine app with the malicious payload. Thus, when the victim executes the app, the backdoor is opened and a reverse-shell starts runs a connection from the vehicle to the attacker Meterpreter session.

In Appendix B we show how to create an app with a malicious payload using msfvenom. Then, we extracted the payload form the smali-code of the app and we injected it in the genuine Android app. In particular, to backdoor the genuine app with executed the following steps:

1. We decompiled the genuine app using the reverse engineering Android apps tool called “Apktool” [45]. It is used for back-smaling, i.e., to pass from the apk-code to the smali-code and, reversely, smaling, i.e., from smali-code to apk-code, Android applications.
2. We injected the malicious payload into the smali and modified the apk-manifest to add permissions (if needed).
3. We built the modified app using apktool.
4. We signed the modified built app.
5. We make it available for download, for instance in a third-party market or exploiting a social engineering campaign.

When the victim downloads and executes the app on the HU, the malicious payload is run and a reverse Meterpreter[5] shell is spawned. From now on, the attacker establishes a remote connection with the HU of the vehicle and the HU starts a connection towards the attacker’s computer.

To achieve the second part of the attack, the attacker leverages the KOFFEE Module presented in Section 5. In particular, over the existing connection the attacker can run the KOFFEE Module and exploit the ACE vulnerability on the “micomd” binary file and remotely attack the vehicle using the available actions.

7 Lessons learned

As consequence of our analysis we now provide the answers to the three research questions we listed in Section 1.1.

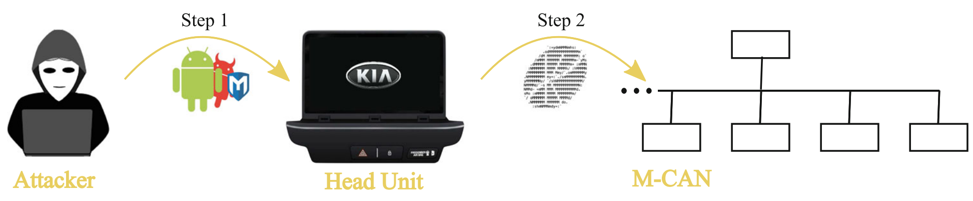
RQ1: Which kind of vulnerabilities may affect a vehicle? The automotive literature and the attacks performed on vehicles already give an overview on which are the most relevant vulnerabilities in this domain. Trend Micro Research [46] studied four relevant automotive attacks, such as the Jeephack of 2015, the TESLA hacks of 2016 and 2017, and the BMW hack of 2018. They observed that the generic attack chain is that these attacks initially start from the vehicle’s Head Unit via WiFi or the mobile network, then exploit a weak point there.

On our side, we started our activity knowing that the CAN bus network is not secure by design: messages are exchanged in clear. However, to exploit this intra-vehicle network vulnerability we need to break other vehicle’s components, e.g., the Head Unit, that allowed us to access to the CAN bus network. Following the attack chain depicted on the Trend Micro Research, we decided to use the HU as entry point of our attack. So, we reverse engineered it to identify the presented vulnerabilities that allowed us to control HU functionalities and inject frames into the M-CAN bus. In particular, referring to Table 1, we identified four vulnerable points. The list of “id” and “payloads” (Vulnerability “id 1”) and Arbitrary Code Execution (Vulnerability “id 2”) allow attackers to compromise the HU and lead to discover the other two vulnerabilities (“id 3” and “id 4”).

RQ2: Which is the attack impact on the vehicle functionalities? And on the driver’s security/safety/privacy? The discovered vulnerabilities can impact on both vehicle functionalities, by allowing the attacker to alter the behaviour

[5] This is the shell that is created when a new Metasploit session is established towards the victim

Fig. 12 KOFFEE workflow



of the vehicle sending crafted messages, and on the security, safety and privacy of the driver, since the HU is able to store data belonging to the driver and, much more risky, can send erroneous information. It is important to note that the actions we trigger with the injection of our crafted messages are not sent directly to safety critical parts of the vehicle. However, altering the HU behaviour, for instance, by suddenly turning the radio up to maximum volume, or altering the user interface within the instruction cluster, may lead to a possible unsafe reaction of the driver.

