Secrets of Simos18

Reversing the secure boot mechanism in an automotive Engine Control Unit

Who Am I?

Brian Ledbetter

@bri3d

Formerly: Corporate Middle Management

Currently: SkySafe

Objectives and Agenda

- Understanding of Simos18 target
- Modification of Calibration (tuning) and Application Software
- High level overview plus random tips and tricks!
- Clean research (not using logs / reverse engineering commercial flash tools)
    - Shoutouts: tinytuning, APR
- Open source: https://github.com/bri3d
- **Two exploit chains**: one in Customer Bootloader and one in Supplier Bootloader

Tools

- Raspberry Pi 3B
- Waveshare CAN hat
- MCP2518 (don’t use 2515!)
- SocketCAN + isotp
- Python
- udsoncan
- pigpio

```
try:
    client.change_session(
        services.DiagnosticSessionControl.Session.programmingSession
    )
```

Target

- Continental/Siemens Simos18.1/6
- **Engine Control Unit**
- Used in 4 cylinder VW AG cars ~2012-2021
- PQ / MQB platforms
- CAN-based (18.4 is FlexRay)
- Specific target: 2016 Golf 7 R


Infineon
TriCore
SAK-TC1791S-512
F240EP AB
84PHCH2B5A4
51829

Target Research: Hardware

- Infineon Tricore **TC1791 -** Automotive ASIL-D Safety Processor
- “**AudoMAX**” family - Look for the TC1978 full user’s manual
- **Full documentation available from Infineon**
- Toolchain available from Hightec
- Compiler is GCC based so it’s GPL!
- https://github.com/volumit/
- https://www.infineon.com/cms/en/product/microcontroller/legacy-microcontroller/other-legacy-mcus/audo-family/tc1798-1793-1791-audo-max/

Target Research: Hardware Properties

- 32-bit RISC-ish CPU, flat address space, memory mapped peripherals, mixed 16 and 32 bit instructions
- Flash and emulated EEPROM (DFlash) are in-package - **nothing external to dump.**
- Can always upload a bootstrap loader, but Flash can be protected
- Debug interfaces appear **disabled.**
- **Boot passwords enabled so Flash is locked.**
- Let’s **attack the software first!**

Target Research: ECU Architecture (AutoSAR)

Application	Role
SBOOT / 1BL	Supplier’s Bootloader. Validate CBOOT. Written in C.
CBOOT / BootUpd / 2BL	Customer’s Bootloader. Validate ASW and perform dealership reprogramming/updates. Written in MISRA-ish C. Only accessible when software damaged or Immobilizer free
ASW	App Software. Run the car. Communicate with other modules. Written in auto-generated MISRA C from Simulink/Labview, ASCET, etc.

Target Research: ECU Software Attack Surface (CAN)

Application	External Surface Area
SBOOT	???
CBOOT	• UDS (Unified Diagnostic Services) Reprogramming
ASW	• UDS Diagnostics • CCP/XCP (CAN Calibration Protocol) • Module comms

Target Research: Things That Didn’t Work

- UDS Diagnostic handlers in ASW
  - ReadMemory and WriteMemory locked down
- CCP / XCP
  - Present (weirdly) but has seed/key. Even once past seed/key, locked down
- Memory safety in CAN communication
  - It’s MISRA enough that there wasn’t anything obvious
- Tampering with firmware and flashing using dealer tool (ODIS)
  - Get stuck in bootloader
- Taking the ECU out of the car
  - Won’t enter CBOOT session, ConditionsNotCorrect
  - Immobilizer

UDS Reprogramming Basics

Replicating the dealership tool (ODIS)

UDS Reprogramming Process and Protections

- Clear DTC (OBD 04)
- Check Programming Preconditions (UDS Start Routine 0x0203)
- Enter Programming Session (UDS Session 02) - **will fail if immobilized**
- Unlock Security Access 0x11 (VW: **SA2 Seed/Key**)
- Write Workshop Fingerprint: LocalIdentifier 0xF15A
- For Each Block
  - EraseMemory (Start Routine 0xFF00, passing block number as parameter)
  - RequestDownload (with **encrypt/compress** type and address via ALFID, in this case a **block number**)
  - TransferData
  - Exit Transfer
  - **Checksum Block** (Start Routine 0x0202, passing block address and checksum as parameter)
- Check Programming Dependencies (localRoutine 0xFF01)
- Reboot

