Attack on Titan M, Reloaded

Vulnerability Research on a Modern Security Chip

Damiano Melotti
Maxime Rossi Bellom

Quarkslab

Who we are

- @DamianoMelotti
- Security researcher @ Quarkslab
- Interested in low-level mobile security and fuzzing

- @max_r_b
- Security researcher & team leader @ Quarkslab
- Working on mobile and embedded software security

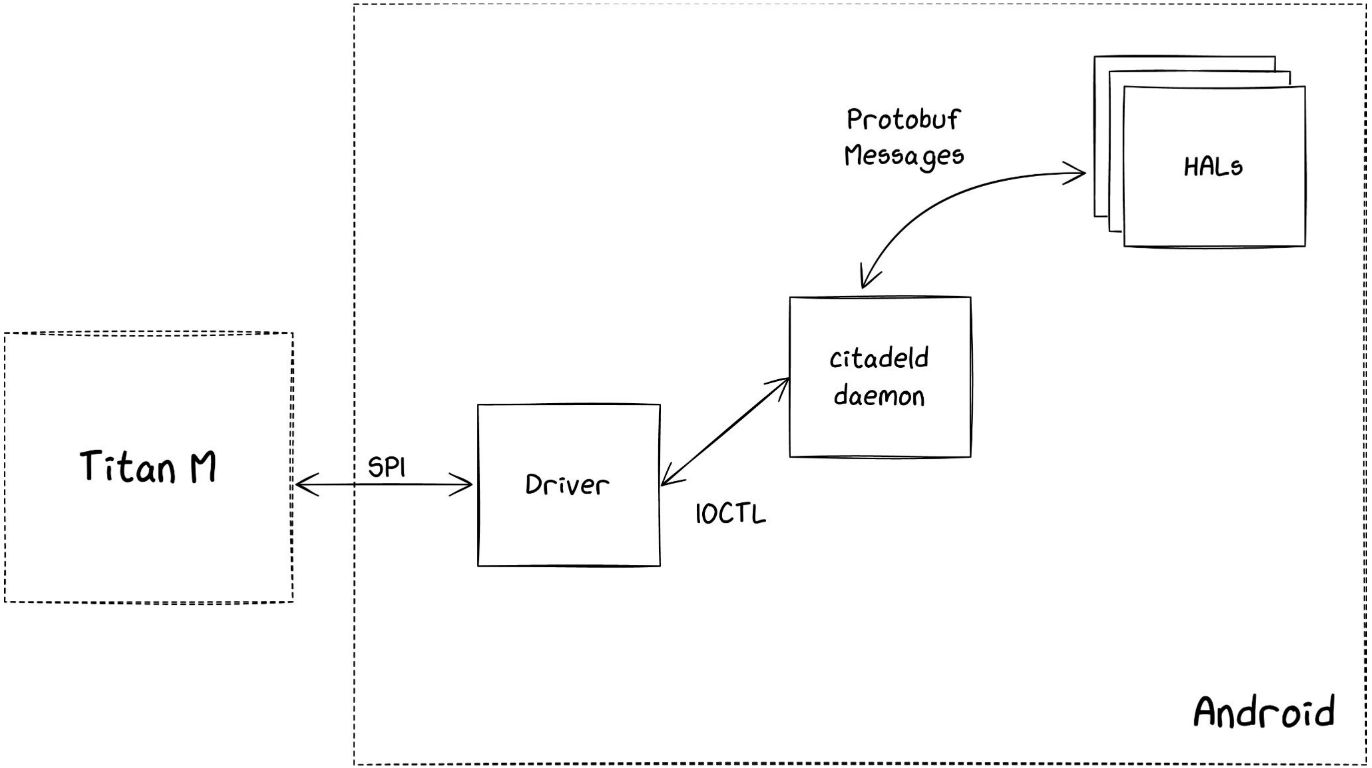
What is Titan M?

- Security chip made by Google, for Pixel devices
- Implements critical security features
  - Keymaster/Strongbox, Weaver, AVB, etc.
- Client-server model
- Introduced to:
  - Mitigate side-channel attacks
  - Protect against hardware tampering

Titan M specs

- Security chip based on ARM Cortex-M3
- Closed source but based on EC
    - An open source OS made by Google
    - Written in C and conceptually simple
    - No dynamic allocation
- Most of the code is divided into tasks
- SPI bus used to communicate with Android
- UART bus used for logs and minimalistic console

Communication with the chip

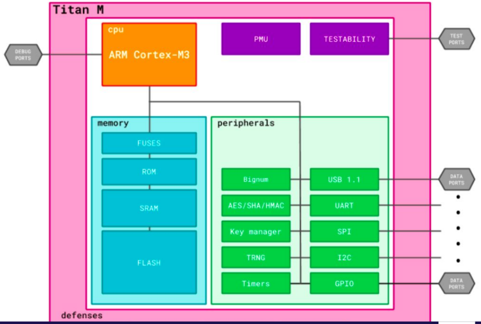


Our previous work in 4 slides

Specification

Hardened SoC based on ARM Cortex-M3

- Anti-tampering defenses
- Cryptographic accelerators & True Random Number Generator
- UART for logs and console
- SPI to communicate with Android

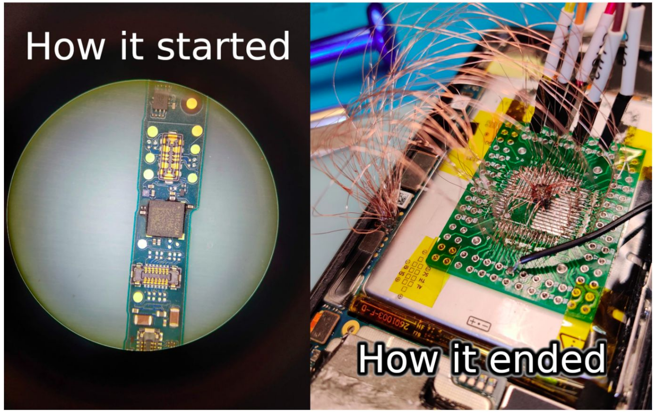


Firmware Security Measures

- Secure boot (images are signed and verified at boot)
- No MMU, but MPU to give permissions to the memory partitions
- Only software protection: hardcoded stack canary checked in the SVC handler

```
if (*CURRENT_TASK->stack != 0xdeadd00d) {
  next = (int)&CURRENT_TASK[-0x411].MPU_RASR_value >> 6;
  log("\n\nStack overflow in %s task!\n",(&TASK_NAMES)[next]);
  software_panic(0xdead6661,next);
}
```

