

Week 11a: Custom Secure Boot + AES in Firmware: Building a CAC-Signed Bootloader and Reverse Engineering It

LEGAL DISCLAIMER: The information, tools, and code provided in this repository and course are strictly for educational, research, and defensive purposes only.

You are explicitly prohibited from using any materials contained herein to access, test, modify, or exploit any device, network, or system that you do not own 100% or for which you do not have explicit, documented, and legally binding authorization to interact with.

By using this repository and course, you acknowledge and agree that:

1. Any illegal, unauthorized, or malicious use of this information is solely your responsibility.
2. The author(s) and contributor(s) of this repository and course shall not be held liable for any damages, legal repercussions, criminal charges, or unauthorized actions resulting from the use, misuse, or abuse of the contents herein.
3. You will comply with all applicable local, state, national, and international laws regarding cybersecurity and computer fraud.

IF YOU DO NOT AGREE WITH THESE TERMS, DO NOT USE THIS REPOSITORY AND COURSE.

What You'll Learn This Week

By the end of this tutorial, you will be able to:

- Understand why consumer and industrial devices use a secure (authenticated) boot chain
 - Explain, with a diagram, how AES-128-CMAC (RFC 4493) authenticates a firmware image
 - Read a custom linker script, modify the MEMORY block, and relocate a program to a higher flash address
 - Inspect a relocated ELF with `readelf`, `nm`, and `objdump` to confirm where every section and symbol lives
 - Use Mbed TLS (already shipped inside the Pico SDK) in your own C code to decrypt data at runtime
 - Flash a raw `.bin`, a manifest, and an application to three different flash addresses with OpenOCD
 - Tamper with a single byte of firmware and watch the bootloader reject it at boot
 - Reverse engineer the bootloader with GDB live on the Pico 2 (breakpoints, memory dumps, single-stepping the jump)
 - Reverse engineer both firmware images in Ghidra, including recovering an AES key, IV, and ciphertext from flash
 - Explain why a static key is not a real secret, and how real products solve secure boot (OTP keys, ROM verification, ECDSA)
-

Part 1: Why Secure Boot? (The Threat Model)

Firmware is a Product, and Products Get Modified

Everything you have flashed onto your Pico 2 so far in this course has been a **single standalone image**. The RP2350 boot ROM reads the image at `0x10000000`, your program runs, done. That is exactly how millions of cheap embedded products work, and it has a serious weakness:

Anybody with a \$5 probe can replace the flash contents.

OpenOCD, a 5-pin SWD connection, and a serial terminal are all an attacker needs to:

1. Dump your firmware (read flash)
2. Modify a byte (change a password, unlock a feature, bypass a license)
3. Flash their own image (rootkit the device)

Now tighten the screw. Imagine your product is:

- A smart lock that talks to the network
- A security camera that signs video
- A commercial battery charger that detects counterfeit batteries
- An update server that pushes "fixes" to thousands of deployed devices

In those products, **running modified or unauthorized firmware is a real-world safety and business problem**. If a device can be made to run code it did not originate, that device cannot be trusted, and neither can the data it produces.

The Defense: A Verified Boot Chain

The modern defense is a **boot chain**. Each stage of the chain refuses to start the next stage until it has proven (cryptographically) that the next stage is authentic:

```
+-----+
| BOOT CHAIN (each stage authenticates the next) |
| |
| +-----+ |
| | STAGE 0: ROM in the chip (frozen in silicon, cannot be changed) | |
| | verifies ... | |
| +-----+ |
| | | trusts | |
| | v | |
| +-----+ |
| | STAGE 1: bootloader in flash (this lab) | |
| | verifies the app with an AES-128-CMAC tag | |
| +-----+ |
| | | trusts (only if the tag matches) | |
| | v | |
| +-----+ |
| | STAGE 2: application firmware (the IR remote control app) | |
| +-----+ |
| |
| An attacker's modified app is REJECTED at stage 1 and never runs. |
| The device is very hard to brick with bad firmware, and cheap |
| modification attacks fail. |
+-----+
```

What this lab does, and what it does not.

*The 0x0026a bootloader authenticates the application using a key that is **embedded in the bootloader itself**. That stops a casual attacker who uses OpenOCD to patch a byte or two. A serious attacker who dumps the flash can read the key and re-sign their own image. Real products push Stage 0 into the ROM*

and store the key in **OTP (one-time programmable) memory** that flash reads cannot access. Part 19 shows how to think about that gap.

Part 2: Choosing the Authentication Primitive

The bootloader must answer one question about the application image: "Did this byte-for-byte image come from the person who holds the key?"

The Candidate Tools

Tool	What it computes	Can an attacker forge it?
CRC-32	A simple checksum	YES, trivially. CRC is designed to catch noise, not attackers.
SHA-256 hash	A one-way digest of the image	YES, if there is no secret. Anyone can compute the hash after editing.
HMAC-SHA256	A keyed digest	NO (without the key). Strong, but needs a keyed hash implementation.
AES-CMAC	A keyed MAC built from a block cipher	NO (without the key). Uses the AES primitive you already have.
ECDSA signature	An asymmetric signature	NO. The verifier holds only a public key, so even a full flash dump cannot forge.

Why AES-CMAC for This Lab

Two practical reasons:

1. **The Pico SDK already ships a fast, well-tested AES.** CMAC is built on AES (it is a CBC-MAC variant), so we do not add SHA code or HMAC glue; we reuse the `MBEDTLS` library that ships inside the SDK (`pico_mbedtls`).
2. **One primitive, two lessons.** The same AES engine that authenticates the image in the bootloader is also used inside the application to decrypt a secret banner (Part 9). You see one cipher doing two classic jobs.

ECDSA would be stronger (asymmetric, so the flash dump does not reveal the signing key), and the RP2350 ROM ships a `secp256k1` verifier. The gap between where this lab leaves off and where the ROM verifier lies is an important "real product" paragraph in Part 19.

Part 3: AES-128-CMAC in Depth (RFC 4493)

A MAC Is a Keyed Checksum

A **Message Authentication Code (MAC)** takes three inputs:

- a secret key `K`
- a message `M`
- (optionally) a length `Tlen`

and produces a short, fixed-size **tag** `T`. Security property: without `K`, an attacker cannot construct `(M, T)` that verifies, even if they are allowed to see thousands of valid `(M, T)` pairs.

