

# Car Hacking Lab

CPS and IoT Security 2022/23 - University of Padua

Denis Donadel - [donadel@math.unipd.it](mailto:donadel@math.unipd.it)

---

Today, when you drive a car, there's nothing that is not mediated by a computer. And at the core of all this is the Controller Area Network or simply called CAN or sometimes CAN Bus, a central nervous system of a car responsible for intravehicular communication. This tutorial is going to be a guide for car hacking as safely as possible on reverse engineering CAN packets.

To practice CAN-Bus exploitation we will be using an ICSim package from Craig Smith. ICSim includes a dashboard with speedometer, door lock indicators, turn signal indicators and a control panel. The control panel allows the user to interact with the simulated automobile network, applying acceleration, brakes, controlling the door locks and turn signals.

## Remarks on CAN bus

Controller Area Network aka CAN is the central nervous system that enables communication between all/some parts of the car. In simple terms, CAN allows various Electronic units in cars to communicate and share data with each other. The main motive of proposing CAN was that it allowed multiple ECU to be communicated with only a single wire.

A car can have multiple nodes that are able to send and/or receive messages. This message consists of essentially an ID, which is a priority of the message and also it can contain CAN message that can be of eight bytes or less at a time.

If two or more node begins sending messages at the same time, the messages sent with the dominant ID will overwrite with that of less dominant. This is called priority-based bus arbitration. **Messages with numerically smaller value IDs are a higher priority and are always transmitted first.** This is how a node detects that higher priority messages are being placed on a bus. For instance, message from Brakes has a higher priority than a message from the audio player.

**Note down that, Lowest ID = Highest Priority.**

If two or more node begins sending messages at the same time, the messages sent with the dominant ID will overwrite with that of less dominant.

CAN bus consists of two different wires. As it is a bus, multiple devices can be connected to these wires. A CAN frame has 3 major parts:

- Arbitration Identifier
- Data Length Code
- Data field

Let's have a look at the CAN data frame:

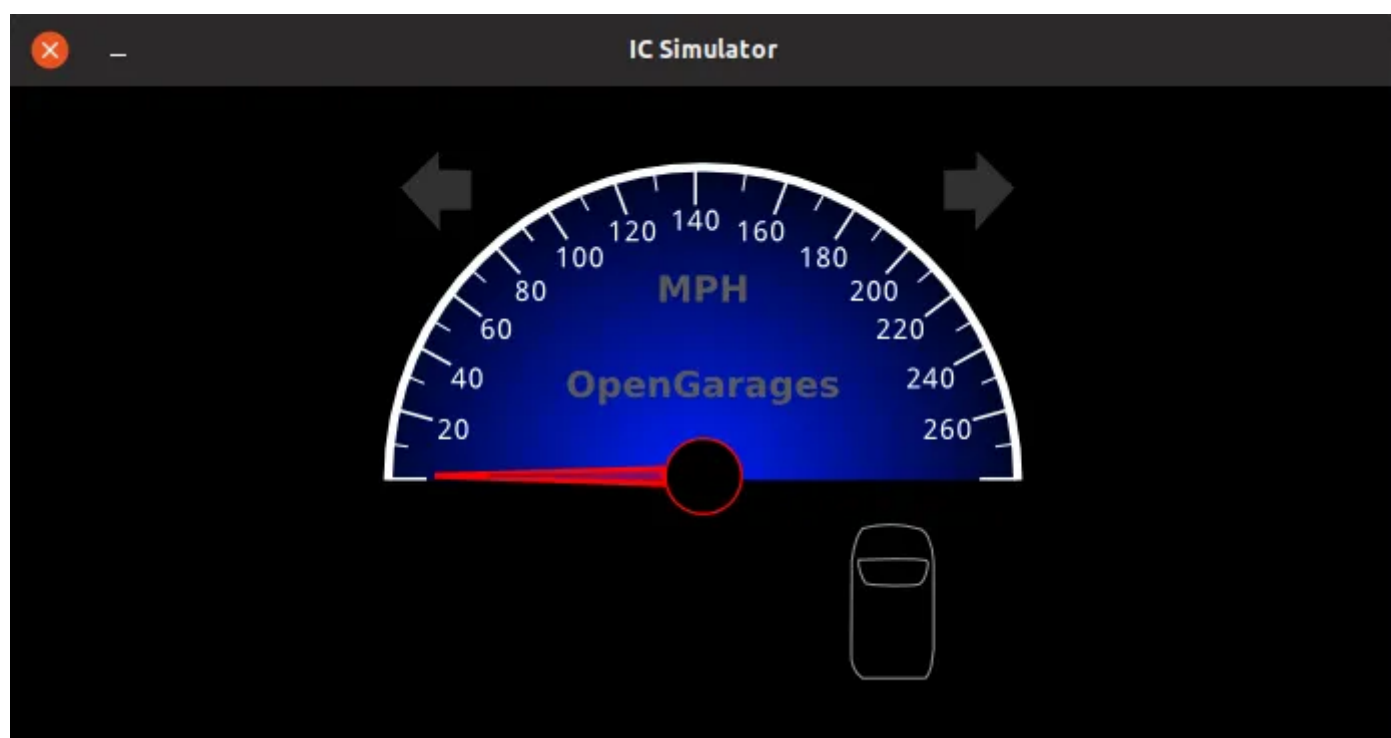


## Part 1: Setting Up the Virtual Lab

You would need:

- Any Linux distributions (tested on debian-based distros)
- can-utils
- ICSim (ICSim is an opensource Instrumentation Cluster Simulator)

Can be downloaded from <https://github.com/zombieCraig/ICSim>



The best and inexpensive way to practice car hacking is by running an instrumentation cluster simulator. Thanks to Craig Smith and his open-source repo called ICSim. Using ICSim, it's pretty easy to set up and inexpensive to learn CAN-Bus exploitation.

Let's do the setup.

SDL is a cross-platform development library for computer graphics and audio. Since ICSim draws and animates a virtual dashboard, this is required. This can be installed via apt-get.

```
sudo apt install libsdl2-dev libsdl2-image-dev -y
```

In order for us to send, receive and analyze CAN packets, we need CAN utils. can-utils is a Linux specific set of utilities that enables Linux to communicate with the CAN network on the vehicle. The canutils consist of 5 main tools that we use very frequently:

- **cansniffer** for sniffing the packets
- **cansend** for writing a packet
- **candump** dump all received packets
- **canplayer** to replay CAN packets
- **cangen** to generate random CAN packets

can-utils can be installed via apt-get

```
sudo apt install can-utils -y
```

The you need to download the Instrumentation Cluster Simulator

Instrumentation Cluster Simulator is used to generate the simulated CAN traffic.