Hardware Reverse: Finding SPI

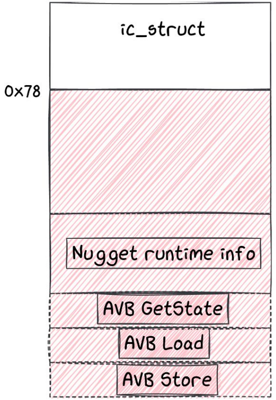


What can we do with the exploit?

Vulnerable buffer placed just before

- runtime data of the chip...
- ... and the list of command handler pointers

➔ overwrite command handler addresses to gain code execution!



What we already did

Firmware reversing → Interact with the chip → Find vulns → First code execution

What we already did

1. Firmware reversing
2. Interact with the chip
3. Find vulns
4. First code execution

What we already did

Firmware reversing

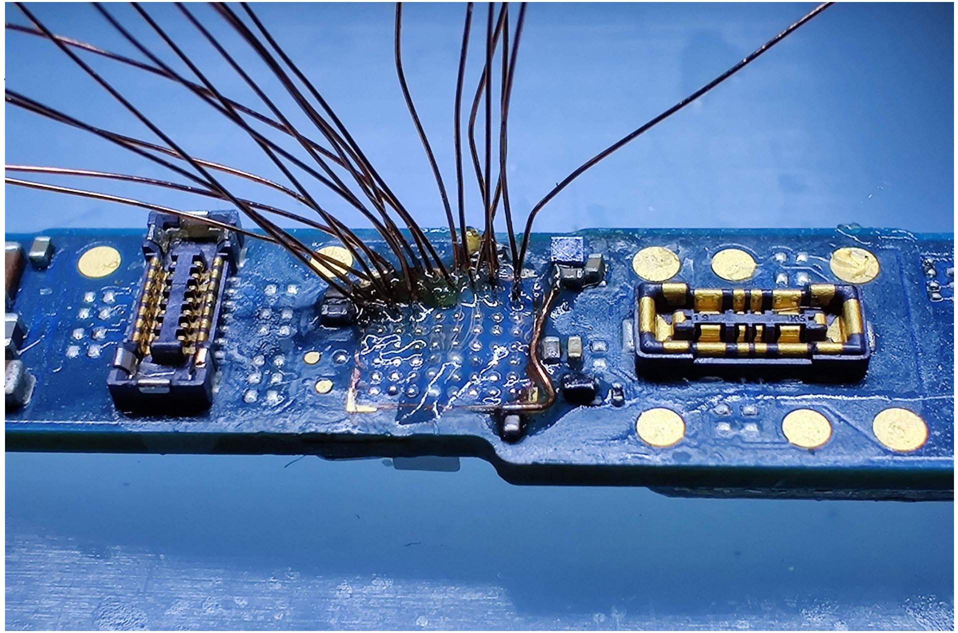
Interact with the chip

Find vulns

First code execution

What we already did

Implemented some tools to interact with the chip

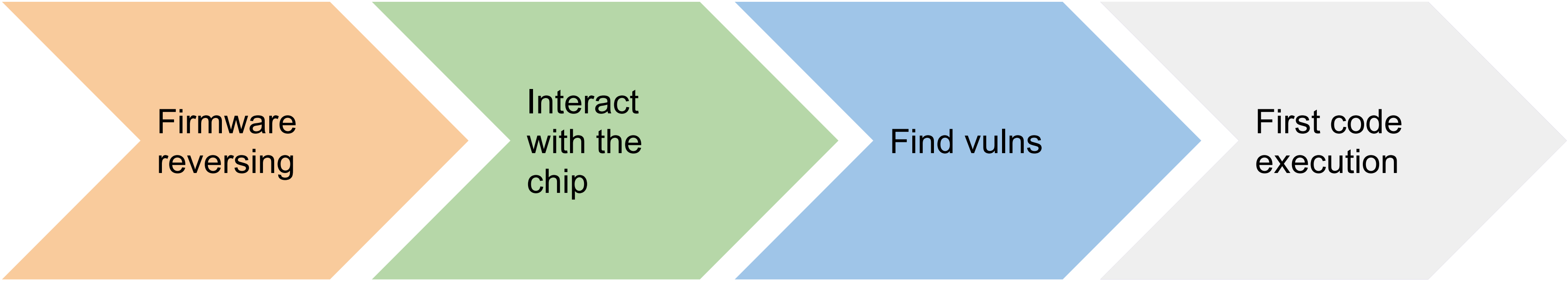


Sniff and send custom commands

- From Android using Frida and our tool **nosclient**
- On this hardware level thanks to @doegox's magic hands

What we already did

Several vulnerabilities reported



- CVE-2021-0939: A memory leak allowing to reveal parts of the Boot ROM
- CVE-2021-1043: A downgrade issue allowing to flash any firmware
  - ➜ With a side effect: all the secrets are erased

What we already did

- Firmware reversing
- Interact with the chip
- Find vulns
- First code execution

Leaked various hidden parts of the firmware, including the Boot ROM

What we show today

- **Fuzzing** is useful also against Titan M
  - Even on such contrainted target, we can get interesting results
- Two approaches
  - Black-box fuzzer vs emulation-based fuzzer
- Exploiting **without** debuggers or stack traces
- How a single **software** vulnerability can lead to
  - Code execution
  - Compromise of the security properties guaranteed by the chip

Blackbox fuzzing

Black Box fuzzing

- Target: tasks
- Arbitrary messages with `nosclient`
  - Known format of the messages
  - We get a return code, and an actual response if successful