Outline of CMAC

CMAC is a keyed block-cipher MAC standardised in **NIST SP 800-38B** and implemented in **RFC 4493** for AES-128. It is a **CBC-MAC with a twist**: the final block is XORed with one of two derived subkeys before the last AES encryption, which removes the classical CBC-MAC extension attack.

```
+-----+
| Subkey derivation (once, at key setup) |
|                                         |
| L = AES_K(0^128)                      AES-encrypt a zero block |
| if MSB(L) == 0: K1 = L << 1           |
| else:      K1 = (L << 1) XOR 0x87      (0x87 = Rb for AES) |
| same step on K1 yields K2             |
|                                         |
| Tagging (per message M, split into 128-bit blocks M1..Mn) |
|                                         |
| M0 = 0^128                             |
| if M is an exact multiple of 128 bits: |
|     Mn' = Mn XOR K1                     |
| else: (pad M with 10...0 to a full block) |
|     Mn' = padded(Mn) XOR K2             |
|                                         |
| for each block: C[i] = AES_K( C[i-1] XOR M[i] ) |
| Tag = C[n] (first Tlen bits)           |
+-----+
```

The two subkeys exist precisely to encode "full last block" vs "short last block" into the tag. If we reused plain CBC-MAC, an attacker could cut the message after a block boundary and still produce a valid tag.

Where Is AES-128-CMAC Not Really "Analysis" But Simple Math?

Study the equation for a single message block:

```
C[1] = AES_K( 0^128 XOR M1 )
```

Because this is a block cipher, a 1-bit change anywhere in `M1` results in a completely unrelated tag. And without `K`, you cannot compute `AES_K(...)` at all. Both facts are what make the tag unforgeable in practice.

The RFC 4493 Test Vectors (how the tool proves itself)

The host signer `tools/sign_firmware.py` implements AES and CMAC from scratch in pure Python, so the build toolchain has zero extra dependency on the development machine. It is validated against the four official test vectors from RFC 4493 using the well-known test key:

```
Key      = 2b7e151628aed2a6abf7158809cf4f3c

Message = (empty)
Tag      = bb1d6929e95937287fa37d129b756746

Message = 6bc1bee22e409f96e93d7e117393172a
Tag      = 070a16b46b4d4144f79bdd9dd04a287c
```

```
Message = 40-byte repeating sample
Tag      = dfa66747de9ae63030ca32611497c827

Message = 64-byte repeating sample
Tag      = 51f0bebf7e3b9d92fc49741779363cfe
```

Read the tool's top-of-file comment; the four vectors are exercised against the signer's internal CMAC routine before any real image is signed.

Part 4: The Hardware and the Flash Map

The Pieces

Component	Role in this lab
Pico 2 (RP2350)	The target. Two Cortex-M33 cores, 512 KB SRAM, executes code in place from XIP flash.
Debug Probe	CMSIS-DAP SWD adapter for flashing, debugging, and UART serial. It owns the SWD port and usually also gives a UART console channel.
IR receiver	GPIO5 input; decodes NEC remote commands in the application.
3 LEDs	GPIO16, GPIO17, GPIO18; driven by decoded commands.
USB-UART/cable	If the Debug Probe's UART is wired to the Pico's UART0, the bootloader and app log to it.

Why 0x10000000?

On the RP2350 the first 4 MiB of the address map (from `0x10000000`) is a **memory-mapped view of external XIP flash** (eXecute In Place). The CPU fetches instructions straight from that window; the flash controller reads the bytes transparently through the XIP cache. The boot ROM boots from this flash window because it is simply memory.

So "flashing at `0x10000000`" means "making the contents of physical flash visible at the addresses your program expects." A raw `.bin` carries no address information; you must always supply the load address out of band (Part 12).

The 0x0026a Flash Layout

BOOTLOADER image (0x0026a_bootloader.bin)	0x10000000
boot.c verify.c manifest.c aes_cmac.c	
image.c main.c (~28 KB total)	~0x10006D00
(free padding)	
MANIFEST sector (4 KB, one flash sector)	0x1001F000
magic, version, app base address, app size,	
and a 16-byte AES-128-CMAC tag	
APPLICATION image (0x0026a_app.bin)	0x10020000

the Week 11 IR remote app plus an AES-128-CBC	
secret banner, linked to run from here	size 0x5498
-----+-----	-----+-----
(free padding)	
-----+-----	-----+-----
End of usable flash (2 MiB region)	0x10200000
-----+-----	-----+-----

Three things to notice:

1. The manifest sits in its **own flash sector** (`0x1001F000`), so updating it never touches the bootloader or the application.
2. The application is **relocated** above both, starting at `0x10020000` . It is not written at `0x10000000` .
3. The bootloader jump at the end of Part 8 hands control to the app's reset vector, which lives at `0x1002015C` .

Part 5: The Manifest Format

Before code runs, the bootloader reads a 4 KB record from flash. The first 28 bytes matter; the rest is `0xFF` padding (erased flash).

```
Declared in bootloader/include/manifest.h
#pragma pack(push, 1)
typedef struct {
    uint32_t magic;      // 0x424C4657 ("WFLB")
    uint32_t version;    // 1
    uint32_t app_base;   // 0x10020000
    uint32_t app_size;   // 0x5498 (21656 bytes)
    uint8_t tag[16];     // AES-128-CMAC of the app image
} fw_manifest_t;
#pragma pack(pop)
```

Annotated hex dump of the real `manifest.bin` produced in Part 11:

```
00000000: 57 46 4c 42 01 00 00 00 00 00 02 10 98 54 00 00 WFLB.....T..
      ~~~~~ ~~~~~ ~~~~~ ~~~~~
      magic      version      app_base      app_size
                        0x10020000 0x5498

00000010: ef 04 1e c5 92 f0 c5 67 27 da 26 51 d3 73 94 7b .....g'&Q.s.{
      ~~~~~
      tag (16 bytes): AES-128-CMAC over the application

00000020: ff ff ff ff ... (0xFF padding to 0x1000)
```

Note the little-endian layout. `00 00 02 10` reads as `0x10020000` , and `98 54 00 00` reads as `0x5498` . The tag is the raw 16-byte CMAC output, not a text string.

Part 6: The Custom Linker Script (Where Your Program Really Lives)

This is the part that answers a question students ask constantly: *"If I want to put a second program above the first one, how does the compiler know where it belongs?"* Answer: **the linker script says so**. Understand this file and you control a device's entire memory layout; ignore it and relocated firmware silently corrupts itself.