This can be downloaded by cloning the project via a git repository.

```
git clone https://github.com/zombieCraig/ICSim
cd ICSim
```

## Preparing the Virtual CAN Network

Once you navigate inside the ICSim directory, there's a shell script called **setup\_vcan.sh**

```
y0g3sh@y0g3sh:~/ICSim$ cat setup_vcan.sh
sudo modprobe can
sudo modprobe vcan
sudo ip link add dev vcan0 type vcan
sudo ip link set up vcan0
y0g3sh@y0g3sh:~/ICSim$
```

The modprobe command here is used to load kernel modules like can and vcan modules. The last two lines will create a vcan0 interface in order to simulate the car network.

You can run the following commands to set up a virtual can interface

```
./setup_vcan.sh
```

To verify vcan0 interface, `ifconfig vcan0` (or `ip a show vcan0` in novel Ubuntu versions) will show

```
ubuntu@y0g3sh:~/ICSim$ ifconfig vcan0
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 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

ubuntu@y0g3sh:~/ICSim$
```

Furthermore, you need to build the simulator. To do so, from the ICSim folder, run:

```
make
```

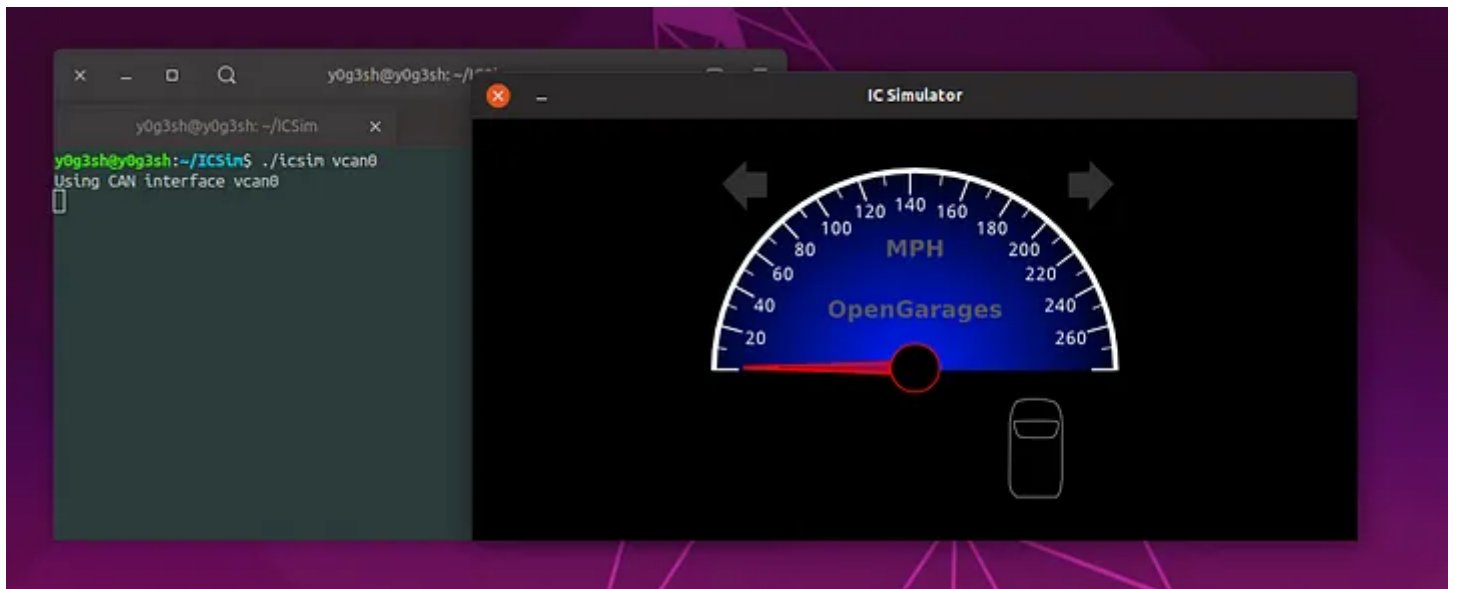
## Part 2: Running the Simulator

Now it's time to run the simulator. Running the ICSim simulator at least requires two components. *A dashboard and a controller* to simulate the acceleration, brakes, controlling the doors, turn lights, etc. You would require at least 3 terminal windows/tabs for these to run. We would require these terminals to run dashboard, controller and another for running can-utils.

### Running the dashboard

In order to run the dashboard, you could run a file called **icsim** with an argument **vcan0**, the interface we created earlier.