Hence, we consider vulnerability “id 1” and “id 2” as high impact because knowing the id and payload of messages and being able to run a code that sends crafted message, the attacker may alter the functionalities of the vehicle and interact with the driver. Vulnerability “id 3” and “id 4” are derived from the first two ones. In fact, by exploiting the vulnerability “id 1”, new messages can be crafted that can be send into the in-vehicle network by exploiting vulnerability “id 2”. Hence, as we describe in Section 6, these vulnerabilities can be exploited by developing a module that allows an attacker to remotely control the HU and inject CAN frame.

RQ3: How strong are the current mechanisms to prevent cybersecurity attacks?
The cybersecurity standard for automotive came out in August 2021 and it is the ISO/SAE FDIS 21434 [11] to draw the guidelines for the cybersecurity of onboard communication. The United Nations Economic Commission for Europe [47] this year has given the directive for over 55 countries regarding the cybersecurity management of vehicle to make vehicles more secure starting from 2022. However, the current state of the art of security solution within vehicles is represented by the internal network separation that aims to divide untrusted zones, such as the multimedia one, from the trusted ones. Partitions are connected through a Central Gateway (CGW) that forwards allowed CAN frames from a partition to another. As we show with our exploit, this solution prevents an attacker to directly interacts with potentially safety-relevant functionalities via the HU. However, the vehicle cannot be considered as a secure system, since, as we discussed in the answer to RQ2, the attack we showed here may impact on the Instrument Cluster (IC) of the vehicle and show erroneous pieces of information to the driver and may lead to driver disorientation since she has not experience on the expected behaviour of the IC.

8 Conclusion

In this paper, we presented a vulnerabilities assessment, through reverse-engineering analysis, of the Kia CEED Head Unit with software version CD.EUR.SOP.003.30.180703. STD_M. We showed how to access the intra-vehicle network exploiting other vulnerabilities that affect the vehicle’s head unit. Moreover, we demonstrated how an attacker may remotely exploit the vehicle when it is connected to the Internet. In particular, the attacker model that we have considered does not require to have fully privileges in the HU operating system making the attack impact higher.

In the last year the HU has got other software upgrade that we had the opportunity to study: CD.EUR.SOP.005.7.181019. STD_M and CD.EUR.SOP.007.1.190212.STD_M. In both software, it is still possible to exploit the presented vulnerabilities. We noticed only minor changes in the working manner of the KOFFEE Module where the majority of the actions work and have the same impact of CD.EUR.SOP.003.30.180703.STD_M version. Thus, all Kia vehicles that have installed one of the three software versions that we tested can be exploited as we have shown in this paper. Moreover, during this study we noticed that Kia shares with another world-wide car manufacturer[6] the same HU software having minimal changes related to the user-interface. Nevertheless, we can guess that the same arbitrary code execution may be present also in these other HUs. However, by not having a car with this HU we were not able to check it, so we leave this other relevant aspect as future work.

Disclaimer

Note that this research activity has followed the responsible disclosure approach in which the vulnerabilities presented in this paper have been reported to Kia Motors. Once we discovered the vulnerabilities, we informed Kia Motors with a technical paper [49]. They internally evaluated our document and after several interactions, we received a new software version to be tested. On this version: CD.EUR.SOP.008.4.200619.STD_M, Kia removed the

[6] As reported by the European Automobile Manufacturers’ Association (ACEA) [48] Kia and the other car manufacturer sharing the same firmware represent the 8.2%, 8.6% and 10.3% percentage of the best European selling car manufacturers in 2019, 2020 and 2021 years.

opportunity to access the engineering menu to install third-party applications that exploit the found vulnerabilities.

As result of the responsible disclosure phase, we published the CVE-2020-8539 that is available online at the CVE MITRE website: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8539.

We declare that all research findings posted in this paper are merely for educational and informational purpose. The replication of the presented results may damage or tamper with vehicle functionalities, they are the user risk and must not be used for malicious purposes.

