Practical Car Hacking CTF Teaser by Willem Melching

Walkthrough by Stephan DB

https://www.linkedin.com/in/dbstephan/

1. Challenge:
Volkswagen CAN Checksum

Task:
The goal of this challenge is to compute the correct one byte checksum XX for the CAN message with payload XX0f0300. The flag is of the format CTF{XX}.

Some background info and a sample python app is given for the Autosar 8bit checksum with the polynomial 0x2F

The examples:
- 74000300
- c1010300
- 31020300
- 84030300
- fe040300

It is stated that the checksum uses a fixed “secret” byte which is added to the data and the checksum then goes over those 4 bytes. So for the first example it is 00 03 00 xx as input with xx as the secret byte for the Autosar crc and the result should be 74.

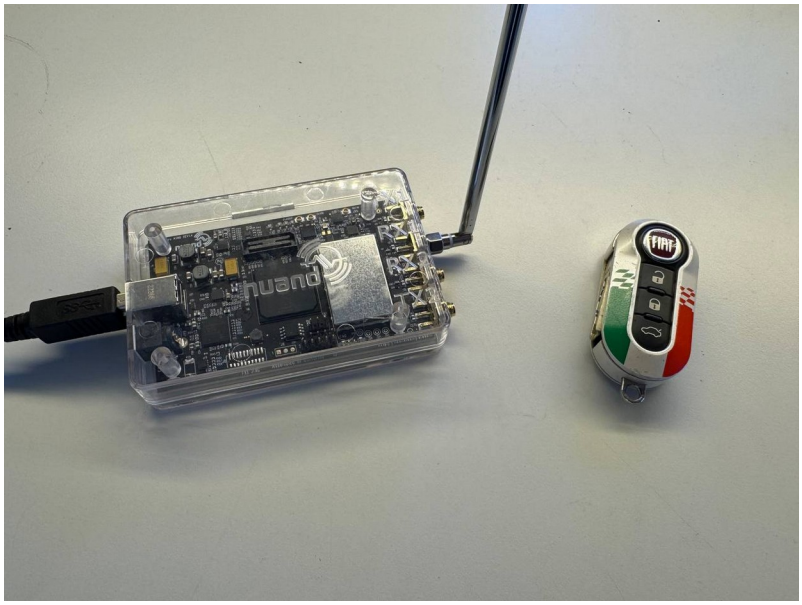
```
import crcmod
crc = crcmod.mkCrcFun(
    poly=0x100 + 0x2F,    #autosar crc poly
    initCrc=0xFF,         #autosar crc init
    rev=False,
)

for secret in range(256):
    data = b'\x00\x03\x00' + secret.to_bytes(1, byteorder='big')
    checksum = crc(data) ^ 0xff
    if checksum == 0x74:
        print(f"Found secret_byte: 0x{secret:02X}")
        data = b'\x0F\x03\x00' + secret.to_bytes(1, byteorder='big')
        checksum = crc(data) ^ 0xff
        print(f"CTF: {checksum:02X}")
```

It finds the correct secret byte 0xC3 and gives then the correct checksum for the CTF payload → 35