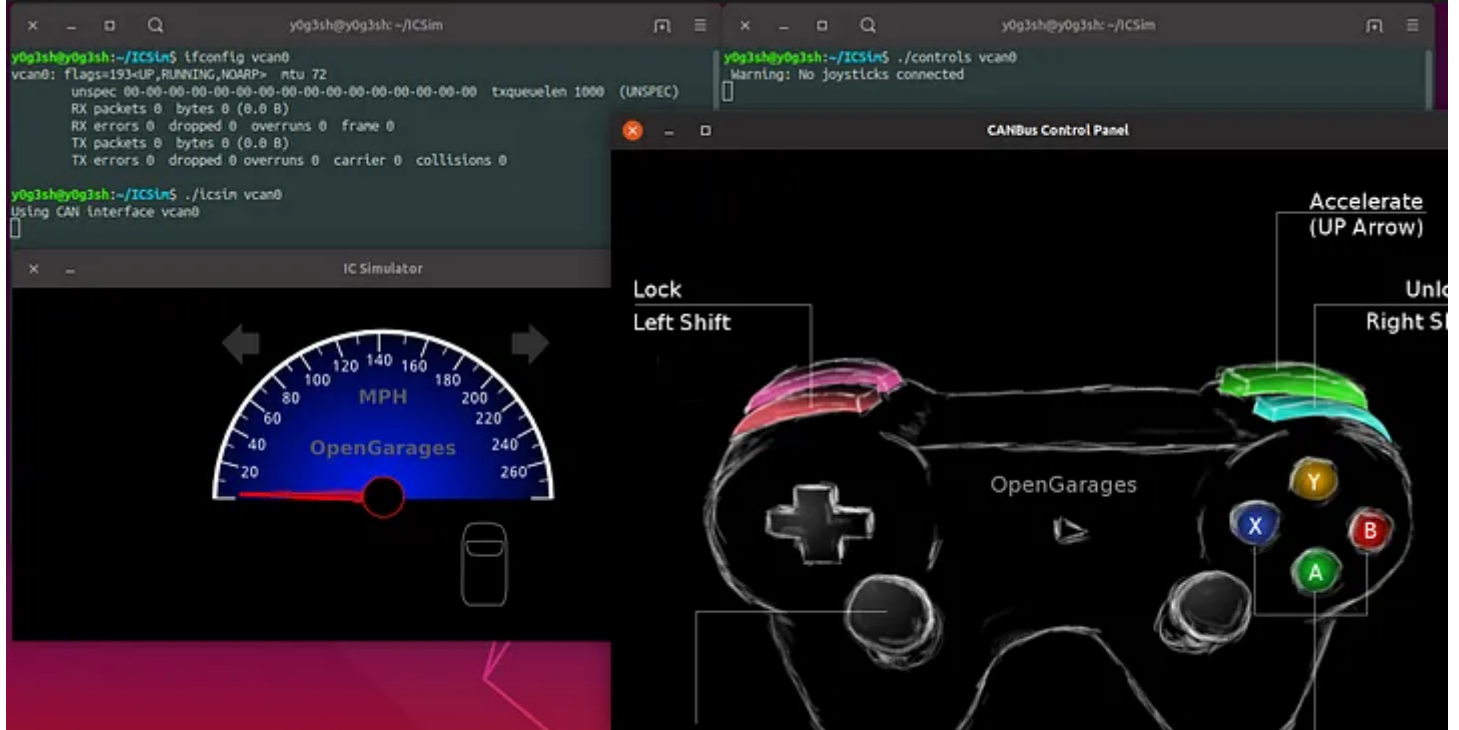
```
./icsim vcan0
```



At this point in time, the dashboard won't be functioning including speedometer, turn lights, brakes or doors. It is because there is no traffic on the interface vcn0. In order to simulate traffic on interface **vcan0**, we need to start the controller in another terminal window.

Control Panel can be started by

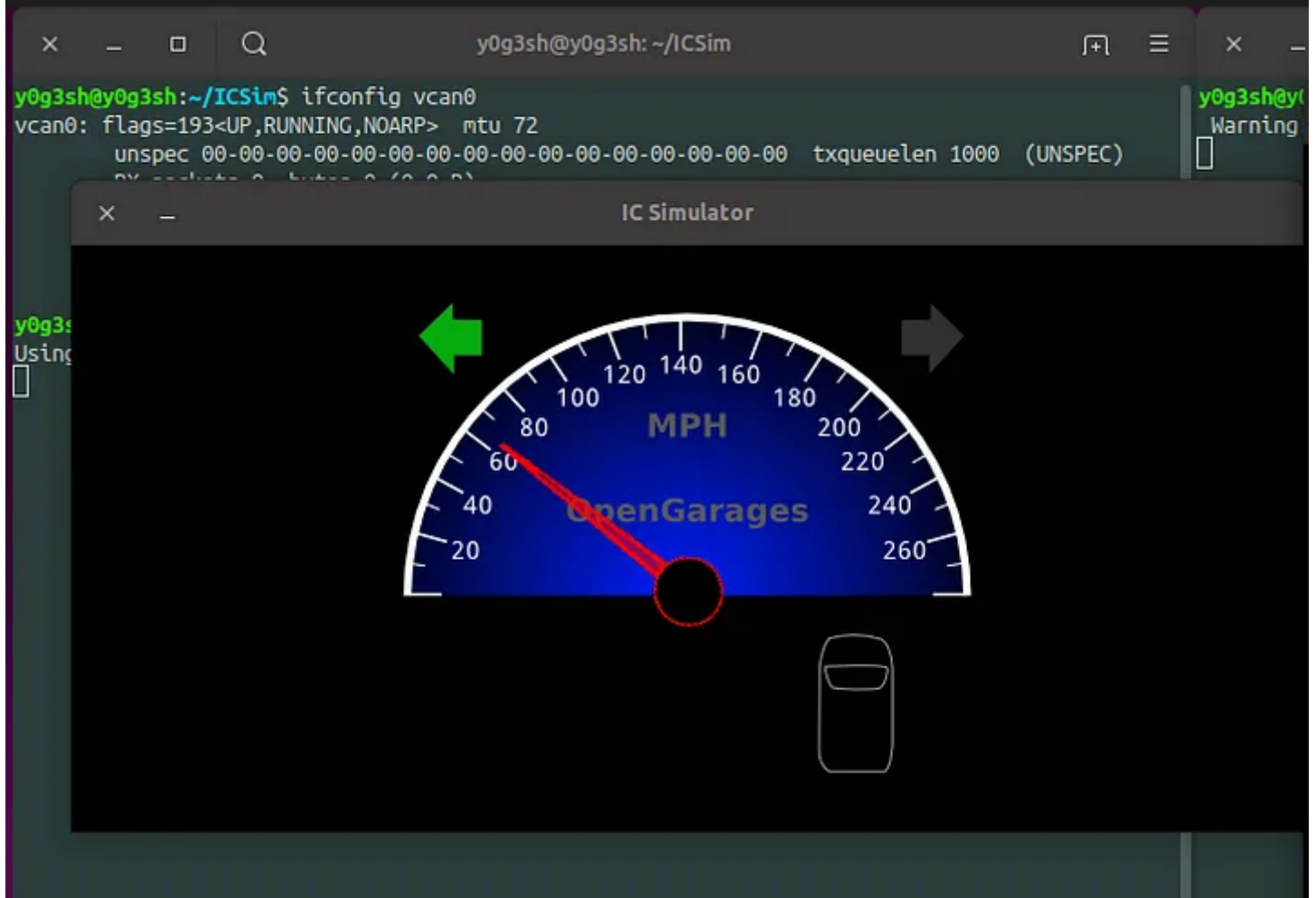
```
./controls vcan0
```



The vcan0 is the virtual CAN interface through which our ICSim will send and receive CAN frames. As soon as you start the control panel, you can observe speedometer making some fluctuations. This is because of the noise simulated by the control panel. Once the control panel has been started, you can use keyboard keys in order to simulate traffic. Using the key combinations below, you can make changes in the ICSim Dashboard.

ICSim Actions	Keys
Accelerate	Up Arrow (↑)
Left/Right Turn Signal	Left/Right Arrow (←/→)
Unlock Front L/R Doors	Right-Shift+A, Right-Shift+B
Unlock Back L/R Doors	Right-Shift+X, Right-Shift+Y
Lock All Doors	Hold Right Shift Key, Tap Left Shift
Unlock All Doors	Hold Left Shift Key, Tap Right Shift

Once you press the up arrow key and left arrow key, this is what you can observe.



---

## Part 3: Exploiting CAN-Bus using Instrument Cluster Simulator

For this first part, you do not need ICSim, so you can close it and reopen it when requested.

