AUTOMOTIVE CAN BUS SECURITY

Marc Barrera Puente

Resum– Els vehicles moderns s'han anat convertint en una xarxa d'ordinadors i sensors, interconnectats entre ells per un bus, amb una implementació CAN, cosa que permet noves possibilitats per ciberatacs. Aquest article explora atacs d'injecció de missatges CAN que es poden dur a terme a través del port de diagnòstic del cotxe amb l'objectiu d'alterar el comportament del vehicle, així com teoritza possibles atacs realistes que es podrien dur a terme amb l'informació obtinguda.

Paraules clau– CAN en vehicles, atac d'injecció CAN, atac OBDII, seguretat en vehicles

Abstract– Modern vehicles have become a network of computers and sensors connected via bus with CAN as the protocol, which opens up new possibilities for cyberattacks. This paper explores simple CAN message injection attacks that can be performed by simply connecting to the diagnostic ports of the vehicle, with the objective of altering the vehicle's behavior, and discusses potential real-world attacks that could be performed with this information.

Keywords– Automotive CAN, CAN injection attack, OBDII attack, vehicle security

1 INTRODUCTION

In the last few years, technology is becoming prevalent in our day to day lives, and people are getting more aware of the security threats and the importance of protecting one's data, with huge scandals like Cambridge Analytica[1], where Facebook's lack of security allowed an app to access the profiles of people who downloaded it, as well as all of their contacts or Equifax[2], the biggest consumer credit reporting agency in the United States, that had a massive breach, and very sensitive information (like Social Security Number, addresses, names, and surnames) of 140 million people got leaked. Because of that, the use of VPNs[3] and other encryption services and new policies for data protection are increasing exponentially. However, with the current trend of IoT (Internet of Things) more and more things are connected and operated by computers, things that usually do not come to mind when thinking about vulnerable systems[4], for example, CCTV systems, printers, and the one we use the most, cars.

Cars have not been an exception to the digitalization of technology, being more complex over time, with more and more computer modules interconnected, and less analog technology. However, the nature of the data transmitted in cars (the information is critical and needs to be processed fast) prevents security from being the number one priority, so it is usually weak or non-existent. For that reason, it is important to test how far you can get into modern vehicles and how easy attacks are to perform, to be able to propose solutions that minimize the risk and still maintain the performance needed in a vehicle.

Modern cars work a lot like a network, with different devices connected between them. For this architecture to be as fast as possible, it uses a bus, where every device dumps the packets, and the intended receptor reads it, while all other components of the networks just ignore it. This bus can use different kinds of protocols, although the most used one is CAN[5], and it is the one we are going to focus on. That means that anyone connected to the bus has access to all information transmitted through it, and it also means that it can be written on by anyone that has physical access to it.

The easiest way to access the CAN bus is the On-Board Diagnostics (OBDII[6]), which uses a standardized digital communications port to provide real-time data in addition to a standardized series of diagnostic trouble codes (coolant temperature, engine load, etc.), and it is mandatory for all gas cars in the EU since 2000, diesel cars following soon after in 2003. To be able to get this information, the OBDII port has a direct line to the bus, and although it is not supported by the OBDII standard, it can be accessed with adequate software.

The objective of this project is to use the OBDII port and the weak security to analyze the traffic, find which packets perform which action in the car, use this information to be able to write to the CAN bus altering the vehicle's behavior and functionalities, such as the RPM gauge, as well as explore other kinds of attacks with the ultimate objective being turning the car on from the computer.

• E-mail de contacte: marcalbert.barrera@e-campus.uab.cat
• Menció realitzada: Tecnologies de la Informació
• Treball tutoritzat per: Víctor García Font
• Curs 2020/21

2 OBJECTIVES

In this section we will introduce the main objectives for this project.

- O1: Affecting vehicle behavior via CAN bus injection: The main goal of the project is to find, interpret and replay packets to the CAN bus to allow us to modify the behavior of the vehicle. This will allow us to expand and use the functionality on future applications. This feat will allow for quite some practical applications, such as periodically turning on your car in order to warm it, and will be expandable using similar techniques and gathered data, for example also turning the heat on.

- O2: Decoding of the CAN bus traffic: In order to get to our main objective, we need to analyze the traffic running in the bus to understand what it does, which bytes refer to what, and how to recreate it. To prove that we have interpreted the fields of the packet correctly, we can use physical inputs to the car. If we set the throttle to a certain position, the RPM gauge should remain in a certain range (for example at 1/4 throttle the RPMs may be at 2000). Knowing that information we can predict the packet fields. Being able to interpret the packets allows us to eventually recreate them modifying it the way we need.

- O3: Write fake packets on the bus: We can use the previously acquired knowledge to create a fake packet, and then write it to the bus, with the objective of affecting the car behaviour. It will work as a simple test to then expand and work towards the ultimate goal of the project as a whole. For this goal to be reasonably done, we will use an easier parameter to search for, a parameter that occurs more frequently: the RPM gauge value. This packet gets sent over and over, to have the RPM value constantly updated, plus it also allows us to have previous knowledge of how this packet should look like (it will have the current RPM value in the data field). That way we can easily prove that we successfully have spoofed the car, by moving the RPM needle to a value that we previously decided in our computer, without input from the throttle. It is important to take this step in order to prove and really gasp the complexity of the main objective, and see how realistic it is.