Practical Car Hacking CTF Teaser by Willem Melching

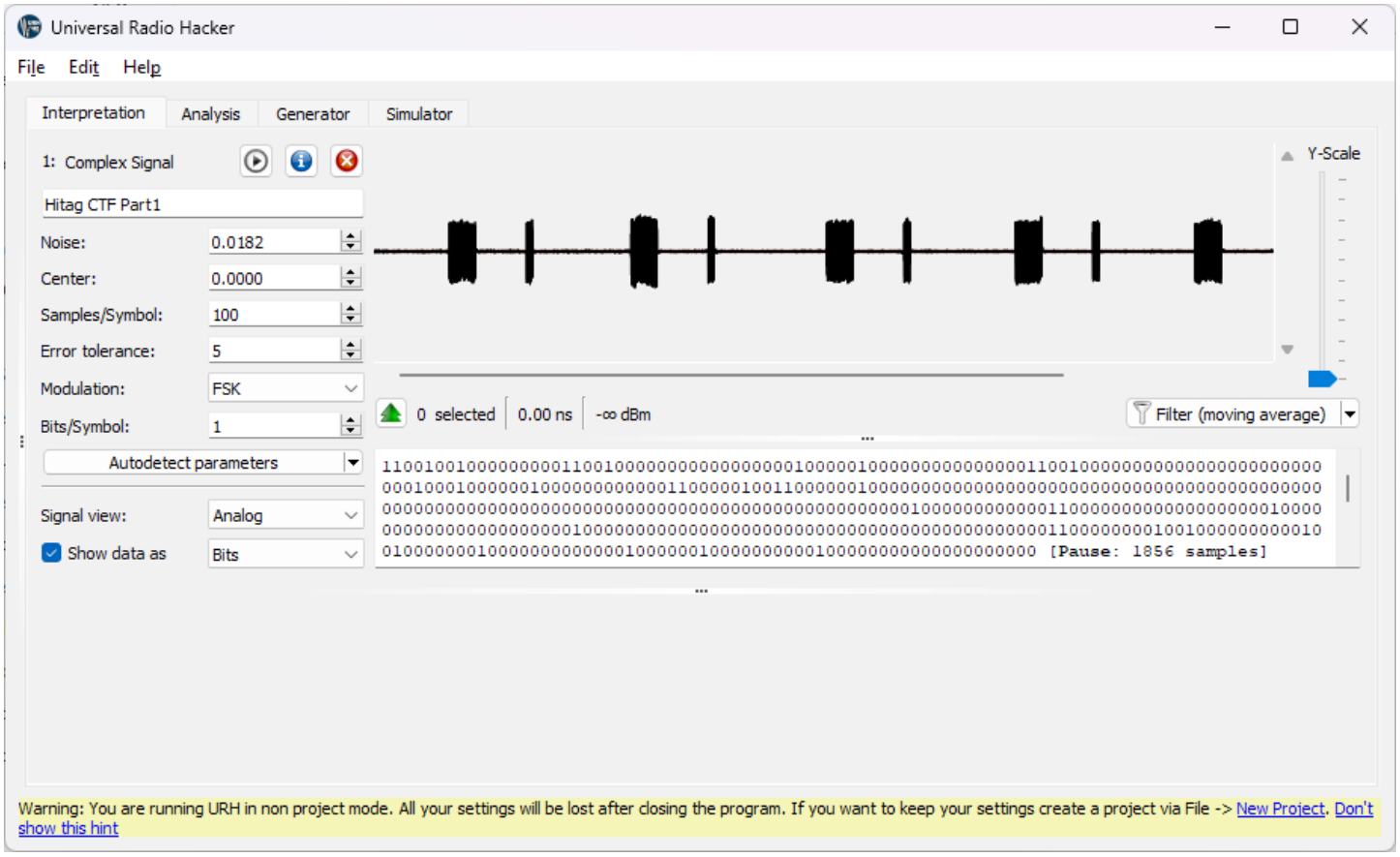
Walkthrough by Stephan DB

2. Challenge:
Hitag2 Keyfob ID (Part 1)



Task:
This challenge contains a recording from a keyfob featuring a Hitag2 cipher for RKE. The keyfob transmits a message containing a plaintext keyfob ID, counter and button followed by a MAC/secret. Attached to this challenge you will find a SDR recording of 6 presses of the unlock button.

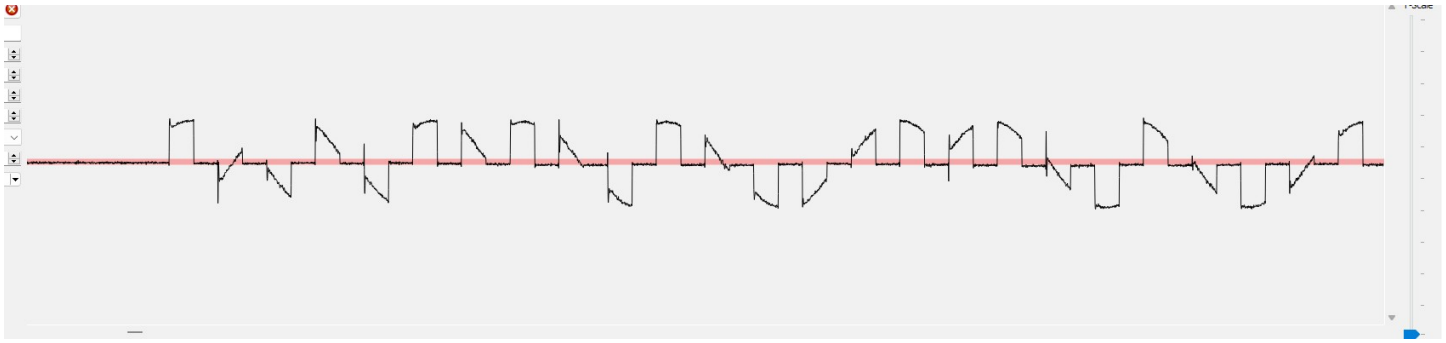
First step is loading the file into Universal Radio hacker. We can see already the signals. One long and one short signal, repeated various times for the several presses of the unlock button.