A eIVIMessage enum struct

Listing 8 eIVIMessage struct code snippet

```
AUDIO_MAIN_CH_CHANGE(IpcSubGroup.ASK_RADIO_AUDIO_MAIN,
        "AUDIO_MAIN_CH_CHANGE", AudioMainChChange.class,
        eMessageType.MICOM),
AUDIO_SUB_CH_CONTROL(273, "AUDIO_SUB_CH_CONTROL",
        AudioSubChControl.class, eMessageType.MICOM),
AUDIO_CH_VOLUME_CTRL(274, "AUDIO_CH_VOLUME_CTRL",
        AudioChVolumeCtrl.class, eMessageType.MICOM),
AUDIO_SETUP_CTRL(275, "AUDIO_SETUP_CTRL", AudioSetupCtrl.class,
        eMessageType.MICOM),
AUDIO_BEEP_PLAY(276, "AUDIO_BEEP_PLAY", AudioBeepPlay.class,
        eMessageType.MICOM),
DSP_MUTE(277, "DSP_MUTE", DspMute.class, eMessageType.MICOM),
AUDIO_AMP_MUTE(278, "AUDIO_AMP_MUTE", AudioAmpMute.class,
        eMessageType.MICOM),
AUDIO_REMOTE_KEY_MUTE(279, "AUDIO_REMOTE_KEY_MUTE",
        AudioRemoteKeyMute.class, eMessageType.MICOM),
AUDIO_SOUND_SETUP_RESET(288, "AUDIO_SOUND_SETUP_RESET",
        AudioSoundSetupReset.class, eMessageType.MICOM),
MAIN_IMAGE_UPGRADE(296, "MAIN_IMAGE_UPGRADE",
        MainImageUpgrade.class, eMessageType.MICOM),
RADIO_FREQ_CHANGE(304, "RADIO_FREQ_CHANGE",
        RadioFreqChange.class, eMessageType.MICOM),
RADIO_PRESET_RECALL(IpcSubGroup.ASK_RADIO_PRESET_RECALL,
        "RADIO_PRESET_RECALL", RadioPresetRecall.class,
        eMessageType.MICOM),
RADIO_PRESET_STORE(IpcSubGroup.ASK_RADIO_PRESET_STORE,
        "RADIO_PRESET_STORE", RadioPresetStore.class,
        eMessageType.MICOM),
RADIO_STATION_SEARCH(IpcSubGroup.ASK_RADIO_STATION_SEARCH,
        "RADIO_STATION_SEARCH", RadioStationSearch.class,
        eMessageType.MICOM),
RADIO_PRESET_DELETE(308, "RADIO_PRESET_DELETE",
        RadioPresetDelete.class, eMessageType.MICOM),
FM_LINKING(325, "FM_LINKING", FmLinking.class,
        eMessageType.MICOM),
DAB_FM_LINKING_STATUS(326, "DAB_FM_LINKING_STATUS",
        DabFmLinkingStatus.class, eMessageType.MICOM),
TMC_AUTO_SEEK_MODE(336, "TMC_AUTO_SEEK_MODE",
        TmcAutoSeekMode.class, eMessageType.MICOM),
TMC_MANUAL_FREQ_CHANGE(340,
        "TMC_MANUAL_FREQ_CHANGE", TmcManualFreqChange.class,
        eMessageType.MICOM),
TMC_PAY_SET(342, "TMC_PAY_SET", TmcPaySet.class,
        eMessageType.MICOM),
TMC_OTHER_COUNTRY_SET(343, "TMC_OTHER_COUNTRY_SET",
        TmcOtherCountrySet.class, eMessageType.MICOM),
ETC_POWER_CONTROL(368, "ETC_POWER_CONTROL",
        EtcPowerControl.class, eMessageType.MICOM),
ETC_BACK_LIGHT_CONTROL(369, "ETC_BACK_LIGHT_CONTROL",
        EtcBackLightControl.class, eMessageType.MICOM),
ETC_LCD_SET_CHANGE(370, "ETC_LCD_SET_CHANGE",
        EtcLcdSetChange.class, eMessageType.MICOM),
...
MICOM_KEY_EVENT(33616, "MICOM_KEY_EVENT",
        MicomKeyEvent.class, eMessageType.MICOM),
ETC_SW_REMOTE_KEY_EVENT(33617,
        "ETC_SW_REMOTE_KEY_EVENT", EtcSwRemoteKeyEvent.class,
        eMessageType.MICOM),
ETC_ACC_SIGNAL_EVENT(33618, "ETC_ACC_SIGNAL_EVENT",
        EtcAccSignalEvent.class, eMessageType.MICOM),
ETC_STATUS_CHANGE_EVENT(33619,
        "ETC_STATUS_CHANGE_EVENT", EtcStatusChangeEvent.class,
        eMessageType.MICOM),
...
```