➔ **Mutate** the message, **check** return code
  - If greater than 1, *something* happened

external/nos/host/generic/nugget/include/application.h [1]

```
enum app_status {
  /* A few values are common to all applications */
  APP_SUCCESS = 0,
  APP_ERROR_BOGUS_ARGS, /* caller being stupid */
  APP_ERROR_INTERNAL,    /* application being stupid */
  APP_ERROR_TOO_MUCH,    /* caller sent too much data */
  APP_ERROR_IO,          /* problem sending or receiving data */
  APP_ERROR_RPC,         /* problem during RPC communication */
  APP_ERROR_CHECKSUM,    /* checksum failed, only used within protocol */
  APP_ERROR_BUSY,        /* the app is already working on a command */
  APP_ERROR_TIMEOUT,     /* the app took too long to respond */
  APP_ERROR_NOT_READY,   /* some required condition is not satisfied */
}
```

[1]: https://android.googlesource.com/platform/external/nos/host/generic/+/refs/tags/android-platform-12.1.0_r1/nugget/include/application.h#307

Implementation

- Plug `libprotobuf-mutator`[2] in `nosclient`
  - Very straightforward
  - `void Mutate(protobuf::Message* message, size_t max_size_hint);`
- Basic corpus generation
  - Messages are quite simple
  - Start from empty ones, but add some non-trivial fields
- Store and triage inputs generating faulty states

[2]: https://github.com/google/libprotobuf-mutator

Results

Firmware: 2020-09-25, `0.0.3/brick_v0.0.8232-b1e3ea340`

- 2 buffer overflows (1 exploited for code exec)
- 4 null pointer dereferences
- 2 unknown bugs causing a reboot

Firmware: latest (at the time), `0.0.3/brick_v0.0.8292-b3875afe2`

- 2 null-ptr deref still make the chip crash
- Bug reported → not a vulnerability

All of this after a few minutes of fuzzing...

Comments and limitations

- ✅ Bugs!
- ✅ Very simple to implement
- ✅ Decent performance: ~74 msg/sec
- ✅ Testing in real world

- ❌ Only “scratching the surface”
- ❌ Prone to false positives
- ❌ Detection is tricky

Bottom line: hard to know what’s going on the target

Emulation-based fuzzing

Switching to emulation-based

- We know how the OS works
- We can leak arbitrary memory with an exploit on an old firmware
  - Helps setting up memory
- With emulation, we control what is executed
  - Good feedback for a fuzzer

Emulating Titan M

- Played with several frameworks
- Choice: Unicorn[3]
- Why?
  - Emulates CPU only
  - We do not care about full-system emulation
  - Easy to setup & tweak
  - Integrates nicely with AFL++

[3]: https://www.unicorn-engine.org/

Fuzzing with AFL++

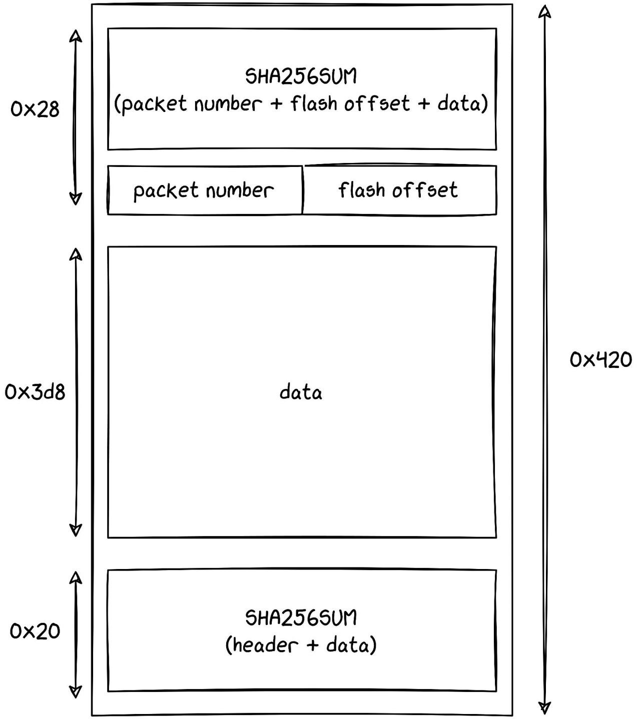
- AFL++ in Unicorn mode
  - Instrument anything that can be emulated with Unicorn
  - Fuzz with the classic AFL experience
- Once the emulator works, not much needs to be done
  - place_input_callback to copy input sample
  - Crashes detected at Unicorn errors (e.g. UC_ERR_WRITE_UNMAPPED)
- Custom mutators depending on needs
  - AFL_CUSTOM_MUTATOR_LIBRARY=<mutator.so>
  - AFL_CUSTOM_MUTATOR_ONLY=1 to use only that one

What to fuzz

- Pretty much anything!
- All you need is:
    - An entry point
    - Valid memory state
    - Registers set at the right values
    - One or more exit points
- Keep attack surface into account

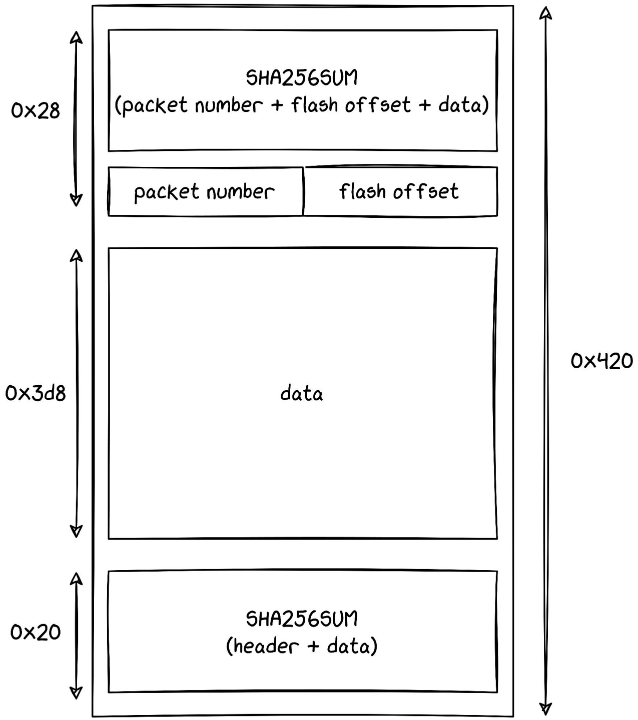
SPI rescue feature

- SPI rescue allows to flash new firmware
  - No password required
  - Wipes user data
  - Can be triggered from bootloader
- Firmware sent as rec file