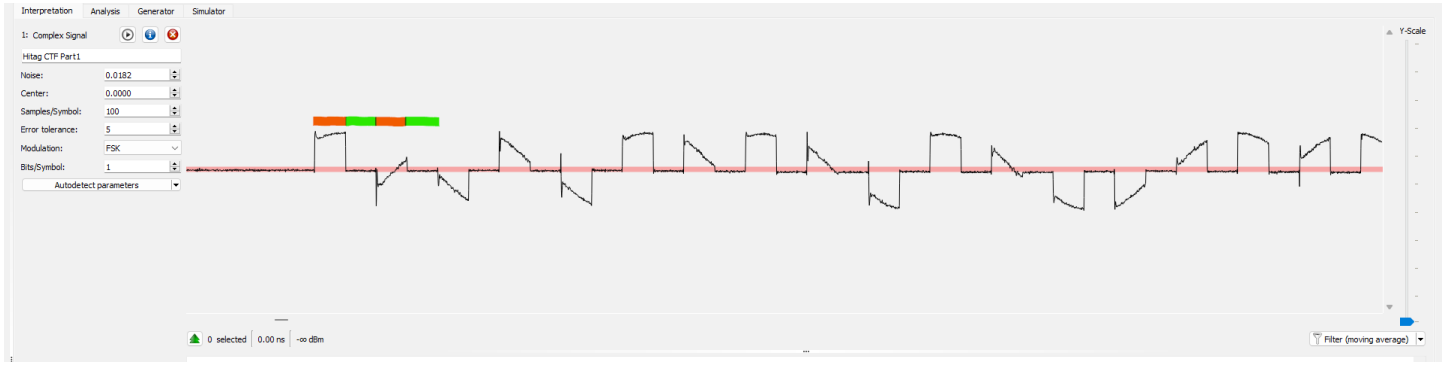
Practical Car Hacking CTF Teaser by Willem Melching

Walkthrough by Stephan DB

When we zoom a bit into the signal we can see this:



Doesn’t seem to be a really good recording, but we will try to get the most out of it.



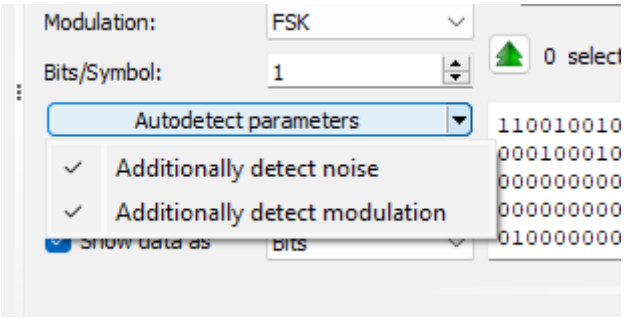
I marked the signal here with red and green. Red is where the signal is present and green where no signal is present. This looks pretty much like OOK (On-Off Keying), where a high amplitude is used to represent a binary 1 and the absence of any amplitude to represent a 0. The morse signal is a well known example of OOK.

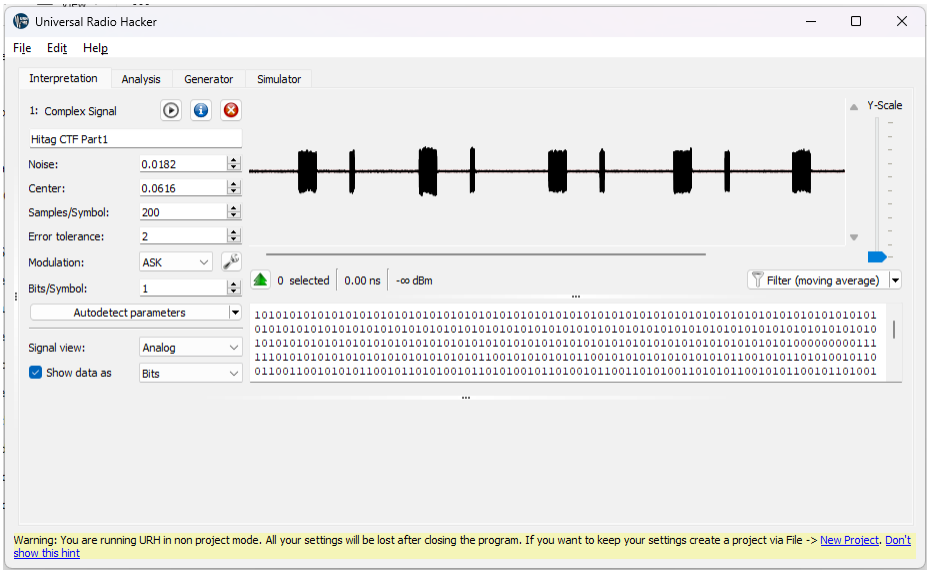
We will convert this signal now to something usable for the two CTFs we need to solve.

