Command Line and Scripting Essentials for Heavy Vehicle Security

Phil Lapczynski

Version 2025-06-09

secure embedded

© 2025 by Philip Lapczynski is licensed under CC BY-SA 4.0.
To view a copy of this license, visit https://creativecommons.org/licenses/by-sa/4.0/

About Me

Phil Lapczynski (lap chin ski)

- Principal engineer at Renesas Electronics America
- ~20 years in transportation security
- Teaching Experience:
  - Automotive embedded security course at UD Mercy
  - Auto-ISAC ACT Program instructor for OTA security
- 8x CyberTruck Challenge Mentor
- SAE, Auto-ISAC, Uptane, Global Platform Contributor
- LinkedIn - https://www.linkedin.com/in/philip-lapczynski/

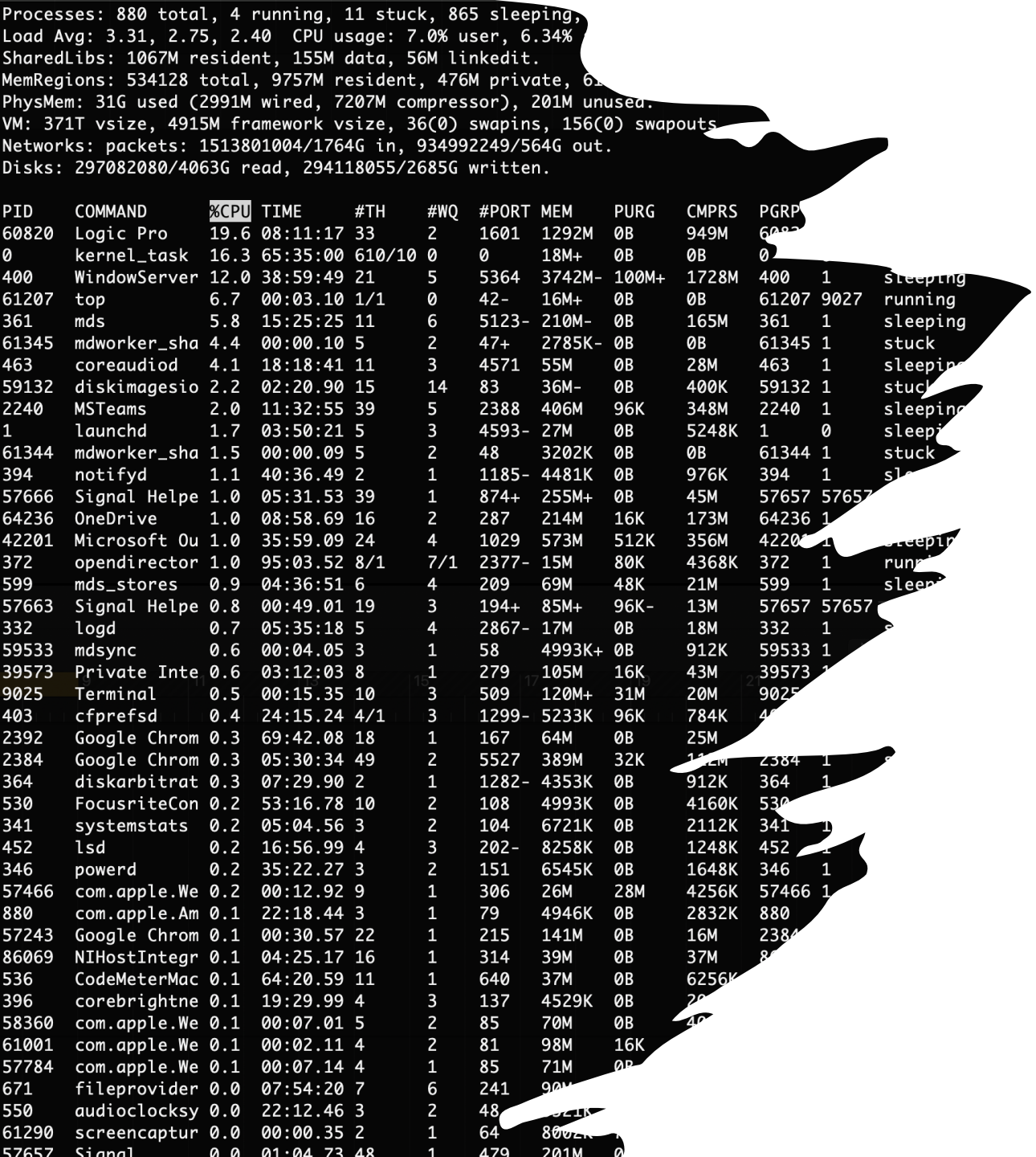
Command Line Interfaces

Who has experience..

...using the command line in Windows?

...using the command line in Linux?

...with CAN and J1939?



What Is the Command Line Interface (CLI)?

- The CLI is a **text-based interface** used to interact with a computer's operating system.
- Unlike graphical user interfaces (GUIs), the CLI uses **typed commands** to perform tasks.
- It's lightweight, powerful, and often the **preferred tool in professional diagnostics and scripting.**
- Common CLIs:
  - **Windows Command Prompt / PowerShell**
  - **Linux Terminal / Bash / ZSH / etc**
- Widely used in **vehicle diagnostics, firmware interaction, and cybersecurity tools**.

Basic Command-Line Operations

Today’s Objectives

Command-line Basics

Overview of essential command-line commands and operations for both Windows and Linux systems.

Scripting Introduction

Introduction to scripting for diagnostics and automation tasks, enhancing efficiency and troubleshooting capabilities.

Python and CAN Bus

Using Python scripts to interact with CAN bus systems for data communication and vehicle diagnostics.

AI Tools for Scripting

Leveraging AI tools like ChatGPT to assist in writing and debugging scripts, optimizing development processes.

Advanced CLI Tools

Overview of advanced command-line tools such as cantools, python-can, TruckDevil, CaringCaribu, and Scapy for enhanced diagnostics and automation.

Let’s Boot Into Linux!

- For a majority of this course we will be in Linux
- We will touch on both Linux and Windows, however it will be fine to stay booted in Linux the whole time

Navigating the File System

Key Commands:

- **`cd`** – Change directory (no argument will show current)
- **`ls`** (Linux) / **`dir`** (Windows) – List contents
- **`pwd`** (Linux) – Show current directory

Quick Examples:

- **`cd`** /home/user/logs (Linux)
- **`cd`** C:\Logs\2024 (Windows)

Basic File Operations

- **Common Commands:**
  - `mkdir` – Make new folder
  - `rm` / `del` – Remove files
  - `copy` (Windows) / `cp` (Linux) – Copy files
  - `move` (Windows) / `mv` (Linux) – Move or rename files
- **Examples:**
  - `mkdir` diagnostics
  - `cp` can_dump.log backup/ (Linux)
  - `copy` can_dump.log backup\ (Windows)

Windows CMD Resources

- https://github.com/security-cheatsheet/cmd-command-cheat-sheet

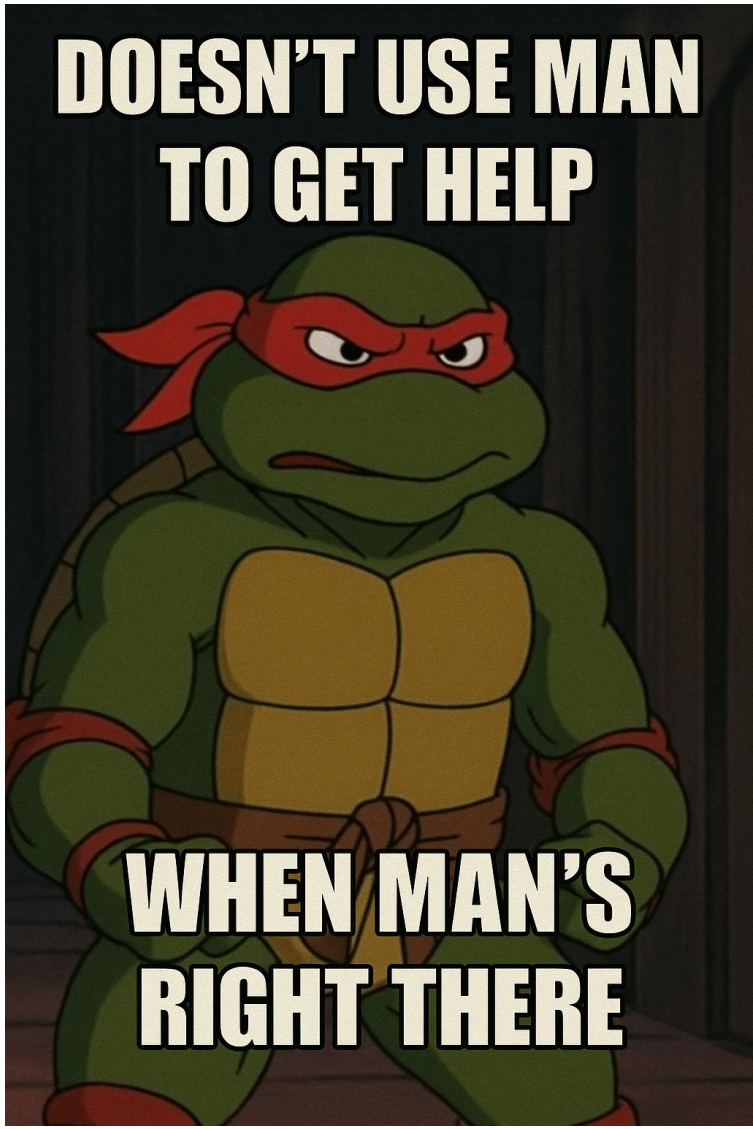
```
> help
```

```
Microsoft Windows [Version 10.0.19043.1586]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Madhushala>help
For more information on a specific command, type HELP command-name
ASSOC          Displays or modifies file extension associations.
ATTRIB         Displays or changes file attributes.
BREAK          Sets or clears extended CTRL+C checking.
BCDEDIT        Sets properties in boot database to control boot loading.
CACLS          Displays or modifies access control lists (ACLs) of files.
CALL           Calls one batch program from another.
CD             Displays the name of or changes the current directory.
CHCP           Displays or sets the active code page number.
CHDIR          Displays the name of or changes the current directory.
CHKDSK         Checks a disk and displays a status report.
CHKNTFS        Displays or modifies the checking of disk at boot time.
CLS            Clears the screen.
CMD            Starts a new instance of the Windows command interpreter.
```

