

Exploiting CAN-Bus using Instrument Cluster Simulator

Anjali Prakash

University of Delhi , Lucideus Technologies

fav.anjaliprakash@gmail.com

Automotive security is really exciting and is an interesting topic of study for many security researchers. Automotive and hardware, for the most part, has always been off the radar. These hacks are always more difficult to pull off and at the same time, they are potentially more devastating. With the arrival of Self Driving cars like Tesla, Automotive security will only become more important.

When you are driving a car today, you are driving a hugely powerful computer that happens to have wheels and steering.

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.

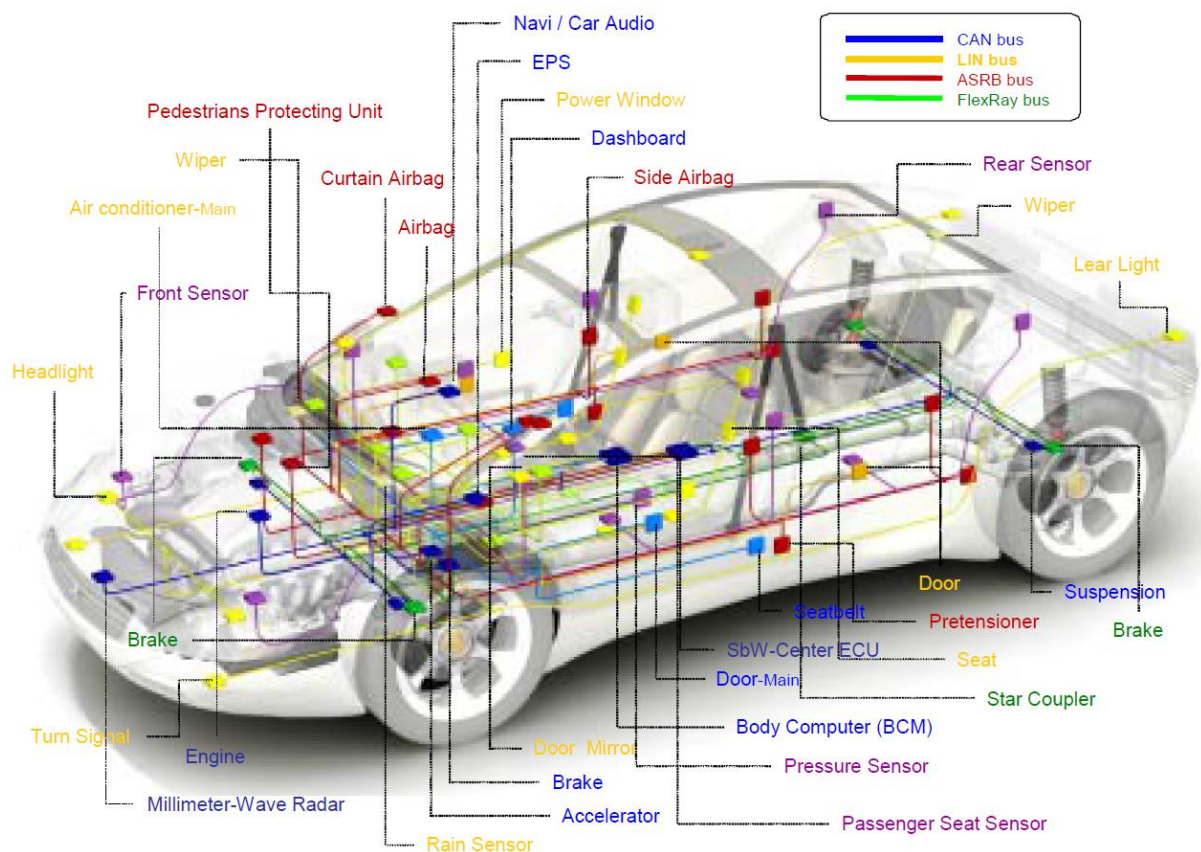
Keywords:

- Car hacking
- Cybersecurity
- CAN bus
- Controller area network
- Automotive security

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.

Introduction to CAN:

CAN (Control Area Network) This is a central nervous system this enables communication between all/some parts of the Car/Vehicle. Before CAN was originally developed by BOSCH in 1985 as an intra-vehicle System automotive Manufacturers used point to point wiring System, as we started adding more and more this become bulky and to expensive to maintain. CAN allows various electronic units in car to communicate and share data with each other.



Working of CAN = The main motive proposing CAN was that it allow multiple ECU to be communicative with only a single wire. A modern car can have as much as ECU (Depends on Car Model).