B Generating an app with msfvenom

We leverage the "msfvenom" utility of Meterpreter to create an app with a malicious payload inside. The app can be obtained with the command in listing 9:

Listing 9 How to create an Android app with a malicious payload using msfvenom

```
msfvenom –p android/meterpreter/reverse_tcp LHOST=<attacjer–IP>
        LPORT=<attacker–port> > ./app.apk
```

The above command generates an apk bundle file needed to install the app in the target device. In particular, with "-p" parameter the Meterpreter reverse shell is chosen as payload and "LHOST" and "LPORT" represent the IP and PORT of the attacker's computer in which there is a running Metasploit module. When, the app is installed the payload connects to the Metasploit module and the Meterpreter shell can be spawned.

References

1. Technology, S.: Automotive ECUs, the Core for Connected Cars. https://www.syrmatech.com/automotive-ecu/. Accessed 22/12/2021 (2022)
2. International Organization for Standardization: Road vehicles—Controller area network (CAN)—Part 1: Data link layer and physical signalling. https://www.iso.org/standard/63648.html. Accessed 22/12/2021 (2015)
3. Dariz, L., Costantino, G., Ruggeri, M., Martinelli, F.: A Joint Safety and Security Analysis of message protection for CAN bus protocol. Adv. Sci. Technol. Eng. Syst. J. **3**(1), 384–393 (2018). https://doi.org/10.25046/aj030147