Linux Command Line Resources

- **Lots of resources out there for help**
  - `man <command>` - Show the users manual

```
MAN(1)                    General Commands Manual                    MAN(1)

NAME
     man, apropos, whatis - display online manual documentation pages

SYNOPSIS
     man [-adho] [-t | -w] [-M manpath] [-P pager] [-S mansect] [-m arch[:machine]] [-p [eprtv]]
         [mansect] page ...

     man -f [-d] [-M manpath] [-P pager] [-S mansect] keyword ...
     whatis [-d] [-s mansect] keyword ...

     man -k [-d] [-M manpath] [-P pager] [-S mansect] keyword ...
     apropos [-d] [-s mansect] keyword ...

DESCRIPTION
     The man utility finds and displays online manual documentation pages.  If mansect is
     provided, man restricts the search to the specific section of the manual.

     The sections of the manual are:
           1.   General Commands Manual
           2.   System Calls Manual
           3.   Library Functions Manual
           4.   Kernel Interfaces Manual
           5.   File Formats Manual
           6.   Games Manual
           7.   Miscellaneous Information Manual
           8.   System Manager's Manual
           9.   Kernel Developer's Manual
```



Linux Command Line Resources (cont.)

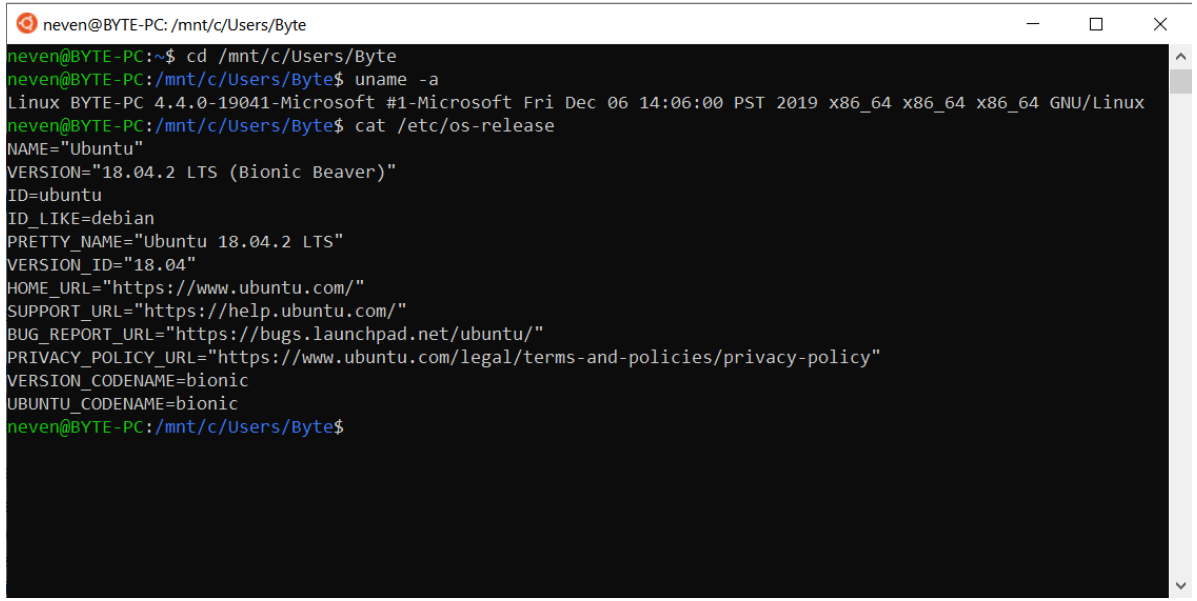
- **Lots of resources out there for help**
  - **tldr** – an attempt to make reading the manuals even easier with examples.

```
➜  ~ tldr man

man

Format and display manual pages.
More information: <https://manned.org/man>.

- Display the man page for a command:
    man command

- Open the man page for a command in a browser (`BROWSER` environment variable can replace `=browse
r_name`):
    man [-Hbrowser_name|--html=browser_name] command

- Display the man page for a command from section 7:
    man 7 command

- List all available sections for a command:
    man [-f|--whatis] command

- Display the path searched for manpages:
    man [-w|--path]

- Display the location of a manpage rather than the manpage itself:
    man [-w|--where] command

- Display the man page using a specific locale:
    man [-L|--locale] locale command

- Search for manpages containing a search string:
    man [-k|--apropos] "search_string"
```

```
➜  ~ tldr pip

pip

Python package manager.
Some subcommands such as `install` have their own usage documentation.
More information: <https://pip.pypa.io>.

- Install a package (see `pip install` for more install examples):
    pip install package

- Install a package to the user's directory instead of the system-wide default location:
    pip install --user package

- Upgrade a package:
    pip install [-U|--upgrade] package

- Uninstall a package:
    pip uninstall package

- Save installed packages to file:
    pip freeze > requirements.txt

- Show installed package info:
    pip show package

- Install packages from a file:
    pip install [-r|--requirement] requirements.txt
```

Windows Subsystem for Linux

- Windows has a subsystem that allows for use of the GNU/Linux environment within Windows without a VM
- **NOTE** – YMMV when trying to use things like socketCAN in WSL
- Better to work directly in Linux if you want to use Linux cantools



Working with SocketCAN in Linux

What is SocketCAN?

- **SocketCAN** is the Linux-native framework for Controller Area Network (CAN) support
- Treats CAN interfaces like network interfaces (e.g., Ethernet) — makes CAN sockets accessible via standard Linux networking APIs
- Developed by Volkswagen Research and integrated into the Linux kernel
- Supports a variety of CAN hardware adapters: USB, PCIe, and virtual interfaces
- Powers tools like **can-utils**, **python-can**, and custom scripts for diagnostics and security

Exercise 1: Enabling a Virtual CAN Interface

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

Since we will be using this many times throughout CyberTruck, is it best to add these commands to a script.

Example:

```
$ ./bringup_vcan.sh
```

Exercise 2: Enabling a HW CAN Interface

STEP 1: Connect your interface and make sure it is connected

- You can use the `lsusb` command to verify that it is connected

STEP 2: Configure interface and bring it up

```
sudo ip link set can0 down
sudo ip link set can0 up type can bitrate 50000
sudo ip link set can1 down
sudo ip link set can1 up type can bitrate 50000
```

Using can-utils – the CANbus hello world

To verify your interface and CAN is set up properly, it is wise to verify functionality using the Linux can-tools. Using two separate terminals…

TERMINAL A

```
$ candump vcan0
```

TERMINAL B

```
$ cangen vcan0
```

__note__ replace **`vcan0`** with your interface (can0, can1, etc)

You should see output!

```
can0  43F   [8]  D6 BA 05 3B 50 C1 47 7C
can0  209   [8]  72 24 42 58 2A 4A 78 71
can0  605   [8]  4E 8B FD 53 4C 6D 7E 1E
can0  16C   [8]  80 D4 CD 19 D3 46 6C 7D
can0  0D7   [3]  24 E0 E7
can0  682   [8]  E6 CF 08 4F 8E AB 49 2D
can0  409   [4]  E2 B2 0E 77
can0  7A7   [8]  9B 90 8D 76 E7 5B EB 14
can0  245   [1]  37
can0  5EC   [8]  78 66 03 0F F1 8B 09 54
can0  555   [7]  3E F9 87 72 C1 F6 53
can0  531   [8]  94 3D C0 5B 08 B6 70 5B
can0  091   [8]  F8 8E 43 5F 13 CF EB 64
can0  277   [8]  A2 7A 35 12 81 56 12 1D
can0  752   [4]  3E DD F5 33
can0  4EF   [8]  E0 82 22 17 34 27 65 4C
can0  14A   [7]  82 43 F0 69 36 17 F6
can0  1C0   [8]  28 A3 FF 17 16 57 F4 69
```

Useful Commands – Send and Receive

Reading from the can bus

```
$ candump any
```

Transmitting CAN frames

```
$ cansend can0 123#DEADBEEF
```

- Explanation:
  - Sends frame with **ID 0x123** and data **DE AD BE EF**

Transmitting 29-bit CAN frames (as used by J1939)

```
$ cansend can0 1823ff40#0123
```

Useful Commands – Logging and Replay

Logging traffic

```
$ candump -l can0
```

Replay a log file

```
$ canplayer -I candump_*.log
```

Getting Saucy With It

Split log files and do a binary search for desired replay behavior

Keep splitting and replaying until you have found the exact message

```
phil@ubuntu:~/Desktop/Projects/canplayground$ ./bifurcator.sh candump-2025-05-24_023038.log
Current segment: bifurcate_current.log (53 lines)
Replaying first half (A)...
Did the behavior occur in A? (y/n):
```

phil@ubuntu: ~/Desktop/Projects/canplayground

```
 can0  0CD   [5]  62 A1 D7 08 65
 can1  0CD   [5]  62 A1 D7 08 65
 can0  452   [6]  D4 80 F4 31 B6 01
 can1  452   [6]  D4 80 F4 31 B6 01
 can0  3F0   [4]  82 0B 9D 3F
 can1  3F0   [4]  82 0B 9D 3F
 can0  3C8   [8]  A6 7C A2 3F B6 F2 B6 7C
 can1  3C8   [8]  A6 7C A2 3F B6 F2 B6 7C
 can0  360   [7]  F9 76 2A 78 09 E0 1D
 can1  360   [7]  F9 76 2A 78 09 E0 1D
 can0  5BC   [8]  03 B7 8F 44 AE 30 61 79
 can1  5BC   [8]  03 B7 8F 44 AE 30 61 79
 can0  1E1   [8]  BD B9 C2 56 34 94 7F 60
 can1  1E1   [8]  BD B9 C2 56 34 94 7F 60
 can0  056   [7]  70 9F 8F 30 23 C1 CE
 can1  056   [7]  70 9F 8F 30 23 C1 CE
 can0  4FC   [2]  88 E8
 can1  4FC   [2]  88 E8
 can0  198   [8]  05 8B 0F 75 88 1D F4 55
 can1  198   [8]  05 8B 0F 75 88 1D F4 55
 can0  730   [7]  44 73 A2 77 F8 0A 72
 can1  730   [7]  44 73 A2 77 F8 0A 72
 can0  7A0   [8]  AE FD 28 42 00 43 F4 57
 can1  7A0   [8]  AE FD 28 42 00 43 F4 57
```

Diving Deeper into can-utils

Linux can-utils are a suite of utilities and tools for SocketCAN which include...