SPI rescue handler

- Are input files parsed and processed correctly?
- Input is structured
  - Let's mutate it smartly :)
- We use `FormatFuzzer`[4]
  - Allows to generate and parse binary files
  - Follows the `bt` template format, from the 010 editor
  - Requires a modified version of AFL++



… also this time, no bugs (but some interesting internals revealed)

[4]: https://github.com/uds-se/FormatFuzzer

Going back to the tasks

- Tasks use protobuf
- Rely again `libprotobuf-mutator`
  - With some tricks to embed the message name in the bytes it generates
- Focused on Identity and Keymaster
  - The largest and most complex tasks
  - We fuzzed Weaver too, but it is not as interesting
- First, can we find the same bugs we know about?

Yes! (apart from one...)

There is no free lunch

- Emulation is not a silver bullet!
- Embedded targets ➔ hw-dependant code everywhere…
  - Lots of hooks
  - Code that can’t be exercised
  - Especially true in system functions
- A bug doesn’t always make Unicorn crash
  - No ASAN-like instrumentation
  - In-page overflows, off-by-ones won’t be detected
- No full system emulation ➔ miss some parts of code
  - No system state
  - The bug we missed makes the scheduler crash
  - … and we don’t emulate the scheduler 🙃

Tweaks

- Much more capabilities compared to pure black-box
- A few heuristics we implemented:
    - Monitor consecutive reads in the Boot ROM ➜ spot buggy `memcpy`
    - Hook accesses to specific global buffers
    - Even more specific ones on different commands
- At the same time, everything comes at a cost
    - Hooks impact performance
    - In our case, not a big deal due to very specific harnesses

The vulnerability

CVE-2022-20233

- `param_find_digests_internal`
  - Checks `DIGEST` tags in `KeyParameter` objects
- Out-of-bounds write of 1 byte to `0x1`
  - Can be repeated multiple times
  - Huge constraints on the offset
- Looks like a minor issue...

```
message KeyParameter {
  Tag tag = 1;
  uint32 integer = 2;
  uint64 long_integer = 3;
  bytes blob = 4;
}

message KeyParameters {
  repeated KeyParameter params = 1;
}
```

CVE-2022-20233

```
ldr.w    r1,[r2,#-0x4]
ldr      r3,[PTR_DAT_0005d808] ; 0x20005
cmp      r1,r3
bne      increment_loop_vars
ldr      r3,[r2,#0x0]
uxtb     r0,r3
cmp      r0,#0x4
bhi      error_exit
movs     r1,#0x1
lsl.w    r0,r1,r0
tst      r0,#0x15
beq      error_exit
strb     r1,[r7,r3]
```

```
if (((nugget_app_keymaster_KeyParameter *)(offset + -1))->tag ==
    0x20005) {
  masked = *offset & 0xff;
  if ((4 < masked) || ((1 << masked & 0x15U) == 0)) {
    return 0x26;
  }
  *(undefined *)(buffer + *offset) = 1;
  *param_3 = *param_3 + 1;
  *param_4 = offset;
}
```

CVE-2022-20233

```
ldr.w    r1,[r2,#-0x4]
ldr      r3,[PTR_DAT_0005d808] ; 0x20005
cmp      r1,r3
bne      increment_loop_vars
ldr      r3,[r2,#0x0]
uxtb     r0,r3
cmp      r0,#0x4
bhi      error_exit
movs     r1,#0x1
lsl.w    r0,r1,r0
tst      r0,#0x15
beq      error_exit
strb     r1,[r7,r3]
```

```
if (((nugget_app_keymaster_KeyParameter *)(offset + -1))->tag ==
    0x20005) {
  masked = *offset & 0xff;
  if ((4 < masked) || ((1 << masked & 0x15U) == 0)) {
    return 0x26;
  }
  *(undefined *)(buffer + *offset) = 1;
  *param_3 = *param_3 + 1;
  *param_4 = offset;
}
```

CVE-2022-20233

```
ldr.w   r1,[r2,#-0x4]
ldr     r3,[PTR_DAT_0005d808] ; 0x20005
cmp     r1,r3
bne     increment_loop_vars
ldr     r3,[r2,#0x0]
uxtb    r0,r3
cmp     r0,#0x4
bhi     error_exit
movs    r1,#0x1
lsl.w   r0,r1,r0
tst     r0,#0x15
beq     error_exit
strb    r1,[r7,r3]
```

```
if (((nugget_app_keymaster_KeyParameter *)(offset + -1))->tag ==
    0x20005) {
  masked = *offset & 0xff;
  if ((4 < masked) || ((1 << masked & 0x15U) == 0)) {
    return 0x26;
  }
  *(undefined *)(buffer + *offset) = 1;
  *param_3 = *param_3 + 1;
  *param_4 = offset;
}
```

0xdeadbeef

CVE-2022-20233

```
ldr.w   r1,[r2,#-0x4]
ldr     r3,[PTR_DAT_0005d808] ; 0x20005
cmp     r1,r3
bne     increment_loop_vars
ldr     r3,[r2,#0x0]
uxtb    r0,r3
cmp     r0,#0x4
bhi     error_exit
movs    r1,#0x1
lsl.w   r0,r1,r0
tst     r0,#0x15
beq     error_exit
strb    r1,[r7,r3]
```