6.1 What a Linker Script Is

Compilation produces **object files** with named sections: `.text` (code), `.rodata` (constants), `.data` (initialized globals), `.bss` (zeroed globals), and so on. The linker (here, the GNU `arm-none-eabi-ld`) merges those sections into a single ELF and decides **which virtual addresses each section gets**. That decision comes from a script, normally `/path/to/lds` or, for the Pico SDK, a script your CMake passes in.

A copy of the RP2350 script lives here:

```
app/memmap_app.ld      <- our modified version, used for the app only
```

The SDK's original (do not edit) is inside the Pico SDK:

```
~/pico-sdk/sdk/2.2.0/src/rp2_common/pico_crt0/rp2350/memmap_default.ld
```

6.2 The MEMORY Command: ORIGIN and LENGTH

The heart of the script is the `MEMORY` block. It tells the linker which address ranges are real and how big they are. The SDK's default is:

```
/* from memmap_default.ld */
MEMORY
{
    INCLUDE "pico_flash_region.ld"
    RAM(rwx) : ORIGIN = 0x20000000, LENGTH = 512k
    SCRATCH_X(rwx) : ORIGIN = 0x20080000, LENGTH = 4k
    SCRATCH_Y(rwx) : ORIGIN = 0x20081000, LENGTH = 4k
}
```

What each line says:

Region	ORIGIN	LENGTH	What lives there
FLASH	0x10000000	board flash size	<code>.text</code> , <code>.rodata</code> , and initial values for <code>.data</code> (XIP).
RAM	0x20000000	512k	<code>.data</code> , <code>.bss</code> , heap
SCRATCH_X	0x20080000	4k	Core 1 stack / time-critical scratch
SCRATCH_Y	0x20081000	4k	Core 0 stack (ends at 0x20082000)

`ORIGIN` is the first byte of the region; `LENGTH` is how many bytes it spans. Anything placed in `FLASH` runs directly from the XIP window; anything placed in a RAM region is copied there at boot by the `_reset_handler`.

6.3 Where FLASH Actually Comes From: `pico_flash_region.ld`

Notice that `FLASH` is not declared inline; it is **included**:

```
INCLUDE "pico_flash_region.ld"
```

The SDK's CMake generates this tiny file for you based on the board's flash size. Ours (from the build directory) is:

```
/* build/pico_flash_region.ld */  
FLASH(rx) : ORIGIN = 0x10000000, LENGTH = (4 * 1024 * 1024)
```

So for the default Pico 2 the whole 4 MiB of flash is one region called `FLASH`, read-only + execute, starting at `0x10000000`. If you build a board with 2 MiB you would see `LENGTH = (2 * 1024 * 1024)`.

6.4 The Modification: Relocating the Application

The bootloader owns the start of flash, so the app must not begin at `0x10000000`. We copy the SDK script and change **one thing in MEMORY**: the FLASH line. In `app/memmap_app.ld`:

```
MEMORY  
{  
    /* Relocated for the custom bootloader: the application is linked to run  
       from 0x10020000, leaving the first 128k of flash for the bootloader and  
       its manifest. */  
    FLASH(rx) : ORIGIN = 0x10020000, LENGTH = 1920k  
    RAM(rwx) : ORIGIN = 0x20000000, LENGTH = 512k  
    SCRATCH_X(rwx) : ORIGIN = 0x20080000, LENGTH = 4k  
    SCRATCH_Y(rwx) : ORIGIN = 0x20081000, LENGTH = 4k  
}
```

Arithmetic behind `1920k`:

```
default FLASH region:    0x10000000 + 2048 KiB    (2 MiB reserved layout)  
reserved for boot/manifest:  
    bootloader image ~28 KiB, manifest at 0x1001F000..0x10020000  
    -> reserve the first 128 KiB (0x20000 bytes)  
application window:      0x10020000 + 1920 KiB = 0x10200000    end of 2 MiB  
1920 KiB = 2048 KiB - 128 KiB
```

Every `> FLASH` and `> RAM` AT> `FLASH` section in the rest of the script now places itself inside this smaller, relocated window. The `SECTIONS` body is otherwise identical to the SDK default.

6.5 Wiring the Script Into the Build (CMake)

The SDK exposes a CMake function that swaps in our script for one target only:

```
# CMakeLists.txt  
pico_set_linker_script(0x0026a_app  
    ${CMAKE_CURRENT_LIST_DIR}/app/memmap_app.ld)
```

The bootloader target gets no such call, so it uses the SDK default (`FLASH` at `0x10000000`) exactly as before.

6.6 Symbols the Script Defines

A linker script does not only place sections; it also **exports symbol values** that startup code uses. Grep `memmap_app.ld` for these while you read:

Symbol	Meaning	Value for the app
<code>__flash_binary_start</code>	First byte of the image in flash	<code>0x10020000</code>
<code>__logical_binary_start</code>	Same as above in this script	<code>0x10020000</code>
<code>__etext</code>	End of code (start of <code>.data</code> initial values in flash)	near <code>0x10026000</code>
<code>__data_start__</code>	Loaded <code>.data</code> base in RAM	<code>0x20000000</code>
<code>__bss_start__/_bss_end__</code>	Zero-initialized globals in RAM	inside <code>0x20000000</code> region
<code>__StackLimit</code>	Heap cap (historically the "stack limit")	<code>0x20080000</code>
<code>__StackTop</code>	Top of core 0 stack (end of <code>SCRATCH_Y</code>)	<code>0x20082000</code>
<code>__HeapLimit</code>	End of RAM = <code>ORIGIN(RAM) + LENGTH(RAM)</code>	<code>0x20080000</code>

The crucial practical consequence: **the first word of the app's vector table must be `__StackTop` = `0x20082000`**. Check it:

```
$ xxd -l 16 0x0026a_app.bin
00000000: 0020 0820 5d01 0210 1b01 0210 1d01 0210  . . ].....
```

- bytes 0-3 `00 20 08 20` = little-endian `0x20082000` = initial SP
- bytes 4-7 `5d 01 02 10` = little-endian `0x1002015d` = reset vector (the `_reset_handler` symbol, Thumb-encoded with bit 0 set)

That `0x1002015d` would be impossible to obtain without the relocated `MEMORY` block. This is the tangible proof that the linker script did its job.

6.7 ENTRY(_entry_point) Versus the Reset Vector

At the top of the script:

```
ENTRY(_entry_point)
```

This names the ELF entry point used by debuggers. In the SDK, `_entry_point` is a tiny stub in `crt0.S` that routes a debugger-launched session back through the boot ROM (so XIP setup happens as on a cold boot). It is **not** the same as the reset vector in the vector table.

