Oct 12th, 1:00 PM - 1:25 PM

Automated Reverse Engineering of Automotive CAN Bus Controls

Charles Barron Kirby
University of North Georgia, cbkirb0968@ung.edu

Bryson Payne
University of North Georgia, bryson.payne@ung.edu

Abstract
This research provides a means of automating the process to reverse engineer an automobile's CAN Bus to quickly recover CAN IDs and message values to control the various systems in a modern automobile. This approach involved the development of a Python script that uses several open-source tools to interact with the CAN Bus, and it takes advantage of several vulnerabilities associated with the CAN protocol. These vulnerabilities allow the script to conduct replay attacks against the CAN Bus and affect various systems in an automobile without the operator's knowledge or interaction.

These replay attacks can be accomplished by capturing recorded network traffic and resending them to find which traffic conducts certain actions. Automobiles are becoming more reliant on computer systems and networks to operate, including the integration of wireless interfaces to interact with these systems (Avatefipour & Malik, 2018). These systems contain numerous vulnerabilities as they were not built with consideration to hacking (Wolf, Weimerskirch, & Paar, 2004). Creating a tool to automate the reverse engineering process allows for a better understanding of the CAN Bus and its vulnerabilities. The aim of this script is to allow the user to identify what specific packets captured from CAN Bus traffic will initiate selected actions in the automobile's controls. The results show the user can repeatedly split and send log files to the CAN Bus to narrow down the files to a single packet that is starting the selected outputs of the CAN Bus using this script.

Location
KSU Center Rm 400

Disciplines
Automotive Engineering | Information Security | Management Information Systems | Technology and Innovation

Comments
The authors are grateful to the reviewer, and have addressed the reviewer's comments and suggestions. Two additional references are added with four previous examples of automotive hacks in recent literature at the reviewer's suggestion. A photo of the Korlan USB2CAN device is added as Figure 1 per the reviewer's comment. We have added a note on the future work we have begun by capturing data from three of four vehicles successfully to date. The only recommendation we did not take was changing the name of the paper, rather, we added two sections explaining the reverse engineering process in the paper.

INTRODUCTION

Computer systems are becoming increasingly integrated into modern cars to improve their performance and safety. This results in an increasing scale and complexity of the systems supporting these automobiles. Modern cars are currently built upon several millions of lines of code in their software to run these systems (Charette, 2009). Electronic Control Units (ECUs) are used to monitor sensors and to control components of an automobile such as turn signals, doors, and the speed of the car (Charette, 2009).

While modern cars have been greatly benefited by these advancements, these systems are vulnerable to malicious attacks, because there have been few safeguards added to prevent them from occurring (Wolf, Weimerskirch, & Paar, 2004). The vulnerabilities of the CAN bus include the lack of any encryption, allowing the attacker to view these packets in plaintext, thus allowing attackers to capture packets from the CAN bus to send back later (Pan, Zheng, Chen, Luan, Bootwala, & Batten, 2017). Another serious vulnerability is the lack of authentication inside the network, because without authentication an attacker can conduct repay attacks without having to appear as a legitimate source (Wolf, Weimerskirch, & Paar, 2004). This lack of encryption and authentication permits even an attacker with no prior knowledge of the particular automobile's controls to reverse-engineer the CAN bus to determine message IDs and data values to interfere with virtually any system connected to that vehicle's CAN bus.

Significant prior work has been published on hacking particular vehicles, from a 2011 Chevy Malibu (Checkoway et al., 2011) to the 2015 Jeep Cherokee (Miller & Valasek, 2018) to 2016 Tesla and 2018 BMW hacks (Payne, 2019). However, in each case, a researcher must do extensive investigative work hands-on with each vehicle, manually reverse-engineering each CAN bus ID and message value for the various controls of each system in the vehicle. And CAN IDs and data values vary not only across manufacturers, but between models from the same manufacturer, and even between the same model across different model years. The purpose of this paper is to provide an easy-to-use application capable of automating much of the process, assisting the researcher in quickly discovering and documenting CAN IDs and data values for relevant control systems in any automobile to which the researcher has physical access.

