2019 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)

CANDY CREAM

hacking infot*ain*ment *and*roid s*y*stems to *c*ommand inst*r*ument clust*e*r via c*a*n data fra*m*e

Gianpiero Costantino
Istituto di Informatica e Telematica
Consiglio Nazionale delle Ricerche
Pisa, Italy
gianpiero.costantino@iit.cnr.it

Ilaria Matteucci
Istituto di Informatica e Telematica
Consiglio Nazionale delle Ricerche
Pisa, Italy
ilaria.matteucci@iit.cnr.it

***Abstract*—Modern vehicles functionalities are regulated by Electronic Control Units (ECU), from a few tens to a hundred, commonly interconnected through the Controller Area Network (CAN) communication protocol. CAN is not secure-by-design: authentication, integrity and confidentiality are not considered in the design and implementation of the protocol. This represents one of the main vulnerability of modern vehicle: getting the access (physical or remote) to CAN communication allows a possible malicious entity to inject unauthorised messages on the CAN bus. These messages may lead to unexpected and possible very dangerous behaviour of the target vehicle. In this paper, we present CANDY CREAM, an attack made of two parts: CANDY aiming at exploiting a vulnerability exposed by an infotainment system based on Android operating system connected to the vehicle's CAN bus network, and CREAM, a post-exploitation script that injects customized CAN frame to alter the behaviour of the vehicle.**

***Keywords*-Automotive, cyber-security attack, infotainment system, Android, remote exploit.**

I. Introduction

The attack performed back to 2015 by Miller and Valasek to the Jeep Cherokee proved that modern vehicles can be hacked like traditional PCs or smart-phones. Vehicles are no longer purely mechanical devices but shelter so much digital technology that they resemble a network of computers. Electronic Control Units (ECUs) regulate all the functionalities of a vehicles. They need to exchange a large amount of data for the various functions of the car to work, and such data must be made secure if we want those functions to work as intended despite malicious activity by attackers.

Similarly to all IoT devices connected to the cyberspace, vehicles are connected with external world through a Wi-Fi connection or via 3G/4G dongle installed in the vehicle to exploit the hundred of functionalities provided by modern infotainment systems. Such systems may present vulnerabilities that may be exploited by attackers. This means that attacks on vehicles may be even disastrous in terms of people safety.

This paper presents CANDY CREAM, an attack that we made to the CAN bus network by exploiting a vulnerability discovered through a vulnerability assessment phase performed on a Android In-Vehicle Infotainment (IVI) system. Following the seven steps described in [1] to perform a remote attack, called CANDY, we gain the remote access to our target Android IVI by exploiting the found vulnerability. Then, we remotely inject a malicious script, named CREAM, that allows us to send CAN frames on the CAN bus in a completely transparent way with respect to the driver. In particular, in our attack, we target an Instrument Cluster (IC) and the CAN frames directed to it. To show the effectiveness of our attack, we use a real Instrument Cluster detached from a vehicle. The observable behaviours of our attack, through the instrument cluster, are on i) the speedometer, ii) the indicator lights, and iii) the red alert indicators. To replicate the proper behaviour of the components of the IC, we executed a reverse engineering procedure in which we discovered which CAN frames activate each IC function and, then exploiting the lack of security of the CAN communication protocol, we forced unexpected actions on the target instrument cluster by sending ad-hoc customized CAN frames.

The paper is organized as follows: next section recalls related literature on other attack to the CAN bus communication protocol. §III recalls primary notions about the CAN bus protocol and §IV details the CANDY CREAM attack. §V draws the conclusion and §VI provides a disclaimer on motivations of this work.