Flash ODX (ODX-F)

```
<FLASHDATA xsi:type="INTERN-FLASHDATA" ID="EMEM_5G0906259L0002.FD_0">
  <SHORT-NAME>FD_0</SHORT-NAME>
  <LONG-NAME>0</LONG-NAME>
  <DATAFORMAT SELECTION="BINARY"/>
  <ENCRYPT-COMPRESS-METHOD TYPE="A_BYTEFIELD">AA</ENCRYPT-COMPRESS-METHOD>
  <DATA>C53FEEEBEFB15149BCECDFEF121CF7B96608F29777A3D231346A41C28605488E42EE84C2
  7F48F63550F3AFE86B29D931DE11D95F3F2779B1A7685A403F74F0666C247D9FDAA89CBDEE2104
  135E962FD849869D1213FE34492EC577547E2F2396727F86AC187BD3460E0C35D1810EDD21BCBF
```

```
<SECURITY-METHOD TYPE="A_ASCIISTRING">CRC32</SECURITY-METHOD>
<FW-SIGNATURE TYPE="A_UINT32">1</FW-SIGNATURE>
<FW-CHECKSUM TYPE="A_BYTEFIELD">00000000</FW-CHECKSUM>
<VALIDITY-FOR TYPE="A_ASCIISTRING">DB_0</VALIDITY-FOR>
```

https://github.com/bri3d/VW_Flash/blob/master/frf/decryptfrf.py

- For VW, FRF container from Flashdaten
  - Encrypted ZIP file
  - Long-ago extracted keys from dealer tools (ODIS)
- Describes flash process:SA2 script, blocks, erase parameters, checksum parameters
- Flashed using dealership tools
- **ENCRYPT-COMPRESS is AA**
- Data has **high entropy** (not just fixed XOR or something)
- Also, no signature and CRC32s are all zero, weird.

SA2 Seed/Key Bypass

- Nothing to it really
- SA2 spec leaked long ago, it’s a bytecode script that runs against the Seed to generate Key
- Even if it hadn’t, it’s all done offline on attacker-controlled machine, so trivial to reverse

```
<SECURITY>
  <SECURITY-METHOD TYPE="A_ASCIISTRING">SA2</SECURITY-METHOD>
  <FW-SIGNATURE TYPE="A_BYTEFIELD">6802814A106804930808200 94A0587221219548249930712201 1824A0587031120108 24A0181494C</FW-SIGNATURE>
</SECURITY>
```

https://github.com/bri3d/sa2-seed-key

Write a Flasher

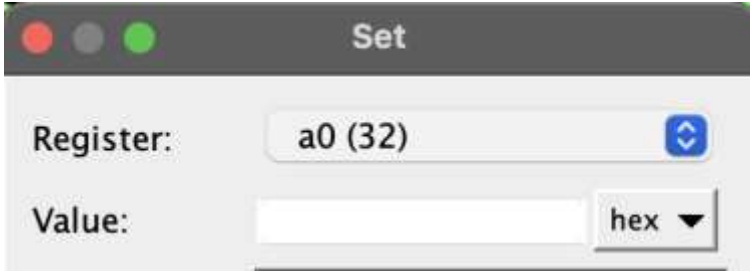
- Perform standard UDS reflashing routine using information from Flash ODX
- This lets us recover when we screw up hacking without needing ODIS!
- Very hard to brick this ECU
- Weird little gotchas:
  - Must send OBD-II “Clear DTC” command before entering Programming Session
  - Must write to the Workshop Code / Programming Fingerprint LocalIdentifier, anything works
  - ISO-TP STMIN_TX - ECU reports 0, reality is different
- Can’t tamper with anything yet…

https://github.com/bri3d/VW_Flash

Static Firmware Analysis

- Partial firmware discovered on forum
- Contains CBOOT, ASW

Ghidra Setup



- Ghidra already has Tricore!
- Add register layout for TC1791
- https://github.com/bri3d/ghidra_tc1791_registers
- File we found loads properly at 0x80000000
- Locate global registers
- Refer to Tricore ABI

```
8001cb94 91 10 00 0d     movh.a      a0,#0xd001
8001cb98 d9 00 40 98     lea         a0,[a0]-0x79c0
8001cb9c 91 10 08 1a     movh.a      a1,#0xa081
8001cba0 d9 11 00 08     lea         a1,[a1]-0x8000
8001cba4 7b 20 00 0d     movh        d0,#0xd002
8001cba8 1b 00 20 0f     addi        d0,d0,#-0xe00
8001cbac 7b 20 00 1d     movh        d1,#0xd002
8001cbb0 1b 01 30 1e     addi        d1,d1,#-0x1d00
```

Table 2-10 Register Assignments

Register	Use
A[0], A[1], A[8], A[9]	System Global Address Registers
D[15]	Implicit data register for many 16-bit instructions.
A[10]	Stack Pointer (SP).
A[11]	Return address register (RA) for CALL, JL, JLA, and JLI Return PC value on interrupts .
A[15]	Implicit base address register for many 16-bit load/store instructions.