- A car Can have multiple nodes that are able to send or receive message, this message consist of essentially the ID which is the priority of the message and also it can contain CAN message that can be of 8 bytes or less at a time.

- If 2 or more begin sending message at the same time the message sent with the dominant ID will overwrite the less dominant msg.
 - Note: This is called priority-based BUS arbitration
- Message with numerically small value ID's are a higher priority and are always transmitted first.

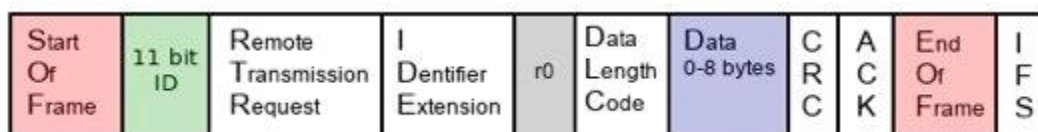
CAN Message Frames:

So what does a CAN message actually look like? The original ISO standard laid out what is called Standard CAN. Standard CAN uses an 11-bit identifier for different messages, which comes to a total of 2^{11} , i.e. 2048, different message IDs. CAN was later modified; the identifier was expanded to 29 bits, giving 2^{29} identifiers. This is called Extended CAN. CAN uses a multi-master bus, where all messages are broadcast on the entire network. The identifiers provide a message priority for arbitration.

CAN uses a differential signal with two logic states, called recessive and dominant. Recessive indicates that the differential voltage is less than a minimum threshold voltage. Dominant indicates that the differential voltage is greater than this minimum threshold. Interestingly, the dominant state is achieved by driving a logic '0' onto the bus, while the recessive state is achieved by a logic '1'. This is inverted from the traditional high and low used in most systems. These two states will be detailed later on in the article. The important thing is that a dominant state overrides a recessive during arbitration.

Standard CAN

The standard CAN message frame consists of a number of bit fields. These are shown in Figure



The first bit is the start of frame (SOF). This dominant bit represents the start of a CAN message. Next is the 11-bit identifier, which establishes the priority of the CAN message. The smaller the identifier, the higher the priority of the message.

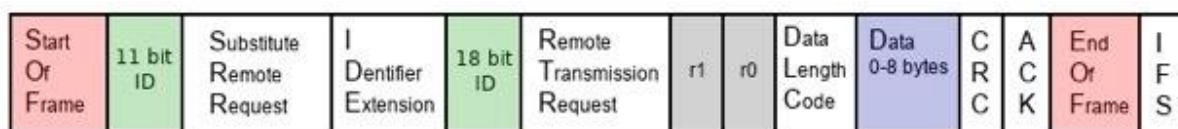
The remote transmission request (RTR) bit is normally dominant, but it goes recessive when one node is requesting data from another. The identifier extension (IDE) bit is dominant when a standard CAN frame is being sent and

not an extended one. The r0 bit is reserved and not currently used. The data length code (DLC) nibble signifies how many bytes of data are in this message.

Next is the data itself, being as many bytes as represented in the DLC bits. The cyclic redundancy check (CRC) is a 16-bit checksum for detecting errors in the transmitted data. If the message is properly received, the receiving node overwrites the recessive acknowledge bit (ACK) with a dominant bit. The ACK also contains a delimiter bit to keep things synchronized. The end of frame (EOF) signifies the end of the CAN message and is 7 bits wide, for detecting bit-stuffing errors. The last part of a CAN message is the interframe space (IFS), used as a time delay. This time delay is precisely the amount of time needed for a CAN controller to move the received message into a buffer for further processing.

Extended CAN

Extended CAN uses a 29-bit identifier along with a few additional bits. An extended message has a substitute remote request (SRR) bit after the 11-bit identifier, which acts as a placeholder to keep the same structure as standard CAN. This time the identifier extension (IDE) should be recessive, indicating that the extended identifier follows it. The RTR bit is after the 18-bit ID and is followed by a second reserve bit, r1. The rest of the message remains the same.



CAN Message Types

Now that you know what a CAN message looks like, you might be wondering what kinds of messages are passed along the bus. CAN allows for four different message types. They are the data frame, remote frame, overload frame, and error frame.

A standard CAN data frame makes use of the identifier, the data, and data length code, the cyclic redundancy check, and the acknowledgment bits. Both the RTR and IDE bits are dominant in data frames. If the recessive acknowledge bit at the receiving end is overwritten by a dominant bit, both the transmitter and receiver recognize this as a successful transmission.