Practical Car Hacking CTF Teaser by Willem Melching

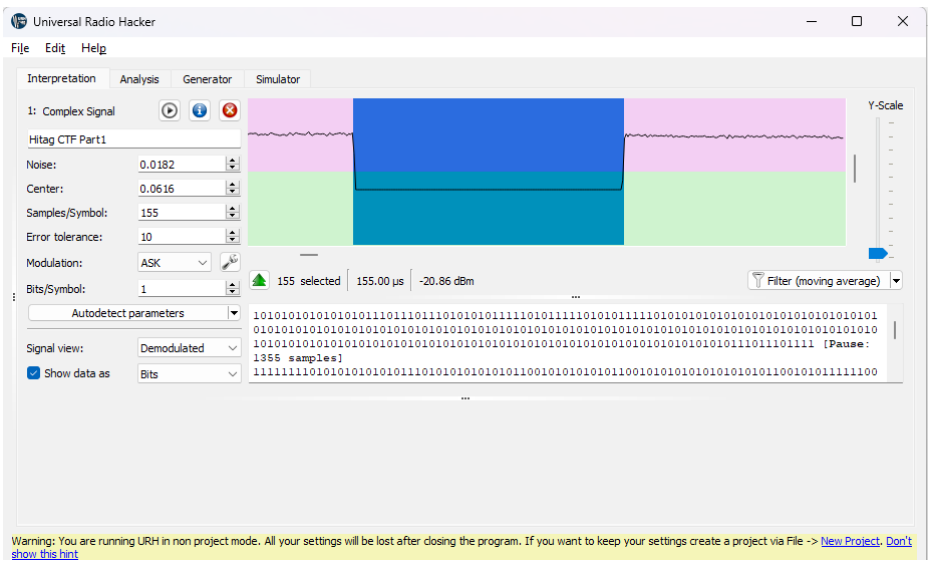
Walkthrough by Stephan DB

We can use the autodetect (make sure to check the two “Additionally” things), but the signal won’t be ok yet. It is detected correctly a ASK/OOK signal. So we can start from there.





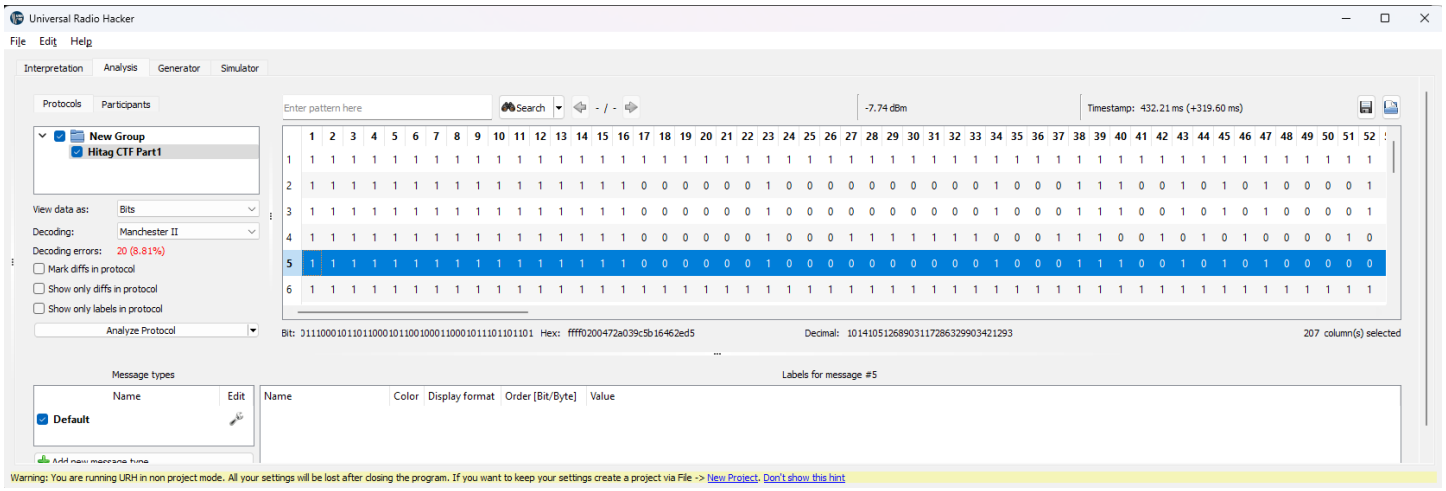
The Autodetect thinks the signals are 200 Samples/Symbol, but this is not correct. When switching the Signal view to “demodulated” we can actually have a better look and measure the correct value there.



Practical Car Hacking CTF Teaser by Willem Melching

Walkthrough by Stephan DB

With the values for the Samples/Symbol now to 155 and an errorrate of 20 we can switch now to the Analysis tab.



The signal is Manchester encoded and we have to select that in the encoding. I use here Manchester II which is just the signal inverted (which is correct) in comparison to Manchester I. Pretending we are looking at a unknown signal we could search for the counter-value and if that is counting down it would be a hint to use the other Manchester encoding. With Manchester II this signal has an incrementing counter and we got confirmation of using “II”.
When we now press on the line numbers we can see in “Hex:”-box the signal as a byte-stream. When it looks like here starting with FFFF and is 13 bytes long, then it should be a good signal. Some will have a bad CRC as the received signal was not that great, but we can test it with the python script. You can also already see the IDE (CTF value) in the hexbox. It is a 4bytes value directly after the FFFF → 0200472A

```
D:\>python hitag.py ffff0200472a039c5b16462ed5
Checksum ok
ide     0200472A
button  0
counter 0E7
secret  16C5918B
```

The ide is what is requested as CTF value, so this is solved.