II. Related work

In real world, cyber-security attacks have been perpetrated to vehicles as a demonstration of the vulnerabilities of connected vehicles. Several examples of remote attacks in the automotive domain have been surveyed in [2]. Among the others, the *Remote Keyless Entry/Start* system is based on key fobs of the user that contain a short-range radio transmitter that communicates with an ECU in the vehicle by sending encrypted data. Such data contain identifying information from which ECU can determine if the key is valid and subsequently lock, unlock, and start the vehicle. A typical attack to this kind of system is the Denial of Service attack (DoS) that would

978-1-7281-1664-8/19/$31.00 ©2019 IEEE
DOI 10.1109/CSE/EUC.2019.00094

not allow the car to be remotely locked/unlocked/started and in some cases it may be possible to unlock/start the car without the proper key fob, as the one perpetrated to the Toyota Prius [2]. In 2010, some security researchers showed how to "kill"a car engine remotely, i.e., make the car engine exploitable, by turning off the brakes so that the vehicle would not stop, and making instruments give false readings [3]. In July 2015 an hijacking has been perpetrated to a Jeep Cherokee [4] and also to General Motors (GM) [5]. Hackers remotely took the control of the engine or stole data from the infotainment system, respectively, by exploiting the Internet connection of the infotainment system and a malicious version of the infotainment software installed on the car. In September 2016, researchers hacked a TESLA Model S [6] by using bugs on the TESLA's bounty program through which vehicles received firmware update. Conversely to all the real case attackers listed above, our attack does not exploit manufacturer's software. It exploits the vulnerabilities of the CAN bus communications in all vehicles with an Android In-Vehicle Infotainment system compatible with the vehicle itself.

Koscher et al. [7] experimentally evaluate the security features of a modern car and demonstrate the fragility of the underlying system structure. They prove that, at a certain point in time, an attacker able to infiltrate virtually any ECU can leverage this ability to completely circumvent a broad array of safety-critical systems, and to control a wide range of automotive functions, including disabling the brakes, selectively braking individual wheels on demand, stopping the engine, and so on.

III. The CAN bus protocol in a nutshell

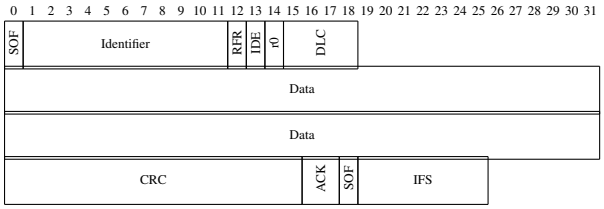

Figure 1: Standard CAN frame format

The CAN protocol is standardised as ISO 11898-1:2015 [8] and, similarly to other networking protocols it is designed as different layers that range from the physical layer to the object layer, in which messages and status are managed. A message exchanged through a CAN data frame contains various fields. These include an Arbitration field carrying the frame ID, also used for arbitration, a Control field for control signals and a Data field for the payload, as pictured in Figure 1. The fields of a CAN frame are described below.

- *Start Of Frame (SOF)* is a dominant bit indicating the beginning of a frame.
- *Arbitration field* consists in: *Identifier*, 11 or 29 bits, according to CAN2.0A *(base frame format)* or CAN2.0B *(extended frame format)*, respectively. This field signifies the priority of the message, with a lower value indicating higher priority, and *Remote Transmission Request* (RTR), 1 bit, which is low for a Data Frame and high for a Remote Frame (one whose Data Field is empty).
- *Control field*, includes the IDE field, 1 bit, to identify whether the payload is of standard length, then r0, 1 bit, reserved for later use, and the Data Length Code (DLC) field, 4 bits, indicating the length of the Data Field.
- *Data* spans over up to 64 bits of data, and carries the payload of the frame.
- *CRC*, 15 bits, is for a cyclic redundancy check code and a recessive bit as a delimiter.
- *Ack*, 2 bits, with the first one being recessive, hence overwritten with a dominant bit by every node that receives it, and the second bit working as a delimiter.
- *EOF*, 7 bits, all recessive, indicates the end-of-frame.
- *IFS*, 7 bits, indicates the time for the controller to move a correct frame into the buffer.

There exist four types of CAN frames:

- *Data frame*: a frame containing node data for transmission.
- *Remote frame*: a frame requesting the transmission of a specific identifier.
- *Error frame*: a frame transmitted by any node detecting an error.
- *Overload frame*: a frame to inject a delay between data or remote frame.