- O4: Create a software to turn on the car periodically: The last objective is the least prioritized, and it would be to create a program or app that allows the user to program the time and for how long he wants the car to turn on automatically, as well as, if possible, other functionalities such as the heat also turning on. This would be closer to a MVP for a hypothetical release to market. The program needs to be easily usable, given that the previous phases already cover the functionality of said program. As mentioned before, this is the least prioritized, because it is bound to the success of the other objectives in a shorter time than estimated, however in case it gets completed, it would be a nice cheap alternative to the paid extras some manufacturers put in their top of the line cars.

2.1 Methodology

The majority of the work will be trying to find ways to recognize certain packets by looking for physical changes that directly correlate to changes in the CAN traffic, and then recreate and replay said packet and see if the car performs the action. This is a very fluid task, as methodology will change the more we understand about the network and the more we progress in the project.

In order to do that we will need to reverse engineer the packets. We are working on a software which code is inaccessible, as well as the hardware. So the only way we can interact with it is by giving it inputs and observing the output. This is called black box reverse engineering [8]. During black box testing, an analyst attempts to evaluate as many meaningful internal code paths as can be directly influenced and observed from outside the system. Black box testing cannot exhaustively search a real program's input space for problems because of theoretical constraints, but a black box test does act more like an actual attack on target software in a real operational environment. With this kind of reverse engineering we will be able to acquire knowledge about certain packets from a vehicle, so we can explore the consequences of replaying them in a malicious manner.

We will follow a FDD[7] (Feature-Driven Development) as our development methodology to complete objective O4, as it is best suited for small teams, such as our developing team, being only formed by only one person. It also complements our objective characteristics seamlessly, given that each one of them is dependent on the ones that come before it, where every objective would be a new feature.To produce tangible software often and efficiently, FDD has five steps, the first of which is to develop an overall model. Next, build a feature list and then plan by each feature. The final two steps (design by feature and build by feature) will take up the majority of the effort.

3 BACKGROUND

Modern automobiles are controlled by a heterogeneous combination of digital components. These components, Electronic Control Units (ECUs), oversee a broad range of functionality, including the drivetrain, brakes, lighting, and entertainment. Indeed, very few operations are not mediated by computer control in a modern vehicle. Charette estimates that a modern luxury vehicle includes up to 70 distinct ECUs including tens of millions of lines of code [9]. In turn, ECUs are interconnected by common wired networks, usually a variant of the Controller Area Network (CAN). This allows for varying features related to safety and critical systems, such as emergency braking and Anti-lock Braking System (ABS), roll-bars deployment in convertibles, seat belt tensioning automatically during a crash, and many others. However, in turn, this architecture creates a surface for attacks, since in a bus every component has implicit access to all of the other components.

3.1 CAN and OBDII

To be able to exploit the aforementioned vulnerabilities, we need to get a deeper understanding of the CAN architecture and protocol. As mentioned earlier CAN runs on a

bus, a multi-master serial bus standard for connecting Electronic Control Units (ECUs) also known as nodes. Two or more nodes are required on the CAN network to communicate. All nodes are connected to each other through a physically conventional two-wire bus, as seen in Fig. 1. The wires are a twisted pair with a 120 Ω (nominal) characteristic impedance.

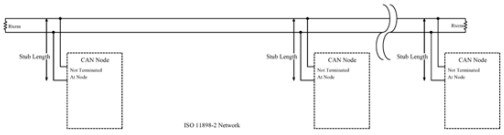

Fig. 1: CAN bus physical diagram

This bus uses differential wired-AND signals. Two signals, CAN high (CANH) and CAN low (CANL) are either driven to a ”dominant” state with CANH $>$ CANL or not driven and pulled by passive resistors to a ”recessive” state with CANH $\leq$ CANL (Fig. 2). A 0 data bit encodes a dominant state, while a 1 data bit encodes a recessive state, supporting a wired-AND convention, which gives nodes with lower ID numbers priority on the bus [5].

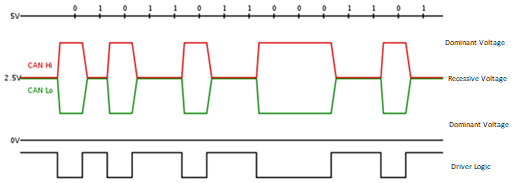

Fig. 2: CAN bus signal example

As can be deduced, anyone with physical access to the bus (wire pair) can access all data transmitted through the bus, fortunately for the success of this project, the OBDII port has direct access to both CANH and CANL, so we won't need to do any destructive entry to the physical bus.

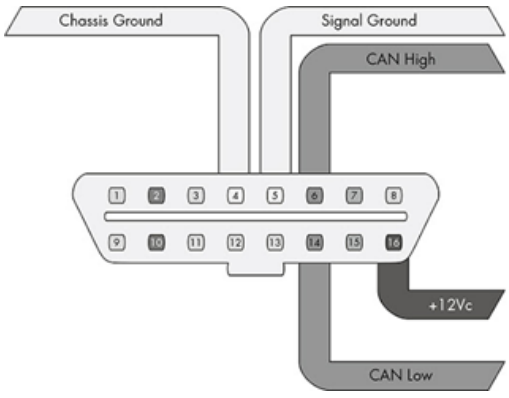

Fig. 3: OBDII connector pinout

Now that we have an entry point, we need to understand how CAN codes the information in the bus, the protocol's frame format. A CAN network can be configured to work with two different messages (or ”frame”) formats: the standard or base frame format (described in CAN 2.0 A and CAN 2.0 B), and the extended frame format (described only by CAN 2.0 B). Extended packets are like standard ones, except that they can be chained together to create longer IDs. Extended packets are designed to fit inside standard CAN formatting to maintain backward compatibility. So if a sensor doesn't have support for extended packets, it won't break if another packet transmits extended CAN packets on the same network.