How does a CAN message looks like?



```

vcan0 166 [4] D0 32 00 18 .....'.2..'
vcan0 158 [8] 00 00 00 00 00 00 00 19 .....
vcan0 161 [8] 00 00 05 50 01 08 00 1C '...P....'
vcan0 191 [7] 01 00 90 A1 41 00 03 '....A..'
vcan0 133 [5] 00 00 00 00 A7 .....
vcan0 136 [8] 00 02 00 00 00 00 00 2A .....*'
vcan0 13A [8] 00 00 00 00 00 00 00 28 .....(
vcan0 13F [8] 00 00 00 05 00 00 00 2E .....
vcan0 164 [8] 00 00 C0 1A A8 00 00 04 .....
vcan0 17C [8] 00 00 00 00 10 00 00 21 .....!'
vcan0 18E [3] 00 00 6B '..k'
vcan0 1CF [6] 80 05 00 00 00 3C '.....<'
vcan0 1DC [4] 02 00 00 39 '...9'
vcan0 183 [8] 00 00 00 0E 00 00 10 2B '.....+'
vcan0 143 [4] 6B 6B 00 E0 'kk..'
vcan0 039 [2] 00 39 '.9'
vcan0 095 [8] 80 00 07 F4 00 00 00 17 .....
vcan0 1A4 [8] 00 00 00 08 00 00 00 10 .....
vcan0 1AA [8] 7F FF 00 00 00 00 67 11 '.....g.'
vcan0 1B0 [7] 00 0F 00 00 00 01 57 '.....W'
vcan0 1D0 [8] 00 00 00 00 00 00 00 0A .....
vcan0 166 [4] D0 32 00 27 '.2.'
vcan0 158 [8] 00 00 00 00 00 00 00 28 .....(
vcan0 161 [8] 00 00 05 50 01 08 00 2B '...P...+'
vcan0 191 [7] 01 00 90 A1 41 00 12 '....A..'
vcan0 133 [5] 00 00 00 00 B6 .....
vcan0 136 [8] 00 02 00 00 00 00 00 39 .....9'
vcan0 13A [8] 00 00 00 00 00 00 00 37 .....7'
vcan0 13F [8] 00 00 00 05 00 00 00 3D .....='
vcan0 164 [8] 00 00 C0 1A A8 00 00 13 .....
vcan0 17C [8] 00 00 00 00 10 00 00 30 .....0'
vcan0 244 [5] 00 00 00 01 80 .....
vcan0 18E [3] 00 00 7A '..z'

```

This is how exactly your CAN messages look like when they are captured via can-utils. If I break down the columns, the first one is the interface, the second one is the arbitration ID, third is the size of the CAN message, this can't be more than 8. If you look at the CAN frame, you will understand better why this can not be more than 8. The fourth is the CAN data itself.

### Making sense of CAN message

ID	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
0x111	0x0B	0xB8	0x00	0x00	0x00	0x00	0x00	0x00

In this example, this is an 8-byte frame. The message is being sent by an arbitration ID 0x111. Once the instrument cluster sees this message, this will first make sure, if it was intended for instrument cluster or not. If it is, then it reads the message which has 0x0BB8, which translated to 3000 in decimals. Now your instrument cluster moves the needle in the tachometer to 3000.



Once you have the understanding of how CAN message makes sense, we can further inject fake/modified packets via OBD-II on the CAN bus to spoof tachometer or anything else.

Before we run into the demo of ICSim, let's look at how other mini utilities of can-utils work. To do this, let's first set up the virtual can interface.

You can do it with the usual `./setup_vcan.sh` script. If you want more insight on what this script is doing, have a look at the appendix of this document.

Once the virtual CAN interface is set up, you are now ready to send/receive the CAN packet in this interface. Let's now use one of the mini utilities from can-utils called cangen to generate the dummy CAN packets.

## cangen

```
CANGEN(1)
NAME
    cangen - manual page for cangen 2018.02.0-1
SYNOPSIS
    cangen [options] <CAN interface>
DESCRIPTION
    cangen - CAN frames generator for testing purposes.

    cangen: generate CAN frames
```

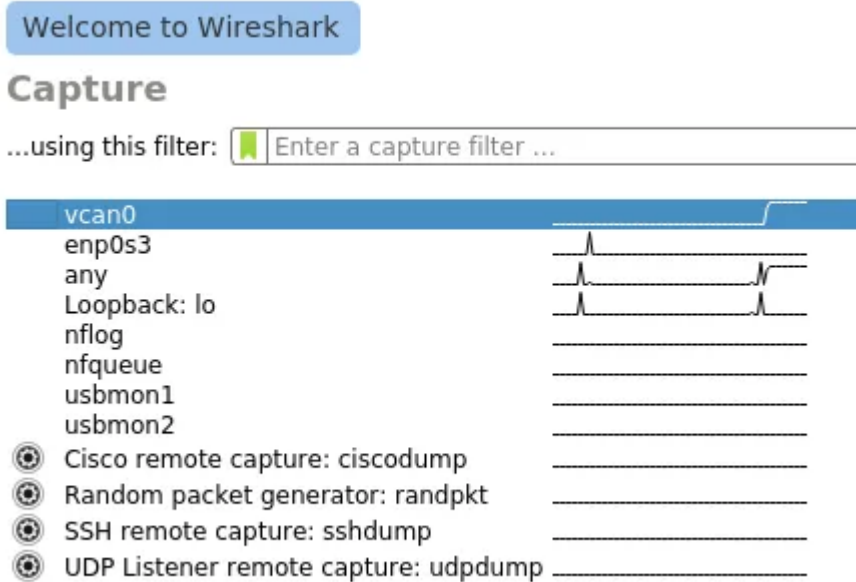
cangen generates the CAN frames for testing purposes. To use cangen, you need to specify the interface in which the CAN frame is to be generated.



```
cangen vcan0
```

*vcan0 is the virtual CAN interface we recently created.*

Since you have already generated CAN frames, there must be a way to look into the frames! There are many utilities available, one among many available is Wireshark. Launch the Wireshark after generating the CAN frames. If not installed, you can install Wireshark using apt.

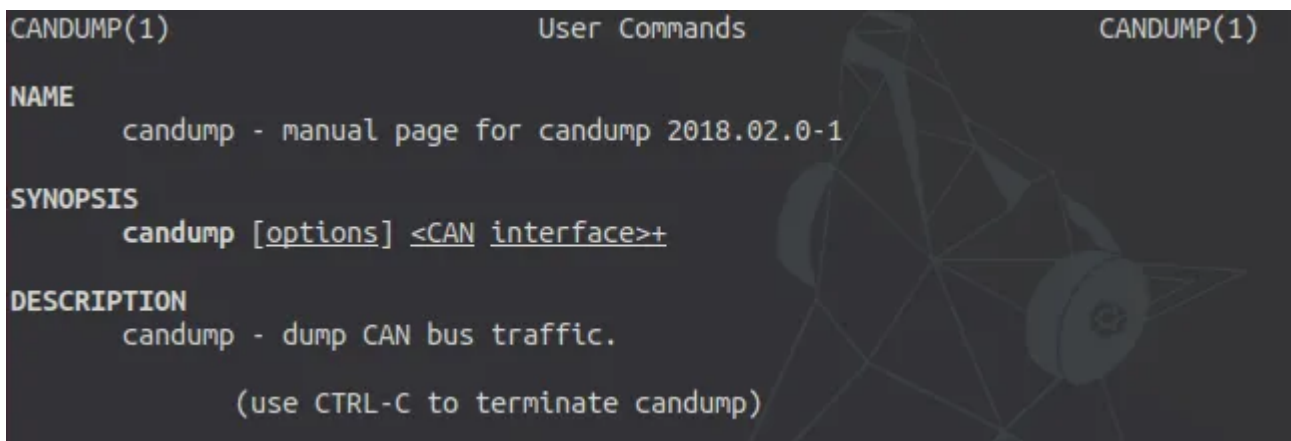


You may see many interfaces available depending on how many interfaces are up, vcan0 is the interface where your CAN frames are being generated.