Block Header

```
00300  00000000 00B06A21 02000000 00C00180 FFC20180 00CA0180
00318  DFFB0380 00000000 00000000 00000000 00000000 00000000
00330  00000000 00000000 00000000 00000000 00000000 F65F5170
00348  03000000 00008480 3F038480 00058480 FF068480 000A8480
00360  DF3B8680 00000000 00000000 00000000 00000000 00000000
```

```
00000  E8CA0180
```

```
                    FUN_8001cae8

8001cae8 6d 00 c0 05     call        FUN_8001d668
8001caec 0d 00 c0 04     isync
8001caf0 0d 00 80 04     dsync
8001caf4 7b 20 00 08     movh        d0,#0x8002
8001caf8 1b 00 a0 0c     addi        d0,d0,#-0x3600
8001cafc cd 40 e2 0f     mtcr        #0xfe24,d0
8001cb00 0d 00 c0 04     isync
8001cb04 7b 00 00 0c     movh        d0,#0xc000
8001cb08 1b 00 00 00     addi        d0,d0,#0x0
8001cb0c cd 00 e2 0f     mtcr        #0xfe20,d0
8001cb10 0d 00 c0 04     isync
```

- Simos-specific
- Found at 0x300 in each flash block
- Simple format: CRC32 followed by number of address ranges, then simple begin->end range specifiers
- That explains why there was no Checksum in the ODX
- Entry point address is also found at start of block (0x0)

CBOOT Interesting Findings

- CBOOT gets copied into RAM in Programming session
- 0x80022000 (Flash) -> 0xD0008000 (SRAM)
- Lots of references to an area of Flash we don’t have yet @ 80014000, which seems to contain cryptography functions
- Obvious giant UDS handler table

```
d0018eb4 94 35 01 d0     addr       31_uds_routinecontrol
d0018eb8 31              ??         31h    1
d0018eb9 10              ??         10h
d0018eba 00              ??         00h
d0018ebb 00              ??         00h
d0018ebc c4 3b 01 d0     addr       34_uds_startdownload
d0018ec0 34              ??         34h    4
d0018ec1 42              ??         42h    B
d0018ec2 00              ??         00h
d0018ec3 00              ??         00h
d0018ec4 28 3d 01 d0     addr       36_uds_transferdata
d0018ec8 36              ??         36h    6
d0018ec9 42              ??         42h    B
d0018eca 00              ??         00h
d0018ecb 00              ??         00h
```

Encryption 0xA

```
bool set_aes_key_iv(void)

{
  int iVar1;

  _DAT_d0003ac8 = 0;
  iVar1 = (*(code *)(_OTP_LIBRARY_SET_AES & 0xfffffffe))(0xd0003acc,&DAT_d0018164,&DAT_d0018154);
  return iVar1 != 0;
}
```

	DAT_d0018164	
d0018164	98	??	98h
d0018165	d3	??	D3h
d0018166	12	??	12h
d0018167	02	??	02h
d0018168	e4	??	E4h
d0018169	8e	??	8Eh
d001816a	38	??	38h
d001816b	54	??	54h
d001816c	f2	??	F2h
d001816d	ca	??	CAh
d001816e	56	??	56h
d001816f	15	??	15h
d0018170	45	??	45h
d0018171	ba	??	BAh
d0018172	6f	??	6Fh
d0018173	2f	??	2Fh

- On **this control module, 0xA = AES-CBC**
- If Encryption Type in UDS 0x34 RequestDownload is 0xA, call a key-setting function in the unknown area of Flash
- Keys are static, plaintext, in flash + RAM. Got em!

Recap So Far

- Decrypted FRF into ODX (using key material from ODIS)
- Loaded ODX blocks to perform standard UDS flashing process
- Seed/Key -> SA2, script is known from ODX, interpreter reimplemented
- Obtained plaintext CBOOT file and analyzed statically in Ghidra
- Encryption -> AES with fixed keys, recovered
- Compression -> LZSS, implemented
- Signature -> RSA-2048, **still not defeated**

Exploit 1: CBOOT Write Without Erase (RSA Bypass)

Security Flags System

- Each block has RSA verified when Checksum request is sent
- Validity flags / “Security Keys” written beyond transfer area
- RSA is **never checked again**
- Can we transfer too much data and overwrite Security Keys area?
- Nope, doesn’t work.
- Can we **transfer data over a block we already checksummed**?
- YES!!!