Standard packets also differ from extended ones in their use of flags. When looking at extended packets in a network dump, you'll see that unlike standard packets, extended packets use substitute remote request (SRR) in place of the remote transmission request (RTR) with SSR set to 1. They'll also have the IDE set to 1, and their packets will have an 18-bit identifier, which is the second part of the standard 11-bit identifier. There are additional CAN-style protocols that are specific to some manufacturers, and they're also backward compatible with standard CAN in much the same way as extended CAN.

This means we can have a pretty good idea of how the packet will be structured when we sniff the traffic from the bus, as the format is pretty well defined and can be seen in Fig. 4, and in a more detailed manner in Table 1.

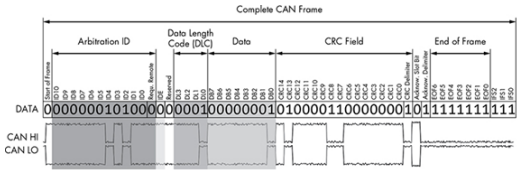

Fig. 4: Standard CAN frame

Table 1: CAN PACKET FIELDS

Field name	Length (bits)	Purpose
Start of frame	1	Denotes the start of frame transmission
Identifier	11	A (unique) identifier which also represents the message priority
Remote Transmission Request	1	Must be dominant (0) for data frames and recessive (1) for remote request frames
Identifier Extension bit	1	Must be dominant (0) for base frame format with 11-bit identifiers
Reserved bit	1	Reserved bit. Must be dominant (0), but accepted as either dominant or recessive.
Data Length Code	4	Number of bytes of data
Data Field	0-64	Data to transmit
CRC	15	Cyclic Redundancy Check
CRC Delimiter	1	Must be recessive (1)
ACK Slot	1	Transmitter sends recessive (1) and any receiver can assert a dominant (0)
ACK Delimiter	1	Must be recessive (1)
End-of-frame	7	Must be recessive (1)

3.2 How it all ties up: inner working of a car

As explained in section 3, a car is a network of ECUs and sensors interconnected via bus. A sensor will send information for an ECU to interpret and use to perform certain actions in the car. The more advanced a car is, the more actions are performed by the ECUs, and fewer analog inputs it has.

This means that the bus carries a lot of information, and creates an issue of response time. Even though the CAN protocol has a priority ID for important packets, what is usually done to prevent this problem is to have separate buses for the critical operations of the car and the media systems, door locks, windows, AC, etc.

A good example is the ”drive-by-wire” trend, which is the use of electrical or electro-mechanical systems for performing vehicle functions traditionally achieved by mechanical linkages, making the driving experience much smoother, and allowing a lot more electronic control into the mix, like traction control, ABS (anti-lock braking system), lane assist and other automatic driving aids. This makes the driver errors much more forgiving, as manufacturers can

correct said mistakes by passing inputs through a program that mitigates them. However, this also means that pretty much all of the car's inner workings are passed through the bus, and accessible by anyone who has physical access to the bus.

An example of this would be the tachometer. In older cars, the RPMs were displayed in the dash using a cable that connected the gauges with the engine, and as the engine rotated, the cable would translate said rotation to the gauge, which would make the needle move as shown in Fig 5. The same applied to the speedometer, a cable connected to the wheel that rotated, and a mechanical device that translated the rotation to a certain position on the gauges.

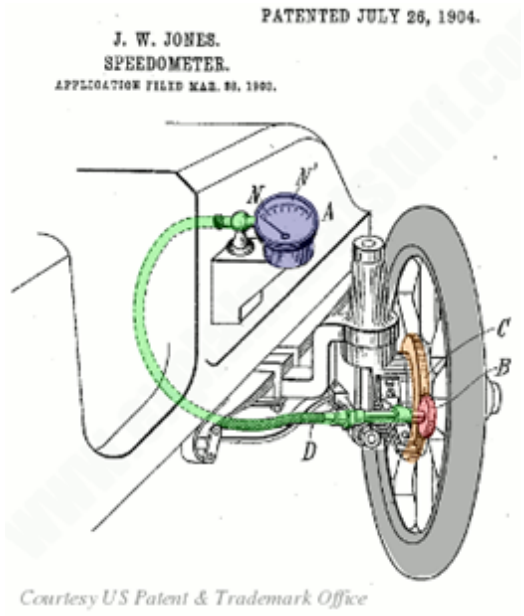

Fig. 5: Analog speedometer

Nowadays this is replaced with a sensor that sends the data through the bus directly to the dash, which is usually digital. This is a good way to see if sending messages through the bus will result in a change in behavior in the car.

3.3 Tools used

In this section, we will briefly discuss the different tools we are going to use during the project, from the physical connector to the OBDII to the various software used during the project. With the information provided so far one can see that we need a OBDII connector (Section 3.3.1) to pass the data from the car to the computer, and we will also need some kind of software to log, interpret and ultimately replay the packets gathered during testing (sections 3.3.2 onward).

3.3.1 OBDII connector

To monitor the activity on the CAN bus, a device that can monitor and generate CAN packets is needed. There are a ton of these devices on the market. The cheap OBD-II devices that sell for under 20$ technically work, but their sniffers are slow and will miss a lot of packets. It's always best to have a device that's as open as possible because it'll work with the majority of software tools/open-source hardware and software.

3.3.2 SocketCAN

The socketcan packet is an implementation of CAN protocols (Controller Area Network) for Linux. We will use its can-utils suite for a command-line implementation, and as open-source software, it's easy to extend functionality to other utilities.