Once you click on the interface you wish to look the packets into, this is how the CAN frame looks like.

Also, there are other utilities inside vcan0 like **cansniffer** and **candump** which does more or less the same stuff Wireshark does. You can use any tools or utility, whichever you feel more comfortable with.

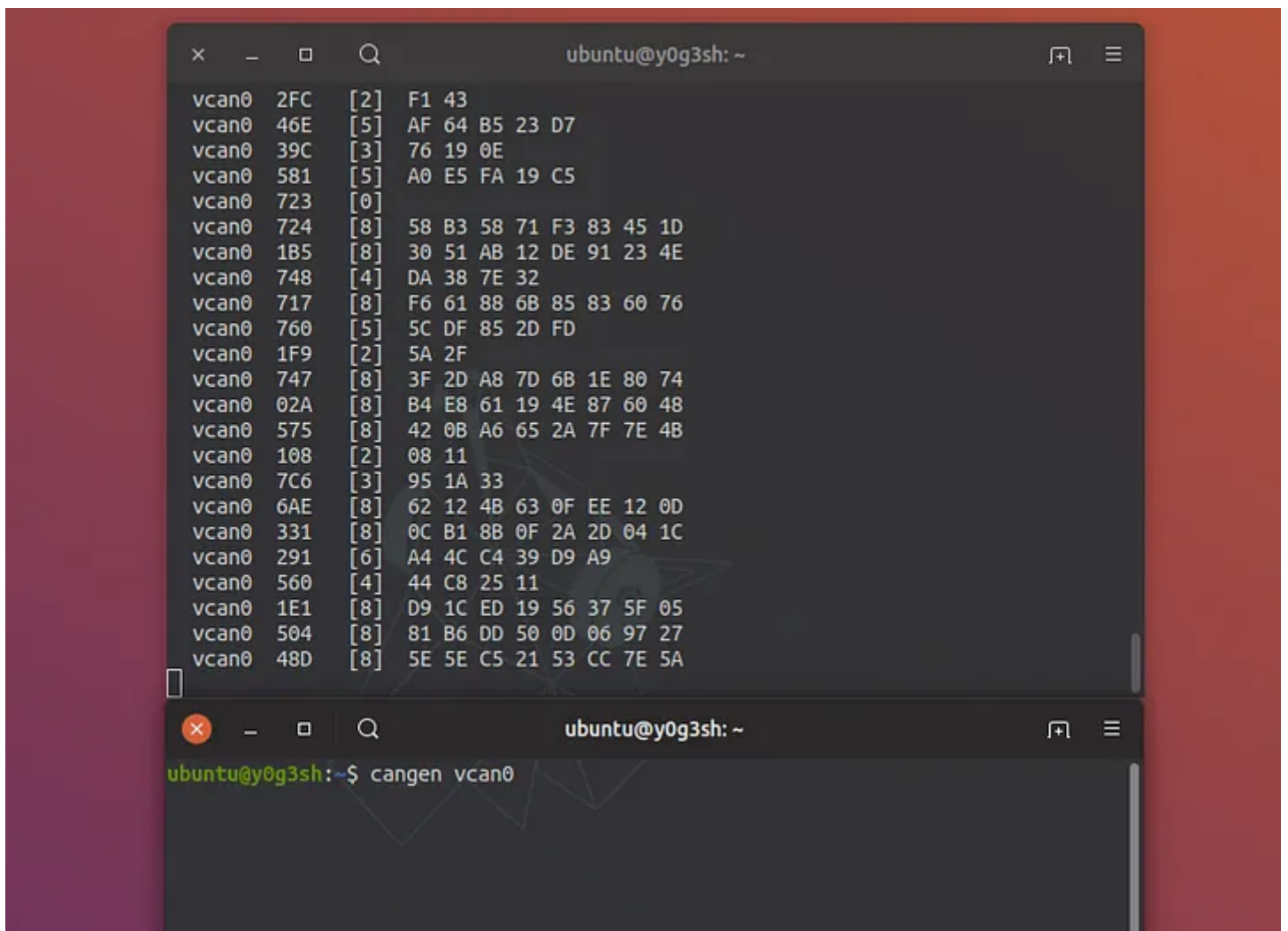
## candump



To dump or log the frames using **candump**, you can use

```
candump vcan0
```

This will be the output from the candump.



```
ubuntu@y0g3sh: ~  
vcan0 2FC [2] F1 43  
vcan0 46E [5] AF 64 B5 23 D7  
vcan0 39C [3] 76 19 0E  
vcan0 581 [5] A0 E5 FA 19 C5  
vcan0 723 [0]  
vcan0 724 [8] 58 B3 58 71 F3 83 45 1D  
vcan0 1B5 [8] 30 51 AB 12 DE 91 23 4E  
vcan0 748 [4] DA 38 7E 32  
vcan0 717 [8] F6 61 88 6B 85 83 60 76  
vcan0 760 [5] 5C DF 85 2D FD  
vcan0 1F9 [2] 5A 2F  
vcan0 747 [8] 3F 2D A8 7D 6B 1E 80 74  
vcan0 02A [8] B4 E8 61 19 4E 87 60 48  
vcan0 575 [8] 42 0B A6 65 2A 7F 7E 4B  
vcan0 108 [2] 08 11  
vcan0 7C6 [3] 95 1A 33  
vcan0 6AE [8] 62 12 4B 63 0F EE 12 0D  
vcan0 331 [8] 0C B1 8B 0F 2A 2D 04 1C  
vcan0 291 [6] A4 4C C4 39 D9 A9  
vcan0 560 [4] 44 C8 25 11  
vcan0 1E1 [8] D9 1C ED 19 56 37 5F 05  
vcan0 504 [8] 81 B6 DD 50 0D 06 97 27  
vcan0 48D [8] 5E 5E C5 21 53 CC 7E 5A  
  
ubuntu@y0g3sh:~$ cangen vcan0
```

In one of the terminals, the lower one is generating the CAN packets, whereas the terminal on the top is running **candump**. If I have to break down the columns for you, the first one you see is the CAN interface. The second is the arbitration ID, the third one is the size of CAN message, and the fourth is the message itself.