Basic tools to display, record, generate and replay CAN traffic

- candump : display, filter and log CAN data to files
- canplayer : replay CAN logfiles
- cansend : send a single frame
- cangen : generate (random) CAN traffic
- cansequence : send and check sequence of CAN frames with incrementing payload
- cansniffer : display CAN data content differences

CAN access via IP sockets

- canlogserver : log CAN frames and serves them
- bcmserver : interactive BCM configuration (remote/local)
- socketcand : use RAW/BCM/ISO-TP sockets via TCP/IP sockets
- cannelloni : UDP/SCTP based SocketCAN tunnel

ISO-TP tools ISO15765-2:2016 for Linux

- isotpsend : send a single ISO-TP PDU
- isotprecv : receive ISO-TP PDU(s)
- isotpsniffer : 'wiretap' ISO-TP PDU(s)
- isotpdump : 'wiretap' and interpret CAN messages (CAN_RAW)
- isotpserver : IP server for simple TCP/IP <-> ISO 15765-2 bridging (ASCII HEX)
- isotpperf : ISO15765-2 protocol performance visualisation
- isotptun : create a bi-directional IP tunnel on CAN via ISO-TP

J1939/ISOBus tools

- j1939acd : address claim daemon
- j1939cat : take a file and send and receive it over CAN
- j1939spy : spy on J1939 messages using SOC_J1939
- j1939sr : send/recv from stdin or to stdout
- testj1939 : send/receive test packet

Can-j1939 Kickstarter

https://github.com/linux-can/can-utils/blob/master/can-j1939-kickstart.md

Can-utils have a dedicated J1939 module

First steps with j1939

Use testj1939

When *can-j1939* is compiled as module, opening a socket will load it, **or** you can load it manually

```
modprobe can-j1939
```

Most of the subsequent examples will use 2 sockets programs (in 2 terminals). One will use CAN_J1939 sockets using *testj1939*, and the other will use CAN_RAW sockets using cansend+candump.

testj1939 can be told to print the used API calls by adding **-v** program argument.

receive without source address

Do in terminal 1

```
testj1939 -B -r can0
```

Send raw CAN in terminal 2

```
cansend can0 1823ff40#0123
```

You should have this output in terminal 1

```
40 02300: 01 23
```

This means, from NAME 0, SA 40, PGN 02300 was received, with 2 databytes, *01* & *23*.

now emit this CAN message:

```
cansend can0 18234140#0123
```

Working with python-can

Working with python-can

Install using:

```
$ pip install python-can
```

Documentation:

https://python-can.readthedocs.io/en/stable/index.html

Why choose this tool?

- Automation and scripting power
- Integration with larger python workflows and tools (cantools, pandas, scapy)
- Programmatic logic and conditional handling
  - Python gives you full control over logic
  - Wait for specific responses
  - Branch based on message contents
  - Retry, delay, or log conditionally

Receiving CAN with python-can

- Define a bus, message, and use the send() API

```
import can

bus = can.interface.Bus(channel='vcan0', bustype='socketcan')
print("Listening for messages...")
for msg in bus:
    print(msg)
```

Sending CAN with python-can

- Define a bus, message, and use the send() API

```
import can

bus = can.interface.Bus(channel='vcan0', bustype='socketcan')
msg = can.Message(arbitration_id=0x123, data=[0x11, 0x22, 0x33], is_extended_id=False)

try:
    bus.send(msg)
    print("Message sent")
except can.CanError:
    print("Message NOT sent")
```

Logging CAN with python-can

- python-can provides a built-in Logger to save CAN messages
- Works with real and virtual interfaces (e.g., can0, vcan0)
- Saves data in a format compatible with can-utils (candump style)
- Ideal for capturing traffic, analyzing diagnostics, and reproducibility

```
import can

bus = can.interface.Bus(channel='vcan0', bustype='socketcan')
logger = can.Logger("logfile.log")

for i in range(10):
    msg = bus.recv(timeout=1.0)
    if msg:
        logger(msg)
        print(f"Logged: {msg}")
```

Tools That Use Python-Can

Working with cantools

Install using:

```
$ python3 -m pip install cantools
```

Documentation:

https://cantools.readthedocs.io/en/latest/

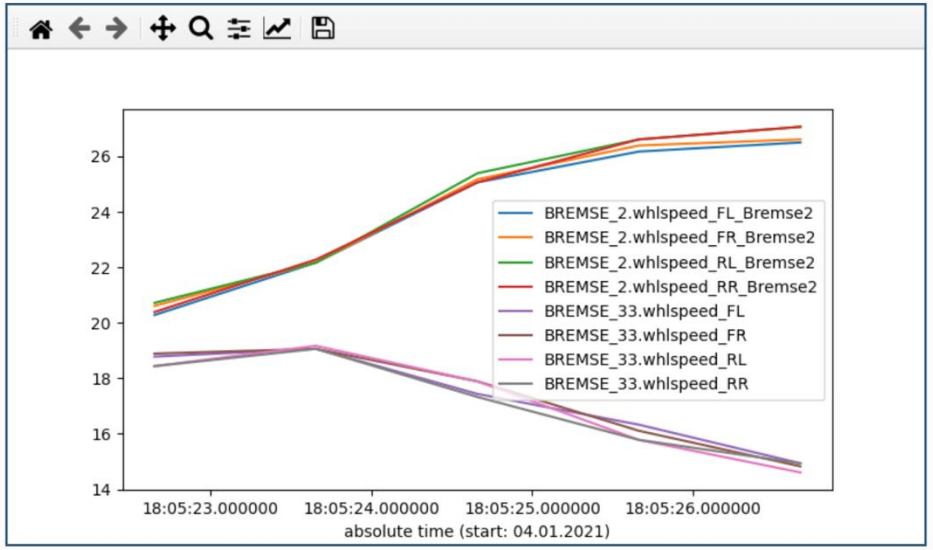
Why choose this tool?

- DBC, KCD, SYM, ARXML 3&4 and CDD file parsing
- CAN message encoding and decoding
- Simple and extended signal multiplexing
- Diagnostic DID encoding and decoding
- candump output decoder
- Graphical plots of signals

The plot subcommand

The plot subcommand is similar to the decode subcommand but messages are visualized using matplotlib instead of being printed to stdout.

```
$ candump -l vcan0
$ cat candump-2021-01-04_180521.log
(1609779922.655421) vcan0 00000343#B204B9049C049C04
(1609779922.655735) vcan0 0000024A#120527052E051905
(1609779923.657524) vcan0 00000343#C404C404CB04C404
(1609779923.658086) vcan0 0000024A#8B058B058B059205
(1609779924.659912) vcan0 00000343#5C04790479045504
(1609779924.660471) vcan0 0000024A#44064B0659064406
(1609779925.662277) vcan0 00000343#15040704F203F203
(1609779925.662837) vcan0 0000024A#8B069906A706A706
(1609779926.664191) vcan0 00000343#BC03B503A703BC03
(1609779926.664751) vcan0 0000024A#A006A706C406C406

$ cat candump-2021-01-04_180521.log | python3 -m cantools plot tests/files/dbc/abs.dbc
```



SAE J1939 for Python

Install using:

```
$ pip install can-j1939
```

Examples:

https://github.com/juergenH87/python-can-j1939/tree/master/examples

Why choose this tool?

- Support for J1939-22 (J1939 on CANFD)
- ECU (CA) Naming according SAE J1939/81
- Full featured address claiming procedure according SAE J1939/81
- Full support of transport protocol (up to 1785 bytes) according SAE J1939/21
- Full support of fd-transport protocol according SAE J1939/22 (J1939-FD)

TruckDevil

Install using:

```
$ git clone https://github.com/LittleBlondeDevil/TruckDevil.git
```

Examples:

https://github.com/LittleBlondeDevil/TruckDevil/tree/master/truckdevil/modules

Why choose this tool?

- Easy to learn module based
- Support for M2 device or any python-can supported device
- Message parsing and display of standard J1939 messages
- Module for UDS, ECU discovery, fuzzing, dumping, and sending J1939 messages
- Easy to add your own modules

Pretty-J1939

Install using:

```
$ pip install pretty_j1939
```

Examples:

https://github.com/nmfta-repo/pretty_j1939

Why choose this tool?