3.3.3 Kayak

Kayak is a Java-based GUI for CAN diagnostics and monitoring, and is one of the best tools for use with socketcan because of the simplicity it offers when dealing with the CAN bus, recording packets, sending them by either creating or replaying the packets, all in a very simple to use interface that speeds up processes that using socketcan alone would need some scripting from our part.

4 State of the art

Adventures in Automotive Networks and Control Units [12] and CAN Message Injection [13] are two studies where the authors managed to affect the behavior of different cars, in the first they accessed the CAN bus directly via the OBDII port, captured and interpreted the messages between the car's ECUs, and by replaying and constructing packets sent back to the bus (regular CAN packets) they turned on and off various lights, activate the horn and other things that don't directly affect the driveability of the car, and they also used diagnostic packets to kill the engine, use the brakes and close the doors, all of this with the restriction that the vehicle would only allow diagnostic packets at very low speeds. However in the second one they kept pressing forward further expanding their research, and managed to reprogram some of the ECUs from the same cars used previously, and allow them to control the acceleration, braking, and steering of said cars while circumventing the main constraint of the diagnostic packets.

Experimental Security Analysis of a Modern Automobile [10] figures out the kind of attacks one could do if given access to the car's internal network. By taking the individual ECUs and getting the code inside of them, as well as writing their own program to read and filter CAN messages, they managed to completely take over the car, unlocking and locking the doors, braking, and killing the engine just to name a few. The study does not discuss the way an attacker can get access to the bus, however, the next study focuses on that.

Comprehensive Experimental Analyses of Automotive Attack Surfaces [11] takes a look at all the different ways a car can be attacked. It is a threat assessment with the previous knowledge that access to the car's internal networks grants them full control of the vehicle. However it does not go into detail about how the attacks are performed as they do in the previous study, it is more of an analysis of how it could be done.

All the studies mentioned above are very lengthy (averaging 80 pages) and complement each other, meaning that the information we provided is a very general summary,

and it doesn't include many in-depth explanations. However there is something very important to understand when it comes to the attacks performed in them, and how they are so successful, and this is the difference between regular CAN messages and diagnostic CAN messages.

Regular CAN messages are the traffic in the bus during normal operation of the vehicle, while diagnostic messages are the ones rarely seen in normal traffic, instead, they consist of direct contact with ECUs that allows mechanics and dealerships to update firmware, perform tests to check the integrity of the systems, etc. This means that they allow more access to the physical actions of the car, which makes them very important to secure. The way these messages are secured is by the ECU performing a cryptographic non-repeatable challenge, where you need a key that is not passed in the CAN bus at any time and is hidden in the ECU firmware. That means that to make an ECU go into diagnostic mode and receive/send diagnostic messages, we need to decompile said ECU's code and find the key, and the procedure for that is to take it out of the car and connect it to a computer. Furthermore, these ECUs only go into diagnostic mode when the car is not moving, or at very low speeds, so to allow the researchers to perform meaningful attacks at speed, they needed to reflash the ECU to allow it to be in accessory mode at any speed, and then send the diagnostic messages.

It is important to note that the results they got are the culmination of 4-5 years of full-time research by a group of people with the knowledge, hardware and software necessary, and the means, so we can't expect to get results at the same level, as we won't be using diagnostic packets nor reprogramming the ECU's firmware.

5 Understanding threat models

Attack surface refers to all the possible ways to attack a target, from vulnerabilities in individual components to those that affect the entire vehicle. When discussing the attack surface, we're not considering how to exploit a target; we're concerned only with the entry points into it (more information on how to define them in [14]).

The attack surfaces on a modern car can vary from make and model, and some of them include the Bluetooth connection for the media systems (music, calls, contacts), the RF receivers for the key fob (both for unlocking the doors and the anti-theft system for authenticating the key for ignition), CD and USB ports and many others. However, for this project, we are focusing on the diagnostics port (OBDII). As it can be seen, there are many ways data can enter the vehicle. If any of this data is malformed or intentionally malicious, it is important to know how the car manages this issues. This is where threat modeling comes in.

Threat modeling, as explained in [15], is a process by which potential threats, such as structural vulnerabilities or the absence of appropriate safeguards, can be identified, enumerated, and mitigation can be prioritized. The purpose of threat modeling is to provide defenders with a systematic analysis of what kind of controls or defenses need to be included, given the nature of the system, the probable attacker's profile, the most likely attack vectors, and the assets most desired by an attacker. However we are going to use this method to identify the ways the CAN bus network in a car can be attacked, and what kind of attacks are more likely to work, to be easier, or to be completely impossible.

6 Attack appraisal

The way we apply the black box reverse-engineering from section 2.1 will be the main workload of the project. Knowing more about the CAN bus inner workings and protocols allows us to exploit it, by reverse-engineering the packets and then retransmitting them into the bus. To reverse engineer the CAN bus, we first have to be able to read the CAN packets and identify which packets control what. We are interested in accessing all the packets that flood the CAN bus that the car uses to perform actions. It can take a long time to grasp the information contained in these packets, but that knowledge can be critical to understanding the car's behavior.

Of course, the goal is to understand the data field in the CAN packets and understand how the information is coded to be able to recreate it. The ID field is also important to correlate to a certain process in the vehicle. Unfortunately, generic packet analysis won't work for CAN because CAN packets are unique to each vehicle's make and model.

To assess if we are able to perform CAN injection attacks successfully we will try to modify one observable physical output from the car via the bus. The easier way to do it is to affect something in the dash, as it has a clear display and it is not a critical system which may have extra security implemented.

Every vehicle will be different in traffic and in the way they send information to the display, IDs, attack mitigation, and other very important factors we will face as we try to perform certain attacks.