The mapping between messages in the payload and vehicle functionalities, e.g., CAN dbc file, is up to the car manufacturer and are normally kept confidential. Each mapping enables ECUs of a specific vehicle to correctly interpret the messages and translate them into signals that carry out the expected functionality.

IV. CANDY CREAM attack

CANDY CREAM is an attack designed to gain the control of a vehicle via its Android In-Vehicle Infotainment (IVI) System. To have a successful attack, we assume that the target Android IVI is powered on and connected to the network. The vehicle can be powered on or not. This, however, impacts on the type of attack that can be performed. In fact, some internal components of the vehicles requires a minimum voltage to work, i.e., 12v, and that other settings are applied, e.g., the vehicle key status is set as "on".

As represented in Figure 2, our attack consists of two parts: CANDY and CREAM. CANDY has additional

Figure 2: CANDY CREAM attack flow

two main phases: i) Android IVI exploitation to take advantages of a misconfiguration discovered through a vulnerability assessment and ii) in-vehicle CAN bus network exploitation to pass through the Android IVI on-board. CREAM is the post-exploitation script to inject forged CAN data frames to alter the behaviour of the vehicle. Note that, CANDY is an inherited name by our previous attack to an Android IVI [9]. Both of them have and Android IVI as target device but they are different in design and development. In fact, conversely with the one we are going to describe here, the one in [9] consists of an Android APP able to steal sensitive information by opening a back-door that allows an attacker to remotely download such information.

A. CANDY: Android IVI exploitation

To obtain the control of a remote Android IVI, we covered all steps described in [1] to exploit a remote device. The target device is shown in Figure 3 and it is equipped with Android 6.0 operating system, quad-core at 1.2 GHZ and 1 Gbyte of RAM. In addition, the Android IVI sports the Wi-Fi and Bluetooth network interfaces and it is possible to get access to the 3G/4G mobile network by plugging-in a USB-dongle.

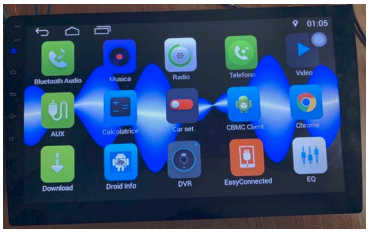

Figure 3: The Android IVI

The attack steps listed below exploit the Wi-Fi connection of the Android IVI where the phase:

- *Recon* checks out the target using multiple sources like intelligence gathering.
- *Scanning* maps out and investigates the network.
- *Exploitation* attacks holes found during the scanning process.
- *Elevation of Privileges* elevates a lower access account to root, or system level.
- *Maintaining Access* uses techniques like backdoors to keep access to the target device.
- *Covering their Tracks* erase logs, and manipulating files to hide the intrusion.

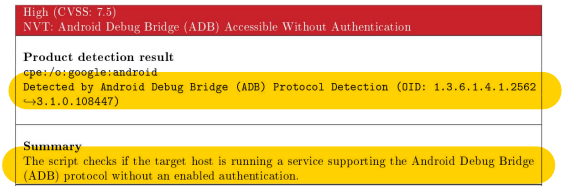

Figure 4: The NVT of the vulnerability

1) Recon: To perform the attack, both the attacking computer and the target device must be visible each other and so covered by the same network. This can apply when a vehicle, which hosts the Android IVI, and the attacker are relatively close, for instance, in a parking area that offers free Wi-Fi connectivity, or in a wider scenario, where the attacker and the vehicle are under the same 3G/4G network.

2) Scanning: This is a very relevant phase since it gives clue of the potential vulnerabilities that the target device may exploit. The scanning phase is done by triggering a *vulnerability assessment* on the target device, in our case, the Android IVI. As we said above, we assume that the vehicle and the attacker are under the same network, so the vulnerability assessment is triggered by specifying the IP of the Android IVI.