4. Dariz, L., Ruggeri, M., Costantino, G., Martinelli, F.: A survey over low-level security issues in heavy duty vehicles. In: Automotive Cyber Security Conference. ESCAR (2016)
5. Dariz, L., Selvatici, M., Ruggeri, M., Costantino, G., Martinelli, F.: Trade-off analysis of safety and security in can bus communication. In: The 5th IEEE International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS 2017), pp. 226–231. IEEE, Piscataway, New Jersey, USA (2017)
6. Strategy Analytics Press Releases: Linux, Android poised to dominate automotive infotainment systems. https://www.telecomtv.com/content/device-software-apps/linux-android-poised-to-dominate-automotive-infotainment-systems-34987/. Accessed 22/12/2021 (2019)
7. Christoph Hammerschmidt: Study: Android challenges automotive OS market. https://www.eenewsautomotive.com/news/study-android-challenges-automotive-os-market. Accessed 22/12/2021 (2019)
8. Sean O'Kane: GM will use Google's embedded Android Automotive OS in cars starting in 2021. https://www.theverge.com/2019/9/5/20851021/general-motors-android-auto-google-infotainment. Accessed 22/12/2021 (2019)
9. Valasek, C., Miller, C.: Remote Exploitation of an Unaltered Passenger Vehicle. http://illmatics.com/Remote%20Car%20Hacking.pdf. Accessed 22/12/2021 (2015)
10. Valasek, C., Miller, C.: Adventures in Automotive Networks and Control Units. https://ioactive.com/pdfs/IOActive_Adventures_in_Automotive_Networks_and_Control_Units.pdf
11. International Organization for Standardization: ISO/SAE FDIS 21434 Road vehicles—Cybersecurity engineering (2021). https://www.iso.org/standard/70918.html
12. AUTOSAR: Specification of Secure Onboard Communication AUTOSAR CP R19-11. https://www.autosar.org/fileadmin/user_upload/standards/classic/19-11/AUTOSAR_SWS_SecureOnboardCommunication.pdf
13. Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S.: Experimental security analysis of a modern automobile. In: 2010 IEEE Symposium on Security and Privacy (SP), pp. 447–462. IEEE Computer Society, Los Alamitos, CA, USA (2010). https://doi.org/10.1109/SP.2010.34. https://doi.ieeecomputersociety.org/10.1109/SP.2010.34
14. Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., Kohno, T., *et al.*: Comprehensive experimental analyses of automotive attack surfaces. In: USENIX Security Symposium, San Francisco (2011)
15. Paganini, P.: How to hack airbag in Audi TT on other models. https://securityaffairs.co/wordpress/41416/hacking/hack-airbag-audi-tt.html. Accessed 22/12/2021 (2015)
16. Zetter, K.: Researchers Hacked a Model S, But Tesla's Already Released a Patch. https://www.wired.com/2015/08/researchers-hacked-model-s-teslas-already. Accessed 22/12/2021 (2015)
17. Solomon, O.: Team of hackers take remote control of Tesla Model S from 12 miles away. https://www.theguardian.com/technology/2016/sep/20/tesla-model-s-chinese-hack-remote-control-brakes. Accessed 22/12/2021 (2016)
18. Greenmberg, A.: GM Took 5 Years to Fix a Full-Takeover Hack in Millions of OnStar Cars. https://www.wired.com/2015/09/gm-took-5-years-fix-full-takeover-hack-millions-onstar-cars. Accessed 22/12/2021 (2015)
19. Paganini, P.: Flaws in BMW ConnectedDrive Infotainment System allow remote hack. https://securityaffairs.co/wordpress/49149/hacking/bmw-connecteddrive-hacking.html. Accessed 22/12/2021 (2016)
20. Mitsubishi Outlander hybrid car alarm 'hacked'. https://www.bbc.com/news/technology-36444586. Accessed 22/12/2021 (2016)
21. Nissan Leaf Can be Hacked via Mobile App and Web Browser. https://www.trendmicro.com/vinfo/it/security/news/internet-of-things/nissan-leaf-can-be-hacked-via-mobile-app-and-web-browser. Accessed 22/12/2021 (2016)
22. A Remote Attack on the Bosch Drivelog Connector Dongle. https://argus-sec.com/remote-attack-bosch-drivelog-connector-dongle/. Accessed 22/12/2021 (2017)
23. Costantino, G., Marra, A.L., Martinelli, F., Matteucci, I.: CANDY: A social engineering attack to leak information from infotainment system. In: 87th IEEE Vehicular Technology Conference, VTC Spring 2018, Porto, Portugal, June 3-6, 2018, pp. 1–5. IEEE (2018). https://doi.org/10.1109/VTCSpring.2018.8417879
24. Keenlab Security Lab: New Car Hacking Research: 2017, Remote Attack Tesla Motors Again. https://keenlab.tencent.com/en/2017/07/27/New-Car-Hacking-Research-2017-Remote-Attack-Tesla-Motors-Again. Accessed 22/12/2021 (2017)
25. Tencent Keen Security Lab: New Vehicle Security Research by KeenLab: Experimental Security Assessment of BMW Cars. https://keenlab.tencent.com/en/2018/05/22/New-CarHacking-Research-by-KeenLab-Experimental-Security-Assessment-of-BMW-Cars/. Accessed 22/12/2021 (2018)
26. Pareja, R.: Fault injection on automotive diagnostic protocols (2018)
27. Subaruoutback: Hacking the Starlink Infotainment System. https://www.subaruoutback.org/threads/hacking-the-starlink-infotainment-system.494535/. Accessed 22/12/2021
28. Volkswagen and Audi car infotainment systems hacked remotely. https://nakedsecurity.sophos.com/2018/05/02/volkswagen-and-audi-car-infotainment-systems-hacked-remotely/. Accessed 22/12/2021
29. Hackers Can Steal a Tesla Model S in Seconds by Cloning Its Key Fob. https://www.wired.com/story/hackers-steal-tesla-model-s-seconds-key-fob/. Accessed 22/12/2021 (2018)
30. Tesla Hacked and Stolen Again Using Key Fob. https://threatpost.com/tesla-hacked-stolen-key-fob/161530/. Accessed 22/12/2021 (2020)
31. Costantino, G., Matteucci, I.: CANDY CREAM - hacking infotainment android systems to command instrument cluster via can data frame. In: Qiu, M. (ed.) 2019 IEEE International Conference on Computational Science and Engineering, CSE 2019, and IEEE International Conference on Embedded and Ubiquitous Computing, EUC 2019, New York, NY, USA, August 1-3, 2019, pp. 476–481. IEEE (2019). https://doi.org/10.1109/CSE/EUC.2019.00094
32. Tencent Keen Security Lab: Experimental Security Assessment on Lexus Cars. https://keenlab.tencent.com/en/2020/03/30/Tencent-Keen-Security-Lab-Experimental-Security-Assessment-on-Lexus-Cars. Accessed 22/12/2021 (2021)
33. Weinmann, R.P., Schmotzle, B.: TBONE—A zero-click exploit for Tesla MCUs. https://kunnamon.io/tbone/tbone-v1.0-redacted.pdf (2021)
34. Zavpasha: HACK Navigation/Multimedia Systems KIA/HYUNDAI and Install Third-party Applications. https://forum.xda-developers.com/t/hack-navigation-multimedia-systems-kia-hyundai-and-install-third-party-applications.3892333/
35. Oviradio: Hyundai and Kia Hidden Android Menu Engineering Mode and Secret Features of Radio and Navigation. https://www.oviradio.cz/hyundai-kia-radio-engineering-mode-en/
36. RAPID4 Metasploit: Metasploit. https://www.metasploit.com. Accessed 22/12/2021 (2021)
37. Denys Vlasenko: Busybox. https://busybox.net (2022)
38. XDA Developers: The differences between Odex and Deodex Files. https://forum.xda-developers.com/t/guide-the-differences-between-odex-and-deodex-files.2336411/. Accessed 22/12/2021 (2013)