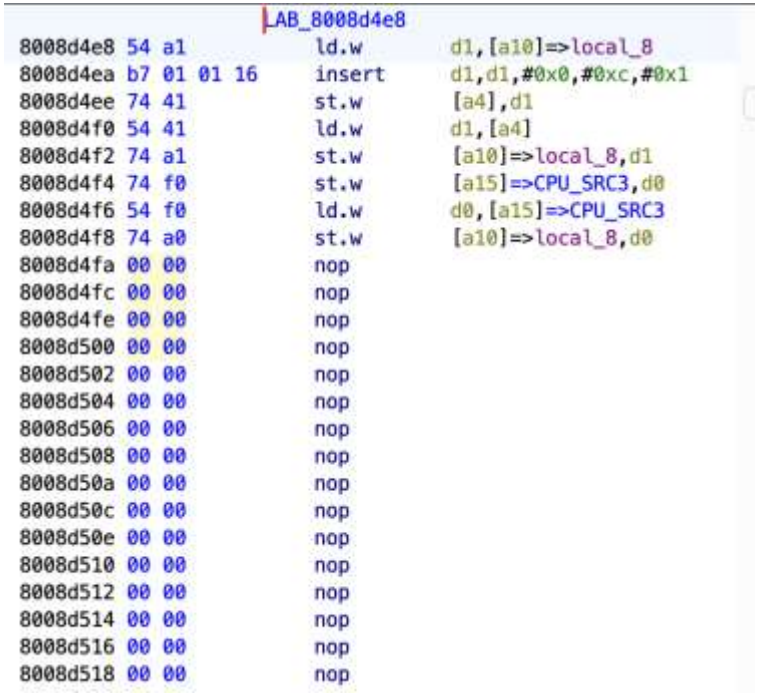
Exploiting Security Flags State Machine Issue

- Enter Programming session as usual
- Request Erase for a block (let’s say block 5, Calibration)
- Request Download to a **different block, let’s say block 2 (ASW)**
- Transfer Data (modified)
- Exit Transfer
- Request download for the Erased block, with unmodified data
- Transfer Data (valid and unmodified)
- Exit Transfer
- Checksum the Erased block, with unmodified data

At this point, all blocks have valid Security Flags, but data has been tampered with!


Software Block
UDS
TransferData
UDS
EraseBlock
Boot Security Flags
UDS
PerformChecksum

Write Without Erase Limitations



```
                        LAB_8008d4e8
8008d4e8 54 a1          ld.w        d1,[a10]=>local_8
8008d4ea b7 01 01 16    insert      d1,d1,#0x0,#0xc,#0x1
8008d4ee 74 41          st.w        [a4],d1
8008d4f0 54 41          ld.w        d1,[a4]
8008d4f2 74 a1          st.w        [a10]=>local_8,d1
8008d4f4 74 f0          st.w        [a15]=>CPU_SRC3,d0
8008d4f6 54 f0          ld.w        d0,[a15]=>CPU_SRC3
8008d4f8 74 a0          st.w        [a10]=>local_8,d0
8008d4fa 00 00          nop
8008d4fc 00 00          nop
8008d4fe 00 00          nop
8008d500 00 00          nop
8008d502 00 00          nop
8008d504 00 00          nop
8008d506 00 00          nop
8008d508 00 00          nop
8008d50a 00 00          nop
8008d50c 00 00          nop
8008d50e 00 00          nop
8008d510 00 00          nop
8008d512 00 00          nop
8008d514 00 00          nop
8008d516 00 00          nop
8008d518 00 00          nop
```

- **64-bit ECC**
- **Breaking ECC makes a brick**
- **CRC runs every boot**
- Could back-calculate ECC (dangerous and difficult)
- 00 on Tricore is nop (hot damn!)
- Can we find 64 bits of nop sled 64-bit aligned?
- YES! (delay in interrupt reconfiguration routine)
- Interrupts are already disabled here too

Payload

```
(*(code *)&SUB_8001f38e)();
puVar3 = &DAT_80022000;
iVar4 = 0x162ff;
puVar5 = &DAT_d0008000;
do {
  *(undefined *)puVar5 = *puVar3;
  bVar1 = iVar4 != 0;
  puVar3 = puVar3 + 1;
  iVar4 = iVar4 + -1;
  puVar5 = (undefined4 *)((int)puVar5 + 1);
} while (bVar1);
DAT_d000ad5c = 0;
uVar2 = DAT_d0008000;
(*(code *)(uVar2 & 0xfffffffe))(0x1200);
return;
```

```
DAT_b000218c = 0;
iVar2 = CHECK_FLASH_STATUS();
if (((iVar2 == 0) && (iVar2 = MCUID_MATCHES_OTP(), iVar2 == 0)) ||
   (uVar1 = DAT_b0000414, iVar2 = MCUID_SUM(uVar1), iVar2 != 0)) {
  uVar3 = 1;
}
else {
  uVar3 = 0;
}
DAT_b000218c = uVar3;
return uVar3;
```

