Introduction to cyber-attacks on vehicles
(vircar)

Halis Duraki 06/2016.

Background
A few months ago, I wrote a blog post about cyber-attacks on vehicles and car computer systems that gathered a lot of attention from various hackers around the world. Most of the people who jump to car hacking have no prior knowledge in electronics, so if this is what you want to do in your life, make sure you dig about Tesla first. Pun intended.

In this research, we will explain information that most media and news portal publish lately: hackers being hackers hacking car systems. Is that so simple as media tell us? How secure are car systems actually? Newer cars vs Older cars?

The information available to the world wide web regarding the cyber attacks on cars and vehicles is pretty low. The reasons for this are mixed but we will state some of them first. Hacking by itself is already expensive; it takes a time to learn about various techniques, you need a good equipment, and it can get a bit overwhelming if you jump into some topic too fast. Now, combine this with car and vehicles hacking, and you already have to be rich to practice it.

We figured out that the best way to practice it is to build own car. So we built and coded a (virtual) car engine in C, which operates as a real car, just in a virtual word – Your PC.

Abstract:
Car. An everyday used object by million of people around the world, including you. With years, the car industry has become a real technology issue, when several of mechanical standards that we used years ago became simply an electronic device. Imagine the same electronic device connected to the network or allowed from third-party to be operated. That gives us a way to manipulate the data that car actually instructs to his brain.

Most of the leading automobile companies became aware of this, so to have a bit of security standards in their products, they started hiring car hackers to keep and protect their software and systems as secure as possible. For example, the common thing between *Mercedes*, *Tesla*, and *Hyundai* is that all of them have a special security sector that deals with computer protection on their products.

Connectivity
The car is split up between two parts: the things that consumer use, and the computer that deal with that operation. There are multiple connection methods allowed, either wireless or wired. For example, in newer cars, we have radio waves, keyfobs, IR, NFC, Wi-Fi, bluetooth etc., but the thing that can help us travel to the kernel space is a wired connection method. In this particular research, we will use CAN-Bus protocol.

CAN Bus, or Controller Area Network (BUS) is a vehicle bus standard designed to allow microcontrollers and devices to communicate with each other without host computer. That means that operations are sent in two ways making the operation registered by the microcontrollers and also doing that action for the consumer.

The CAN is open port, which means it's available as an open platform, making this standard already available for hacking.

The way CAN communicates is by sending signals, holding various data information called frames to the ECU. These information are sent through two signals: CANH (high) and CANL (low) that represents data frames.

In this research, we will use an OBD-2 connectivity standard that has CAN signal position – CANH (position 6 pin) & CANL (position 14 pin).

ECU
Earlier in connectivity paragraph, I've noted that CAN communicate by sending signals to the ECU. ECU, or Electronic Computer Unit is an embedded system that controls one or more of

the electronic and electrical system or subsystem in a transport vehicle. There are number of ECU types ranging from engine operations to the transmission, brakes, central ECU and such.

Think about ECU as a way that consumer operation deals directly with the car. For example, if a consumer turned on a headlights, the operation with frame **HEADLIGHTSON** will be registered on the ECU, and it will return either if operation succeed or not.

This is also the way vehicle repair shop work when they use a laptop to find out what doesn't work on your car. The type of software that works using these techniques are simple yet many people think it's some magic. The software actually just sends a CAN data frame and check if the returned message is either 1 (succeed) or 0 (fail). It's good to note that frame does not actually respond with a Boolean (1/0), but for easier explanation we will just use this reference.

Virtual CAN Bus

Normally, when you learn new techniques especially jumping into new security sector, you set up the lab, a hacking lab. This way, you can see what happens in background, and you will also spare money if you break something *too hard*. We all know Linux is awesome, but reading this paragraph will show you how awesome it actually is.

Linux kernel comes with CAN bus protocol that a group of WV researchers and developers contributed to. It's called *Linux-CAN* - and it's used as a SocketCAN user space application, freely and possibly already available on your system. SocketCAN package include various userspace utilities including (but not limited to): *canbusload*, *candump*, *cangen*, *cansend*, *cansniffer*.

All these packages can be downloaded and compiled from GitHub @ **linux-can/can-utils**.

We can now study about CAN bus through the virtual emulation of CAN protocol and linked hardware module.

Here is a simple device linking and virtual lab setup.