The vulnerability assessment was done using two different open-source, widely used and well-known tools: i) Nmap [10] to network discovery and security auditing and ii) OpenVas [11] that allows the tester to deeply analyse the found vulnerabilities and discover the related Common Vulnerabilities and Exposures (CVEs). Nmap offers several options to scan network and discover open ports of a target device and several add-ons have been developed to extend its core functionalities. OpenVas has been developed as vulnerability assessment system and includes unauthenticated, authenticated testing, and more than 50,000 Network Vulnerability Tests (NVTs).

The first scan was run through Nmap to discover the opened ports on the target device: it highlighted a few opened ports and, in particular, a service running on port 5555. At first glance, this information did not provide particular hints on the security status of the Android IVI. So, we decided to run an additional scanning through OpenVas and the report provided more details on the vulnerable service running on port 5555. Figure 4 shows part of the NVT description.

As the NVT shows, the vulnerability score given is 7.5 points in a range from 1 to 10, meaning that the vulnerability may be exploited to access the target device. In particular, the report says: *"The script checks if the target host is running a service supporting the Android Debug Bridge (ADB) protocol without an enabled authentication."* and the impact states that *"This issue may be exploited by a remote attacker to gain access to*

sensitive information or modify".

The Android Debug Bridge (ADB) [12] is a tool developed by Google and used by developers for debugging purposes. ADB allows developers to remotely access the device. As reported in the on-line developer Android page, *"Android Debug Bridge (ADB) is a versatile command-line tool that lets you communicate with a device. The ADB command facilitates a variety of device actions, such as installing and debugging apps, and it provides access to a Unix shell that you can use to run a variety of commands on a device".*

The ADB is a tool that should be used only during the configuration phase of the devices and should be stopped when not needed anymore. This action, however, is not always performed by manufacturers and it may expose devices to relevant cybersecurity risks. In fact, as reported by Beaumont in [13], thousand of Android devices can be exploited since hardware manufactures sell their device with the `adb` port, i.e., `5555`, opened and without any authentication mechanism.

3) Exploitation: To exploit the vulnerability found and described in the previous step, we used the `adb` command line client, available for the majority of operating systems, spanning from Microsoft Windows to Mac Os X. This command line tool allows a developer to connect to the device with a simple command and to interact with it by exploiting other additional commands. So, we just needed the target device IP address. Once, the connection is up, it is possible to control the Android IVI exploiting the set of commands available through the `adb`. For instance, entering the command `adb shell`, we spawn a remote shell on the device that allow us to explore the remote file-system. In addition, with the command `adb push`, it is possible to transfer local files to the remote device.

4) Elevation of Privileges: This phase is usually performed by attackers to get higher privileges on the controlled devices, e.g., obtaining *root* privileges. As it is customary, processes running on computers should never run with high privileges unless the access to important operating system files is requested, then the root access is mandatory.

In our attack, the `adb` command tool gave a remote control of the device with root privileges and no additional vulnerabilities were required to have more privileges on the Android IVI.

5) Maintaining Access and Covering their Tracks: These last two phases are out of scope of this paper. However, they should not be left uncovered when an attack is performed. In fact, the maintaining access step refers to find a mechanism to create a persistent link between the target device and the attacker. This is needed when, for some reasons, a link disconnection appears, for instance the Android IVI has been powered off. Thus, when the device will be switched on again, it should automatically reconnect to the attacker computer to make again a usable link.

Covering their tracks step refers to the procedure to delete all indications that may be used by a controller to identify unauthorised access from users or processes. For instance, this action could be performed by cleaning the log files and the command history executed in the target device.

B. CANDY: In-vehicle CAN bus network exploitation

Figure 5: Real Instrument Cluster

The second phase of the attack consists in gaining the control of the vehicle. To reproduce it in our test-bed with an Android IVI connected to the CAN bus, we leveraged a real Android IVI (Figure 3) connected to an real instrument cluster (Figure 5).

The target instrument cluster is connected via USBtin [14], which is a simple USB to CAN interface able to monitor CAN bus and to transmit CAN messages, to the CAN interface of the Android IVI (Figure 6).