- Load CBOOT into RAM
- Patch **Series/Sample** mode function
- Use CBOOT to write unsigned code blocks
- Complete control of ECU… **provided IMMO is free (in car) and ECU isn’t bricked**
- **Dump entirety of Flash**

We Need to Go Deeper

Gaining access to the Bootstrap Loader

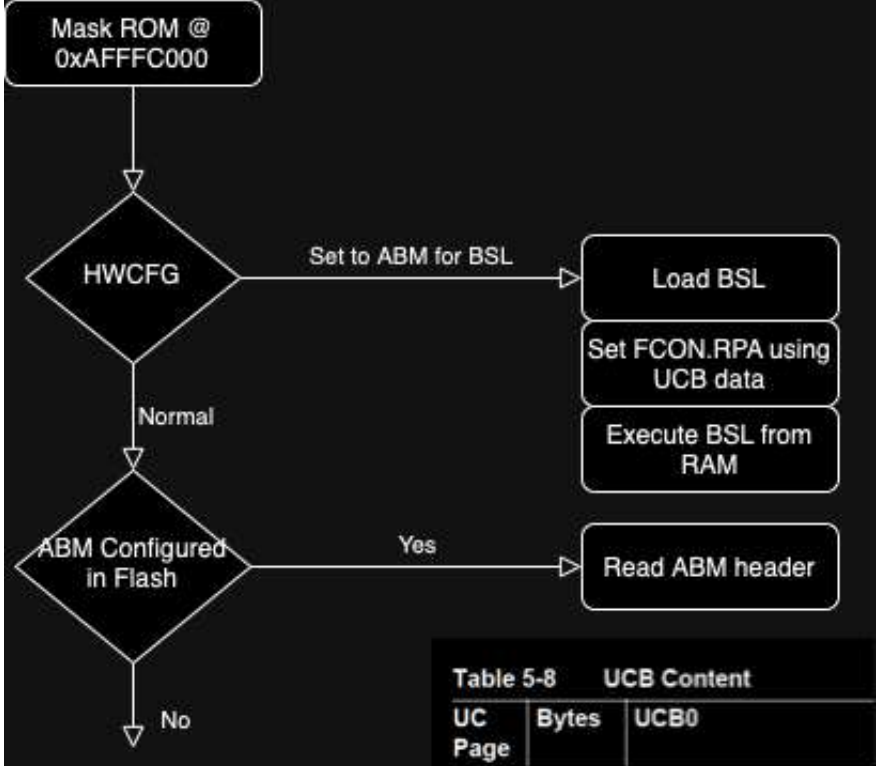


Table 5-8 UCB Content

UC Page	Bytes	UCB0
0	0–3	PROCON0
	4–7	SHEBOOT0
	8–11	PROCON0 (copy)
	12–15	SHEBOOT0 (copy)
	16–19	PW0 of User 0
	20–23	PW1 of User 0
	24–27	PW0 of User 0 (copy)
	28–31	PW1 of User 0 (copy)

Tricore Boot Process

- UCB = User Configuration Blocks
- Secret areas of flash accessible to Boot ROM at initialization
- Programmed with protection state, passwords, SHE configuration, debug configuration
- Boot ROM reads UCB, then configures Flash controller and locks UCB access
- Bootstrap Loader is **always accessible, but Flash is locked if UCB is set up that way**
- Flash access can be unlocked with passwords from UCB

Write a BSL

- Use DAVE to generate hardware primitives
- Use Hightec toolchain to compile
- https://github.com/bri3d/TC1791_CAN_BSL


Infineon
TriCore


GND
PWM1: 5V Level
Shifter to GPIO 13
PWM2: 5V Level
Shifter to GPIO 12
+12
+12
CAN
L
CAN
H
+12

Upload BSL

- Awesome, it works!
- **We can dump RAM!**
  (unexpected but makes sense)
- CBOOT AES key extraction is easy now
- Hangs on Flash access
- Dump flash configuration -> Protected by UCB (plus an OTP region)