- Pretty-print J1939 traffic captured in candump logs with dissected data for J1939
- Convert a J1939 Digital Annex (Excel) file into a JSON structure
- ***note*** This will not parse out proprietary PGNs unless the info is added to your JSON file

Pretty-J1939

```
usage: pretty_j1939.py [-h] [--da-json [DA_JSON]] [--candata] [--no-candata] [--pgn] [--no-pgn
                       [--link] [--no-link] [--include-na] [--no-include-na] [--real-time] [--
                       candump

pretty-printing J1939 candump logs

positional arguments:
  candump               candump log, use - for stdin

optional arguments:
  -h, --help            show this help message and exit
  --da-json [DA_JSON]   absolute path to the input JSON DA (default="./J1939db.json")
  --candata             print input can data
  --no-candata          (default)
  --pgn                 (default) print source/destination/type description
  --no-pgn
  --spn                 (default) print signals description
  --no-spn
  --transport           print details of transport-layer streams found (default)
  --no-transport
  --link                print details of link-layer frames found
  --no-link             (default)
  --include-na          include not-available (0xff) SPN values
  --no-include-na       (default)
  --real-time           emit SPNs as they are seen in transport sessions
  --no-real-time        (default)
  --format              format each structure (otherwise single-line)
  --no-format           (default)
```

Some examples of pretty printing

Formatted content (one per line) next to candump data:

```
$ pretty_j1939.py --candata --format example.candump.txt | head
(1543509533.000838) can0 10FDA300#FFFF07FFFFFFFFFF ; {
                                                    ;     "DA": "All(255)",
                                                    ;     "PGN": "EEC6(64931)",
                                                    ;     "SA": "Engine #1(  0)",
                                                    ;     "Engine Variable Geometry Turbocharge
                                                    ; }
(1543509533.000915) can0 18FEE000#FFFFFFFFB05C6800 ; {
                                                    ;     "DA": "All(255)",
                                                    ;     "PGN": "VD(65248)",
                                                    ;     "SA": "Engine #1(  0)",
```

Single-line contents next to candump data:

```
$ pretty_j1939.py --candata example.candump.txt | head
(1543509533.000838) can0 10FDA300#FFFF07FFFFFFFFFF ; {"SA":"Engine #1(  0)","DA":"All(255)","P
(1543509533.000915) can0 18FEE000#FFFFFFFFB05C6800 ; {"SA":"Engine #1(  0)","DA":"All(255)","P
(1543509533.000991) can0 08FE6E0B#0000000000000000 ; {"SA":"Brakes - System Controller( 11)","
(1543509533.001070) can0 18FDB255#FFFFFFFF0100FFFF ; {"SA":"Diesel Particulate Filter Controll
(1543509533.001145) can0 0CF00400#207D87481400F087 ; {"SA":"Engine #1(  0)","DA":"All(255)","P
(1543509533.001220) can0 18FF4500#6D00FA00FF00006A ; {"SA":"Engine #1(  0)","DA":"All(255)","P
(1543509533.001297) can0 18FEDF00#82FFFFFF7DE70300 ; {"SA":"Engine #1(  0)","DA":"All(255)","P
(1543509533.001372) can0 1CFE9200#FFFFFFFFFFFFFFFF ; {"SA":"Engine #1(  0)","DA":"All(255)","P
(1543509533.001447) can0 18F00131#FFFFFF3F00FFFFFF ; {"SA":"Cab Controller - Primary( 49)","DA
(1543509533.001528) can0 18FEF131#F7FFFF07CCFFFFFF ; {"SA":"Cab Controller - Primary( 49)","DA
```

Caring Caribou

Install using:

```
$ git clone https://github.com/CaringCaribou/caringcaribou.git
$ cd caringcaribou && python setup.py install
```

Examples:

https://github.com/CaringCaribou/caringcaribou/tree/master/caringcaribou/modules

Why choose this tool?

- Dedicated modules for CAN message reverse engineering
- Powerful module for UDS RE such as attacking security seeds, DID enumeration, ECU discovery
- Support for DoIP and XCP reverse engineering

Caring Caribou

```
(venv) phil@ubuntu:~/Desktop/Projects/canplayground/caringcaribou$ caringcaribou --help
usage: caringcaribou [-h] [-i INTERFACE] module ...

------------------
CARING CARIBOU v0.7 - python 3.10.12 (main, Feb  4 2025, 14:57:36) [GCC 11.4.0]
\_\_    _/_/
    \__/
    (oo)\_______
    (__)\       )\/
        ||-----||
        ||     ||
------------------

A friendly car security exploration tool

positional arguments:
  module        Name of the module to run
  ...           Arguments to module

options:
  -h, --help    show this help message and exit
  -i INTERFACE  force interface, e.g. 'can1' or 'vcan0'

available modules:
  dcm, doip, dump, fuzzer, listener, module_template, send, test, uds, uds_fuzz, xcp
```

```
(venv) phil@ubuntu:~/Desktop/Projects/canplayground/caringcaribou$ caringcaribou -i vcan0 uds -h

------------------
CARING CARIBOU v0.7 - python 3.10.12 (main, Feb  4 2025, 14:57:36) [GCC 11.4.0]
------------------

Loading module 'uds'

usage: caringcaribou uds [-h]
                         {discovery,services,subservices,ecu_reset,testerpresent,security_seed,dump_dids,read_mem,auto}
                         ...

Universal Diagnostic Services module for CaringCaribou

positional arguments:
  {discovery,services,subservices,ecu_reset,testerpresent,security_seed,dump_dids,read_mem,auto}

options:
  -h, --help            show this help message and exit

Example usage:
  caringcaribou uds discovery
  caringcaribou uds discovery -blacklist 0x123 0x456
  caringcaribou uds discovery -autoblacklist 10
  caringcaribou uds services 0x733 0x633
  caringcaribou uds ecu_reset 1 0x733 0x633
  caringcaribou uds testerpresent 0x733
  caringcaribou uds security_seed 0x3 0x1 0x733 0x633 -r 1 -d 0.5
  caringcaribou uds dump_dids 0x733 0x633
  caringcaribou uds dump_dids 0x733 0x633 --min_did 0x6300 --max_did 0x6fff -t 0.1
  caringcaribou uds read_mem 0x733 0x633 --start_addr 0x0200 --mem_length 0x10000

(venv) phil@ubuntu:~/Desktop/Projects/canplayground/caringcaribou$
```

https://github.com/CaringCaribou/caringcaribou/tree/master for help

Scapy

Install using:

```
$ git clone https://github.com/secdev/scapy
$ cd scapy
$ ./run_scapy
```

Examples:

https://munich.dissec.to/kb/chapters/can/can-scapy.html
https://scapy.readthedocs.io/en/latest/layers/automotive.html

Why choose this tool?

- Protocol stack and flexibility (CAN, ISO-TP, UDS message support)
- Built in sniffing and injection
- Built in packet manipulation and automation
- Support for other protocols (learn one tool for ethernet, bluetooth, etc)

Scapy – CAN send message

```
1  from scapy.all import *
2  load_layer("can")
3
4  # Configure Socket
5  conf.contribs['CANSocket'] = {'use-python-can': False} # default
6  load_contrib('cansocket')
7  socket = CANSocket(channel='vcan0')
8
9  #Send a Packet
10 frame = CAN(flags='extended', identifier=0x18da00f1, data=bytes.fromhex('021101'))
11 socket.send(frame)
```

Scapy – CAN receive messages

```python
from scapy.all import *
load_layer("can")

# Configure Socket
conf.contribs['CANSocket'] = {'use-python-can': False} # default
load_contrib('cansocket')
socket = CANSocket(channel='vcan0')

# Recieve CAN
while True:
    rx_packet = socket.recv()
    rx_packet.show()

```

Scapy – Low Level CAN attacks

- https://munich.dissec.to/kb/chapters/can/can-lowlevelattacks.html
- Enumerates many low-level attacks on CAN devices including
  - Bus Flood Attacks
  - Simple Frame Spoofing
  - Adaptive Frame Spoofing
  - Frame Tampering
  - Error Passive Spoofing Attacks
  - Bus-off Attacks

Scapy – Protocol Layers

- Scapy has support for discecting the following protocol layers that you might find useful
  - ISO-TP
  - UDS
  - CCP/XCP
  - ODB
  - AUTOSAR SecOC
- https://scapy.readthedocs.io/en/latest/layers/automotive.html#layers

Using python-can in Windows

Connecting PCAN to python-can

1) Install PEAK drivers from
   - https://www.peak-system.com/quick/DrvSetup
2) Verify connected device using PEAK tool.
3) Specify channel and interface

```python
import can

bus = can.interface.Bus(channel = 'PCAN_USBBUS1', interface='pcan', bitrate=500000)
print("Connected to PEAK CAN device")

can_id = 0x0CF00331
data = [0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08]
message = can.Message(arbitration_id=can_id, data=data, is_extended_id=True)

bus.send(message)
```

GUI Based Tools

Wireshark

Install using:

```
$ sudo apt install wireshark can-utils
$ sudo usermod -aG wireshark $USER
$ newgrp wireshark
```

Examples:

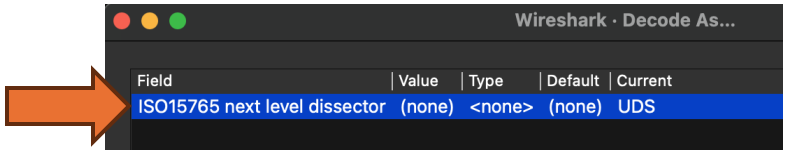
https://github.com/hexsecs/cybertruck/blob/main/chapters/wireshark.md

Why choose this tool?

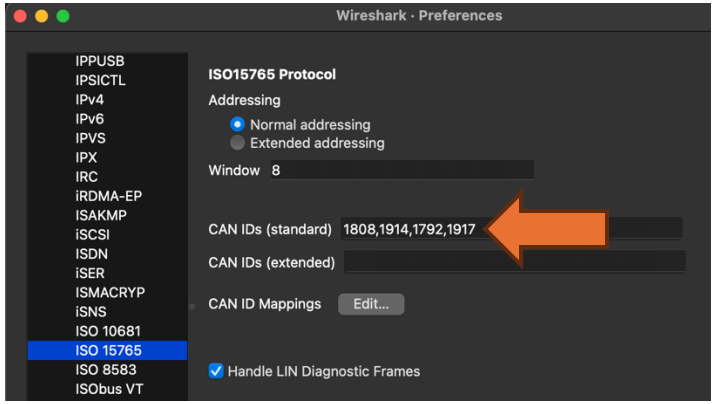
- Dissectors for CAN, ISO-TP, J1939, DoIP, etc
- Powerful logging and filtering functions
- Support for many other protocols (learn one tool for ethernet, DoIP, bluetooth, etc)