Practical Car Hacking CTF Teaser by Willem Melching

Walkthrough by Stephan DB

The pythonscript for the two hitag tasks:

```
#!/usr/bin/env python3

#read the papers/pdfs. All the bits are explained in them. It is actually quite simple.

import sys
if len(sys.argv) < 2:
    print("Usage: {} ffff0200472a03a05749e7facf ".format(sys.argv[0]))
    sys.exit(1)

byte_array = bytearray.fromhex(sys.argv[1])   #from RADIO HACKER

if len(byte_array)!=13:
    raise ValueError("The provided string got bad length")
checksum=0
for i in range(2,12):
    checksum=checksum^byte_array[i]
if checksum==byte_array[12]:
    print("Checksum ok")
else:
    raise ValueError(f"Bad checksum should be {checksum:02X} but is {byte_array[12]:02X}")

ide = int.from_bytes(byte_array[2:6], byteorder='big')
print(f"ide     {ide:08X}")
button=(byte_array[6]>>4)&0x3
print(f"button  {button:01X}")
counter=byte_array[6]&0xF
counter=counter<<6
counter+=byte_array[7]>>2  #the two lsb are the two msb of the secret/crypt
print(f"counter {counter:03X}")
secret = int.from_bytes(byte_array[8:12], byteorder='big')
secret=secret>>2
secret+=(byte_array[7]&0x3)<<30  #the two bits after the counter are the high bits of the
secret/crypt
print(f"secret  {secret:08X}")

#this is needed for the next CTF
print("\ndata for htcrack:")
print(f"UID     {ide:08X}")
#IV is in this case just the counter plus 4bits with the button value
iv=(counter<<4)+button
print(f"nRx     {iv:08X}")
print(f"aRx     {secret^0xFFFFFFFF:08X}")
```

Practical Car Hacking CTF Teaser by Willem Melching

Walkthrough by Stephan DB

3. Challenge:
Hitag2 Crack Secret Key (Part 2)

Task:
This challenge contains a recording from a Keyfob featuring a Hitag2 cipher for RKE. The keyfob transmits a message containing a plaintext keyfob ID, counter and button followed by a MAC. Attached to this challenge you will find a SDR recording of 6 presses of the unlock button.
Use URH to decode the messages from the keyfob and figure out the keyfob ID, button and keystream. Use this to crack the (equivalent) key that's inside the keyfob.

The loading of the file and the analysing is same as on the task before. You should download the proxmark code and compile the ht2crack tool as this is needed in this task. Works also fine under Windows using MSYS.

So first we need to get the data from two signals (one seems to be actually a lock signal, as we got a different button value). The nRx value is just the countervalue shifted by 4 to the left plus the button value. aRx is secret value inverted (xor FFFFFFFF).

```
D:\>python hitag.py ffff0200472a039c5b16462ed5
Checksum ok
ide     0200472A
button  0
counter 0E7
secret  16C5918B

data for htcrack:
UID     0200472A
nRx     00000E70
aRx     E93A6E74
```

```
D:\>python hitag.py ffff0200472a13a4d7af8e2608
Checksum ok
ide     0200472A
button  1
counter 0E9
secret  35EBE389

data for htcrack:
UID     0200472A
nRx     00000E91
aRx     CA141C76
```

And these are then used for the ht2crack programm.

```
$ ./ht2crack5opencl.exe -D 2 0200472A 00000E70 E93A6E74 00000E91 CA141C76
Selected 1 OpenCL Device(s)

 0 - gfx90c

Attack 5 - opencl - start (Max Slices 16384, FORWARD order)

[0] Slice 2166/16384 ( 13.2% )
Key found @ slice 2166/16384 [ 72B7C3CCE726 ]

Attack 5 - opencl - end in 206.311291 second(s).
```

After some time the key is found and that is also our CTF value.