We had access to a few cars, some from the last 2-4 years, and others were 10+ years old, so we were able to observe all kind of traffic and security measures, as well as different architectures of ECUs, sensors and displays, where we tried to change the RPM display as a starter. Some of the cars we tried to use didn't even implement a CAN network, meaning our connector was no good in those cases. This is why in the next sections we will discuss which cars were significant (only the ones that had a CAN protocol implementation in their bus), and the results we got from each of them .

6.1 Scenario 1

The first car we attempted to breach was a 2018 Mercedes Benz GLA 220d. It is a fairly modern vehicle, with tons of electronic aid, meaning a lot of the systems run on the bus. As expected there is a fair amount of traffic in the bus, which means there's a lot of functions controlled in some way by the bus.

We were able to correlate some physical inputs to the car with a certain field of a certain packet (filtered by ID). However, after successfully reverse-engineering some packets including the one that controls the RPMs, we tried to test if our assumptions were correct by replaying them into the bus, both by recreating it from scratch and replaying them as they appeared in the bus, but we weren't able to affect the display.

6.1.1 Why it didn't work: an educated guess

To understand the reasoning behind the explanation of our failure, we need to take a look at the traffic and keep in mind the CAN packet fields explained in detail at section 3.1.

```
/  time    ID  data ...                    < cansniffer can0 # l=2 h=10 t=50 >
0.199884    1  C2 81 1F AE AA 03 20 9D ...... .
0.199278    3  10 1C 10 00 04 FF E0 B3 ........
0.199915   45  00 FF 00 00 00 00 C0 36 .......6
0.189372   73  28 20 00 32 00 80 B8 9A ( .2....
0.200200   dd  0B 80 00 00 00 00 C0 AA ........
0.219976  105  3F FF FF FF FF 3F D7 07 ?....?..
0.199277  201  00 00 00 00 FF 81 B0 ED ........
0.999976  204  02 30 35 37 FF FF FF FF .057....
0.219988  245  7F D9 7D 7A 7D 7F B0 84 ..}z}...
0.199851  249  1C 1F 00 20 3E 00 FF FF ... >...
0.999942  37d  00 00 03 C1 19 00 FF FF ........
0.199979  39d  FF 02 FF FF 00 00 00 18 ........
0.999848  39f  0D 0A 2D 16 0A 14 12 FF ..-.....
1.003798  3e9  07 E4 0A 16 0B 0A 11 94 ........
4.008174  3eb  00 00 05 13 10 7A 0F D1 .....z..
```

Fig. 6: CAN bus traffic

Fig 6 shows filtered traffic from the bus, only appearing packets that are constantly changing, with the bytes in light gray being the ones that change. As we can see the last two bytes are constantly changing, and those bytes represent the CRC field (check Table 1). This allows manufacturers to use those bytes to create a checksum to secure the packets, which means that there is a function to calculate this field that is unique to the make and model (or at least it is not standardized), and it is not public knowledge. The calculation should be simple enough to take little time, as the packets are from critical systems (as we can see from the low ID value, meaning high priority), so it should be possible to discover how it is calculated if given enough time. However the duration of the project did not allow us to perform this other feat of reverse engineering, so there wasn't any way to send packets without the ECU detecting the wrong checksum and discarding our packets.

Our guess is that being a model made after the studies mentioned in section 4, the manufacturer made sure to take the concerns raised by said studies into account when designing new models that use the CAN bus, so we needed to find another car that was new enough to have a bus and some electronic functions, but older than 2014 (when those studies started coming out).

6.2 Scenario 2

The second car we used to try to breach was a 2010 Toyota Yaris. As explained in the section above we tried to look for an old enough vehicle, which would probably mean it would have a more lax security. This car has a CAN bus network and doesn't perform CRC checks, so it is easier to exploit.

We found ourselves to be more successful in this car when it comes to decoding packets. We were able to decode quite a few of them, which fields meant what (Section 6.2.4), and we managed to perform an injection attack on the RPM display (Section 6.2.1).

All the packets we identified we tried to replay into the bus, however most of them didn't seem to have an effect on the behavior of the car, and the reason is similar enough between all of them that we deemed redundant explaining them individually. Section 6.2.3 explains why we think it didn't work as one would initially expect.

6.2.1 RPM display

After taking a look at the traffic in the bus, we were able to identify the packet that controls the RPM display, and which bytes refer to the value shown on the screen.

Creating the packet from scratch, we can set the RPM value (only on the screen, the car doesn't get revved) to whatever value we desire. To prove that we got the correct bytes and values, we created a simple script that changes the RPM from 1K up to 7K every 2 seconds and repeats the cycle periodically.

Fig. 7: RPM display changed with the car in accessory mode (engine not running)

6.2.2 Speedometer

Another packet we were able to identify and decode is the speed of the vehicle. As we can see in Fig 8 the second byte is a direct representation in Kph of the vehicle's current speed as shown in Fig 8.

Fig. 8: packet showing speed with car in motion

However when we try to send the packet with a different value the speedometer doesn't change at all, instead showing the correct speed no matter what the packet says.

6.2.3 Why it didn't work: an educated guess

Knowing that we are able to change the RPMs without much trouble, we can conclude that the value we get from the packet does not refer to the speed displayed on the speedometer. The question is, what does it stand for? Our best guess is that the speed shown in the packet is used for another system, such as the ABS or traction control, or in a less significant way, the trip's average speed meter. We believe the most likely answer to why we can't affect the vehicle speedometer has to do with the odometer, as manufacturers don't want the mileage of the car to be doctored in order to commit fraud, so it is probably directly wired to the speedometer and odometer display and doesn't go through the bus.