**candump** can also log the can frame for you. If you wish to perform a replay attack, you can first log the frames and then use mini utility like canplayer to replay the frames. Logging of CAN frames can be enabled using -l flag.

```
candump -l vcan0
```

When you log the CAN frames, a file will be created prefixed by candump followed by the date. If you wish to see the contents of the dump file, you can always use **cat** command in Linux to see the contents.

The frames we captured using **candump** can be replayed using a utility like **canplayer**.

## canplayer

```
CANPLAYER(1)                                User Commands                                CANPLAYER(1)

NAME
    canplayer - manual page for canplayer 2018.02.0-1

SYNOPSIS
    canplayer <options> [interface assignment]*

DESCRIPTION
    canplayer - replay a compact CAN frame logfile to CAN devices.
```

As the name suggests, the **canplayer** will replay the can frames. Ideally, this is useful when you have to do the replay attack. You would first dump/log the CAN frames and then playback the CAN frame using the canplayer.

Imagine a scenario where you wish to spoof the tachometer, and you have no idea on which arbitration ID the tachometer reading works, you have no idea what's in the CAN message. So ideally you would first dump and log the frames using candump with -l flag, and then use canplayer to replay the frames that were logged.

canplayer requires -l option to accept the input file.

```
canplayer -l canfile.log
```

canplayer has several other really useful options, you can find out them using **man canplayer**.

## cansniffer

```
CANSNIFFER(1)                                User Commands                                CANSNIFFER(1)

NAME
    cansniffer - manual page for cansniffer 2018.02.0-1

SYNOPSIS
    cansniffer [can-interface]

OPTIONS
    -m <mask>
        (initial FILTER default 0x00000000)

    -v <value>
        (initial FILTER default 0x00000000)

    -q      (quiet - all IDs deactivated)
```

CAN sniffer is used to see the change in CAN traffic. This is very useful to see a change in a particular byte. cansniffer has an option -c very useful for seeing the byte change in a colorful way. What this does is, it will compare the earlier byte and the current byte, if there's a difference then it is indicated by the change in the color of the byte. This is very useful when you wish to know if there was a change when you had performed a certain operations in a car.

```
cansniffer -c vcan0
```

I find cansniffer very helpful because the cansniffer allows filter by IDs as well. So if you wish to see the frames only from a particular ID, say 0x011, you can do that as well.

This can be done once you start sniffing, press **-** and then 000000. This will first clear all the frames. Now, you can start adding the IDs using + and then the ID you want to display and hit Enter. This way you can filter the frames of individual IDs.

## cansend

```
CANSEND(1) User Commands
NAME
    cansend - manual page for cansend 2018.02.0-1
SYNOPSIS
    cansend - simple command line tool to send CAN-frames via CAN_RAW sockets.
DESCRIPTION
    Usage: ./cansend <device> <can_frame>.  <can_frame>:

    <can_id>#{R|data}
        for CAN 2.0 frames

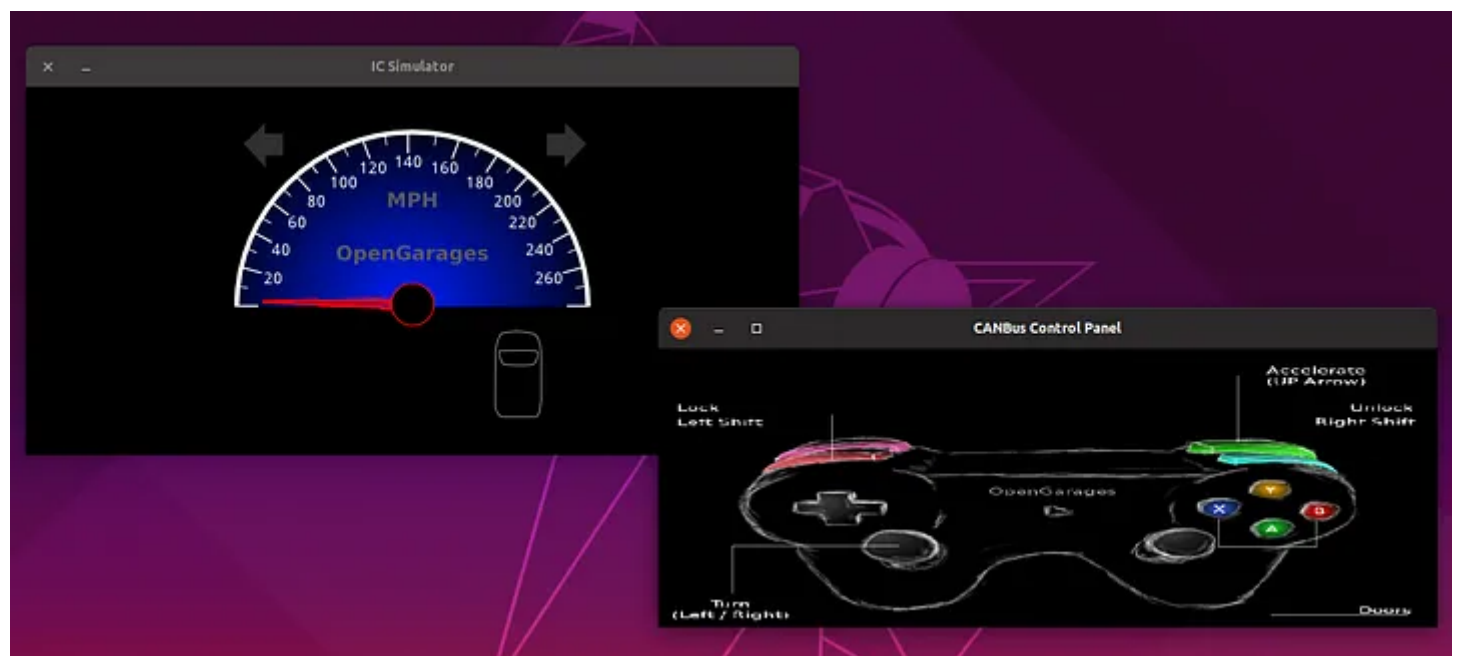
    <can_id>##<flags>{data}
        for CAN FD frames
```

cansend is used to send the CAN frames to a specific CAN interface. It's usage is

```
cansend interface frame
```

We will use all of these utilities with ICSim. Let's launch the ICSim as you already are able to do, and sniff the CAN frames.

```
./icsim vcan0
./controls vcan0
```