A CAN remote frame looks similar to a data frame except for the fact that it does not contain any data. It is sent with the RTR bit in a recessive state; this indicates that it is a remote frame. Remote frames are used to request data from a node

Practical

Prerequisite

If you decide to practice this tutorial, you would need:

- Any Linux distributions (I'm using Kali Linux)
- can-utils
- ICSim (ICSim is an opensource Instrumentation Cluster Simulator)
Can be downloaded from <https://github.com/zombieCraig/ICSim>

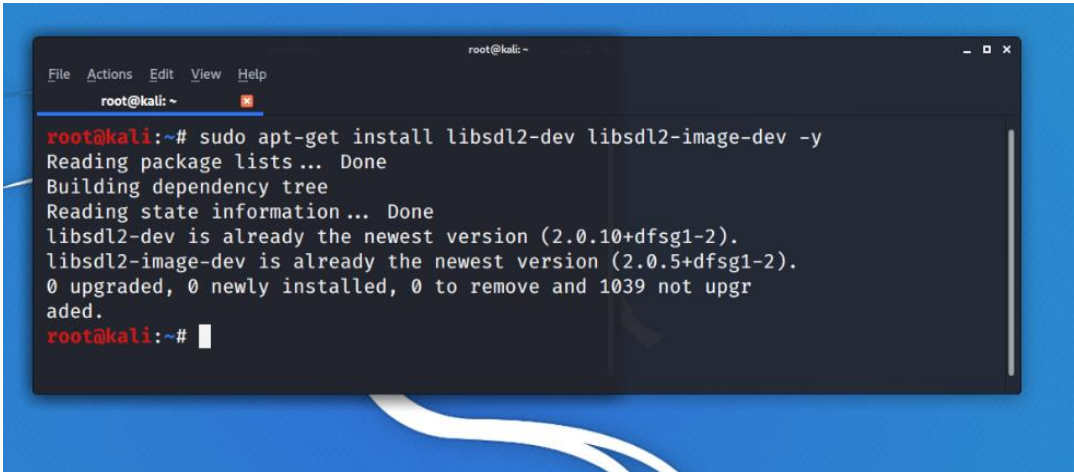
Setting up the virtual Environment

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.

Instrument Cluster Simulator requires SDL libraries

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.

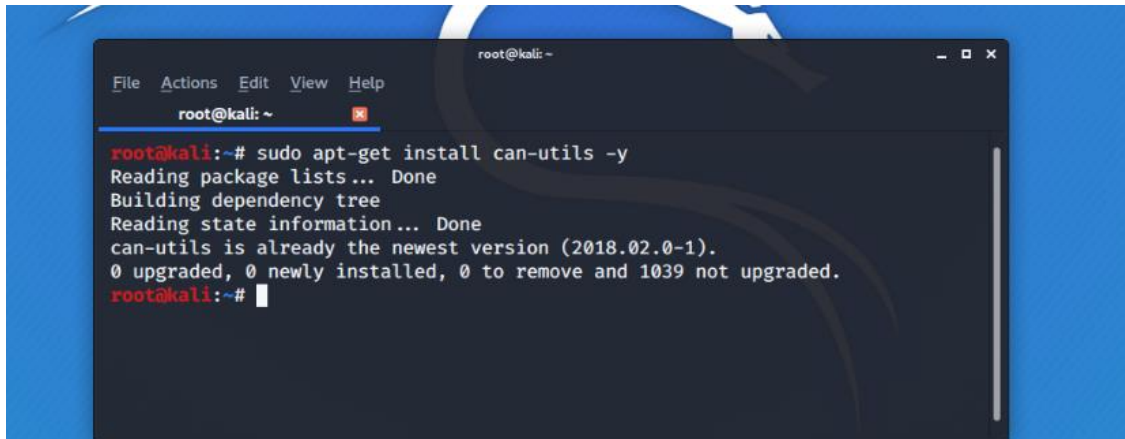
```
sudo apt-get install libsdl2-dev libsdl2-image-dev -y
```



```
root@kali: ~  
File Actions Edit View Help  
root@kali: ~  
root@kali:~# sudo apt-get install libsdl2-dev libsdl2-image-dev -y  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
libsdl2-dev is already the newest version (2.0.10+dfsg1-2).  
libsdl2-image-dev is already the newest version (2.0.5+dfsg1-2).  
0 upgraded, 0 newly installed, 0 to remove and 1039 not upgraded.  
root@kali:~#
```

After the LibSDL libraries are installed, add the CAN utilities. CAN is short for controller area network, the primary network in modern automobiles. The CAN utilities are included in some Linux distributions, but not in Kali as of this writing. The CAN utilities can be installed using the command:

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

A terminal window titled 'root@kali: ~' showing the execution of the command 'sudo apt-get install can-utils -y'. The output indicates that the package is already installed at the latest version (2018.02.0-1) and no further action is required.

```
root@kali:~# sudo apt-get install can-utils -y
Reading package lists... Done
Building dependency tree
Reading state information... Done
can-utils is already the newest version (2018.02.0-1).
0 upgraded, 0 newly installed, 0 to remove and 1039 not upgraded.
root@kali:~#
```

Once the LibSDL and CAN utilities software dependencies are in place, as shown, we can proceed to download and install the ICSim car hacking tools.