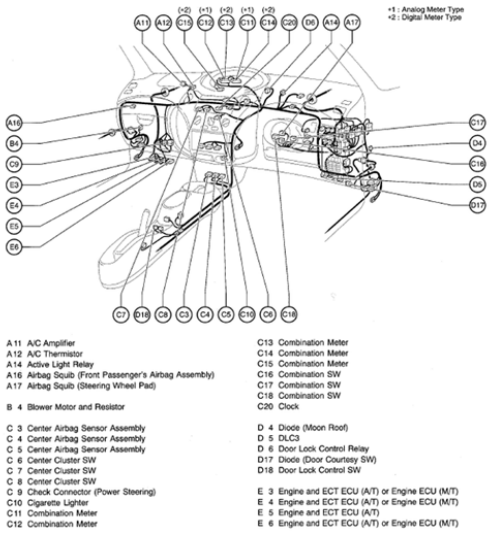

Fig. 9: Dashboard wiring diagram from Toyota

This hypothesis is further strengthened when we look at the wiring diagram (Fig 9), and we can see that where the speed display is located in the dash, it says that it is an analog meter type.

6.2.4 Other identified packets

Following the same procedure from the last packets we obtained, we were also able to find other functions running through the bus, or at least some that trigger a packet to be sent. However similarly to the speedometer, replaying them did not have any effect on the car's behavior, and those are:

- Locking/unlocking the doors
- AC compressor turning on
- Wheel speed (left and right) for ABS module
- Throttle pedal position

Fig. 10: packets mentioned above in order

7 Potential attacks

Having obtained all the information mentioned in all subsections of section 6, we can speculate about how it can be used maliciously. Other than the RPM (section 7.1) all other attacks could not be proven to work, as it would be dangerous to try and perform them outside a track or any kind of regulated environment, which we do not have access to, so they are hypothetical.

7.1 RPM display

The RPM display allows the user to know what the engine speed currently is, however, we can change that as we please, so we could have a program that subtracts some RPMs when the car goes above idle, making the user believe that everything is working as intended, but in reality, the attacker makes him over-rev the engine, causing an early deterioration of the head, pistons, and valves, diminishing the reliability of the car and the engine life expectancy by a while, depending on the abuse that it sustains if the user is not used to the other RPMs indicators (such as vibrations, sound, the velocity at given RPM and other ones that come with experience).

7.2 ABS

As mentioned in section 6.2.4 we were able to identify the individual wheel speed, information that gets fed to the ABS module. The ABS module avoids a wheel to block by disengaging the brake periodically when the speed of one wheel is slower than the contiguous wheel. This happens when the car is slipping because of wet surface or one with lots of debris, or in very hard braking.

Knowing this we could make a program that always sends packets where all 4 wheels are rotating at the same speed, even when not true, tricking the ABS into thinking that the car is not sliding/blocking the wheels, and rendering it useless. This is a very situational attack, however, it will work in a very bad situation, which can make it very dangerous.

7.3 DoS (Denial of Service) attacks on CAN

Probably the simplest software attack on the CAN network is the denial of service [16]. This is thanks to the way the CAN protocol works.

As explained in 2.1, the CAN protocol has a fixed format, and each field contains different information. The field that allows this attack to be so simple is the ID, more specifically the way the IDs are prioritized, the lower the ID value the more priority the packet has. This means that every time an ECU wants to send a packet, it checks the bus and if a packet with a lower ID is sent, it will wait for it to be over and then send the packet. We can exploit this fact by sending packets with an ID of 0 at a high rate, preventing all other transmitters connected to the bus to send their packets, shutting down every functionality of the vehicle that relies on that bus. The contents of the fake packet are irrelevant, as our goal is just to disrupt the normal functionality of the car.

To make this attack meaningful, we can perform it when a certain criterion is met, for example preventing the vehicle from starting and moving, by injecting packets as soon as the OBDII port receives current in the pins, which means that the key is in the car and it is at least in accessory mode. An even more malicious application would be to wait for

the car to reach a certain speed (we already have located the packet and field where speed is shown) to then prevent the vehicle from responding correctly in more dangerous situations. Again this was not tested, as we didn't have the right conditions to do it safely.

It has been shown that a CAN protocol-compliant DoS attack like the one described earlier consume up to 98% of the CAN network bandwidth, as they found in [17]. The same study also proposes a way to mitigate DoS attacks, however they do so by having a separate wireless network performing the checks, which would mean a great increase in price and development for each vehicle, so only simpler methods are actually applied if at all. That means that if the attacker sends packets in a regular rate that matches that of the bus, as well as changes the ID or content of the package each time, the attack can still be successful.

8 Conclusions

Automobiles have been getting increasingly safer with the addition of modern electronic aids, however there can be no safety without security. In this project, we explored the possible attack surfaces of one of these vehicles, and performed successfully a CAN injection attack, as well as discussed the potential effects this attack could have caused in a real scenario.

The idea of the project came when I tried to solve an issue I had in my life when I moved to the United States for a year and had to leave my motorcycle at my house. I asked my brother to turn it on for me every once in a while so the battery wouldn't die, and I could use it as soon as I got back. However my brother took it for a ride and crashed it, so I figured there had to be a better way to automate that, so I searched online for a little while and found out about the CAN bus in the automotive industry. I read a little bit about what had been done and figured I could try it as a senior project. Once involved with the project and done more research, I quickly found that it wouldn't be as straightforward as I foresaw, and this notion ratified itself as I got more hands-on with it.