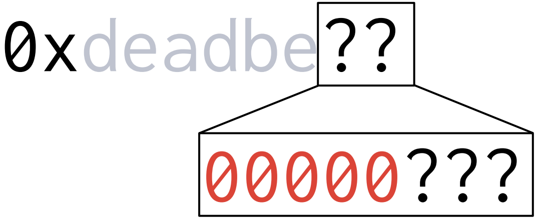
```
if (((nugget_app_keymaster_KeyParameter *)(offset + -1))->tag ==
    0x20005) {
  masked = *offset & 0xff;
  if ((4 < masked) || ((1 << masked & 0x15U) == 0)) {
    return 0x26;
  }
  *(undefined *)(buffer + *offset) = 1;
  *param_3 = *param_3 + 1;
  *param_4 = offset;
}
```

0xdeadbeef

CVE-2022-20233

```
ldr.w     r1,[r2,#-0x4]
ldr       r3,[PTR_DAT_0005d808] ; 0x20005
cmp       r1,r3
bne       increment_loop_vars
ldr       r3,[r2,#0x0]
uxtb      r0,r3
cmp       r0,#0x4
bhi       error_exit
movs      r1,#0x1
lsl.w     r0,r1,r0
tst       r0,#0x15
beq       error_exit
strb      r1,[r7,r3]
```

```
if (((nugget_app_keymaster_KeyParameter *)(offset + -1))->tag ==
    0x20005) {
  masked = *offset & 0xff;
  if ((4 < masked) || ((1 << masked & 0x15U) == 0)) {
    return 0x26;
  }
  *(undefined *)(buffer + *offset) = 1;
  *param_3 = *param_3 + 1;
  *param_4 = offset;
}
```



CVE-2022-20233

```
ldr.w    r1,[r2,#-0x4]
ldr      r3,[PTR_DAT_0005d808] ; 0x20005
cmp      r1,r3
bne      increment_loop_vars
ldr      r3,[r2,#0x0]
uxtb     r0,r3
cmp      r0,#0x4
bhi      error_exit
movs     r1,#0x1
lsl.w    r0,r1,r0
tst      r0,#0x15
beq      error_exit
strb     r1,[r7,r3]
```

```
if (((nugget_app_keymaster_KeyParameter *)(offset + -1))->tag ==
    0x20005) {
  masked = *offset & 0xff;
  if ((4 < masked) || ((1 << masked & 0x15U) == 0)) {
    return 0x26;
  }
  *(undefined *)(buffer + *offset) = 1;
  *param_3 = *param_3 + 1;
  *param_4 = offset;
}
```

0xdeadbe??

00000??0

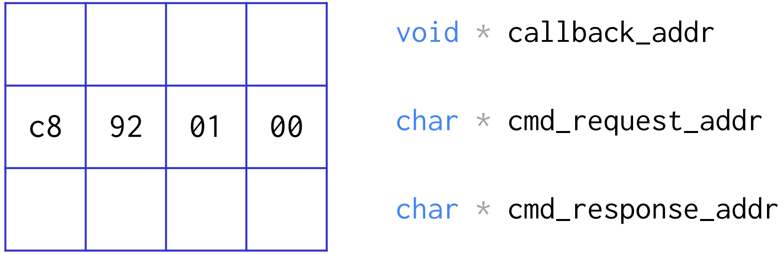
What can we do?

- Multiple ways to reach the vulnerable code
  - A few different command handlers call it
  - Different base addresses for the OOB-write
- Titan M's memory is completely static
  - All structures are always located at the same addresses
- Setting one byte can be enough to break the system

Our approach

- Generate all writable addresses
- Highlight them in Ghidra
- …🔍

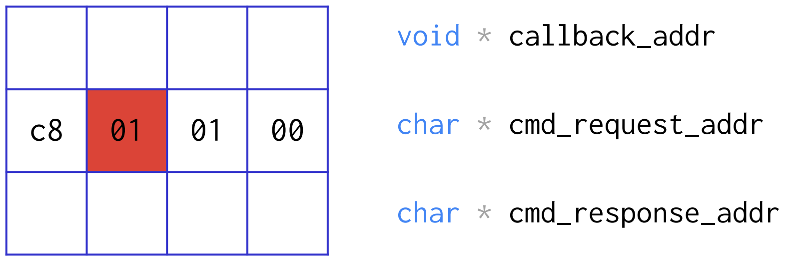
KEYMASTER_SPI_DATA



Our approach

- Generate all writable addresses
- Highlight them in Ghidra
- ...🔍

KEYMASTER_SPI_DATA



What to overwrite

`KEYMASTER_SPI_DATA`

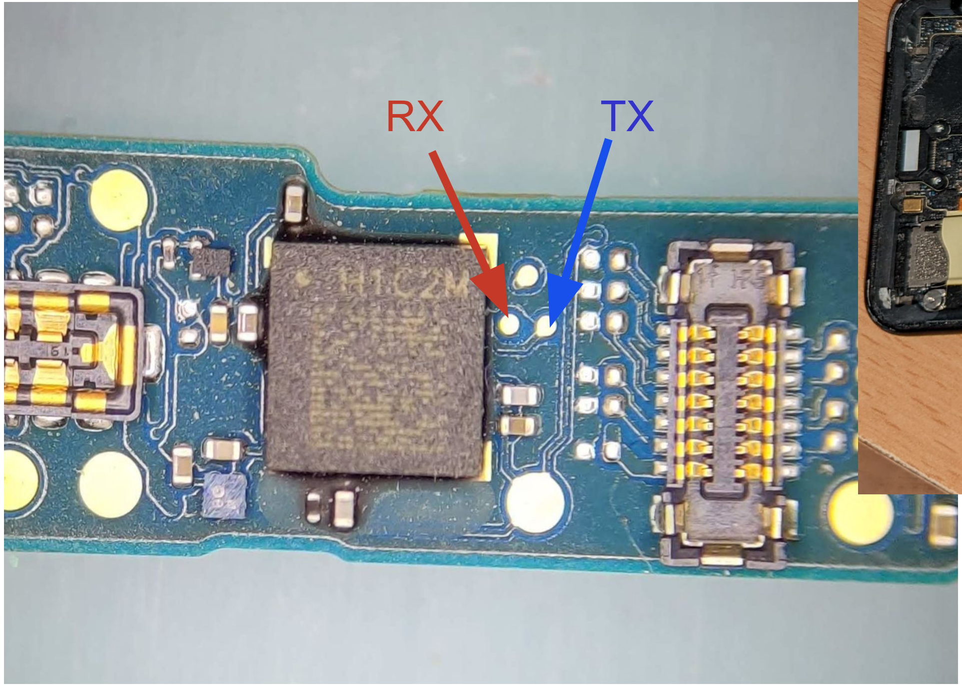
- Global structure
- Stores info about SPI commands
- `cmd_request_addr`: where to store incoming Keymaster requests
- `0x192c8` → `0x101c8`

😈

But first...