The bootloader does not use `_entry_point` at all. It jumps straight at the app's **reset vector** (`vector[1]`), which is `_reset_handler`, the function that actually initializes `.data`, zeros `.bss`, and calls `main`. Part 8 shows that branch instruction. This distinction matters when you are poking a relocated image with a debugger: breaking on `_entry_point` and breaking on `_reset_handler` are different sessions.

6.8 Inspecting the Result With readelf and nm

After building (Part 10), look at what the linker produced. The section table:

```
$ ~/.pico-sdk/toolchain/14_2_Rel1/bin/arm-none-eabi-readelf -S build/0x0026a_app.elf
[ 0] null
[ 1] .flash_begin    PROGBITS  10020000  00001000  000000  00 WA  0  0  1
[ 2] .text            PROGBITS  10020100  00001100  005e34  00 AX  0  0 16
...
[ 5] .rodata          PROGBITS  10025f40  00006f40  000235  00  A  0  0  8
...
```

Note that every flash section is stamped `1002xxxx`, never `1000xxxx`. The program headers (use `readelf -l`) likewise show `LOAD` segments whose vaddr sits in the `0x10020000` window. `nm` shows symbol addresses:

```
$ ~/.pico-sdk/toolchain/14_2_Rel1/bin/arm-none-eabi-nm -n build/0x0026a_app.elf
...
1002015c T _reset_handler
10020234 T main
1002027c T ir_init
...
```

6.9 One UF2 for the Whole Chain

No single image is the whole story. The device needs the bootloader at `0x10000000`, the manifest at `0x1001F000`, and the app at `0x10020000`. A standalone `0x0026a_app.uf2` would flash only the app region, leaving the other two to be written separately, so the app is deliberately built with explicit `.bin / .hex / .dis / .map` outputs instead of its own UF2.

To bring a bare board up in one move the lab ships a single combined UF2, `0x0026a_full.uf2`, that carries all three images. It is built by laying the images out flat - offset 0 mapped to `0x10000000`, the reserved gaps filled with `0xFF` - and handing the result to picotool as one image:

```
$ PICOTOOL=~/.pico-sdk/picotool/2.3.1/picotool/picotool \
python3 tools/make_full_uf2.py
```

picotool numbers the blocks, tags them with the `rp2350-arm-s` family, and injects the RP2350-E10 "absolute block" (family `0xE48BFF57` at `0x10FFFF00`) that tells the boot ROM to write every block to its **absolute** address rather than into a partition. A single BOOTSEL drag then restores the whole chain. Part 11 uses it. Note the contrast with the naive file layout: a UF2 is not restricted to `0x10000000`, but a multi-region UF2 is only correct once the blocks and the E10 absolute block are laid out the way picotool produces them.

Part 7: The Bootloader Source, Module by Module

The bootloader applies the standard we use for every module in this course: **main.c contains only the sequence; each job has its own .c + .h pair.**

```
bootloader/
include/
  flash_layout.h  addresses of the three flash partitions
  manifest.h      fw_manifest_t type + read/validate API
  boot.h          console, halt, launch API
  verify.h        firmware authenticity check
```

aes_cmac.h	MBEDTLS front-end + constant-time compare
image.h	low-level jump to the app
src/	
main.c	ONLY the 5-step boot sequence
boot.c	banner console, fatal-haunt loop, handoff
manifest.c	flash-read + magic/version validation
verify.c	the static key + CMAC recompute + tag compare
aes_cmac.c	computes the tag with mbedtls AES-128-CMAC
image.c	VTOR/MSP rebase + branch to reset vector

7.1 main.c : Only the Sequence

The whole of `main.c` :

```
int main(void) {
    fw_manifest_t manifest;
    boot_init();
    manifest_read(&manifest);
    if (!manifest_is_valid(&manifest)) boot_halt("manifest invalid");
    if (!verify_manifest(&manifest)) boot_halt("firmware signature invalid");
    boot_launch(&manifest);
}
```

Read it as a check-list, one step per line:

1. `boot_init()` - stdio up, 2 s pause, banner print
2. `manifest_read` - copy the 28 useful bytes out of the `0x1001F000` sector
3. `manifest_is_valid` - magic WFLB and version == 1?
4. `verify_manifest` - recompute CMAC over the app, compare to stored tag
5. `boot_launch` - print success, jump to the app reset vector

There is **no** stdio, no key, no crypto in this file. Everything below is a different module.

7.2 verify.c : The Secret and the Verdict

```
/** @brief AES-128 key that authenticates the application image. */
static const uint8_t fw_key[FW_TAG_SIZE] = {
    0x49, 0x72, 0x6F, 0x6E, 0x4E, 0x65, 0x74, 0x42,
    0x6F, 0x6F, 0x74, 0x4B, 0x65, 0x79, 0x21, 0x21
};

bool verify_manifest(const fw_manifest_t *manifest) {
    const uint8_t *base = (const uint8_t *)manifest->app_base;
    uint8_t computed[FW_TAG_SIZE];
    int rc = aes128_cmac(fw_key, base, manifest->app_size, computed);
    if (rc != 0) return false;
    return cmac_tags_equal(computed, manifest->tag);
}
```

What is important here:

- The key is 16 bytes "IronNetBootKey!!". It is `static const`, so the compiler puts it in `.rodata`, right in the binary, at a predictable address (you will extract it in Ghidra in Part 15).
- The CMAC is computed over **the raw bytes the processor will execute**, starting at `manifest->app_base` for `app_size` bytes. That is exactly the `0x0026a_app.bin` content (21656 bytes, `0x5498`).
- The comparison is deliberately a full 16-byte compare with no shortcut, so the runtime does not leak *which* byte mismatched (timing side-channels).

7.3 `boot.c` : Console and Handoff

```
void boot_init(void) {
    stdio_init_all();
    sleep_ms(2000);
    printf("\n=== 0x0026a secure bootloader ===\n");
}

void boot_halt(const char *reason) {
    printf("BOOT FAILED: %s\n", reason);
    while (true) {
        sleep_ms(500);
    }
}

void boot_launch(const fw_manifest_t *manifest) {
    printf("signature valid, entering app at 0x%08lx\n",
        (unsigned long)manifest->app_base);
    image_start(manifest->app_base);
}
```

A 2 000 ms delay before the banner gives a human time to open a serial terminal; the blink of that pause is also a simple "alive" signal. On failure, `boot_halt` loops forever with a 500 ms heart-beat so you can see the red LED blinking and the exact reason on the console.