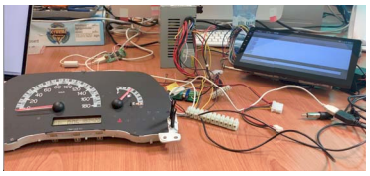

Figure 6: Our test-bed

1) ADB connection to the Android IVI: As we explained in §IV-A2, the Android IVI exposes a relevant vulnerability that allows an attacker to establish unauthorised connections to the device exploiting port `5555`. In particular, in §IV-A3 by using the adb command line it is possible to establish a remote connection to the target Android IVI. So, to obtain access to the remote device, as attacker, we opened a new shell in our local operating system with the adb tool installed, and we inserted the command:

```
adb connect TARGET_IP
```

Then the attacker obtains a direct connection with the target devices. Now, the connection with the vehicle has been achieved and the next step is to access the CAN

Figure 7: Speedometer activation

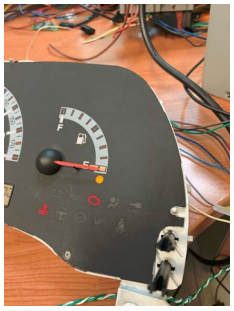

Figure 8: Alert indicators on

Figure 9: Light indicators on

bus network. This is made by using the USBtin interface that physically connects the Android IVI to the CAN bus of the instrument cluster.

2) Environment preparation: To execute our CREAM post-exploitation script, a python environment must be available within the Android IVI. Since, the Android IVI operating system does not host by default the python environment, it can be installed using one of the applications presented in the Android store. This step would require a physical interaction with the Android IVI. This action, however, would make the attack more and more complicated since the attacker should access the vehicle, switch on the radio and install the python environment. Instead, to bypass all these steps, we leverage again the adb command to remotely install the application to obtain the python environment running on the Android IVI. In particular, the application installed in the python environment is *qpython3*, which is available at: https://github.com/qpython-android/qpython3/releases. So, once the application is downloaded from the store, it can be installed in the Android IVI with this two simple commands:

```
adb push qpython3-app-release.apk /sdcard/
adb shell pm install /sdcard/qpython3-app-release.apk
```

The first command copies the application from the attacker's computer to the target device. Then, the second command installs the python environment on the Android IVI. However, when reproducing the attack, we observed that through the adb shell pm install command, not all needed files to run the python environment are properly installed. This issue can be simply overtaken using this additional command:

```
adb push org.qpython.qpy3/ /data/data/
```

This command will copy the missing files from the attacker's computer to the Android IVI to achieve a complete and working python environment.

C. CREAM: post-exploitation script

After exploiting the Android IVI, we are ready to perform our CREAM post-exploitation script as illustrated in Figure 2. CREAM script is written in python and uses the remote server to inject forged CAN data frames to the instrument cluster.

CREAM is written to trigger unexpected actions against our instrument cluster. in particular, it can:

1) Send data frames to activate the speedometer;
2) Show the alert indicators;
3) Show the lights indicators.

Within *Send data frames* option, CREAM will inject into the CAN bus crafted data frames that will move the speedometer indicator into a particular positions (Figure 7). Within the second option, CREAM will send a CAN frame that turns on the alert indicators, such as problem to the engine, the absence of oil, too high water temperature (Figure 8), while the last option will send a CAN frame to active the lights indicators (Figure 9).

To install CREAM into the Android IVI, we exploited the connection established with the adb command line. In particular, to move the exploit from our local computer to the victim device, we run the command:

```
adb push CREAM.py /sdcard/
```

At this point, CREAM resides on the Android IVI but to be properly executed it must leverage the python environment installed before. So, the next step is to active the python environment by entering the following commands on the attacker's shell:

```
adb shell
cd /data/data/org.qpython.qpy3/files/bin
./qpython-root.sh
```

Last command will execute the python environment as it is shown in Figure 10. Reached this phase, we are ready to run CREAM by simply executing the following two commands:

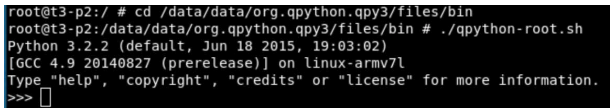

Figure 10: Python environment on the Android IVI

```
import subprocess
subprocess.call(['python','/sdcard/CREAM.py'])
```

These commands execute the exploit. Below part of the code to activate the speedometer till ∼40km/h.

```
can = serial.Serial(port, baud, timeout=timeout)
can.write(("S2\r").encode('ascii'))
sys.stdout.write("Opening CAN channel\n");
     sys.stdout.flush()
can.write(("O\r").encode('ascii'))
sys.stdout.write("Sending command to set speedometer
     at ~40kmh...\n"); sys.stdout.flush()
can.write(("t0E520196\r").encode('ascii'))
sys.stdout.write("Command sent\n"); sys.stdout.flush()
```

V. Conclusion

Modern vehicles are connected with the world around them via WiFi Connection or 3G/4G network. This provides to drivers innumerable advantages in terms of services and smart functionalities. On the other hand, it also entails real risks in terms of security and privacy that can affect passengers' safety. In this paper, we present CANDY CREAM as a real example of a possible attack that can be perpetrated on vehicles by exploiting possible existing vulnerabilities of Android IVI. In this paper, we describe how, by using open source tools and following well defined steps, we discovered a vulnerability on our target Android IVI, remotely took control of it and, consequently of the vehicle on which it has been installed by exploiting the absence of security on the CAN bus communication. We are currently working to provide a protocol able to secure the CAN communication, named TOUCAN [15]. This solution may not prevent an attacker to gain the access to an Android IVI, but it may prevent the attacker to take the control of the vehicle.

VI. Ethical Issue

We are conscious that the description of the attack is detailed enough to be replicated not only for research purposes but also for malicious activities. We declare that our purposes are research-oriented and we aim at pinpointing these kind of issues in order to carry on activities to overcome them.

Acknowledgement

The authors would like to thank A. Brusca and V. Melani for their great work during their stage period. This work has been partially supported by the GAUSS national research project (MIUR, PRIN 2015, n2015KWREMX) and by H2020 EU-funded projects C3ISP (GA n700294).

References

[1] D. W. Dieterle, *Basic Security Testing with Kali Linux 2*. Copyright © 2016 by Daniel W. Dieterle, 2016.

[2] C. Miller and C. Valasek, "A survey of remote automotive attack surfaces," *Black Hat USA*, 2014.

[3] [Online]. Available: https://www.bbc.com/news/10119492

[4] [Online]. Available: https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/

[5] [Online]. Available: https://insideinternetsecurity.wordpress.com/2015/08/05/ownstar-yet-another-car-hack/

[6] [Online]. Available: https://www.pcworld.com/article/3121999/researchers-demonstrate-remote-attack-against-tesla-model-s.html

[7] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham *et al.*, "Experimental security analysis of a modern automobile," in *Proceedings of IEEE Symposium on Security and Privacy*, 2010, pp. 447–462.

[8] 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, 2015.

[9] G. Costantino, A. L. Marra, F. Martinelli, and I. Matteucci, "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*, 2018, pp. 1–5. [Online]. Available: https://doi.org/10.1109/VTCSpring.2018.8417879

[10] [Online]. Available: https://nmap.org

[11] [Online]. Available: http://www.openvas.org

[12] [Online]. Available: https://developer.android.com/studio/command-line/adb

[13] Beaumont. [Online]. Available: https://tinyurl.com/yd7mvp2f

[14] [Online]. Available: https://www.fischl.de/usbtin/

[15] G. Bella, P. Biondi, G. Costantino, and I. Matteucci, "TOUCAN: A protocol to secure controller area network," in *Proceedings of the ACM Workshop on Automotive Cybersecurity, AutoSec@CODASPY 2019, Richardson, TX, USA, March 27, 2019*, 2019, pp. 3–8. [Online]. Available: https://doi.org/10.1145/3309171.3309175