- Remainder:
  - Communication through `nosclient`
  - Send request using Android libs
  - Get a return code and (maybe) a response
  - A few logs on logcat
- What if we crash the chip?
  - Error code 2
- That’s it
- Debugging an exploit is... challenging

Accessing the UART



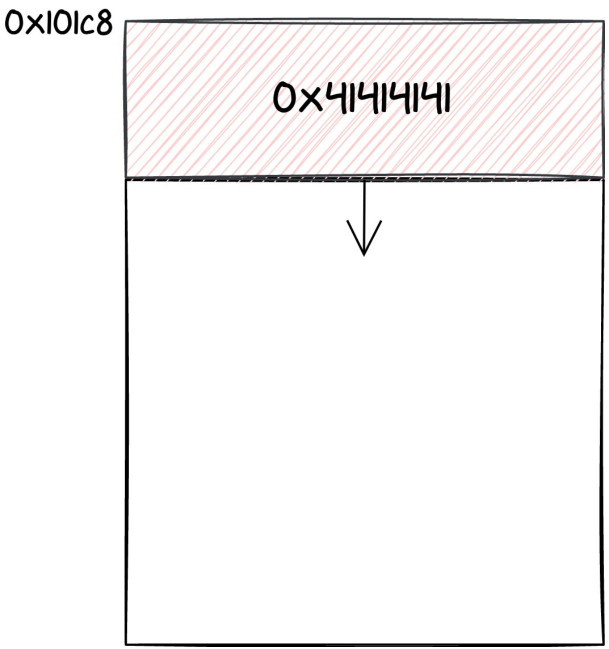
UART console

```
$ picocom /dev/ttyUSB0 -b 115200
[Image: RW_A, 0.0.3/chunk_ab7976980-a9084b7 2021-12-07
18:40:23 android-build]
[1.694592 Inits done]
[1.695460 update_rollback_mask: stop at 1]
[1.695884 gpio_wiggling: AP_EL2_LOW_IRQ = 0]
Console is enabled; type HELP for help.
>
> help
Known commands:
  apfastboot    history     repo        taskinfo    version
  board_id      idle        sleepmask   timerinfo
  help          reboot      stats       trngstats
HELP LIST = more info; HELP CMD = help on CMD.
```

- Allows basic interaction
- Prints logs
  - Useful when exploiting

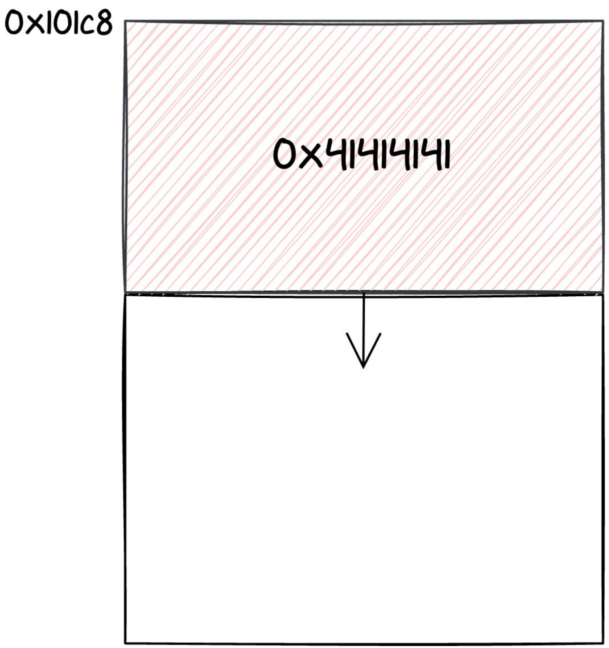
So, what's in 0x101c8?

- Data doesn't seem to be used
- How do we hijack execution flow?
- Idea:
  - Send progressively bigger payloads
  - In parallel monitor the UART
  - ... and see what happens



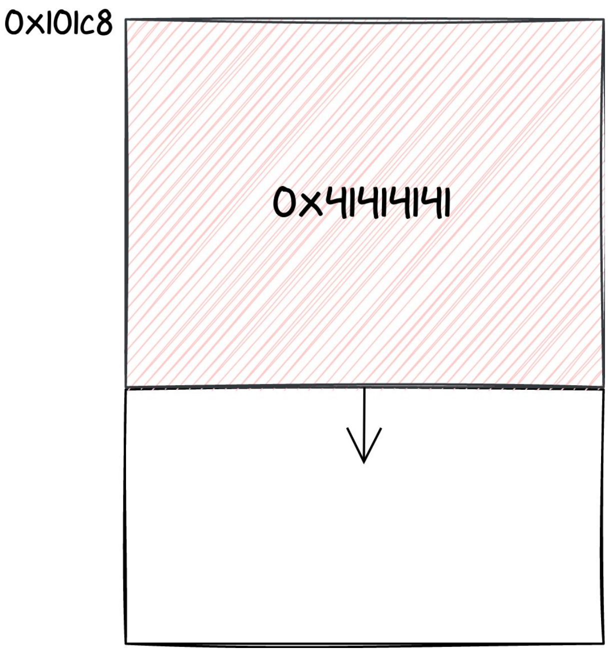
So, what's in 0x101c8?