Practical Car Hacking CTF Teaser by Willem Melching

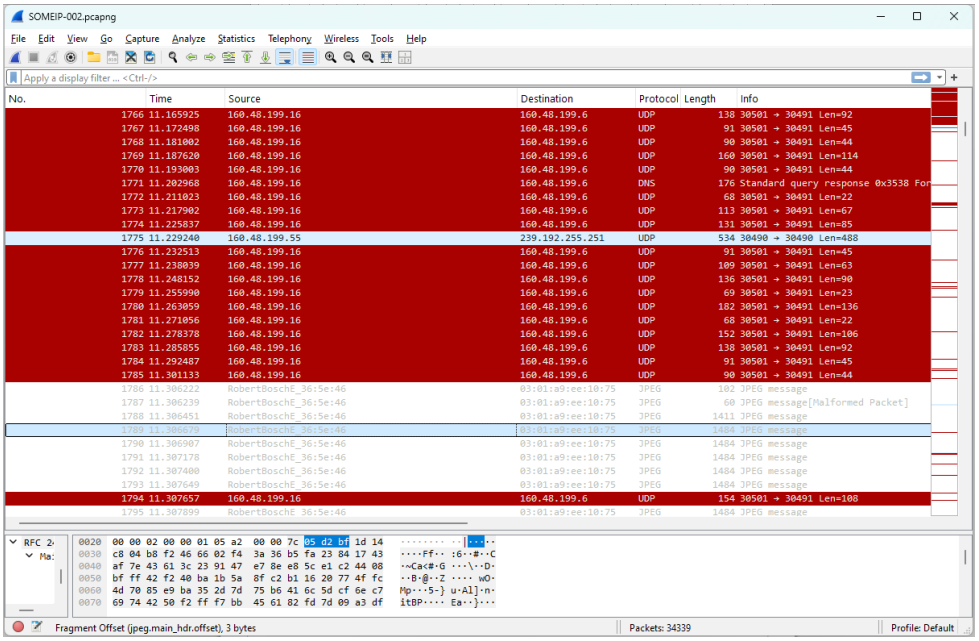
Walkthrough by Stephan DB

4. Challenge:
AVTP Video

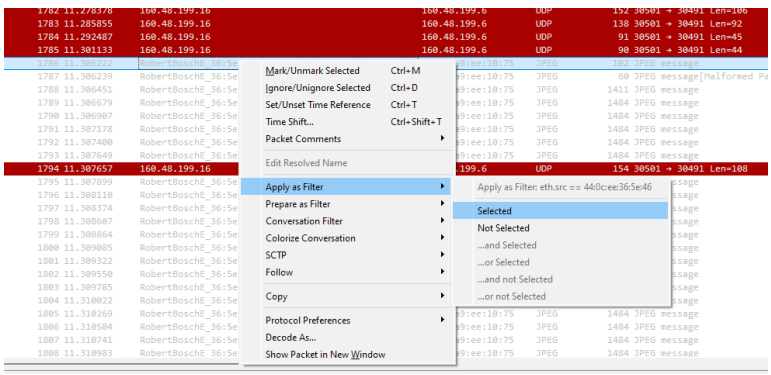
Task:
Attached to this challenge you will find a pcap captured from a G30 BMW. This capture was taken using a TAP on the automotive ethernet connection between the BDC (e.g. the gateway/BCM) and the rear view camera.

In this pcap you will first see SOME/IP traffic. When the car is put in reverse a video stream is started alongside the SOME/IP traffic. The goal of the challenge is to decode this video stream. The rear view camera is pointed at a piece of paper containing the flag.

As you can see on the timeline of the CTF this one was my nemesis. First step is to load pcap into wireshark and take a look:



At time 11.306222 in the log the network traffic with “JPEG” as marked protocol starts. I updated my wireshark, so maybe in previous versions it will have another ID and not “JPEG”, but the start time will be same. You can take that first message, select the destination and apply as filter and the log will be cleaned from all other traffic.



Practical Car Hacking CTF Teaser by Willem Melching

Walkthrough by Stephan DB

Lets check the first packet:

```
0000   03 01 a9 ee 10 75 44 0c ee 36 5e 46 81 00 a0 56
0010   22 f0 03 80 01 01 44 0c ee 36 5e 46 10 75 00 00
0020   00 00 02 00 00 01 00 20 00 00 67 42 00 29 e3 50
0030   16 87 a4 20 00 00 7d 00 00 18 6a 0d 18 00 0c e0
0040   00 04 86 bd e0 00 40 00 00 00 00 00 00 00 00 00
0050   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0060   00 00 00 00 00 00
```

22 f0 is the AVTP ID and wireshark states the 03 after is a flag for compressed video. This frame seems pretty short for compressed video, so it might be just some setup.

Next packet:

```
0000   03 01 a9 ee 10 75 44 0c ee 36 5e 46 81 00 a0 56
0010   22 f0 03 80 02 01 44 0c ee 36 5e 46 10 75 00 00
0020   00 00 02 00 00 01 00 04 00 00 68 ce 3c 80 00 00
0030   00 00 00 00 00 00 00 00 00 00 00 00
```

Even shorter than the first, so pretty clueless up to now.

Next packet (only the first 0x40 bytes as it's big):

```
0000   03 01 a9 ee 10 75 44 0c ee 36 5e 46 81 00 a0 56
0010   22 f0 03 80 03 01 44 0c ee 36 5e 46 10 75 00 00
0020   00 00 02 00 00 01 05 59 00 00 7c 85 88 84 00 00
0030   6b 7b e9 8f f2 c1 90 00 10 12 f4 45 23 c8 a5 d8
0040   df 27 4e 07 ed 29 f1 c0 0c 76 23 77 9f 15 de 7c
```

There are very little changes in the first 0x28 bytes of the packets. The two bytes at 0x26/0x27 seem to be related to the size, and when we verify this on the big package it looks like the "payload" starts from 0x2A (from the 7C). This assumption can also be verified with all following "big" packets. The two small ones seem to be padded with 00s, so they jump out a bit. The 0x67 and 0x68 from the first two packets must be some setup and then the video follows with the 7C blocks.