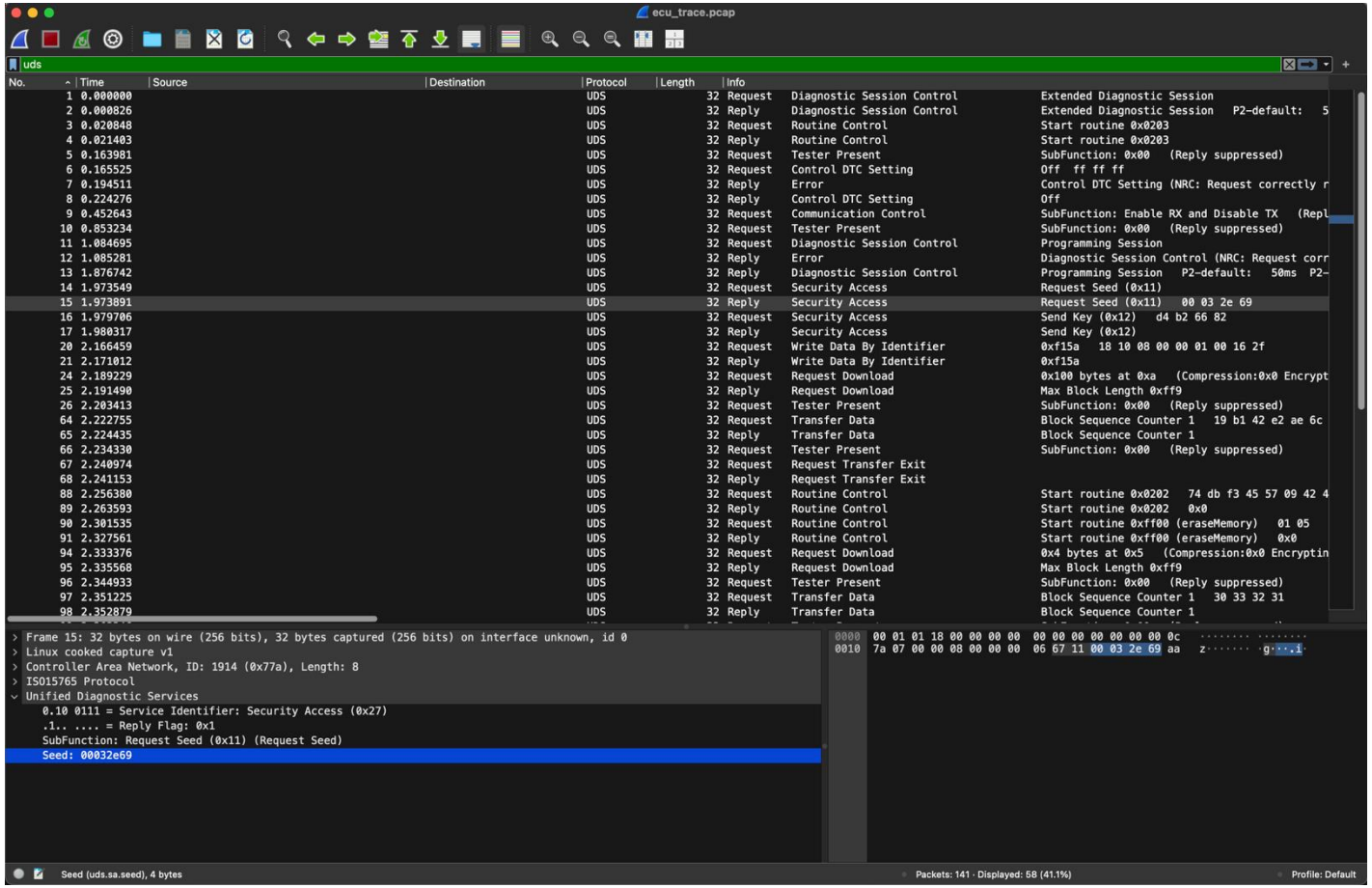
Wireshark – Dissecting UDS

Add a UDS dissector…



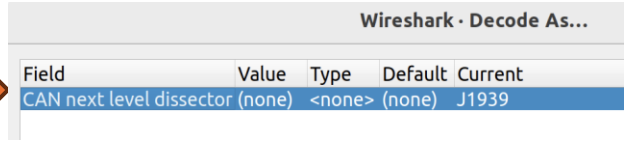
Configure the diagnostic IDs…

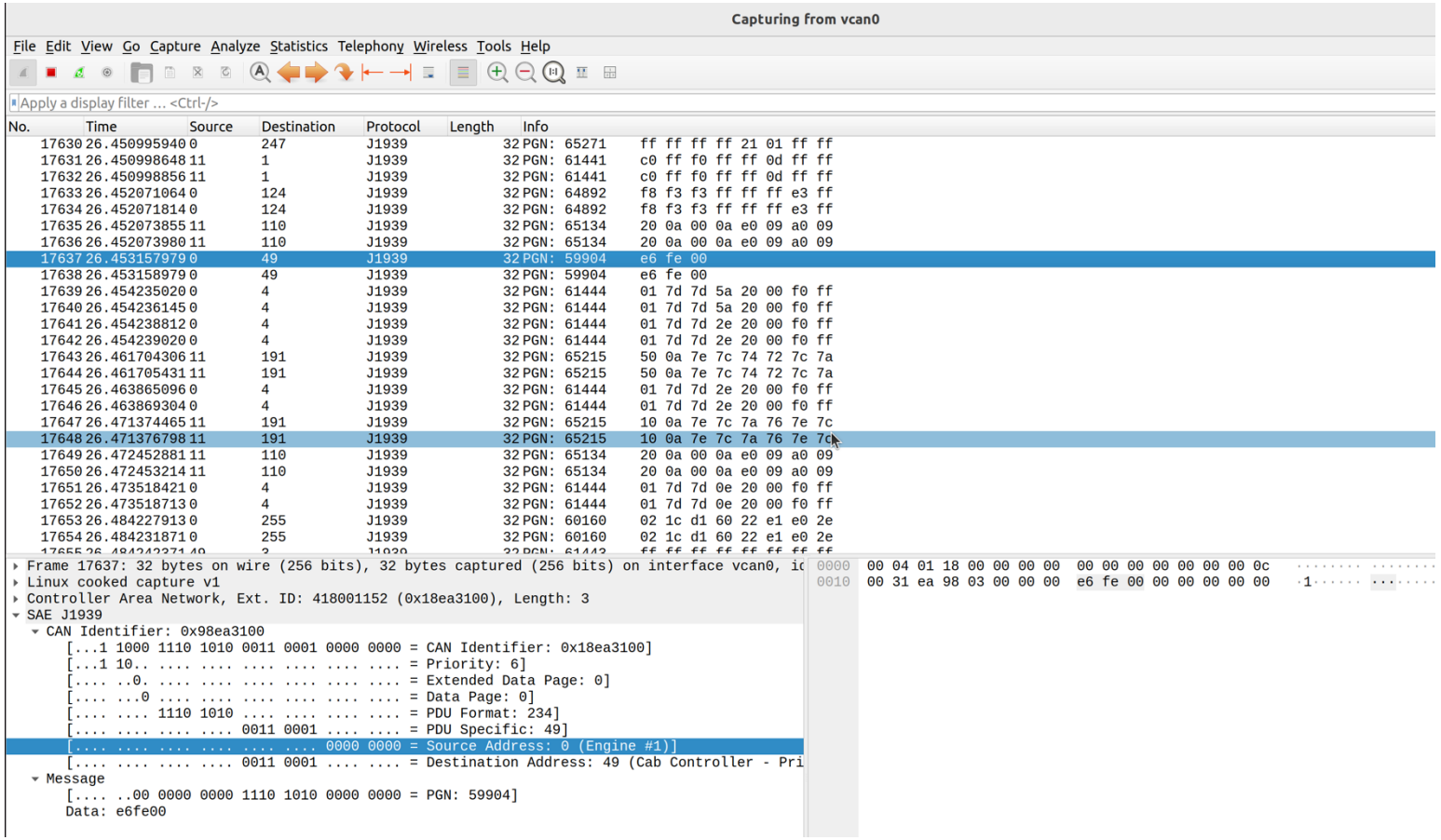




Wireshark – Dissecting J1939

Add a J1939 dissector...





SavvyCan

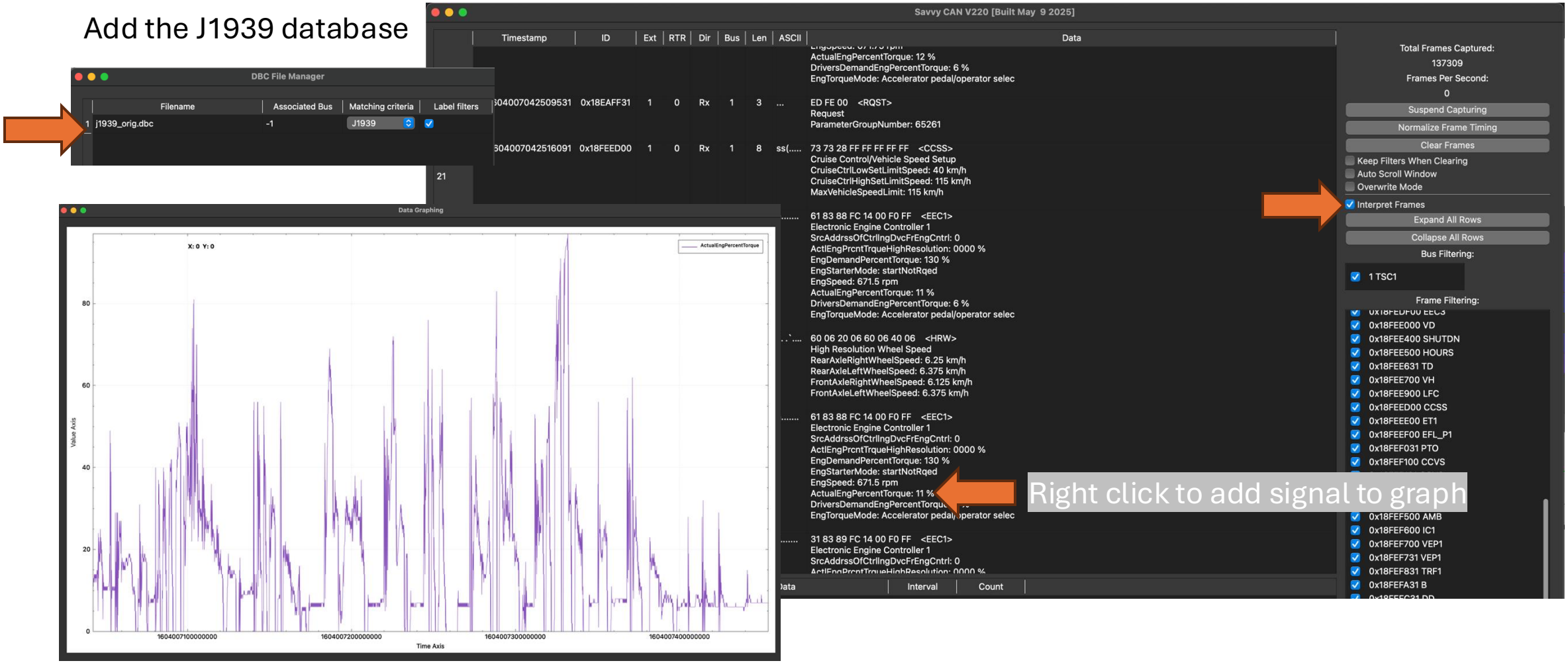
Install using:
https://github.com/collin80/SavvyCAN/releases
Get latest app image and install.