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

To inspect if device is up and working:

```
$ ifconfig | grep vcan
```

What we did now is create a CAN virtual device that will wait for further instructions and operations that shall be sent over CAN high and CAN low pins in a way of frames. To inspect and dump frames we could use *candump* – a CAN utility to dump data frames that are sent through our device. This will help us examine the log data for further use.

```
$ candump -td vcan0
```

After the above command is executed, the *candump* will wait for frames sent through and on to **vcan0** device. As there are not data frames currently sent over our device, we might use **cansend** or **cangen** to send a set of instructions to the vcan0.

To send a specific set of frame data we use parameters **device can_id#can_frame.**

```
$ cansend vcan0 123#DEADBEEF
```

To send a random generated frames data to the vcan0 device, we may use a can-utils subsystem package called **cangen**.

```
$ cangen vcan0
```

If you tried any of above commands, you may see that the **candump** starts displaying the data frames sent to the vcan0 device. The cangen is quite a tool that can be used to fuzz an actual car ECUs with various options available.

Vir(*tual*) **Car**

Now when we know how does CAN operates, and we know that CAN is usable in a virtual environment, we can use the same technique to code our own car – because you wouldn't download a car, would you?

We won't cover all details on how to code your own car but we will cover most of the code from our car engine called **vircar**. Vircar is an open-source car engine platform built on top of

Linux CAN kernel sys. The same technique used on these package subsystems will be developed in our engine. Vircar is also freely available on my GitHub @ **dn5/vircar**.

Later, we will show car hacking techniques, and also develop a tool that will show us an example on how our car could be hacked from a fuzzing perspective.

Vircar is a highly customizable car engine written in pure C, that represent a car in a limited spirit and form. It can be used as an example of CAN interface communication through device linking and car hacking lab. Vircar currently supports a set of several operations that are sent to the ECU and these are: ENON (engine ON), ENOFF (engine OFF), LOCK (Door lock), DOCK (Door unlock), and KILL (destroy the car, remove CAN device).

The vircar is so simple even female can understand it! First, it opens a virtual bus device, then starts listening the operations, and of course last but most important – waits for consumer operation and check if the operation with specific frame is available. If it is, the vircar execute the operation, displaying the data, and if it isn't valid, it proceed to wait for other commands send by the consumer.

The method create_car() creates a device in a virtual environment (this requires sudo or appropriate):

```
void create_car() {
…
system("modprobe vcan");
system("sudo ip link add dev vircar type vcan");
system("sudo ip link set up vircar");
```

The main() function firstly creates a structure of the CAN frame, including the current operation received, and secondly call a create car method.

```
int         s;
struct      can_frame frame;
int         size, i;
static      struct ifreq ifr;
static      struct sockaddr_ll sl;
char        *instance = "vircar";
int         ifindex;
char        current[8];
```

The socket will create a CAN endpoint for communication in a raw type:

```
s = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_CAN));
```

Bind will assign the address to the struct *sl*. The simplest to say is this will assign a name to a socket.

```
if (bind(s, (struct sockaddr *)&sl, sizeof(sl)) < 0) {
            …
}
```

Currently, our car is running a virtual CAN device, opening the socket and waiting for an input. The thing that is left is to register any operation received to the device. Please, don't be me and avoid using *while()* – I just love to play with fire.

```
while (1) {
  /* frame validation */
  if ((size = read(s, &frame, sizeof(struct can_frame))) < 0) {

  /* size of frame */
  } else if (size < sizeof(struct can_frame)) {

  /* frame seems fine */
 if (frame.can_id & CAN_EFF_FLAG)

  /* copy frame data into current */
  strcpy(current, frame.data);
```

Once we copied the received frame data into current, we may want to use it in a various way. Vircar is so simple it will basically check the value of *current* with the operations that are valid. Please, refer to **car.h** file in vircar engine for explanation of frames that are valid.

```
  /* check if frame is valid operation */
  if (strcmp(current, *VALID) == 0) {
```

And that's it! Simplest open car engine that can be ridden on your computer! Don't you love it? Now when you know how **vircar** operates, we can examine it a bit. Please, *git clone*, *make*, and *chmod +x* it! *Git pull* is **highly** appreciated.

```
$ ./vircar
Welcome to vir(tual) car.
~
=strip=
https://github.com/dn5/vircar

# waiting for operation
```

Your car is officially usable now. Please refer to *vircar* documentation to found out more about it. Remember when we said earlier that we could use Linux-CAN to send the data. Here it is again. The *cangen* or *cansend* can be used to send frames to our car.

```
$ cangen vircar –D DEADBEEF
```

The vircar will start displaying the data received. We don't know what are the frames that actually are usable in our car so we need to examine what available techniques are possible for pwning the car.