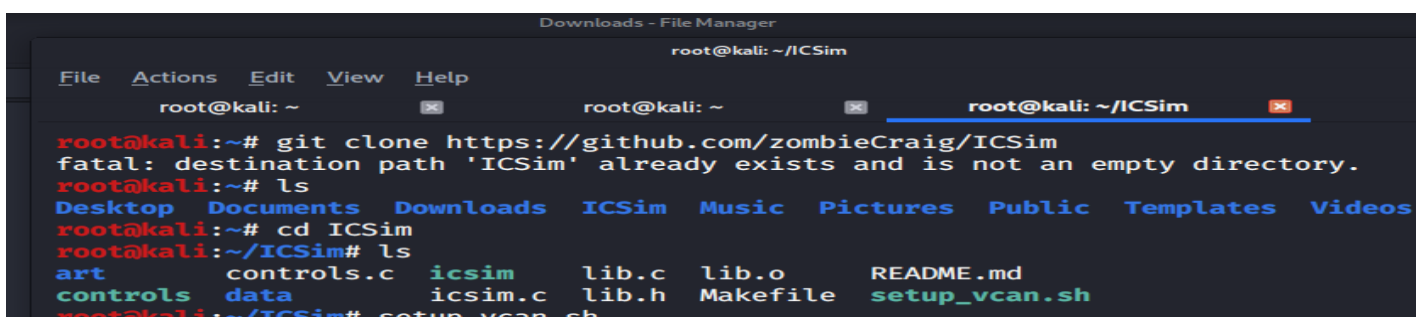
After installing LibSDL and CAN Utilities software dependencies, we can download and install the ICSim software

Downloading and Installing ICSim

Download and expand the ICSim project files using git with the command:

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

Git will download the project files for ICSim into a folder labeled ICSim in your home directory. Change into the ICSim folder and list the contents:

A terminal window titled 'root@kali: ~/ICSim' showing the cloning of the ICSim repository and the listing of its contents. The output shows the directory structure and files within the ICSim folder.

```
root@kali:~# git clone https://github.com/zombieCraig/ICSim
fatal: destination path 'ICSim' already exists and is not an empty directory.
root@kali:~# ls
Desktop  Documents  Downloads  ICSim  Music  Pictures  Public  Templates  Videos
root@kali:~# cd ICSim
root@kali:~/ICSim# ls
art          controls.c  icsim      lib.c      lib.o      README.md
controls    data       icsim.c   lib.h      Makefile   setup_vcan.sh
root@kali:~/ICSim# setup_vcan.sh
```



```
cd ICSim/
```

```
ls
```

You should see several files, including two executable files labeled `controls` and `icsim`, inside the `ICSim` folder, as shown in Figure

Preparing the Virtual CAN Network

View the contents of the shell script `setup_vcan.sh` by typing the `more` command:

You should see the following four lines of shell commands:

```
root@kali:~/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
root@kali:~/ICSim#
```

The `modprobe` command is used to load kernel modules, like the CAN and vCAN network modules from the CAN utilities library, and the first two lines of the script will load these two modules to be able to communicate using CAN protocols on a virtual controller area network (vCAN) for our car-hacking simulator. The final two lines will create a new network device called `vcan0` of type vCAN and turn the link on.

Either type and run those four lines above, or run the shell script by typing:

```
sh setup_vcan.sh
```