A bit of searching the internet then revealed that 0x67 and 0x68 are actually NAL (Network Abstraction Layer) unit headers for H.264/AVC video. Nice. We know the video format. A bit more digging then explained the usage of 00 00 00 01 (Annex-B format) to preceed those NALs to have a video which you can use then with ffmpeg to convert to a MP4. Python with scapy is your friend here to extract all those packages. I used the found length identifier to extract every payload with the leading NAL byte from every networkpacket, put the 00 00 00 01 bytes in front of each and saved this then stitched together. Only to see ffmpeg then hitting me with errors and the file was not recognized as video… So this was a fail.

Practical Car Hacking CTF Teaser by Willem Melching

Walkthrough by Stephan DB

Something is missing. Lets take a look at the structure again. So first is the 0x67 packet, then the 0x68 and then a lot of 7C packets before it starts again with 0x67.

First packet after the 0x68 packet
```
0000   03 01 a9 ee 10 75 44 0c ee 36 5e 46 81 00 a0 56
0010   22 f0 03 80 03 01 44 0c ee 36 5e 46 10 75 00 00
0020   00 00 02 00 00 01 05 59 00 00 7c 85 88 84 00 00
```

loads of these ones
```
0000   03 01 a9 ee 10 75 44 0c ee 36 5e 46 81 00 a0 56
0010   22 f0 03 80 04 01 44 0c ee 36 5e 46 10 75 00 00
0020   00 00 02 00 00 01 05 a2 00 00 7c 05 d2 bf 1d 14
```

last packet before the next 0x67 packet is sent
```
0000   03 01 a9 ee 10 75 44 0c ee 36 5e 46 81 00 a0 56
0010   22 f0 03 80 5d 01 44 0c ee 36 5e 46 10 75 00 00
0020   00 00 02 00 00 01 01 5e 04 00 7c 45 22 a1 8b 2c
```

So when looking at the 7C packets we can see that the first got the byte after the 7C as 0x85, then all next packets got 0x05 and the last packet 0x45.
Further internet search then revealed that 7C 85 is the start of a FU-A (fragmentation unit) for a H264 IDR slice which got normally NAL 0x65. The two most significant bits of the second byte tell what fragment it is. 10xxxxxx or 0x8X means it is the start/first fragment. 01xxxxxx or 0x4X means it is the end/last fragment. When none of the bits is set → 00xxxxxx or 0x0X it means it is a middle fragment.
When we look at the other packets in the network stream we can see 5C x1 which is the FU-A (fragmentation unit) for a H264 non-IDR slice which got normally NAL byte 0x41.

Lets adapt the python script again. The handler for the 0x67 and 0x68 packets is fine. When we find the 0x7C 0x85 we treat that as a start of the 0x65 slice. Putting 00 00 00 01 in front, then the payload. From the payload we remove the 0x7C 0x85 and replace with 0x65 (IDR slice). From the 7C 05 messages we only take the payload after the 7C 05 and stitch them to the previous payload. When we encounter the 0x7C 0x45 frame we also remove the first two leading bytes from that payload (7C 45) and stitch that to our 0x65 slice. Now this NAL message is complete.

With the following network packets we do the same. The only difference is that the marker is 5C x1 for a non-IDR slice (NAL byte 0x41), but process is identical.

The resulting file is saved and we can let ffmpeg do its job.

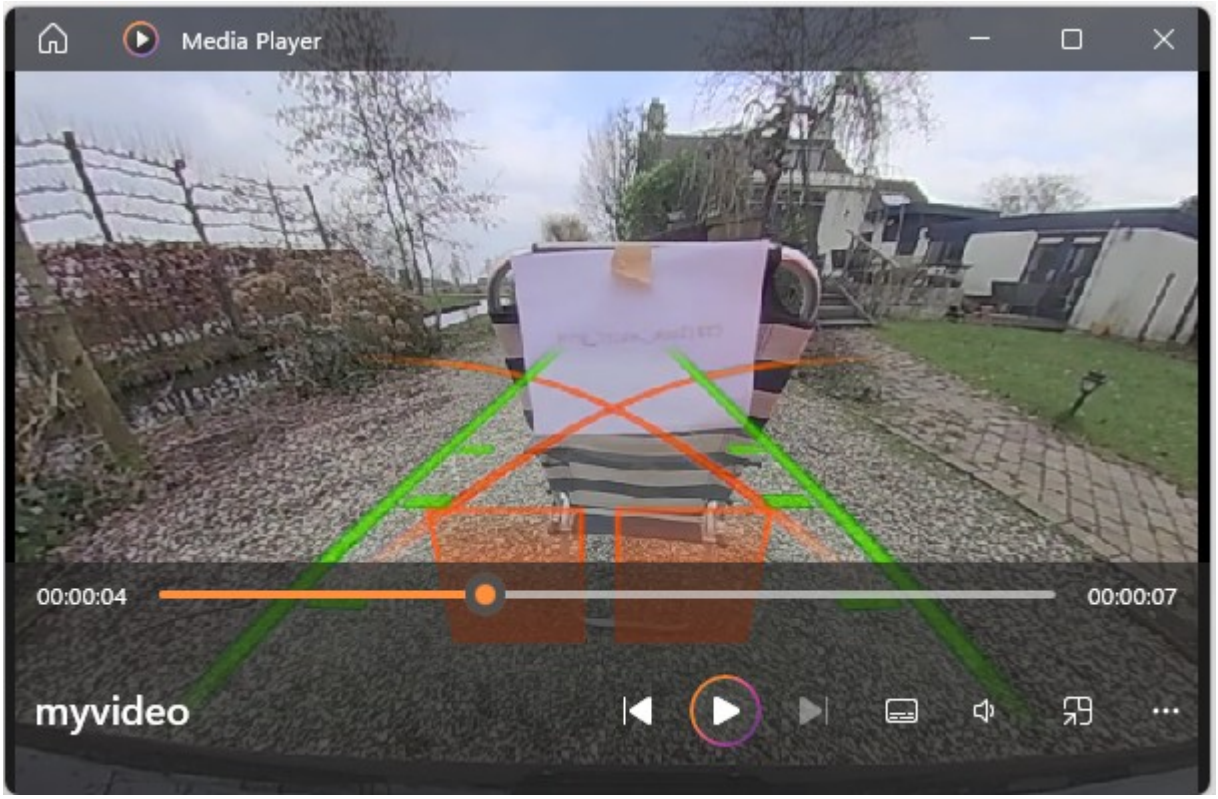
Practical Car Hacking CTF Teaser by Willem Melching