Pwning techniques

When it comes to hacking the actual car, there are several options available. I know there are more out there but here are the one I will cover: **DBCT**, **Trial and error**, and of course **fuzzing** technique.

The DBCT

The DBC technique actually isn't a hacking technique but rather a direct access to the frames that ECUs are registered in some vehicle. When car manufactures are developing a new vehicle, they must know what frame is used for what with all details included. Imagine this as an API or methods that are available for car itself. These are in fact CAN data frames in their original format.

The great thing is, these frames can be sniffed (check out **cansniffer**) but most of the time, the data are actually encrypted and protected by the manufacture. Some car companies may additionally provide these original frames to the third-party services and shops.

The frames originally come in DBC file, a CAN database format that holds all information and frames regarding a specific instruction set. Sometimes, these files are sold on black market or leaked by employees. Most of the time, they are not, and are actually pretty secured.

Trial and error

Ah, the good old, fundamental method of solving a problem. Most developers use it to come up with a solution for their working code solution. The same technique can be used in cyber attack on vehicles and car systems simply by rechecking the pattern of some data frame. Turn the headlight on, and CAN sniffer may display a hundred of frames that are sent. Turn them off and then turn them on again? Some of the frames are missing from the previous sniff and some are there. Rinse and repeat until you get appropriate data frame. This might not be the best technique but I'm quite sure it's cheaper then the DBCT or fuzzing one.

Fuzzzzzzzing

You must have heard of this one too! It's one of the most famous and oldest method of finding bugs and vulnerabilities. You fuzz every possible combination until you either: 1) crash the system or 2) find the bug.

This is the technique we will show in this research paper. I don't advice trying this technique on a real car and I've heard the stories your engine may fail or you can really break your car system and ECUs. I am sure you don't want that.

The next paragraph will go in depth with fuzzing technique on **vircar** using a Ruby script called **vircar-fuzzer**.

Am I a car hacker, yet?

We can start off with basic ECU and CAN information we have. The later will be assumptions on data frames and the algorithm.

By investigating our car further through linked third-party device, we found out that there are four ECU types in our car. The additional ECU (KILL) is in the documentation we get in car paper and usage. The arbitration ID (**can_id**) is avoided because the car we just bought is simple and there is only one ID that operates everything. Maximum of four char operations, 8-bits (1 byte) ASCII encoded, and there are only A-Z letters with no numbers.

We now have n(4) number of possibilities which comes out as 456976. That's a lot of frames to fuzz but we built a script that will help us do this. For more information on the script, please visit my GitHub @ **dn5/vircar-fuzzer**.

Use **vircar-fuzzer g** to generate a fuzz list. If you pulled latest commit from GitHub, the vircar-fuzzer also includes a already generated

list with a lot of data already in. In between, some actual frames are available that are registered on ECU.

We can now start fuzzing our car. First, create and link the car (**vircar**).

```
$ ./vircar | tee fuzz.txt
```

In another window, run the **vircar-fuzzer** and you started fuzzing the car.

```
$ ruby vircar-fuzzer.rb
```

Our Ruby script will open the already generated wordlist and start sending the data frames to the vircar virtual CAN bus device. Please be aware that this may take a long, long time. Once the script finished, you may cat the file and grep the output.

```
$ cat fuzz.txt | grep vircar
```

The greped file should display valid data frames that we can send manually over **cansend** for future use. Please bear in mind that the script is coded under one minute and generating same as fuzzing is really slow. Please make a pull request if you rewrite the script to be faster!

Outro

That is it. We successfully hacked a car! A virtual one, but still, this should give you an example of various techniques, and it's also a great introduction to the cyber attacks on vehicles.

This paper is written for BalCCon 2k16 with a headline "Introduction to cyber-attacks on vehicles" and is written by Halis Duraki.

I'm available and social:

Blog : https://dn5.ljuska.org
Twitter : https://twitter.com/dn5__
Email : duraki.halis@nsoft.ba

Brought to you by Halis Duraki (dn5) & Nsoft.

https://nsoft.com