7.4 `image.c` : The Actual Jump

```
static void halt_irq_and_rebase(uint32_t base) {
    __asm volatile ("cpsid i" ::: "memory");
    *(volatile uint32_t *)0xE00ED08U = base;
}

void image_start(uint32_t vector_base) {
    uint32_t *vectors = (uint32_t *)vector_base;
    image_entry_t entry = (image_entry_t)vectors[1];
    halt_irq_and_rebase(vector_base);
    __asm volatile ("msr msp, %0" :: "r" (vectors[0]) : "memory");
    __asm volatile ("dsb\n\tisb" ::: "memory");
    entry();
}
```

Step by step:

1. Read the app's vector table at `vector_base` (`0x10020000`).
2. `entry = vectors[1]` - the reset vector (`0x1002015d`), bit 0 set for Thumb mode. Branching to a Thumb-encoded address is how you switch modes.
3. `cpsid i` - mask interrupts so nothing fires mid-handoff.
4. Write `0xE000ED08` (the **VTOR** register) with the new vector table base. From here on, exceptions use the app's vectors.
5. `msr msp` - load the app's initial stack pointer from `vectors[0]` (`0x20082000`).
6. `dsb / isb` - force the memory and instruction pipelines to catch up.
7. `entry()` - call the reset handler. It never returns to the bootloader.

The assembly, as it appears in the ELF (this is exactly what you will disassemble in GDB and Ghidra):

```
10000394 <image_start>:
10000394: 6843      ldr r3, [r0, #4]      ; entry = vectors[1]
10000396: b672      cpsid i          ; mask IRQs
10000398: mov.w r2, #0xe000e000    ; SCB base
1000039c: str.w r0, [r2, #3336]     ; VTOR = r0 (0xE000ED08)
100003a0: ldr r2, [r0, #0]         ; sp = vectors[0]
100003a2: msr MSP, r2
100003a6: dsb sy
100003aa: isb sy
100003ae: bx r3                  ; branch to reset handler
```

When you set a breakpoint on `image_start` and single-step, you will see the PC jump from `100003ae` to `1002015d` and disappear into application startup.

Part 8: Using AES in the Application (a Fully Worked Example)

The application is the Week 11 IR remote firmware, but with one addition: a module (`secret.c`) that decrypts a banner at runtime using the **Mbed TLS AES that ships inside the Pico SDK**. This part teaches you the SDK's real AES API on real code, and it plants the exact artifact you will hunt for in the reverse-engineering parts.

8.1 The API You Need (all from mbedtls, all shipped in the SDK)

Four calls. That is the entire surface we touch:

Call	Purpose
<code>mbedtls_aes_init(ctx)</code>	Give the context a clean base state.
<code>mbedtls_aes_setkey_dec(ctx, key, 128)</code>	Expand the 16-byte key for decryption.
<code>mbedtls_aes_crypt_cbc(ctx, mode, len, iv, src, dst)</code>	Decrypt one or more whole blocks.
<code>mbedtls_aes_free(ctx)</code>	Clean up (releases nothing for a stack context).

`mbedtls_aes_crypt_cbc` is the workhorse. Its important contract:

- `len` must be a multiple of 16 (a whole number of AES blocks).
- The IV is **read and updated in place**; you must give it a mutable copy.

- `src` and `dst` may be the same pointer - the operation is **in-place**. That is how `secret_decrypt` reuses one 33-byte buffer for ciphertext, then plaintext.

8.2 The Secret Banner Link: Host Side Production

The plaintext banner is exactly 32 bytes (two AES blocks), so no padding is needed. It was encrypted on the host with OpenSSL using the same key and IV:

```
$ printf '%s' 'SECRET: IR LINK SECURED ONLINE ' \
| openssl enc -aes-128-cbc \
-K 49726f6e4e6574426f6f744b65792121 \
-iv 41707053656372657449563132333435 \
-nopad | xxd -p
baec4a1925861bb92dfd9f13f79b3e80cf12b698db9165be8fd7c3477bb2e76e
```

The hex strings are:

```
KEY = 49 72 6F 6E 4E 65 74 42 6F 6F 74 4B 65 79 21 21 = "IronNetBootKey!!"
IV  = 41 70 70 53 65 63 72 65 74 49 56 31 32 33 34 35 = "AppSecretIV12345"
CT  = ba ec 4a 19 25 86 1b b9 2d fd 9f 13 f7 9b 3e 80
      cf 12 b6 98 db 91 65 be 8f d7 c3 47 7b b2 e7 6e
PT  = "SECRET: IR LINK SECURED ONLINE " (32 bytes, two spaces padded)
```

Notice: the ciphertext bytes in the disassembly of `secret_print_banner` (Part 16) are **exactly** `baec4a19...76e6`. You will re-discover that the reverse engineers' job is rarely to break AES (you cannot); it is to find where the key, IV, and ciphertext are stored and feed them to a decryptor. Here the key is 70 bytes away from the ciphertext.

8.3 The Application-Side Module (`secret.c`)

```
static int secret_cbc(mbedtls_aes_context *aes, uint8_t *iv, uint8_t *buf) {
    int rc = mbedtls_aes_crypt_cbc(aes, MBEDTLS_AES_DECRYPT, APP_BANNER_LEN,
                                   iv, buf, buf);

    return rc;
}

static int secret_decrypt(uint8_t *buf) {
    mbedtls_aes_context aes;
    uint8_t iv[APP_AES_BLOCK];
    memcpy(iv, banner_iv, APP_AES_BLOCK);
    mbedtls_aes_init(&aes);
    mbedtls_aes_setkey_dec(&aes, banner_key, 128);
    int rc = secret_cbc(&aes, iv, buf);
    mbedtls_aes_free(&aes);
    return rc;
}

int secret_print_banner(void) {
    uint8_t buf[APP_BANNER_LEN + 1];
    memcpy(buf, banner_ct, APP_BANNER_LEN);
    buf[APP_BANNER_LEN] = '\0';
    int rc = secret_decrypt(buf);
}
```

```

    if (rc != 0) return rc;
    printf("app banner: %s\n", (char *)buf);
    return 0;
}

```

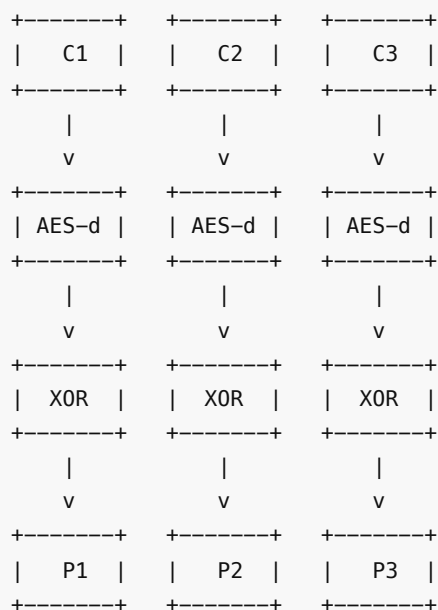
Walk through it:

1. `memcpy(buf, banner_ct, 32)` - pull the two ciphertext blocks into a working buffer (with a spare byte for the terminating `'\0'`).
2. `banner_iv` is `static const`, so it cannot be safely mutated by the CBC routine; copy it to a local array first.
3. Init + expand the key ("setkey_dec" stores the pre-computed decryption key schedule once, so `crypt_cbc` is fast).
4. `crypt_cbc` writes the plaintext back into `buf`: `src == dst`.
5. Print. Release.