Besides a direct physical connection to the on-board diagnostic type two (OBD-II) port, wireless connections can be made to modern vehicles by using external interfaces such as Bluetooth, 4G LTE, and GPS (Avatefipour & Malik, 2018). This makes the vulnerabilities of the CAN bus more problematic, as it expands the possible methods attackers can use to access the car's CAN bus (Avatefipour &

Malik, 2018). The Python script developed in this paper exploits these vulnerabilities to allow a researcher, or an attacker, to reverse engineer the CAN bus inside the majority of automobiles sold in the US in the past twenty years.

SETTING UP THE ENVIRONMENT

The process begins with connecting to the car using an OBD-II converter, such as a $40 CANtact CAN to USB converter, or a slightly more expensive $70 Korlan USB2CAN adapter like the one shown in Figure 1 (Kirby, 2019).

Figure 1: The Korlan USB2CAN adapter ($70 USD) connects a laptop or desktop computer to the OBD-II port in most cars and trucks sold in the US since 1996.

VirtualBox or VMware can be used to create a Kali or Ubuntu Linux virtual machine in order to create an environment suitable for intercepting and replaying CAN traffic (Payne, 2019). Once a Linux VM is installed, the proper tools needs to be installed so the script can be run. Fortunately, the user only needs to install *can-utils*. Can-utils is an open-source Linux tool that allows the user to interact with the CAN bus by issuing commands on the terminal. This allows the user to interact with the CAN bus by sending individual packets, sending entire log files, and display CAN bus traffic. Can-utils can be installed with the following terminal command:

```
sudo apt-get install can-utils
```

Next, the user needs to use an OBD-II Adapter cable, a CANtact v1.0 CAN to USB Converter, and A-Male to B-Male USB 2.0 cable to hook up to the car. The OBD-II Adapter is plugged into the OBD-II port of the vehicle. The OBD-II port is commonly located beside or below the steering wheel. In order to use the CANtact

or Korlan converter to look at the car's CAN bus traffic, the *slcand* daemon needs to run and a CAN interface should be enabled using the following commands:

```
sudo modprobe can
sudo slcand /dev/ttyACM0 can0
sudo ip link set up can0 type can bitrate 500000
```

Issuing a `candump` command like the one below helps capture all traffic in the CAN bus and inserts it into a log file for the user to use in the script:

```
candump -L can0 > action.log
```

It is important to note that the `candump` command must be issued before doing anything in the car that the user wishes to record to the log file, otherwise the actions will not be captured. More than one action can be performed in a single candump. For example, the user can turn on the headlights, use both turn signals, turn on the radio and air conditioning, press the brake pedal, and rev the engine during a single logging session. The same log file can then be replayed to determine the CAN message IDs and data values for each of the controls captured, one at a time. The user can press CTRL-C to stop the capture.

In many cases, a control will remain active after it is initiated (like turning on the headlights or radio), so a second short candump capture is required to serve as a "baseline" to reset the control to its off state:

```
candump -L can0 > baseline.log
```

After issuing this command, the user can turn off the relevant controls—including turning off more than one system at a time. This baseline file will be replayed after each successful replay of the first log file to turn off the appropriate controls for each successive round of refinement. (Again, press CTRL-C to stop the capture.)

IMPLEMENTATION

A Python script is used to automate the reverse engineering process by sending repeated replay attacks using a binary search to replay recursively smaller halves of the log file until only a single CAN bus message remains. Python is used because it was faster and more flexible to design the script with and the requirements do not demand the fast execution of compiled languages.

The script only needs the user to provide two log files: the log file containing captured traffic with the original output and a log file to toggle off any functions turned on by the previous log file. The syntax to run the script is:

```
python CANReverseEngineer.py action.log baseline.log
```

where *action.log* is the log file containing the actions the user wishes to reverse-

engineer into CAN IDs and message values, and *baseline.log* is the log file containing a clear (off) signal for the same control.

The script uses a straightforward binary search approach, cutting the log file into halves and playing these halves for the user across the CAN bus cable to the vehicle, or to the emulator if preferred. After a half is played, the user decides if the file sent the output the user is looking for (like brake lights, radio turned on, turn signals, etc.). If the desired control is activated, that file's half is split into smaller halves and those halves are played until there is only one line remaining, containing the command that generates the desired output.

If the first half is played with no desired output the user can play the second half to find if the desired output is in the second half instead. If the user finds the desired output from the car, the half that was played is set as the file to be split again at the start of the while loop, the number of lines is updated for the new half-file, and the file counter is incremented to match the number of times the file was split. But, if at any point both halves are played and no desired response is found, the script will stop, requiring the researcher to start from the original file, or possibly recapture the CAN bus data from the source.