Walkthrough by Stephan DB

```
ffmpeg -i output.ts myvideo.mp4
```

```
->  Stream #0:0: Video: h264 (Baseline),
yuv420p(progressive),720x480, 25 fps, 50 tbr, 1200k tbn
```

That looks nice.



CTF found.

Final words… A big “Thank you” to Willem for providing these interesting riddles.

Stay curious!

Practical Car Hacking CTF Teaser by Willem Melching

Walkthrough by Stephan DB

The pythonscript for the AVTP task:

```
#!/usr/bin/env python3
import sys
import subprocess
from scapy.all import rdpcap, Ether, Dot1Q

def extract_avtp_stream(pcap_file, output_ts):
    """
    CTF extractor by Stephan DB. Python 3.10+ required
    """
    print(f"Loading packets from {pcap_file}...")
    packets = rdpcap(pcap_file)
    with open(output_ts, "wb") as f:
        for pkt in packets:
            avtp_payload = None
            if Ether in pkt:
                #VLAN-tagged frame; check Dot1Q layer for AVTP type
                if pkt.haslayer(Dot1Q):
                    dot1q = pkt.getlayer(Dot1Q)
                    if dot1q.type == 0x22F0:
                        raw_data = bytes(dot1q.payload)
                        length_bytes = raw_data[0x14:0x16]
                        nal_length = int.from_bytes(length_bytes, byteorder='big')
                        #print(hex(nal_length))
                        if len(raw_data) > 12:
                            avtp_payload = raw_data[24:24+nal_length]
                            if avtp_payload[0]==0x67:
                                avtp_payload = b'\x00\x00\x00\x01' + avtp_payload
                            if avtp_payload[0]==0x68:
                                avtp_payload = b'\x00\x00\x00\x01' + avtp_payload
                            match avtp_payload[0]:
                                case 0x7C:
                                    match avtp_payload[1]:
                                        case 0x85:
                                            print("fragment IDR slice start")
                                            avtp_payload=avtp_payload[2:]
                                            avtp_payload = b'\x00\x00\x00\x01\x65'+avtp_payload
                                        case 0x5:
                                            print("fragment IDR slice followup")
                                            avtp_payload=avtp_payload[2:]
                                        case 0x45:
                                            print("fragment IDR slice end")
                                            avtp_payload=avtp_payload[2:]
                                case 0x5C:
                                    match avtp_payload[1]:
                                        case 0x81:
                                            print("fragment non-IDR slice start")
                                            avtp_payload=avtp_payload[2:]
                                            avtp_payload = b'\x00\x00\x00\x01\x41'+avtp_payload
                                        case 0x1:
                                            print("fragment non-IDR slice followup")
                                            avtp_payload=avtp_payload[2:]
                                        case 0x41:
                                            print("fragment non-IDR slice end")
                                            avtp_payload=avtp_payload[2:]
                        else:
                            print("Warning: Packet too short, skipping.")
            if avtp_payload:
                f.write(avtp_payload)
    print(f"Extraction complete. Stream written to {output_ts}")

def convert_ts_to_mp4(ts_file):
    cmd = ["ffmpeg", "-i", ts_file," myvideo.mp4"]
    print(cmd)
    print("Converting to MP4 using FFmpeg...")
    subprocess.run(cmd, check=True)
    print(f"Conversion complete. Playable file saved as myvideo.mp4")

def main():
    if len(sys.argv) < 2:
        print("Usage: {} <pcap_file> ".format(sys.argv[0]))
        sys.exit(1)

    pcap_file = sys.argv[1]
    ts_output = "output.ts"

    extract_avtp_stream(pcap_file, ts_output)
    convert_ts_to_mp4(ts_output)

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