39. Universal Deodexer: Universal Deodexer V5. https://forum.xda-developers.com/showthread.php?t=2213235. Accessed 22/12/2021 (2013)
40. Android IPC: Android Interface Definition Language (AIDL). https://developer.android.com/guide/components/aidl. Accessed 22/12/2021 (2021)
41. CAN Hacker Website: CAN Hacker. https://canhacker.com/projects/can-hacker-3-0/. Accessed 22/12/2021 (2021)
42. CARBUS Analyzer: CARBUS Analyzer. https://canhacker.com/projects/carbus-analyzer/. Accessed 22/12/2021 (2022)
43. Costantino, G., Matteucci, I.: KOFFEE Module. https://github.com/rapid7/metasploit-framework/blob/master/modules/post/android/local/koffee.rb
44. Costantino, G., Matteucci, I.: KOFFEE Module Documentation. https://github.com/rapid7/metasploit-framework/blob/master/documentation/modules/post/android/local/koffee.md
45. apktool: Apktool: A Tool for Reverse Engineering Android Apk Files. https://github.com/iBotPeaches/Apktool
46. Huq, N., Gibson, C., Vosseler, R.: Driving security into connected cars:threat model and recommendations. Technical report, Trend Micro Research (2020). https://documents.trendmicro.com/assets/white_papers/wp-driving-security-into-connected-cars.pdf
47. for Europe, U.N.E.C.: Un regulation no. 155 - uniform provisions concerning the approval of vehicles with regards to cyber security and cyber security management system. Technical report (2021). https://documents.trendmicro.com/assets/white_papers/wp-driving-security-into-connected-cars.pdf
48. ACEA: European Automobile Manufacturers' Association. https://www.acea.auto/about-acea/ (2022)
49. Costantino, G., Matteucci, I.: KOFFEE—Kia OFFensivE Exploit. Technical report, Consiglio Nazionale delle Ricerche, Istituto di Informatica e Telematica (2020)