Once a log file with only one line is generated, the while loop stops, and the script returns the name of the log file that contains the packet the user is looking for. The user can check the final log file to see the packet that triggers the desired output, and they can check if it really gives this desired output by sending the following command:

```
cansend [packet from final log file]
```

A binary search method to split the files into halves and run those halves was used for this script. The script checks for an even number of lines before splitting in half. If the number of lines in the log file is not even, the middle of the file is pushed to one more line before spitting. A variable that increments after each cycle through the while loop is used to keep track of how many times the file has been split and is used as part of the naming convention for newly split log files. The naming convention goes **x** followed by the number of files played followed by **aa** if it's the first half or **ab** if it is the second half. For example, file "**x1aa**" would be from the first time a file was split and is the original file's first half. The code below demonstrates how the files are split:

```
if (numberOfLines % 2 == 0):
        os.system("split -l %i %s x%i" % (findMiddleOfFile(file), file,
fileCounter))
        firstHalf = "x%iaa" % fileCounter
        secondHalf = "x%iab" % fileCounter
```

```
elif (numberOfLines % 2 == 1):
            os.system("split -l %i %s x%i" %
(findMiddleOfFile(file)+1, file, fileCounter))
            firstHalf = "x%iaa" % fileCounter
            secondHalf = "x%iab" % fileCounter
```

A line of code is presented below showing the terminal command to replay a log file across the CAN bus.

```
canplayer -I [file name]
```

The current implementation inside the Python script wraps this terminal command in a Python `os.system()` command that sends this command automatically through the Linux terminal without user interaction (Kirby, 2019).

Any function of the car that toggles on and off requires the function to be turned off before commands to turn on the feature can be used again, so a file of captured traffic containing packets to turn off the turn signals is used as well. The command to play the log file is placed inside of a `for` loop to repeat the command multiple times. The current implementation is seen in the code below:

```
for x in range(0, 5):
      os.system("canplayer -I %s" % firstHalf)
os.system("canplayer -I %s" % baseline)
```

The final resulting line of a CAN log file after several rounds of halving the file and indicating "Y" or "n" should look like the one below. The featured CAN message shown here is the command used to turn on the right side turn signal inside a simulated CAN network:

```
vcan0 188#02000000
```

Finding the packets that trigger the acceleration and deceleration in a CAN bus can be more difficult, as they do not represent a simple toggle on and off like turn signals (Payne, 2019). This means the response will be harder to see by the user as the responses will mean smaller upward movements from the speedometer. Finding harder to detect desired outputs can be done with a `for` loop as it replays the output multiple times, so it gives the user multiple chances to detect the output (Kirby, 2019). To take full advantage of this script, it is recommended that the user create a database to hold CAN bus data of reverse-engineered cars to use at a later time.

RESULTS

The resulting seventy-line Python script can successfully reverse engineer many of the functions controlled by an automobile's CAN bus. The user interaction required

is limited to capturing the original CAN bus log with the desired controls activated and a "baseline" log with all controls turned off, then selecting "Y" or "n" to indicate whether the automated analysis script was able to activate the control during the replay phase. Using a binary search approach to split the files is fast. However, the speed that the script can narrow down the log files to one line varies by the log (base 2) of the number of lines in the capture file.

Another factor to consider is user error. If the user misses any input coming from the CAN bus, the user needs to delete all the files generated by the script and restart with the original log file. Many functions were easy to discover with the script including functions that toggled on and off such as the turn signals, door locks, and so on (Kirby, 2019). We have begun testing on a range of production automobiles to amass a database of CAN IDs and message values for future work, and successfully captured control signals in three of four vehicles tested (a 2006 Volkswagen Jetta failed to connect successfully to the laptop). The user can successfully discover which arbitration ID belongs to a particular control along with the data field values that can be manipulated to send different commands.