Docs:
https://www.savvycan.com/docs/

Why choose this tool?
- Dissectors for CAN, ISO-TP, J1939, etc
- Powerful reverse engineering tools
- Signal parsing and graphing for J1939 dbc (ask mentors for dbc)
- Powerful logging and filtering functions
- GUI based and easy to learn

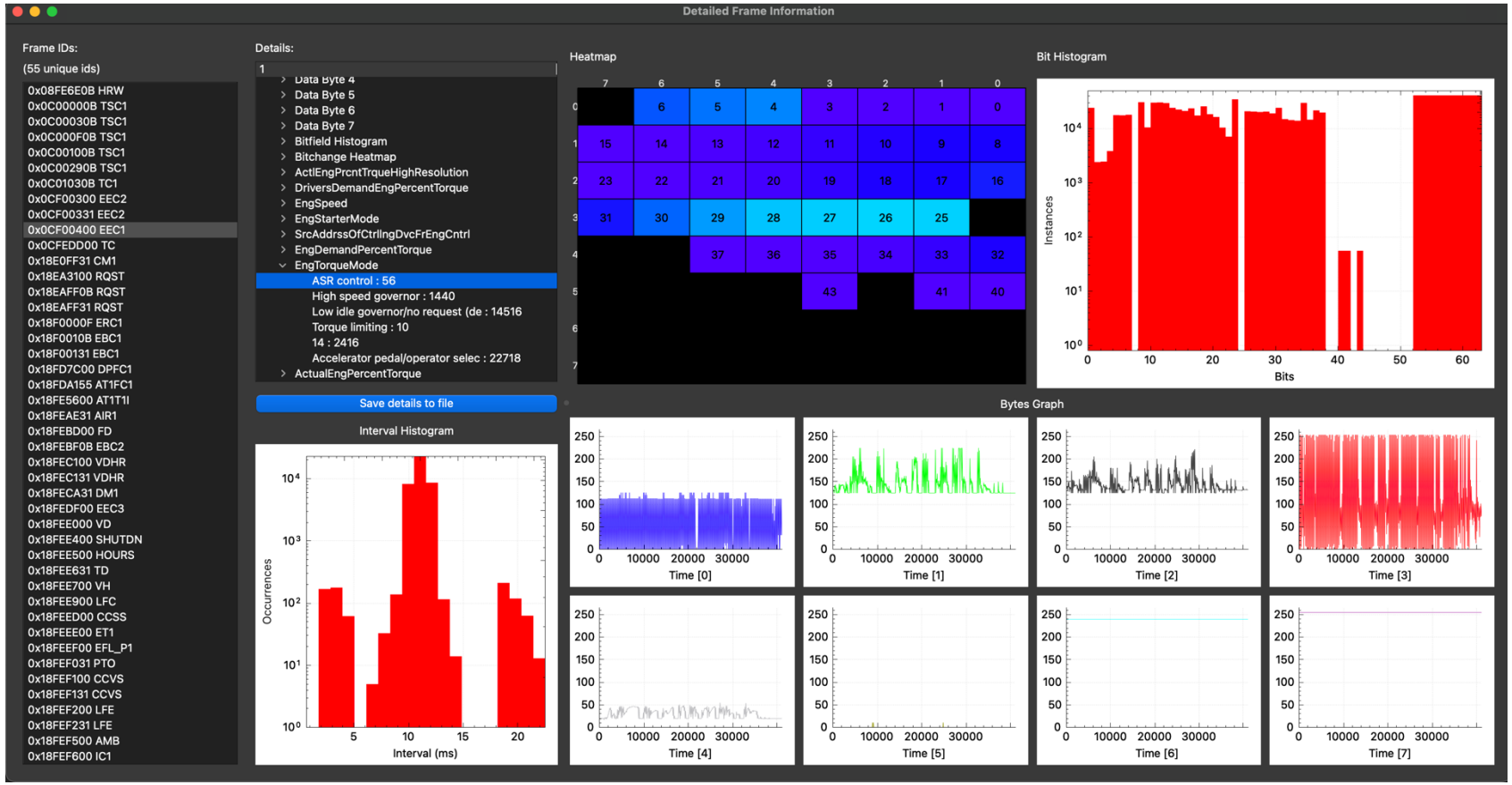
SavvyCAN – J1939 Signal Parsing

Add the J1939 database



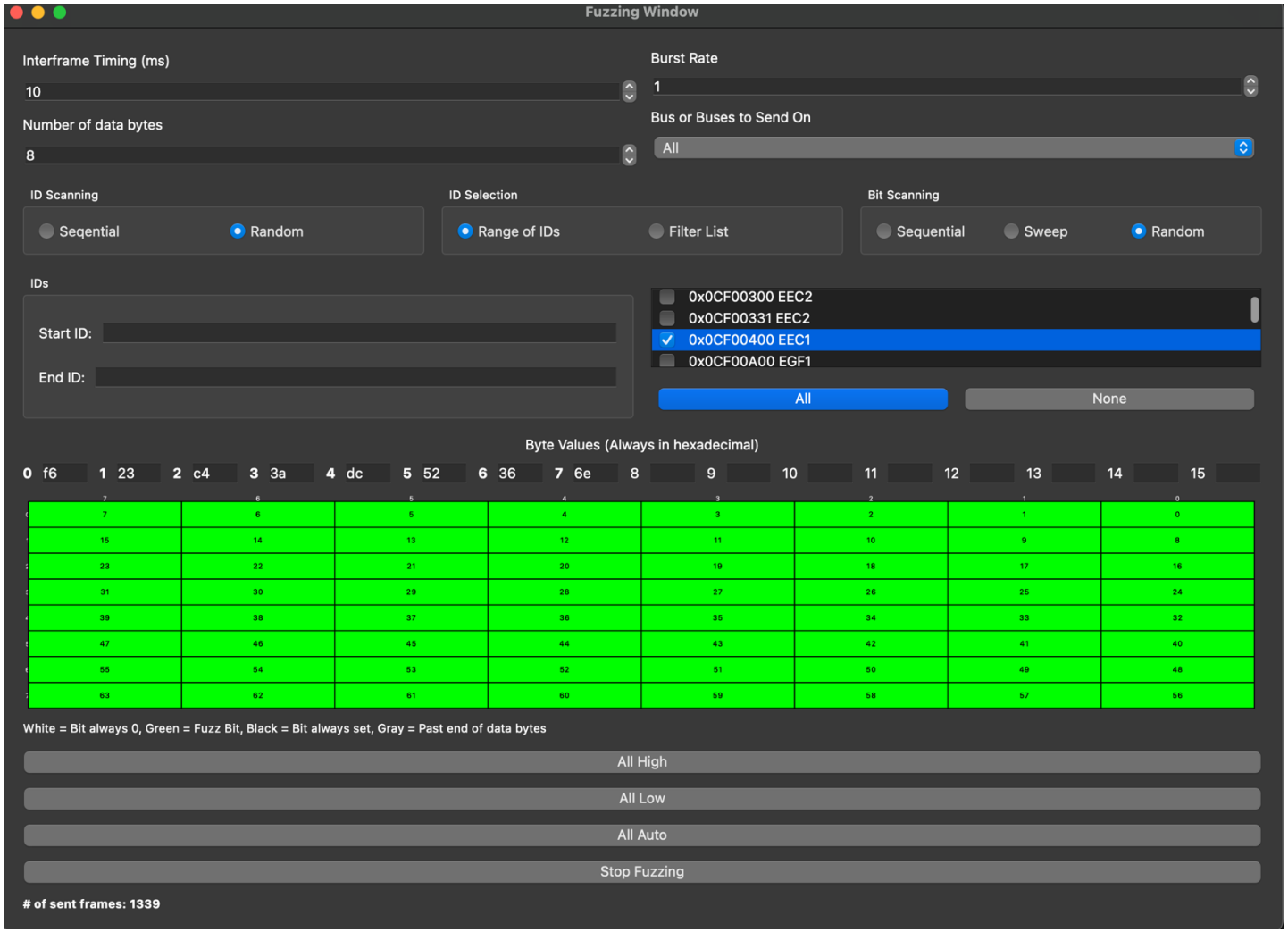
SavvyCAN – Frame Reverse Engineering

Detailed frame information view can give clues to reversing signals.