- Data doesn't seem to be used
- How do we hijack execution flow?
- Idea:
  - Send progressively bigger payloads
  - In parallel monitor the UART
  - … and see what happens



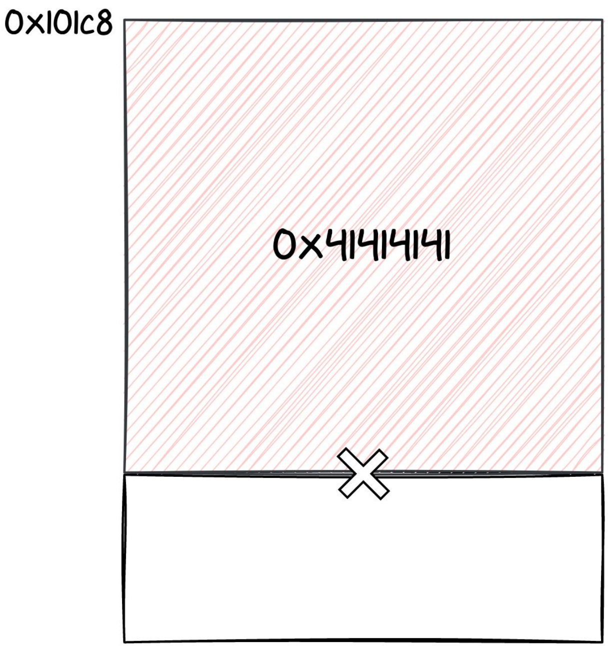
So, what’s in 0x101c8?

- Data doesn’t seem to be used
- How do we hijack execution flow?
- Idea:
  - Send progressively bigger payloads
  - In parallel monitor the UART
  - … and see what happens



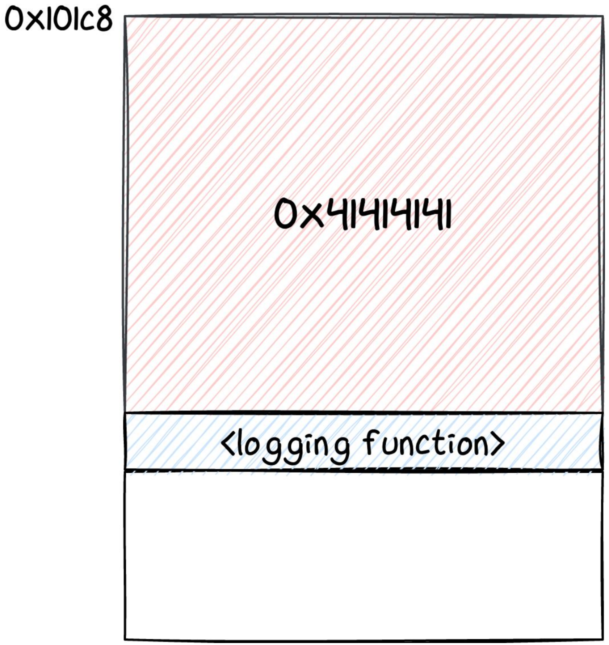
So, what's in 0x101c8?

- Data doesn't seem to be used
- How do we hijack execution flow?
- Idea:
  - Send progressively bigger payloads
  - In parallel monitor the UART
  - … and see what happens
- At some point, the chip starts crashing



So, what's in 0x101c8?

- Data doesn't seem to be used
- How do we hijack execution flow?
- Idea:
  - Send progressively bigger payloads
  - In parallel monitor the UART
  - … and see what happens
- At some point, the chip starts crashing
- What if we put a valid address at the end?



So, what's in 0x101c8?

- Data do
- How do
- Idea:
  - Send
  - In pa
  - ... an
- At some
- What if w
- \o/

```
--- UART initialized after reboot ---
[Reset cause: power-on]
[Retry count: 1]
[Image: RW_A, 0.0.3/chunk_ab7976980-a9084b7 2021-12-07 18:40:23 android-build]
[0.001684 Inits done]
[0.002532 update_rollback_mask: stop at 1]
[0.002952 gpio_wiggling: AP_EL2_LOW_IRQ = 0]
Console is enabled; type HELP for help.
> EVENT: 2 0:0x00000001 1:0x00000000 2:0x00000000 3:0x00000000 4:0x00000000 5:
0x00000000 6:0x00000000 7:0x00000000 8:0x00000000 9:0x00000000
[0.073052 Retry count: 1 -> 0]
Task Ready Name         Events      Time (s)  StkUsed   Flags
   0 R << idle >>       00000000    0.000116    80/ 512  0000
   1 R HOOKS            20000000    0.001708   152/ 640  0000
   2   NUGGET           00000000    0.012108   168/1024  0000
   3   FACEAUTH         00000000    0.000456    80/2048  0000
   4   AVB              00000000    0.004440    88/4096  0000
   5   KEYMASTER        00000000    0.015640    88/9600  0000
   6   IDENTITY         00000000    0.000220    88/1952  0000
   7   WEAVER           00000000    0.010372   240/1024  0000
   8   CONSOLE          00000000    0.024296    80/ 576  0000
```

Exploiting

- Our guess:
  - We are actually in the stack of a task (`idle`)
  - We overwrite a function pointer that was pushed to the stack
  - At some point, the function jumps back to it
- From here on, things get complex
  - No space to write a ROP chain there
  - We need to move `$sp`
- In the end, we send another command to complete the exploit
- Blogpost arriving soon :)

Impact

- Control the execution flow of the chip
  - We did not try to reconfigure the MPU
  - … but we can do pretty much anything using ROP
- We implemented again a leak command
  - This time based on a 0-day
  - Data is not erased by the downgrade like before!
  - We can leak all the secrets stored in the chip's memory

```
sargo:/data/local/tmp # ./nosclient leak 0x0 0x10
00 00 02 00 99 14 00 00 b9 3e 00 00 b9 3e 00 00
```

Impact

Can we leak Strongbox keys?