Getting Passwords

Disable Write Protection	Address Data	.5554 ..xx**AA**	.AAA8 ..xx**55**	.553C **UL**	.AAA8 **PW 0**	.AAA8 **PW 1**	.5558 ..xx**05**
Disable Read Protection	Address Data	.5554 ..xx**AA**	.AAA8 ..xx**55**	.553C ..xx**00**	.AAA8 **PW 0**	.AAA8 **PW 1**	.5558 ..xx**08**

```
undefined4 UNLOCK_FLASH_WITH_PASSWORDS(int param_1)

{
  int iVar1;

  if (param_1 == 0) {
    iVar1 = FUN_8000421c();
    if (iVar1 != 0) {
      return 1;
    }
  }
  else if (param_1 == 0x1d) {
    iVar1 = SEND_PASSWORDS_TO_FLASH(&PASSWORD_USR0_0,&PASSWORD_USR0_1,0);
    if (iVar1 != 0) {
```

```
                    PASSWORD_USR0_0

    8001420c d7          ??         D7h
    8001420d 52          ??         52h
    8001420e 73          ??         73h
    8001420f 19          ??         19h
```

- Search for XRefs to Flash controller magic address 0xA0005558
- Locate methods where passwords are sent to flash controller to see how they are calculated
- They’re in plaintext in OTP area!
- Verify by reading using WriteWithoutErase exploit and sending with BSL.
- **It works!**
- **Need a read primitive**

Exploit 2: SBOOT Command Processor (TSW)

SBOOT

- Supplier Bootloader
- Located at 0x80000000 (entry point)
- Validates CBOOT
- Promotes CBOOT_Temp over CBOOT
- Has a backdoor command shell (TSW) in it!

SBOOT entry function

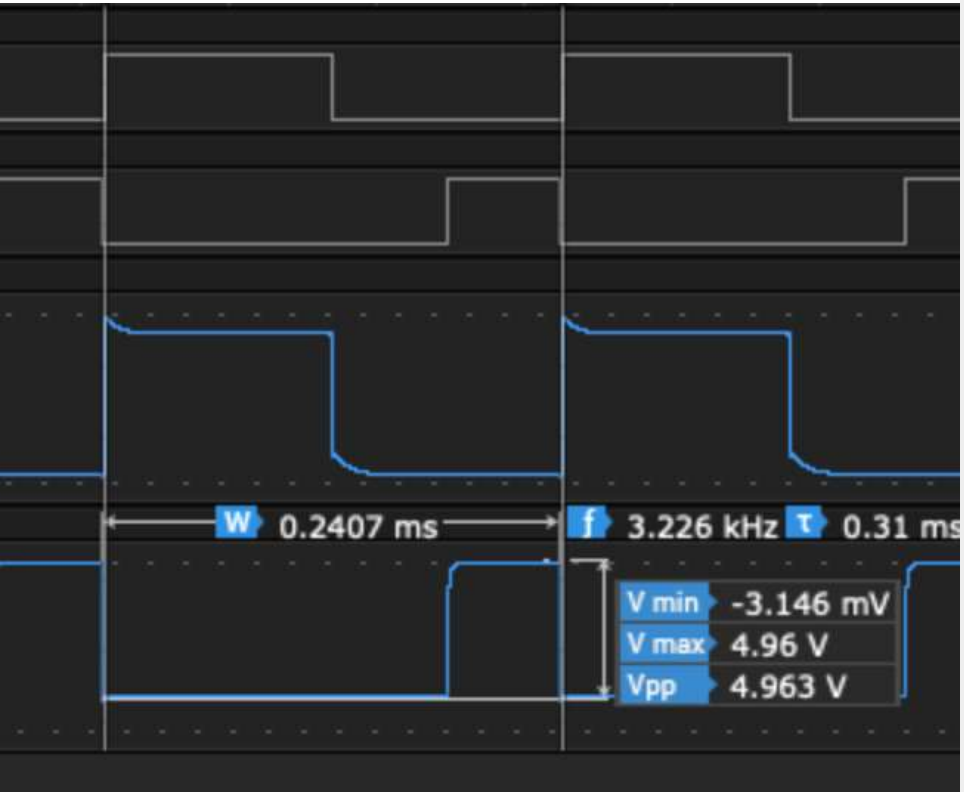
```
BREAK_INTO_COMMAND_SHELL = 0;
iVar3 = CHECK_PWM_SIGNALS();
if (iVar3 == 0) {
  RESET_GPTA_PWM();
}
else {
  iVar3 = WAIT_FOR_CAN_59_45();
  if (iVar3 == 0) {
    RESET_GPT_BREAKIN_59_45_fAILED();
  }
  else {
    iVar3 = WAIT_CAN_6B();
    if (iVar3 == 0) goto LAB_80004290;
    RESET_GPT_SUCCESS();
    BREAK_INTO_COMMAND_SHELL = 1;
  }
```