If the SDK's mbedtls were misconfigured (say, you forgot to enable `MBEDTLS_AES_C` in `common/include/mbedtls_config.h`), `crypt_cbc` returns a negative error instead of decrypting, and the module refuses to print garbage.

8.4 CBC, In Pictures

CBC decryption undoes each block with the *previous* ciphertext block:



That is why the IV is the "previous block" for block 1. In-place operation is safe because C1 is no longer needed once P1 is produced ... except block two needs C1, so the ciphertext *before* it was overwritten must still be intact. mbedtls handles this correctly internally; you only owe it an untouched IV copy.

8.5 How the Application Is Structured (modules, not a mega-file)

This week the app got the same treatment as the bootloader: `app_main.c` is only entry + the poll loop. Everything else lives in a module:

```

app/
include/
  ir.h      NEC receiver API (ir_init, ir_getkey)
  led.h     LED controller type + API (simple_led_ctrl_t, led_default,
        led_init, led_handle_ir)
  secret.h AES banner API (secret_print_banner)
src/
  app_main.c ONLY stdio, banner, led_init, ir_init, then the poll loop
  ir.c       NEC timing decoder
  led.c      3-LED controller (mapping, blink, hold-on)
  secret.c   the AES-128-CBC banner module

```

`app_main.c` in full:

```

static void ir_poll_loop(simple_led_ctrl_t *leds) {
    while (true) {
        int key = ir_getkey();
        if (key >= 0) led_handle_ir(leds, key);
        else sleep_ms(1);
    }
}

int main(void) {
    stdio_init_all();
    secret_print_banner();
    simple_led_ctrl_t leds = led_default();
    led_init(&leds);
    ir_init(IR_PIN);
    printf("IR receiver on GPIO %d ready\n", IR_PIN);
    ir_poll_loop(&leds);
}

```

The `simple_led_ctrl_t` type carries a Doxygen block in `led.h`; notice the struct fields (`led1_pin ... led3_state`) are visible in `ir_getkey`'s callers but the types stay in the header, exactly the WEEK 11 lesson about structs being flattened by the compiler (see the `led_handle_ir` disassembly in Part 16 for the flattened, inlined version).

Part 9: Building Everything

Step 1: Configure the CMake build

From the project directory:

```

$ cd 0x0026a_custom-bootloader

$ cmake -B build -G Ninja \
  -DPICO_BOARD=pico2 \
  -DPICO_PLATFORM=rp2350 \
  -DCMAKE_BUILD_TYPE=Release \

```

```
-DPICO_SDK_PATH="$HOME/.pico-sdk/sdk/2.2.0" \  
-DPICO_TOOLCHAIN_PATH="$HOME/.pico-sdk/toolchain/14_2_Rel1"
```

Step 2: Build both images

```
$ cmake --build build
```

You should see two linked executables, no warnings:

```
[6/7] Linking CXX executable 0x0026a_app.elf  
[7/7] Linking CXX executable 0x0026a_bootloader.elf
```

Step 3: Check the artifacts

```
$ ls -l build/0x0026a_bootloader.bin build/0x0026a_app.bin  
-rwxr-xr-x 27772 0x0026a_bootloader.bin  
-rwxr-xr-x 21656 0x0026a_app.bin
```

The bootloader includes the second-stage loader and binary-info, so it is larger; the app is 21656 bytes (0x5498), which is the number baked into the manifest.

Step 4: Confirm the app really is relocated

```
$ ~/.pico-sdk/toolchain/14_2_Rel1/bin/arm-none-eabi-nm -n build/0x0026a_app.elf | grep  
-E ' _reset_handler| main$| ir_getkey| secret_print_banner'  
1002015c T _reset_handler  
10020234 T main  
100202a4 T ir_getkey  
10020568 T secret_print_banner
```

Every address is 0x1002xxxx . If you see 0x1000015c for _reset_handler , the linker script was not applied and the bootloader would jump the wrong way.

Part 10: Signing the Image on the Host

Step 5: Compute the tag and write the manifest

```
$ python3 tools/sign_firmware.py build/0x0026a_app.bin --out build/manifest.bin  
image      : build/0x0026a_app.bin (21656 bytes)  
app_base   : 0x10020000  
cmac tag   : ef041ec592f0c56727da2651d373947b  
manifest   : build/manifest.bin (4096 bytes)
```

Step 6: Verify the manifest against the image

```
$ python3 tools/sign_firmware.py 0x0026a_app.bin --verify build/manifest.bin  
computed : ef041ec592f0c56727da2651d373947b  
result    : MATCH
```

MATCH means the exact bytes in `0x0026a_app.bin` produce exactly the tag the bootloader will read from flash. Signing is not magic: it is one CMAC computation over the bytes, plus a 36-byte header written into a 4 KB sector.

Step 7: Read the manifest back byte by byte

```
$ xxd build/manifest.bin | head -3
00000000: 5746 4c42 0100 0000 0000 0210 9854 0000  WFLB.....T..
00000010: ef04 1ec5 92f0 c567 27da 2651 d373 947b  ....g'.&Q.s.{
00000020: ffff ffff ffff ffff ffff ffff ffff ffff  ....
```

Cross-reference with Part 5: magic, version 1, base `0x10020000` , size `0x5498` , then the 16-byte tag, then `0xFF` padding.

Part 11: Flashing the Three Images

Three images, three addresses. The bootloader first (it owns the system), then the manifest, then the app. If you flash the app before the manifest, the bootloader happily boots it anyway (the manifest is already in flash from a previous lab run) - order only matters on a bare board.