You can verify that the `vcan0` network link is active by typing `ifconfig`

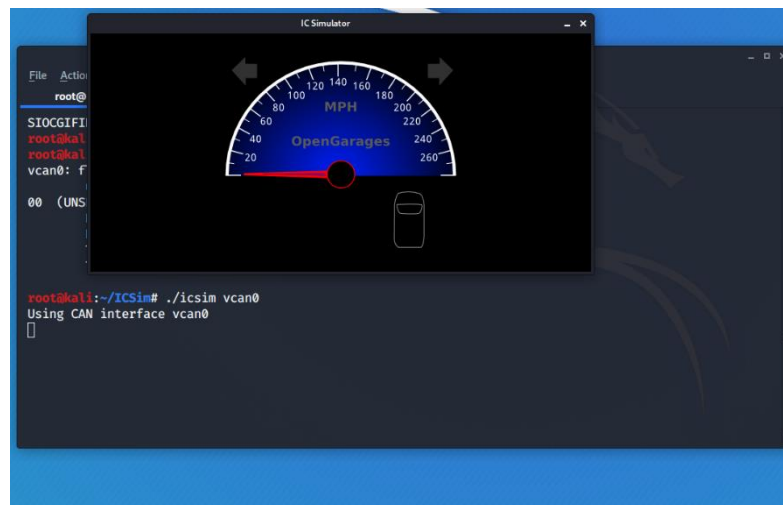
Running the ICSim Software

The standard setup for running ICSim includes at least two components, the `icsim` Instrument Cluster Simulator program file, which simulates an automobile's dashboard instrument panel, and the `controls` executable, which gives the user control of the virtual automobile, including acceleration, steering, door locks, and turn signals. For a first experience for beginning car hackers, it's also instructive to open a third terminal window running a network sniffer to view packets on this new virtual CAN network.

Open three terminal windows. In the first window, open the Instrument Cluster Simulator application, icsim, on the vcan0 virtual CAN network interface we created:

```
~/ICSim/icsim vcan0
```

That's vcan0, with a zero, denoting the virtual CAN network we created by running setup_vcan.sh above. The dashboard instrument panel simulator will appear as shown

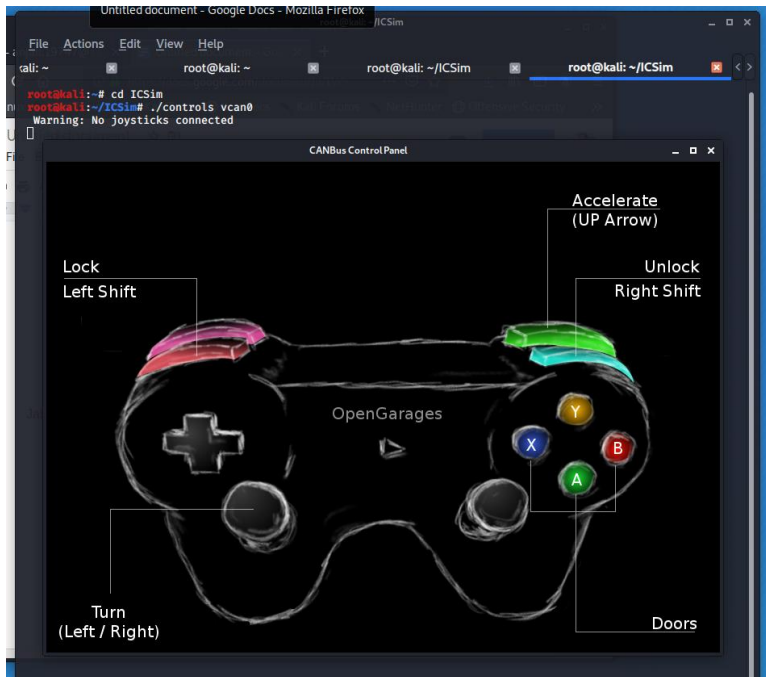


Nothing on the dashboard will light up, and the speedometer will remain fixed, because there's no traffic on the vcan0 network yet. We'll address that by starting the ICSim's controls.

In a second terminal window, open the controls app

```
: ~/ICSim/controls vcan0
```

The CANBus Control Panel application will appear on the screen



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



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 indicate the change in bytes of the frame.

```

root@kali:~# cansniffer -c vcan0
00 delta ID data ... < cansniffer vcan0 # l=20 h=100 t=500 >
9.999999 166 D0 32 00 36 .2.6

01 delta ID data ... < cansniffer vcan0 # l=20 h=100 t=500 >
9.999999 39 00 2A .*
9.999999 95 80 00 07 F4 00 00 00 35 .....5
9.999999 133 00 00 00 00 98 .....
9.999999 136 00 02 00 00 00 00 00 1B .....
9.999999 13A 00 00 00 00 00 00 00 19 .....
9.999999 13F 00 00 00 05 00 00 00 00 .....
9.999999 143 6B 6B 00 C2 kk..
9.999999 158 00 00 00 00 00 00 00 0A .....

```

Reference

- <https://www.allaboutcircuits.com/technical-articles/introduction-to-can-controller-area-network>
- https://en.wikipedia.org/wiki/CAN_bus
- <https://www.ni.com/en-in/innovations/white-papers/06/controller-area-network--can--overview.html>