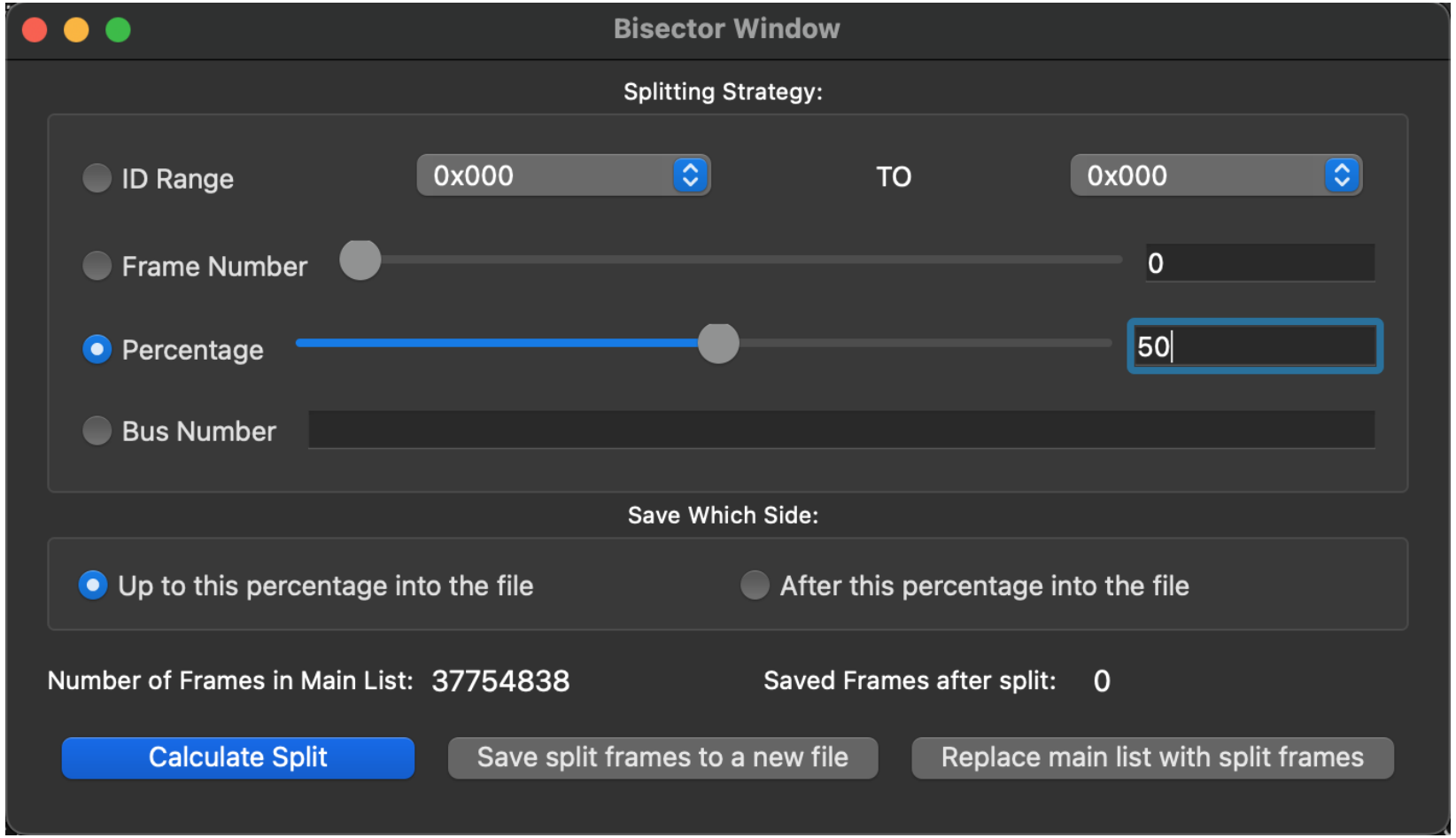


SavvyCAN – CAN Fuzzing Module

CAN fuzzing module



SavvyCAN – Log Bisection and Replay



Commercial Scan Tools

Examples:

- Cummins PowerSpec
- Bendix ACOM
- WABCO Toolbox
- Jpro
- DG Tech Diagnostics

Why choose these tools?

- May implement proprietary diagnostic routines
- Different flavors that are less supported in OSS tools (J1939 diag, KWP2000, proprietary diag)
- They may contain the seed-key unlock routines
- Often times used to reverse engineer proprietary messages / diagnostics
- They often use a “standardized” RP1210 DLL (your PEAK interface supports this RP1210 API)

Misc Tools

Interfacing with J1708

- **Use RP1210 Adapters with J1708 support**
  - Examples: DG Tech DPA4, DPA5, Nexiq USBLINK
- **Use py-hv-networks tool to interface over python**
  - https://github.com/TruckHacking/py-hv-networks
  - Windows – use RP1210 DLL (MAY NEED TO USE 32-BIT!!!!!!)
  - Linux – You can use the TruckDuck hardware
- **Mentors who are knowledgeable with J1708 and PLC**
  - Ben Gardiner

Coding with AI (or vibe-coding your way through CyberTruck)

Do’s and Don’ts

DO NOT!

- Transmit any information or data that is in scope of your NDA
- This includes
  - Private information
  - CAN logs
  - Files and documents
  - Pictures
  - If you are not 100% sure it’s okay, you should not send it

Do’s and Don’ts

What you can do…

- Ask AI about public knowledge
- Ask AI about how to use a public tool (such as the tools in can-gen, scapy, TruckDevil, etc.)
- Ask AI to write you code or code samples, provided you aren’t breaking any NDA restrictions!

Do’s and Don’ts

Examples of acceptable use…

- “Write a bash script to bifurcate a candump log file and replay it using canplayer. Ask me after each round if I found what I was looking for. If yes, continue splitting the confirmed file and replaying until the single CAN message is identified.”

- “Write a python-can script to send out periodic tester present messages at a rate of 2 seconds.”

AI-Generated Script to Read CAN Messages

```python
import can

bus = can.interface.Bus(channel='can0', bustype='socketcan')

print("Listening on can0...")
for msg in bus:
    print(msg)
```

Prompt to AI:

“Generate a Python script that listens on can0 using python-can and prints each message.”

Adding complexity

Prompt to AI:

“Write a python-can script that will iterate through sending a read request for all diagnostic DIDs from 0x0000 to 0xffff. The script should use 29bit IDs.

Copy Edit

```python
import can
import time

# Constants
TX_ID = 0x18DB33F1  # 29-bit CAN ID: Functional request (example)
SERVICE_ID = 0x22   # UDS ReadDataByIdentifier

def send_did_requests(channel='can0', bustype='socketcan'):
    bus = can.interface.Bus(channel=channel, bustype=bustype, fd=False)

    print("Starting DID scan on extended CAN ID...")

    try:
        for did in range(0x0000, 0x10000):  # 0x0000 to 0xFFFF
            # Format DID into 2 bytes
            did_bytes = did.to_bytes(2, byteorder='big')

            # Construct UDS request: [SID, DID_High, DID_Low]
            data = bytes([SERVICE_ID]) + did_bytes

            # Pad to 8 bytes if necessary
            data += bytes(8 - len(data))

            # Create CAN frame
            msg = can.Message(arbitration_id=TX_ID,
                              is_extended_id=True,
                              data=data)

            # Send frame
            bus.send(msg)
            print(f"Sent: SID=0x{SERVICE_ID:02X}, DID=0x{did:04X}")

            time.sleep(0.01)  # Slight delay to avoid bus overload

    except KeyboardInterrupt:
        print("\nScan interrupted by user.")
    except Exception as e:
        print(f"Error: {e}")

if __name__ == "__main__":
    send_did_requests()
```

Choosing the Right Tool

Security Use Case Mapping

Use Case	Recommended Tools
J1939/PGNs Message Decode	Wireshark, SavvyCAN, pretty-j1939, TruckDevil
Message Logging	can-utils, SavvyCAN, Wireshark, TruckDevil, Python-can
Low-level message injection/scripting	Scapy, can-utils, TruckDevil, python-can
PGN fuzzing / injection testing	TruckDevil, Scapy, SavvyCAN, python-can
UDS SID brute-force or diagnostics	Caring Caribou, Scapy, python-can
Layered protocol scripting	Scapy
Batch analysis / automated decoding	cantools, pretty-j1939, Wireshark, TruckDevil
XCP research	Wireshark, Caring Caribou

Tool Comparison

Tool	J1939 Support	Security Usefulness	Packet Crafting	Sniffing	Fuzzing	Pros	Cons
Caring Caribou	✅ Basic	✅ UDS brute force, DoS	⚠️ Basic scripts	✅ Yes	⚠️ Limited	Purpose-built for security testing	Outdated, limited PGN/J1939 support
Scapy	⚠️ Manual	✅ Custom attacks, fuzzing	✅ Full stack	✅ Yes	✅ Scriptable	Full control via Python, layered protocols (CAN/ISOTP/UDS)	Requires custom J1939 definitions, less user-friendly
TruckDevil	✅ Excellent	✅ ECU probing & PGN fuzzing	✅ CLI-driven	✅ Yes	✅ Native	Built for J1939 & truck ECU testing	CLI only, limited documentation
pretty-j1939	✅ Excellent	⚠️ Passive only	❌	✅ piped input	❌	Best CLI-based J1939 PGN decoder	No fuzzing or message injection
cantools	✅ via DBC	⚠️ Passive only	❌	✅ File/stream	❌	Excellent for DBC decoding & validation	Requires DBC files, no attack features
can-utils	✅ Basic	⚠️ Low-level sniff/send, quick and dirty	✅ Hex based	✅ Yes	❌	Lightweight & fast, great for CLI testing	No PGN decoding, no protocol awareness
SavvyCAN	✅ via DBC	✅ Visual fuzzing/injection	✅ GUI + script	✅ GUI/CLI	✅ Visual	Great for reverse engineering, signal tracking, fuzzing	GUI-heavy, scripting uses Qt/JavaScript
Wireshark	✅ Basic	⚠️ Passive + protocol insight	❌	✅ GUI	❌	Excellent decoding, PGN/SPN stats, time analysis	No crafting or active manipulation