Strongbox

- `StrongBox`: hardware-backed version of `Keystore`
  - Generate, use and encrypt cryptographic material
- Titan M does not store keys
  - Key blobs encrypted with a **K**ey **E**ncryption **K**ey
    - This KEK is derived in the chip from various internal elements
  - Key blobs are sent to the chip to perform crypto operations
  - `root` can **use** any key, but not **extract** it

Strongbox

There are 3 commands to use strongbox keys:

- *BeginOperation*
  - Contains the keyblob and the characteristics of the key
  - The chip will decrypt the keyblob
  - And save it for later into a **fixed address**
- *UpdateOperation*
  - Contains the data on which the operation is performed
  - Return the output bytes
- *FinishOperation*
  - Contains the data on which the operation is performed
  - Return the output bytes
  - End the operation

Leak strongbox keys

Our strategy:

1. Get the keyblob from the device
   - Stored in `/data/misc/keystore/persistent.sqlite`
2. Forge a *BeginOperation* request
3. Leak the decrypted key from the chip memory

“Live demo or it didn’t happen!”

Conditions

- Ability to send commands to the chip
  - Being root
  - Or direct access to the SPI bus
- Access to the keyblobs
  - Being root
  - Or find a way to bypass FBE...

Mitigation

```
KeyGenParameterSpec spec = new KeyGenParameterSpec.Builder("key_name",
    KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
          .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
          .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
          .setIsStrongBoxBacked(true)
          .setUserAuthenticationRequired(true)
```

Conclusion

Conclusion

- Titan M was an interesting target
  - Limited attack surface, but enough to expose some vulnerabilities
- With black box fuzzing, you easily get the surface bugs
- Emulation-based fuzzing is particularly effective of such target
  - Yet few tricks are required to optimize the results
- We found a critical 0-day
  - Allowed us to execute code on the chip
  - Permit to leak anything from the chip's memory
- A single software vulnerability is enough to leak strongbox keys

Tools & resources:
https://github.com/quarkslab/titanm

Thank you!

@max_r_b
@DamianoMelotti

contact@quarkslab.com

Quarkslab

Backup - EC Tasks

`idle`
`hook` → system events, timers

`nugget` → system control task

`AVB` → secure boot management

`faceauth` → biometric data

`identity` → identity documents support

`keymaster` → key generation and cryptographic operations

`weaver` → storage of secret tokens

`console` → debug terminal and logs

Backup - Communication with the chip

```
package nugget.app.keymaster;
// ...
service Keymaster {
  // ...
  rpc AddRngEntropy (AddRngEntropyRequest) returns (AddRngEntropyResponse);
  rpc GenerateKey (GenerateKeyRequest) returns (GenerateKeyResponse);
  // ...

message AddRngEntropyRequest {
  bytes data = 1;
}
message AddRngEntropyResponse {
  ErrorCode error_code = 1;
}

message GenerateKeyRequest {
  KeyParameters params = 1;
  uint64 creation_time_ms = 2;
}
```

- Protobuf-based
  - Serialization framework by Google
  - Language agnostic
  - Titan M uses the nanopb library
  - Limited risk of input validation bugs
- Protobuf definitions are part of the AOSP

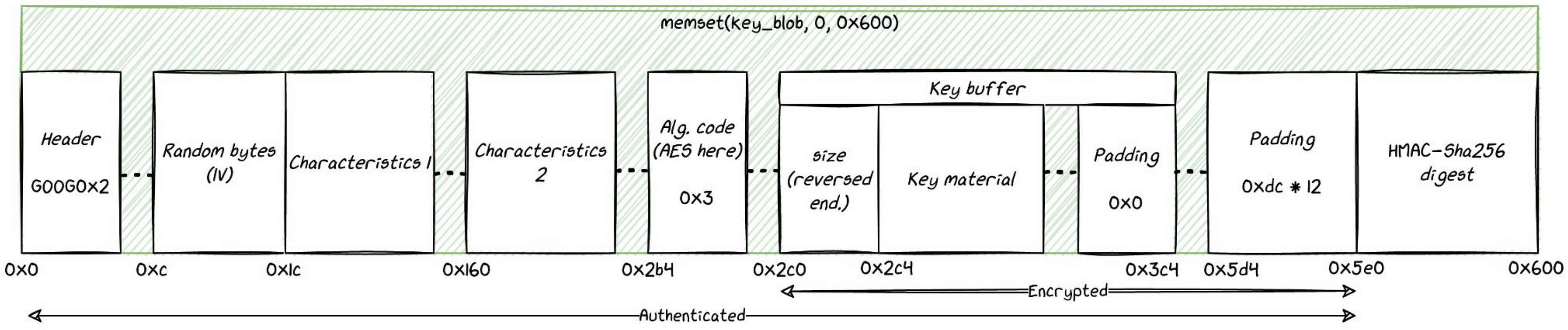
Backup - Command Handling Example on Titan M

```
uint32_t keymaster_AddRngEntropy (...,
    keymaster_AddRngEntropyRequest *request, ...,
    keymaster_AddRngEntropyResponse *response) {

  // ...

  iVar1 = pb_decode_ex(param_1,param_2,request,(uint)param_4);
  if (iVar1 == 0)
    return 1;

  km_add_entropy(request,response);
  iVar1 = pb_encode(param_4,param_5,response);

  return iVar1 == 0 ? 2 : 0;
}
```

Backup - Firmware security

- No dynamic allocation ➔ no UaF and similar
- **Secure boot** (images are signed and verified at boot)
- **MPU** to give permissions to the memory partitions
  - Custom interface to set the eXecute permission
  - No WX permissions by default
- **Only** software protection: hardcoded stack canary

Backup - Key Blob Structure



KEK: `SHA256(Root of Trust || salt || req1 || req2 || flash_bytes)`

HMAC KEY: `SHA256(Root of Trust || salt || flash_bytes)`

Backup - Fuzzing the Boot ROM

- Thanks to the 1-day exploit, we leaked the Boot ROM
- A bug there would be disastrous
- Not much code to test (only 16 KB)
- Idea: fuzz the image loader
  - We could flash them with SPI rescue

… no interesting results

- The function is simple, and not processing much
- Samples are just image headers

Backup - Strongbox

KEKs are derived from a **key ladder**

- Still quite mysterious since we did not reverse it
- It uses
  - An internal *root key*
    - Not readable from the Titan M firmware
  - A *Root Of Trust* provided by the bootloader at first boot
  - A *salt* that is randomly generated when RoT is provisioned

➜ We can leak most of the secrets, but not the key ladder root key