The whole chain in one drag (recommended)

On a bare board skip straight to one file: hold BOOTSEL, connect the Pico 2, and copy `0x0026a_full.uf2` to the `RP2350` drive. That single UF2 writes all three images at once - no OpenOCD, no second drag. Section 6.9 explains how it is built; the hardware-verified behavior is:

- BOOTSEL accepts it and unmounts the drive;
- the boot ROM writes bootloader, manifest, and app to their absolute addresses;
- on reset the bootloader verifies the manifest and jumps to the app.

The steps below are the three-image equivalent, which is what you use while iterating on a single image or when you prefer OpenOCD over the button.

A. The bootloader: two options

Option 1 - BOOTSEL (no extra tools): hold the BOOTSEL button, connect the Pico 2, copy `0x0026a_bootloader.uf2` to the drive, which lands the bootloader at `0x10000000` (or drag `0x0026a_full.uf2` to write the whole chain in one shot).

Option 2 - OpenOCD:

```
$ "$HOME/.pico-sdk/openocd/0.12.0+dev/openocd" \
-s "$HOME/.pico-sdk/openocd/0.12.0+dev/scripts" \
-f interface/cmsis-dap.cfg -f target/rp2350.cfg \
-c "adapter speed 5000" \
-c "program 0x0026a_bootloader.bin 0x10000000 verify reset exit"
```

The flags, in the repo's own words:

Flag	Meaning
<code>cmsis-dap.cfg</code>	The Debug Probe is a CMSIS-DAP v1 adapter

rp2350.cfg	RP2350 target: dual Cortex-M33 with a B0-style DAP
adapter speed	SWD clock in kHz; 5000 is a conservative write-path default
program	Write + verify (read-back compare) + reset + exit

B. The manifest (always its own sector)

```
$ "$HOME/.pico-sdk/openocd/0.12.0+dev/openocd" \
-s "$HOME/.pico-sdk/openocd/0.12.0+dev/scripts" \
-f interface/cmsis-dap.cfg -f target/rp2350.cfg \
-c "adapter speed 5000" \
-c "program manifest.bin 0x1001F000 verify reset exit"
```

C. The application (relocated target)

```
$ "$HOME/.pico-sdk/openocd/0.12.0+dev/openocd" \
-s "$HOME/.pico-sdk/openocd/0.12.0+dev/scripts" \
-f interface/cmsis-dap.cfg -f target/rp2350.cfg \
-c "adapter speed 5000" \
-c "program 0x0026a_app.bin 0x10020000 verify reset exit"
```

Each `program` invocation ends with `verify reset exit`, so OpenOCD reads the bytes back and aborts loudly on a mismatch. `verify` here is straight flash read-back - a completely different check from the CMAC the bootloader does at boot.

While you are at it, note that the repo's `flash.sh` helper always programs at `0x10000000` - convenient for the bootloader, wrong for the manifest and app. The three explicit commands above are the tutorial forms.

Part 12: Running It (First Boot, Serial Console)

Step 8: Open a serial console

Open a terminal on the Debug Probe's UART channel (or the board's USB-UART bridge if you wired one) at 115200 baud:

```
$ screen /dev/cu.usbmodem311402 115200
```

Replace the device name with whatever `ls /dev/cu.usb*` shows.

Step 9: Reset the board and watch the bootlog

Press the RESET button (or let OpenOCD's `reset` do it). After roughly two seconds you see the bootloader banner, then the app banner:

```
=== 0x0026a secure bootloader ===
signature valid, entering app at 0x10020000
app banner: SECRET: IR LINK SECURED ONLINE
IR receiver on GPIO 5 ready
```

The three lines tell you the whole story: the bootloader ran, it verified the image against the manifest (Part 10), it jumped to `0x10020000`, and inside the app `secret_print_banner()` decrypted the banner with the SDK AES and printed it (Part 8). Press keys on the remote and the `NEC` command: lines appear.

What you just watched, in one trace (the exact functions and where they live):

Line on screen	Function that printed it	Address
=== 0x0026a secure bootloader ===	boot_init	0x10000278
signature valid, entering app at ...	boot_launch	0x100002ac
app banner: SECRET...	secret_print_banner	0x10020568
IR receiver on GPIO 5 ready	main (app)	0x10020234

Part 13: The Tamper Test (Breaking the Chain)

Now do the attacker's favorite move: change one byte, keep everything else, reflash, and see what the bootloader does.

Step 10: Flip a byte in the app image (do NOT re-sign)

```
$ cp 0x0026a_app.bin rogue_app.bin
$ printf 'X' | dd of=rogue_app.bin bs=1 seek=0 count=1 conv=notrunc
```

(The byte at offset 0 is the low byte of the initial SP; any change breaks the CMAC, so the location is irrelevant for this test.)

Step 11: Flash the rogue image over the good one

Re-run the app `program` command from Part 11 but with `rogue_app.bin`. Do not touch the manifest - the tag inside it still corresponds to the *original* image.

Step 12: Reset and observe the rejection

```
=== 0x0026a secure bootloader ===
BOOT FAILED: firmware signature invalid
```

The second line is missing. The app never ran. The blue/yellow LED heartbeat on the board keeps blinking (500 ms), which is `boot_halt()`'s idle loop. No matter how many times you re-flash a modified image without a matching tag, the bootloader refuses.

Step 13: Confirm it still boots after restoring the good image

Re-flash `0x0026a_app.bin` (fresh, matching the manifest) and reset - or just drag `0x0026a_full.uf2` onto the BOOTSEL drive again to restore the whole chain in one move. The banner returns. The bootloader is doing real enforcement, not just complaining once.

Stop and think about the security statement this demonstrated:

- **Patch a byte, no re-sign -> reject.** The device cannot be modified cheaply.

- **Dump flash, read the key, re-sign -> accept.** Because the key lives in the binary (Part 7.2), a full flash dump defeats this scheme. Part 19 fixes that in a real product.

Part 14: Reverse Engineering the Bootloader with GDB (Live)

Live debugging on the Pico 2 uses the repo's two scripts: `debug-server.sh` owns the Debug Probe as a GDB server, and `pico-gdb-memory.gdb` confines GDB memory reads to valid flash/RAM so bogus register previews cannot fault SWD.

Step 14: Start the GDB server (Terminal 1)

```
$ ./debug-server.sh
```

The script resets the target and leaves it *running* (`reset run`), exposing `127.0.0.1:3333` .