- Configures GPTA timer / logic analyzer peripheral to measure some values
- Waits for specific CAN messages

SBOOT PWM

```
while ((bVar2 = PWM_NUM_MATCHES, bVar2 < 6 &&
       (iVar4 = TIM0_VAL_CURRENT, iVar3 = TIM0_VAL_INITIAL, (uint)(iVar4 - iVar3) < 60000))) {
  CHECK_GPTA_TIMER_COUNTER(&DAT_8000661c);
  iVar3 = GPTA_TIMER_VALUE;
  iVar4 = STM_TIMER_DIFFERENCE;
  if ((iVar3 - 0x692U < 0x178) && (iVar4 - 0x8adU < 0x1fd)) {
    cVar1 = PWM_NUM_MATCHES;
    PWM_NUM_MATCHES = cVar1 + '\x01';
  }
  else {
    PWM_NUM_MATCHES = 0;
  }
  uVar5 = STM_TIM0;
  TIM0_VAL_CURRENT = uVar5;
}
cVar1 = PWM NUM MATCHES;
```

SBOOT PWM



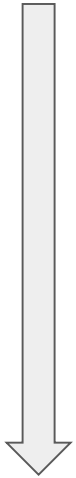
- GPTA Timer Counter measures posedge(sig1) -> posedge(sig2). Count must be < 0x80A
- System timer measures time between interrupts at posedge(sig2), must be < 0xAAA
- 8.75MHz tick clock
- 235uS sig1 -> sig2
- 312uS sig2 -> sig2
- 3.22kHz phase-shifted by 1/4

SBOOT Command Processor (TSW)

```
can_command = CAN_MESSAGE_DATA0;
if (can_command == 0x30) {
  recovery_state_2 = RECOVERY_SHELL_STATE;
  if (((recovery_state_2 == 0) || (recovery_state_2 == 6)) || (recovery_state_2 == 7)) {
    RETURN_STATUS = CHECK__8000081C_AND_WRITE_D0000010(&CAN_MESSAGE_DATA1);
    if (RETURN_STATUS == 1) {
      RETURN_CAN_A0();
      RECOVERY_SHELL_STATE = 2;
      return;
    }
    RETURN_CAN_91();
    RECOVERY_SHELL_STATE = 0;
    return;
  }
```

SBOOT Command Processor (TSW)

Command (Sent using ISO-TP Framing)	**Behavior (A0 = Positive Response)**
7D XX XX XX	Has a few weird little subcommands, does not affect state
6B	Ping! Always replies with A0 02
30 ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??	Sets parameters in RAM at D0000010, escalates state
54	Request Seed. Returns a fixed value followed by 0x100 bytes of high-entropy data
65 ?? x 0x100	Send Key. Requires 0x100 bytes of data, checks value against data generated during Request Seed process
78 AA AA AA AA XX	Set Byte. Sets a byte in the RAM scratchpad area at B0010000 - B0015000. Strong bounds checking. Requires Seed/Key first
79	Boot. Invokes validation process against RAM area



SBOOT Seed/Key

```
else if (current_state == 2) {
  bVar2 = SBOOT_RSA_KEY_INDEX;
  ENCRYPT_MEMORY_USING_RSA_PUBKEY
            (PTR_RSA_PUBLIC_KEY_LOCATIONS_80014e80[bVar2],&MERSENNE_TWISTER_DATA);
  iVar6 = CHECK_RSA_RETURN_STATUS(1);
  if (iVar6 == 3) {
    iVar6 = COPY_RSA_OUTPUT(&ENCRYPTED_DATA_USING_136_PUBKEY,2);
    if (iVar6 == 4) {
      COPY_CRYPTED_DATA_TO_SBOOT_SEED_AREA();
      SWAP_ENDIAN(&SBOOT_SEED_MATERIAL,0x40);
      SWAP_ENDIAN(&MERSENNE_TWISTER_DATA,0x41);
      SBOOT_SEED_SETUP_COMPLETE = 1;
    }
```

- Generate 256 bytes of PRNG
- Encrypt using RSA public key
- Expect original 256 bytes to be sent back (decrypted using private key)
- Public key not vulnerable to common factoring issues
- e=65537, not really vulnerable to common large-modulus / payload stuffing, plus all 256 bytes are checked