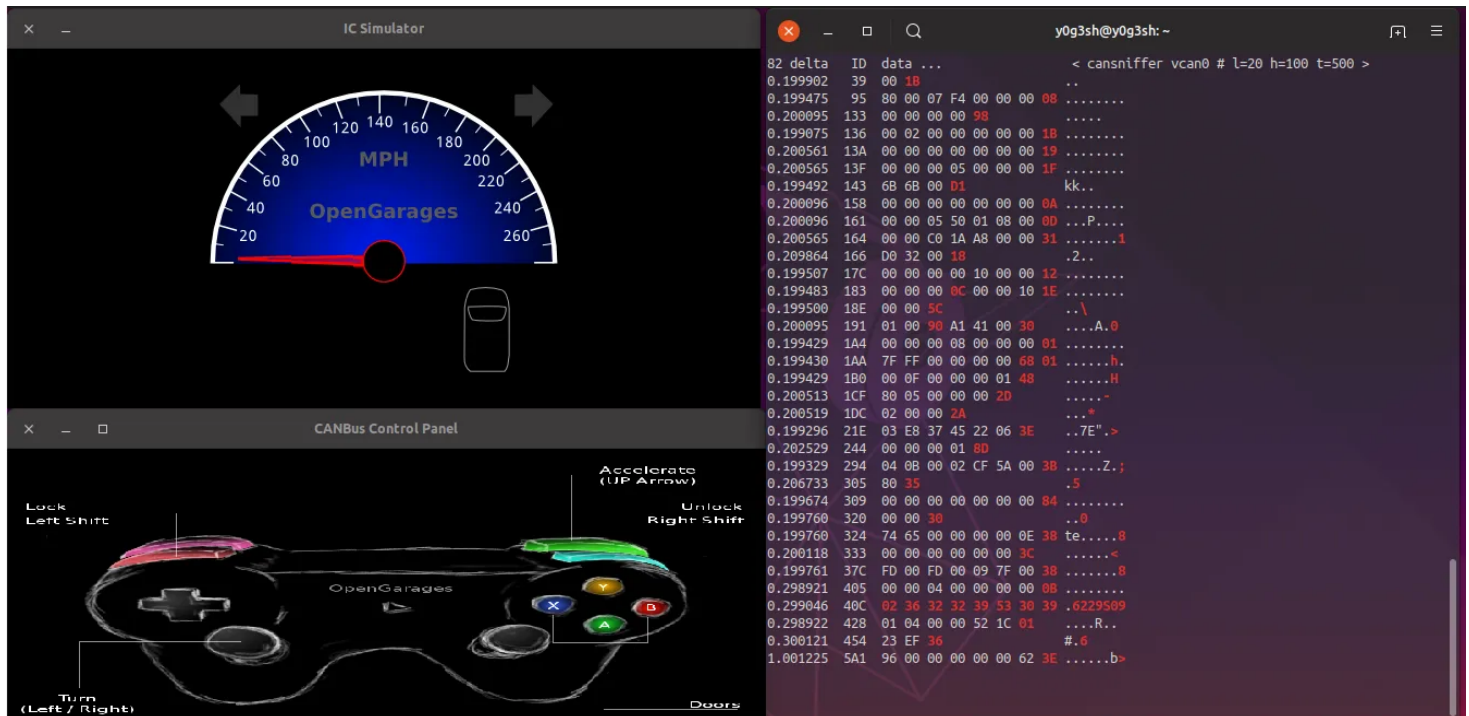
If you have followed every step discussed in the earlier post, you should be able to see this. Also, you can notice that the speedometer needle is moving back and forth, which is expected behavior because of the noise present.

## Sniffing the CAN frames generated by ICSim

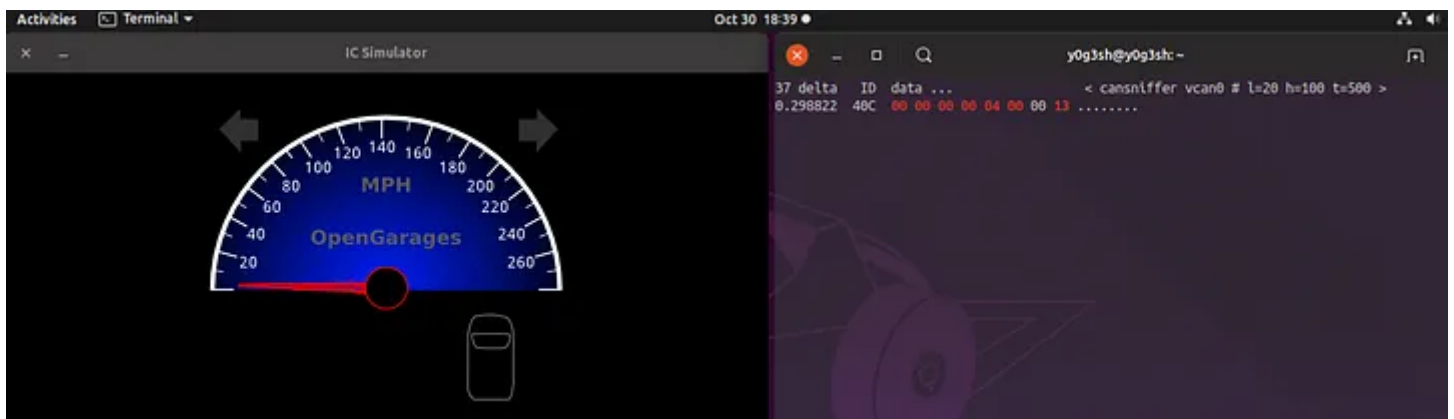
We will use **cansniffer**, a utility provided by can-utils, to sniff the packets. You can open up a new terminal and start cansniffer by

```
cansniffer -c vcan0
```

The **-c** option is used to highlight the change in bytes of the frame.



You can see very quick changes in the CAN frames, difficult to keep up with the rate at which communication is happening. In a real car, this communication would happen fast. To keep up with it, you can use arbitration ID filtering. If you only wish to see the frames from ID **40C**, you can always press **-** and then 000000 followed by Enter key. This will clear all the IDs from cansniffer and you can then press **+** followed by ID to filter out and then press Enter key.



As an example, I have filtered the ID **40C** only using the same steps mentioned above.

You can try playing around pressing the Up arrow key to increase the throttle and then notice how quickly CAN frames are being changed. The change is again indicated by the coloring. You can always play around with this and see how things are working under the hood.

---

## Task 4: Replay Attack

Making sense of this huge data is going to be a difficult task. Also, finding the arbitration ID in which you have to inject the frames is an impossible task to do from this big data.

So you would start sniffing the packets, then perform some action like turning on the turn signal indicators or pushing the throttle, once it is logged, then divide the packets into two halves, perform the replay attack on the first half and see if it works. If it doesn't, move on to the other half. This other half chunk of frames must work. Again this other half is still going to be huge, go ahead and divide the frames into two halves again, repeat this until you are left out with a single frame.