It was found that some of the objectives were not possible given the resources and time constraints of this project, as O4 was dependant on being able to find the packet that turned the car on, which was not possible. However, the main core of the project was to write on the can bus and affect the vehicle's behavior as detailed in O1. This was accomplished, and even though some parts of the project were left out, such as the turning on of the car periodically, we still felt that it was interesting enough to move forward with it in a similar format we originally planned.

We would consider the project successful in a general sense, even tough he original scope was too ambitious, and some of the objectives were impossible to complete, the project focused on regular CAN messages and which behavior we could affect with them, which proved to be fairly limited, so in future work getting deeper into the diagnostic CAN messages could prove to be extremely interesting and much more fruitful in resulting in meaningful attacks.

Acknowledgments

I would like to thank Dr. Heinrich Foltz and Dr. Emmett Tomai from UTRGV for helping me kicking off the project, and also Víctor García for his help during the development of it. I also want to thank my family and friends for allowing me to fiddle with their cars as well as for their support.

References

[1] The Guardian, 2018, 50 million Facebook profiles harvested for Cambridge Analytica in major data breach, Carole Cadwalladr and Emma Graham-Harrison.

[2] Issues in Information Systems, Volume 19, Issue 3, pp. 150-159, 2018.

[3] 2018. Proceedings of the Internet Measurement Conference 2018. Association for Computing Machinery, New York, NY, USA.

[4] T. Alladi, V. Chamola, B. Sikdar and K. R. Choo, "Consumer IoT: Security Vulnerability Case Studies and Solutions," in IEEE Consumer Electronics Magazine, vol. 9, no. 2, pp. 17-25, 1 March 2020, doi: 10.1109/MCE.2019.2953740.

[5] L. Almeida, P. Pedreiras and J. A. G. Fonseca, "The FTT-CAN protocol: why and how," in IEEE Transactions on Industrial Electronics, vol. 49, no. 6, pp. 1189-1201, Dec. 2002, doi: 10.1109/TIE.2002.804967.

[6] Directive 98/69/EC of the European Parliament and of the Council of 13 October 1998 relating to measures to be taken against air pollution by emissions from motor vehicles and amending, Council Directive (70/220/EEC)

[7] Steve R. Palmer and Mac Felsing. 2001. A Practical Guide to Feature-Driven Development (1st. ed.). Pearson Education.

[8] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. 2018. A Survey of Methods for Explaining Black Box Models. ACM Comput. Surv. 51, 5, Article 93 (January 2019), 42 pages.

[9] IEEE Spectrum, 2009, This Car Runs on Code, Robert N. Charette

[10] K. Koscher et al., "Experimental Security Analysis of a Modern Automobile," 2010 IEEE Symposium on Security and Privacy, Berkeley/Oakland, CA, 2010, pp. 447-462, doi: 10.1109/SP.2010.34.

[11] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. 2011. Comprehensive experimental analyses of automotive attack surfaces. In Proceedings of the 20th USENIX conference on Security (SEC'11). USENIX Association, USA, 6.

[12] Charlie Miller and Chris Valasek. Adventures in Automotive Networks and Control Units. IOActive Comprehensive Information Security, 2014.

[13] Charlie Miller and Chris Valasek. CAN Message Injection. 2018

[14] P. K. Manadhata and J. M. Wing, "An Attack Surface Metric," in IEEE Transactions on Software Engineering, vol. 37, no. 3, pp. 371-386, May-June 2011, doi: 10.1109/TSE.2010.60.

[15] P. Torr, "Demystifying the threat modeling process," in IEEE Security & Privacy, vol. 3, no. 5, pp. 66-70, Sept.-Oct. 2005, doi: 10.1109/MSP.2005.119.

[16] C. Meadows, "A formal framework and evaluation method for network denial of service," Proceedings of the 12th IEEE Computer Security Foundations Workshop, Mordano, Italy, 1999, pp. 4-13, doi: 10.1109/CSFW.1999.779758.

[17] W. Si, D. Starobinski and M. Laifenfeld, "Protocol-Compliant DoS Attacks on CAN: Demonstration and Mitigation," 2016 IEEE 84th Vehicular Technology Conference (VTC-Fall), Montreal, QC, 2016, pp. 1-7, doi: 10.1109/VTCFall.2016.7881182.