SOURCE CODE

```
from subprocess import check_output
import os, sys


"""
    --CANReverseEngineer.py--:
        Reverse engineers CAN bus systems
             to find a specific CAN bus ID and
             message values corresponding to a
             particular automitive control msg
             by automating replay attacks.
"""


# Global Variables
# if arguments provided at command line, set log and baseline files
if len(sys.argv) > 2:
       log = sys.argv[1]
       baseline = sys.argv[2]
else:
```

```
      print("Usage: python CANReverseEngineer.py {logfilename}
{baselinefilename}")
      print("-----------------------------------------------------------
---------")
      print("      Where {logfilename} is a candump of the desired CAN
messages,")
      print("      and {baselinefilename} is candump of all controls
turned off.")



def countNumberOfLines(file):
    return int(check_output(["wc", "-l", file]).split()[0])

def findMiddleOfFile(file):
    return int(countNumberOfLines(file)/2)

#Sends automated replay attack against CAN bus
def sendReplayAttack(file):

    numberOfLines = countNumberOfLines(file)
    fileCounter = 1
    correctOutput = True

    while(numberOfLines > 1):
        #Splits files
        if(numberOfLines % 2 == 0):
            os.system("split -l %i %s x%i" %
(findMiddleOfFile(file),file,fileCounter))
            firstHalf = "x%iaa" % fileCounter
            secondHalf = "x%iab" % fileCounter
        elif(numberOfLines % 2 == 1):
            os.system("split -l %i %s x%i" %
(findMiddleOfFile(file)+1,file,fileCounter))
            firstHalf = "x%iaa" % fileCounter
```

```
            secondHalf = "x%iab" % fileCounter

        print("[*]Playing first half of %s" % file)
        for x in range(0, 5):
            os.system("canplayer -I %s" % firstHalf)
        os.system("canplayer -I %s" % baseline)

        answer1 = input("[*]Did the first half %s send correct output?
[Y/n]?" % file)

        if(answer1 == "Y"):
            file = firstHalf
            numberOfLines = countNumberOfLines(file)
            fileCounter+=1
        elif(answer1 == "n"):
            print("[*]Sending second half of %s" % file)
            for y in range(0, 5):
                os.system("canplayer -I x%iab" % fileCounter)
            os.system("canplayer -I %s" % baseline)

            answer2 = input("[*]Did the second half of %s send the
correct output?? [Y/n]?" % file)

            if(answer2 == "Y"):
                file = secondHalf
                fileLength = countNumberOfLines(file)
                fileCounter+=1
            elif(answer2 == "n"):
                print("[*]Desired output is not in file.")
                correctOutput = False
                break

    if(correctOutput == True):
```

```
        print("[*]%s contains correct output" % file)


#Executes script
sendReplayAttack(log)
```

REFERENCES

Avatefipour, O., & Malik, H. (2018). State-of-the-Art Survey on In-Vehicle Network Communication (CAN-Bus) Security and Vulnerabilities. Retrieved from http://search.ebscohost.com.libproxy.ung.edu/login.aspx?direct=true&db=edsarx&AN=edsarx.1802.01725&site=eds-live&scope=site

Charette, R. N. (2009, February 01). This Car Runs on Code. Retrieved from https://spectrum.ieee.org/transportation/systems/this-car-runs-on-code

Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., ... & Kohno, T. (2011, August). Comprehensive experimental analyses of automotive attack surfaces. *USENIX Security Symposium* (pp. 77-92). http://www.autosec.org/pubs/cars-usenixsec2011.pdf

Kirby, C. B. (2019, March). Automating Reverse Engineering of Automotive Networks. UNG 24th Annual Research Conference, Poster Session. March 22, 2019.

Miller, C. & Valasek, C. (2018). Securing Self-Driving Cars. Presentation at Black Hat USA 2018. Retrieved September 9, 2019 from http://illmatics.com/securing_self_driving_cars.pdf

Pan, L., Zheng, X., Chen, H. X., Luan, T., Bootwala, H., & Batten, L. (2017). Cyber security attacks to modern vehicular systems. *Journal of Information Security and Applications, 36,* 90–100. https://doi-org.libproxy.ung.edu/10.1016/j.jisa.2017.08.005

Payne, B. R. (2019). Car Hacking: Accessing and Exploiting the CAN Bus Protocol. *Journal of Cybersecurity Education, Research and Practice*, *2019*(1), 5.

Wolf, M., Weimerskirch, A., & Paar, C. (2004). Security in Automotive Bus Systems. Retrieved from http://www.weika.eu/papers/WolfEtAl_SecureBus.pdf