GUI vs CLI Tool Comparison

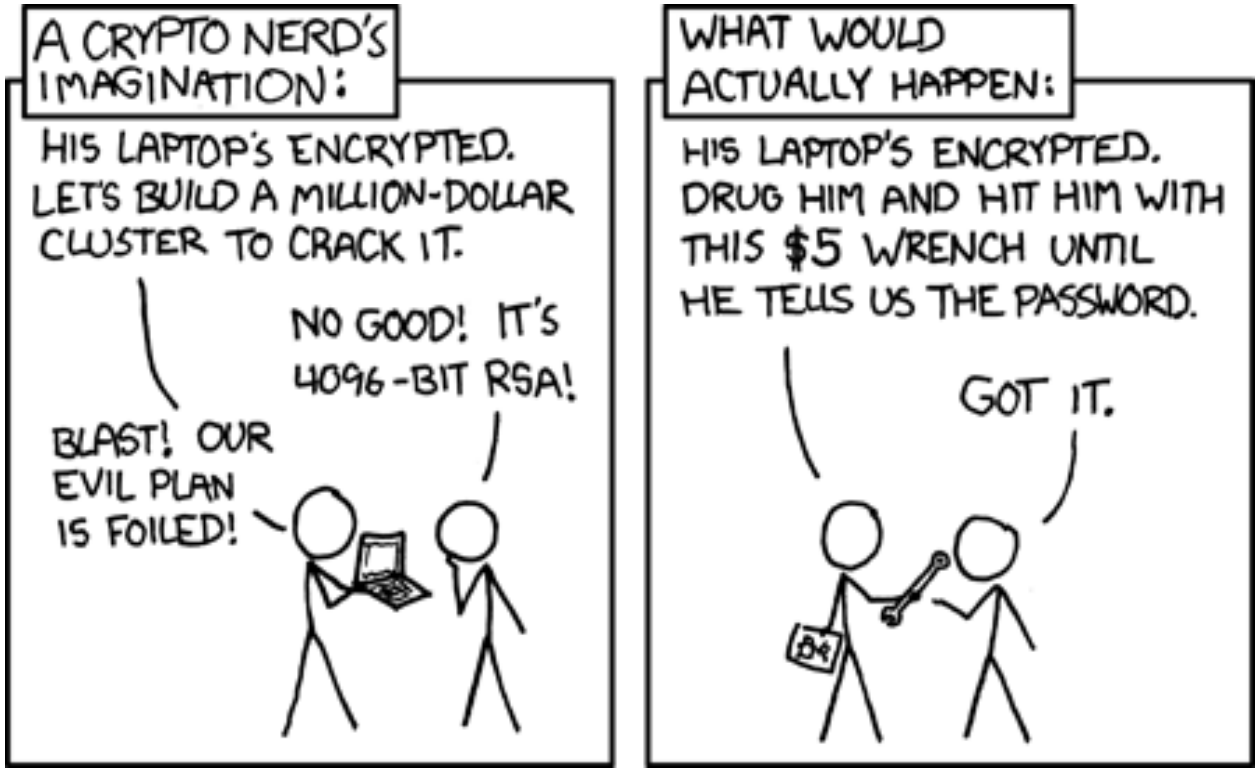
Tool	CLI	GUI	Scriptable	Best For
Scapy	✅	❌	✅ Python	Custom CAN/ISOTP/UDS/J1939 message crafting
TruckDevil	✅	❌	✅ Python	Targeted J1939 security modules including PGN decode
SavvyCAN	⚠️ Basic	✅ Full	✅ JavaScript	Visual reverse engineering, batch injection
Wireshark	✅ (tshark)	✅ Full	❌	PGN/SPN visualization and timing analysis
pretty-j1939	✅	❌	❌	Clean readable PGN decoding in pipes/logs
cantools	✅	❌	✅ Python	Scripting PGN analysis with DBCs
can-utils	✅	❌	✅ Bash	Quick injection and sniffing from the shell, logging

Final Thoughts

ABCD – “Always Be CAN Dumping”

- There are **TONS** of tools that can parse though through CAN logs
- The standard SocketCAN candump format can be converted to most any other format you need (such as pcap)
- Candumps are useful to record your work, as well as replay or reverse engineer messages

KISS – “Keep It Simple Students”



Don’t overthink it. If there is a simple solution, try that first.

Don’t Be Afraid to Ask for Help

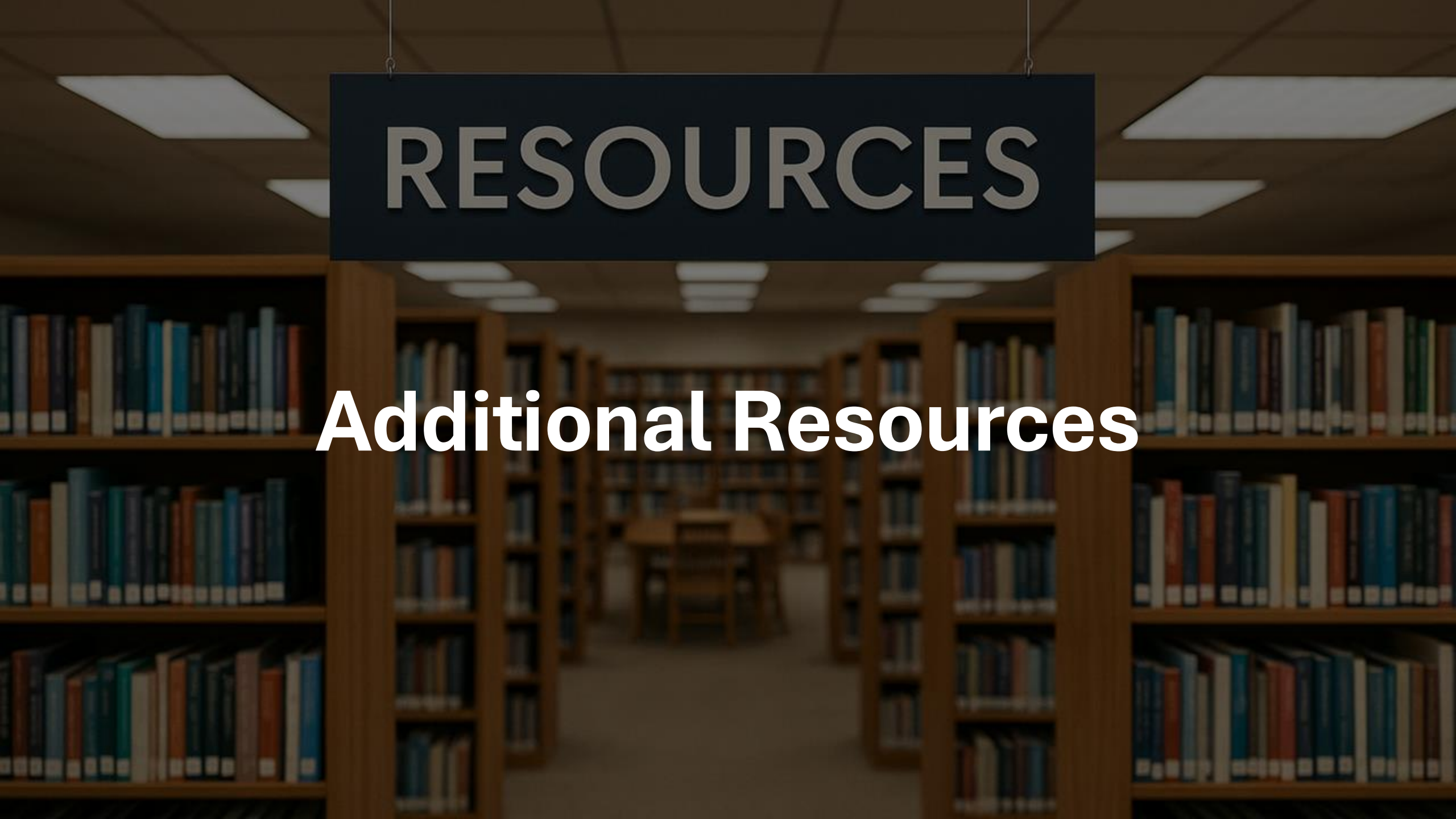
- If you aren’t sure what tool to use or how to accomplish a task, **ask your mentors and truck teams**
- Many mentors, instructors, and truck teams have participated in CyberTruck many times
- If they don’t know themselves, they will likely know who to talk to
- Remember, the goal is to learn, the mentors are here to **help** you, but not **carry** you

Use Your Time Wisely

What seems like a lot of time for the assessment goes by **very fast.**

Experts often spend days or months doing security research.

1. If a truck team or mentor is nudging you in a different direction, you may be going down the wrong rabbit hole
2. **If a truck team or mentor is nudging you in a different direction, you may be going down the wrong rabbit hole**
3. **If a truck team or mentor is nudging you in a different direction, you may be going down the wrong rabbit hole**
4. If they've told you multiple times…see steps 1-3 above



Additional Resources

Additional Resources

- **My Cybertruck Repository**
  - https://github.com/hexsecs/cybertruck
- **Awesome CANbus**
  - https://github.com/iDoka/awesome-canbus
- **Sharika Kumar – 2023 – Diagnostic Systems (Includes RP1210 Shimming)**
  - https://www.cybertruckchallenge.org/wp-content/uploads/2023/06/Diagnostic-Systems-by-Sharika-Kumar.pdf
- **Dr. Daily - Intro to J1939 (Includes RP1210 Shimming)**
  - https://www.cybertruckchallenge.org/wp-content/uploads/2023/06/Introduction-to-SAE-J1939-CyberTruck-2023.pdf
- **Hannah Silva – 2022 CyberTruck Training (Includes TruckDevil)**
  - https://www.cybertruckchallenge.org/wp-content/uploads/2022/08/Silva_Heavy_Vehicle_Networks.pdf
  - https://www.cybertruckchallenge.org/wp-content/uploads/2022/08/Silva_Vehicle_Network_Security.pdf
  - https://www.youtube.com/@NMFTA-Inc <-- Truck Devil Training Videos
- **NMFTA – Blind Wireless Seed Key Unlock**
  - https://nmfta.org/wp-content/media/2025/01/Blind-Wireless-Seed-Key-Unlock-Whitepaper-final.pdf

Additional Resources

- **Heavy Vehicle CAN data**
  - https://www.engr.colostate.edu/~jdaily/J1939/candata.html
- **Automated Reverse Engineering and Fuzzing of the CAN bus**
  - https://downloads.distrinet-research.be/software/sancus/publications/werquin2019.pdf
- **Expanding the Attack Scenarios of SAE J1939: A Comprehensive Analysis of Established and Novel Vulnerabilities in Transport Protocol**
  - https://arxiv.org/pdf/2406.00810v1