APPENDIX

A.1 Work planning

- Task 0: Documentation and reports Throughout the project we will need to design tests and procedures, as well as det the results of those tests, so to keep track of our progress and to be able to recreate it later in a rigorous manner we will need to keep a record of all the information we gather, as well as making reports about the successfulness of our testing, and finally a final report detailing all the project's obstacles and conclusions.
- Milestone 1 (M1): Reverse engineering of a packet completed
- Task 1.1: Get the right setup The OBDII port gives access to standardized parameters, for that reason there is a lot of available software that can display it without much complication. To be able to access those parameters, the OBDII port is connected to the CAN bus (or other architectures underneath), where the modules feed it the information. This is the vulnerability we are going to exploit, and for that we are going to need a very low-level software, so we can interact with the packets in the medium (2nd layer of communication). For phase 1 the software used throughout the project will need to be capable enough to read packets other than the ones offered by the OBDII standard. It also needs to be flexible and easily usable and have some functionalities that allow packets to be read, recorded, created/replayed and written in the bus.
- Task 1.2: Design test There isn't a set in stone way to reverse engineer a piece of equipment, and in this study it is particularly tricky as the car modules cannot be accessed physically (not possible to remove ECUs from the vehicle), and the source code for them is not accessible. Effectively we are working on a black box that only has inputs and outputs that we can observe (inputs being replayed packets, outputs the packets sniffed out of the bus). To prove that we have interpreted the fields of the packet correctly, we can use physical inputs to the car, the easiest critical system would be the throttle.
- Task 1.3: Log the results and interpret the packets Probably be the longest and hardest one to achieve, and should take a lot of time, as it involves a lot of trial and error. If we set the throttle to a certain position, the RPM gauge should remain in a certain range. Knowing that information, we can predict the packet fields. We should be able to understand the fields of it and modify them in order to replay it in the bus and consequently affecting the vehicle's behavior
- Milestone 2 (M2): Send packets to the bus successfully
- Task 2.1: Recreate the packet An easy functionality to modify and prove it did would be once again the RPM gauge, setting it to a certain value for a certain amount of time without any input to the vehicle, only through the bus.
- Task 2.2: Research spoofing methods Find out which spoofing methods there are, and narrowing them down to the ones that we can use in this project.
- Task 2.3: Design the tests for the different spoofing methods To be able to do that we have to find a good way to trick the receiver into ignoring the real packets, and only reading our fake ones. There are two options suited for this, first is to send the fake packet at a much higher frequency than the real one, so even if the module receives the real value, shortly after will get our desired one. The other option is to wait for the packet to appear in the bus, and then send a bunch of fake packets synchronized with the real one.
- Task 2.4: Decide the best spoofing method After running some tests, we should be able to evaluate the different spoofing methods and select the one that works best for the project's goals
- Milestone 3 (M3): Turn on the car from the computer consistently
- Task 3.1: Design tests Pretty similar to M1 and M2, but the target packet would be harder to find, as it is not constantly being broadcasted (only sent when the button is pressed). The best way to find the right packet is to record a bunch of instances of the data running on the bus as the vehicle is started, and then search for the packets that appear in all instances. Then discard packets that are not coming from the ECU if information is found about the identifiers.
- Task 3.2: Log the packets Logging the traffic from the bus as we start the car multiple times should allow us to narrow down the packet we need to recreate, so using a program, we can log all those instances of the car turning on to then evaluate.
- Task 3.3: Discover and replay the packet that turns on the car Analyzing the logs and using trial and error, we can find the packet that the ECU recognizes as the keyless ignition switch, and then using the same method as before, replay it into the bus.
- Milestone 4 (M4): Create a program to periodically turn on the car
- Task 4.1: Plan the functionalities The functionalities are very flexible, given that time is our main constraint. The functionalities need to be planned keeping the deadline in mind, and molding them to the feasibility of them being completed.
- Task 4.2: Write the code for the functionalities
- Task 4.3: Design the UI The UI is secondary compared to the features, and is also dependent on the flow of the project, and as such should be mindful of the time at hand.
- Task 4.4: Implement the UI

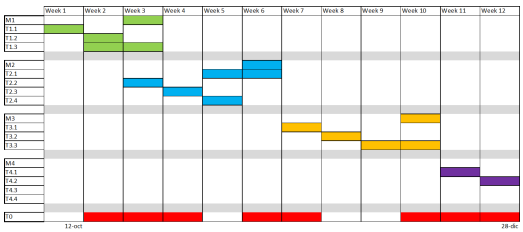

Fig. 11: Gantt chart of the work plan

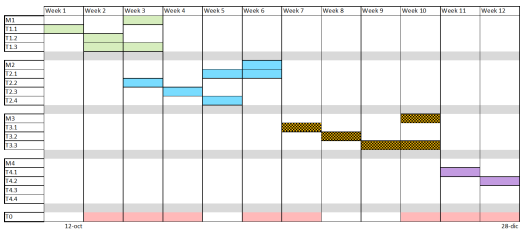

Fig. 13: Gantt chart of the work plan at the end of the project

A.2 Work updates

A.2.1 Work planning update (I)

Currently in week 4 of the schedule, and tasks 1.1 and 1.2 have been completed, however we are having some difficulties with task 1.3, as it seems that the packets intercepted have a CRC or checksum of some kind, which makes the replay of those packets more difficult, and in turn testing if the reverse engineering results are correct. However, if this problem is resolved by week 6 (when task 2.1 needs to be completed), we could still be on time in this exact schedule.

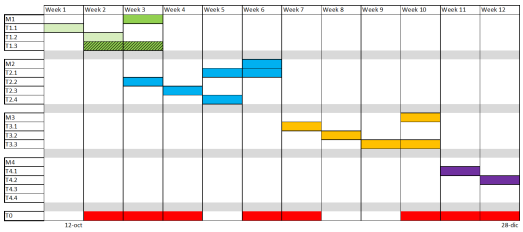

Fig. 12: Gantt chart of the work plan updated

A.2.2 Work planning update (II)

Currently on week 10 we have hit a brick wall, the various methods of replaying packets that we have tried haven't been successful, and even though we are almost certain that we have identified the IDs of some of the car's functionality we haven't been able to affect the vehicle's behaviour by injecting packets into the bus. The reason for this is the security implemented by the manufacturer is complex enough for it to take more time to unravel than time we got to complete the project (see appendix A1) . For this reason, we decided to use an older car, in hopes that the security is more lax.

A.2.3 Work planning update (III)

Working on an older car has proven to be more fruitful as there are less security measures in the network, however it also limits the amount of systems that use the bus, so the number of functions of the vehicle we are able to affect is diminished. As we found during the experiments, we were only able to affect the RPM display, and other functionalities have proven to be unalterable, which has proven our original objectives to be a little bit too ambitious.

The whole milestone referring to turning on the car had to be scratched, and the program we wrote to affect the vehicle changed from turning it on to spoofing the RPM display.