SBOOT Seed/Key

```
void SEED_MERSENNE_TWISTER_SGENRAND(void

{
  bool bVar1;
  uint uVar2;
  int *piVar3;
  int iVar4;
  int *piVar5;

  uVar2 = READ_TIM0_VALUE();
  DAT_b0000b6c = 0x270;
  piVar3 = &MERSENNE_TWISTER_STATE;
  MERSENNE_TWISTER_STATE = uVar2 | 1;
  piVar5 = (int *)&DAT_b0000b6c;
```

- Inadequate entropy: can easily precalculate all 2^31 possible plaintext -> ciphertext pairs practically on modern hardware.
- Especially inadequate entropy because it's the system timer.
- https://github.com/bri3d/Simos18_SBOOT/blob/main/twister.c

SBOOT code-block RSA

- Uses same block structure / headers and crypto lib in OTP as main RSA
- Seems fairly robust
- **RSA2048-PKCS#1.5-SHA256 with proper padding / whole-message comparison**
- No block reuse attack as SHA256 data has addresses mixed in
- No obvious memory safety issues
- **Are we stuck?**

FAIL!

```
CRC_START = ::CRC_START;
if (((CRC_START < 0xb0010131) && (puVar8 = (undefined *)CRC_END, &UNK_b001012f < puVar8))
    && (RETURN_STATUS = SETUP_CRC_VARIABLES(), RETURN_STATUS != 0)) {
  RECOVERY_SHELL_STATE = 8;
  return;
```

- CRC_START can be set to any address before 0xB0010131
- What's before 0xB? 0x8…
- CRC_END still needs to be after 0xB001012F though…

Read primitive

```
                    RAM_CRC_BASE

d0010770 00 00 00 00     undefined4  00000000h

                    RAM_CRC_LENGTH

d0010774 00 00 00 00     undefined4  00000000h

                    RAM_CRC_VALUE

d0010778 00 00 00 00     undefined4  00000000h
```

- Code block has a header with CRC addresses
- CRC is checked before RSA
- Bounds check inadequate
- Can control beginning of CRC region freely
- Checksum the Flash passwords?
- Can’t control end of region
- Wait… we can read RAM in BSL!
- Reset timing attack to dump state

No Demo :(

- Send PWM signals to signal SBOOT breakin
- Wait fixed time before sending Seed request
- Bruteforce Seed ciphertext until we find matching Key plaintext
- Send seed/key
- Send block header with bounds set to flash password location
- Back-calculate CRC to extract Flash passwords

Recap (Architecture Revisited)

Application	Purpose
SBOOT @ 0x80000000	● CBOOT promotion ● TSW - vulnerable
Crypto @ 0x80014000	● AES ● RSA ● Hitag-looking LFSR used for DFlash encryption
CBOOT @ 0x8001C000	● UDS Reprogramming - vulnerable
ASW @ 0x80040000	● UDS Diagnostics ● CCP/XCP ● Inter-module communication

What should Continental have done differently?

- PRNG is only as strong as seed
- Never use timer for PRNG seed
- Need more than 31 bits of entropy in today’s world
- Most strict possible bounds checking
- CRC is a read primitive
- Trust Flash less (verify more)
- Why not just discard the read passwords?

Applicability to Modern ECUs

- Very similar exploits in Bosch MED17/MG1 until 2021
- Broadly applicable in concept to almost everything European 2010-2020
- This is “last gen” now
- Newer ECUs use Aurix Tricore variant with HSM (ARM TrustZone core inside of Tricore!) for Sample Mode and key storage

Research ideas

- Glitch Tricore BootROM + Flash controller
  - Flash controller and debug interfaces initialize in insecure state
  - Felix Domke (tmbinc) did this on previous generation Tricore with static voltage and it worked (lol!)
  - Looking at BootROM, pretty sure EMFI could be very plausible too
  - Also decapping and accessing Flash die probably practicable (if you want to try this look at DSG which has Tricore on flexible package)

Thanks

- tinytuning - original research, especially around WriteWithoutErase and LZSS
- Joedubs - lots of early code
- AaronS3, ConnorHowell, switchleg1 - various code contributions, Haldex and DSG research and development, drivers, firmware, etc.

Additional reading

- JinGen Lim - https://github.com/jglim/UnsignedFlash/blob/main/document_pub.pdf - very similar flash exploit in Mercedes IC204 instrument cluster
- Willem Melching (hi!) - https://github.com/I-CAN-hack/pq-flasher and https://icanhack.nl/blog/vw-part1/ - VW Power Steering research and a great intro series
- Felix Domke - https://media.ccc.de/v/33c3-7904-software_defined_emissions - static voltage glitching against previous-generation Tricore

Questions?