Now to perform replay attack with ICSim, you must have already started ICSim, you should be able to see the frames using **cansniffer**. Now we will use **candump** with **-l** option to log and save the frames, in the meanwhile we will increase the throttle, press <left> and <right> arrow keys to turn on the turn signal indicator.

```
candump -l vcan0
```

Now we will stop candump and you will see a file as candump-XXXXXX.log being created.

## Replaying the CAN frames

To replay those packets we will be using **canplayer**. Since we will be using a file as input for canplayer, we need **-l** option to be enabled.

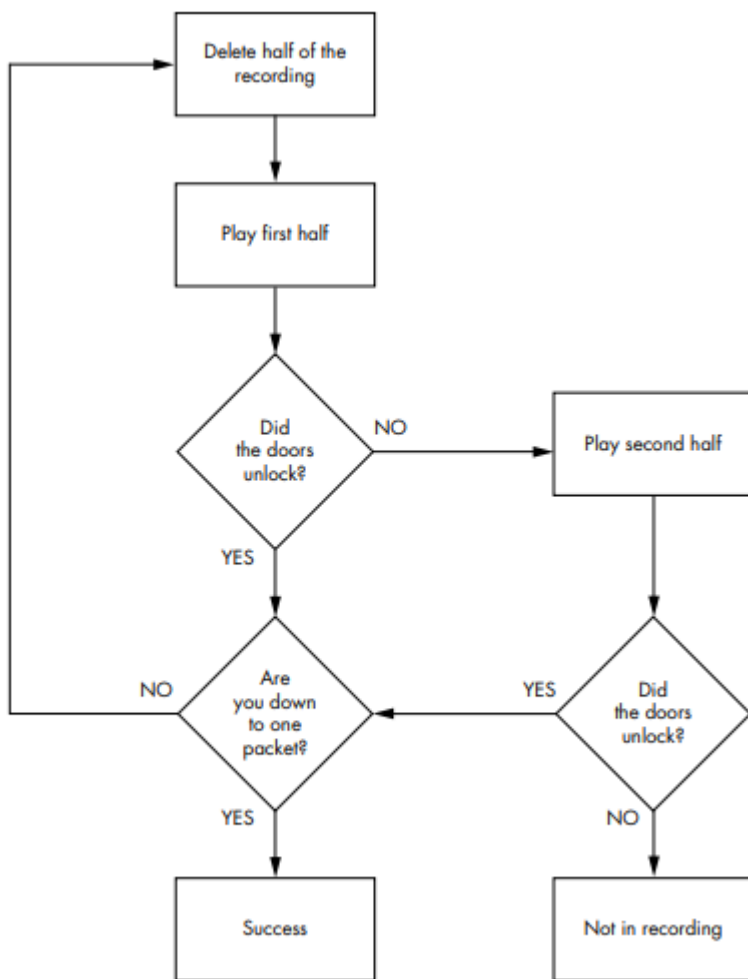
```
canplayer -l candump-2019XXXXXX.log
```

You can see that a replay attack has been performed, the turn signals, speedometer should be working as you had done earlier.

## Creative Packet Analysis

In a real car, CAN bus can be a lot noisier and CAN frames can appear a lot faster, so identifying the arbitration ID could be a difficult task. So to easily identify arbitration ID, you can follow this schema:





## Dividing the CAN frames and performing Replay

If you wanted to divide the CAN frames into two half and perform the replay on each of them, the best way to do is capture the CAN frames using `candump` and use **wc** utility to count the number of CAN frames, then use **split** to divide the log into two half equally:

```

ubuntu@y0g3sh: ~
ubuntu@y0g3sh:~$ wc -l candump-2019-11-23_201326.log
43722 candump-2019-11-23_201326.log
ubuntu@y0g3sh:~$ split -l 25000 candump-2019-11-23_201326.log frame1

-rw-r--r--  1 ubuntu ubuntu   1102600 Nov 23 20:37 frame1aa
-rw-r--r--  1 ubuntu ubuntu    825650 Nov 23 20:37 frame1ab

```

Now, you can use the `canplayer` to replay these CAN frame independently.

## Task 5: Reverse Engineering of CAN packets

Now you should have all the needed information to guess and reverse the meaning of different packets you saw in the CAN bus.

For this task, the challenge is to identify the arbitration ID for throttle, doors, and turn signals.

To do so, you may follow the graph we saw before and fine-tune a recording up to having one single packet. Another option could be to play with `cansniffer` and look for values chaining when performing specific actions on the controller. You can use filters to help you during the process.

## Task 6: Packet Injection

Now that you have reversed some of the packets, you can play as an attacker who has access to the canbus of a victim's car. You can use cansend to send recordings and manipulate what the car is doing without using the controller. If you prefer, you can also use cangen.

For this task, try to write a script that makes the vehicle press the throttle to the max without using the controller. You can do it in a one-line bash script, but if you are more comfortable with other languages, feel free to use them.

---

## References

- CSS Electronics <https://www.csselectronics.com>
- Car Hacking 101: Practical Guide to Exploiting CAN-Bus using Instrument Cluster Simulator — Part I: Setting Up <https://medium.com/@yogeshojha/car-hacking-101-practical-guide-to-exploiting-can-bus-using-instrument-cluster-simulator-part-i-cd88d3eb4a53>
- Car Hacking 101: Practical Guide to Exploiting CAN-Bus using Instrument Cluster Simulator — Part II: Exploitation <https://medium.com/@yogeshojha/car-hacking-101-practical-guide-to-exploiting-can-bus-using-instrument-cluster-simulator-part-ee998570758>

---

## Appendix: setup the virtual canbus interface

What follows is what the `./setup_vcan.sh` script is doing.

```
sudo modprobe can
```

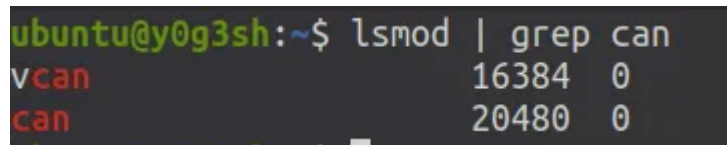
This will load the kernel module for CAN. Also, we need to load the kernel module for virtual can as well.

```
sudo modprobe vcan
```

If you wish to verify if the required kernel modules are loaded or not, you can use

```
lsmod | grep can
```

This will display if CAN and VCAN have been loaded or not.



```
ubuntu@y0g3sh:~$ lsmod | grep can
vcan                16384  0
can                 20480  0
```

Let's now set up the virtual interface

```
sudo ip link add dev vcan0 type vcan
sudo ip link set up vcan0
```

You can verify if virtual CAN interface is set up or not using `ifconfig vcan0` (or `ip a show vcan0` in novel Ubuntu versions).