Step 15: Start GDB on the bootloader (Terminal 2)

```
$ ~/.pico-sdk/toolchain/14_2_Rel1/bin/arm-none-eabi-gdb build/0x0026a_bootloader.elf
```

Connect and confine memory access:

```
(gdb) target extended-remote :3333
(gdb) source ../../pico-gdb-memory.gdb
```

```
delete mem
mem 0x10000000 0x10400000 ro
mem 0x20000000 0x20082000 rw
set mem inaccessible-by-default on
info mem
```

Step 16: Stop the running target cleanly

```
(gdb) monitor halt
(gdb) monitor reg pc
```

You will land somewhere in the bootloader or app depending on how far the resets progressed - either is fine.

Step 17: Disassemble `main` and read the boot sequence

```
(gdb) disassemble 0x10000234,+80
```

You should see the miniature version of Part 7.1:

```
0x10000234 <main>:      push    {lr}
0x10000238 <main+4>:      bl      0x10000278 <boot_init>
0x1000023e <main+12>:     bl      0x100002c8 <manifest_read>
0x10000244 <main+16>:     bl      0x100002fc <manifest_is_valid>
```

```
0x1000024c <main+24>:    bl      0x100003b0 <verify_manifest>
0x10000254 <main+32>:    bl      0x100002ac <boot_launch>
```

Note `main` does almost nothing itself - it is a sequence of calls, exactly the design we built.

Before accepting `0x10000234` as "the start", note that it is not the start of the image at all. The image begins at `0x10000000` with the Cortex-M vector table, and `main` sits `0x234` bytes in because the vector table, the default ISRs, the SDK binary-info header, the RP2350 IMAGE_DEF block, the reset handler, and `crt0` all run before it. Appendix A.2 derives that layout from first principles and proves it on the board. It is also why you will see `0x10000234` for the bootloader and `0x10020234` for the app: the same `0x234` offset, with the base shifted by `0x20000`.

Step 18: Stop inside the crypto (the payoff breakpoint)

The interesting code is `verify_manifest` at `0x100003b0` and the mbedtls front-end `aes128_cmac` at `0x1000031c`. Re-arm a hardware breakpoint and reset:

```
(gdb) hbreak *0x100003b0
(gdb) monitor reset halt
(gdb) continue
```

After 2 s (the startup pause) the core stops at the top of `verify_manifest`, with `r0` pointing at the manifest copy on the stack. Step a little and watch the argument registers the mbedtls call receives:

```
(gdb) disassemble 0x100003b0,+50
(gdb) x/4bx 0x10006064          # low nibbles of the key? no: dump the key
(gdb) x/16bx 0x10006064         # the 16-byte fw_key from .rodata
0x10006064: 49 72 6f 6e 4e 65 74 42 6f 6f 74 4b 65 79 21 21
```

`0x49 0x72 0x6f 0x6e ...` is "IronNetBootKey!!". You just pulled the boot key out of live flash with three GDB commands. (Refresh: the key's virtual address is `0x10006064`; `nm` reports it as `10006064 r fw_key`.)

Step 19: Watch the verdict

Let it run to the compare:

```
(gdb) hbreak *0x100003d0          # the bl cmac_tags_equal
(gdb) continue
```

`cmac_tags_equal` compares the freshly computed tag (on the stack) against the 16 bytes at offset +16 of the manifest (which came from flash). Step over it (`next`) and check the return:

```
(gdb) next
(gdb) info registers r0
```

`r0 == 1` means the tags matched and `boot_launch` is next.

Step 20: Single-step the application jump

Set a breakpoint at the tail of `image_start` and keep going:

```
(gdb) break *0x100003ae
(gdb) continue
```

At `0x100003ae` the core is one `bx r3` away from the app. Inspect what it is about to execute:

```
(gdb) info registers r3
r3 0x1002015d
(gdb) x/4bx 0x10020000      # the app's vector table
0x10020000: 00 20 08 20 5d 01 02 10    # SP=0x20082000, reset=0x1002015d
```

`r3` and `vectors[1]` agree. Step into the jump:

```
(gdb) stepi
```

Watch the PC: it is now `0x1002015c` (the unencoded reset-vector address of `_reset_handler`) and the stack pointer is `0x20082000`. The bootloader is finished; the application startup code owns the machine.

Step 21: Re-verify from the application's side

Quit and relaunch GDB against the app ELF (the core is already running it, so no reset needed - reattach so symbols load):

```
(gdb) file build/0x0026a_app.elf
(gdb) target extended-remote :3333
(gdb) hbreak *0x10020234      # app main
(gdb) monitor reset halt
(gdb) continue
```

When you hit `main`, the banner has already been decrypted and printed (it runs before `main`). To watch the AES itself, break at `secret_print_banner` instead:

```
(gdb) hbreak *0x10020568
(gdb) monitor reset halt
(gdb) continue
```

Then single-step the four `mbedtls` calls from Part 8.1. At `0x10020598` you can read the key argument:

```
(gdb) disassemble 0x10020568,+110
(gdb) x/16bx 0x10024e6c      # banner_key in .rodata
```

`0x24e6c` is the offset of the key inside the app image (Part 16/17 confirms by Ghidra). You are reading the same constant three different ways: C source, GDB, and now Ghidra.

Part 15: Reverse Engineering the Bootloader in Ghidra

Step 22: Import the ELF (not the raw .bin)

Ghidra will happily import either, but the **ELF** brings symbols, section names, and types. For a stripped factory image you would import the raw `.bin` as a "Raw Binary" at base `0x10000000`; for this lab, use the ELF and check the green "found symbols" messages after auto-analysis.

1. **File > New Project** (non-shared),

2. **File > Import File**, select `build/0x0026a_bootloader.elf`,
3. Language: confirm `ARM LE:LE32:v8M / Cortex-M33` view,
4. **Analyze** (auto-analysis defaults are fine).

Step 23: Find the interesting functions

Open **Symbol Tree > Functions**:

Function	Address	What you will see
main	0x10000234	Five calls, no logic of its own
boot_init	0x10000278	stdio_init_all, a wait, a puts
verify_manifest	0x100003b0	the 16-byte key push + aes128_cmac + compare
aes128_cmac	0x1000031c	mbdctl_cipher_info_from_type + mbdctl_cipher_cmac
image_start	0x10000394	the VTOR/MSP/bx block from Part 7.4

Double-click `verify_manifest`. Ghidra's decompiler shows something close to our C - including the constant table it pulled from `.rodata`. Open the **Data** panel and navigate to `0x10006064`: Ghidra lists a 16-byte array. It is the key. Rename it by selecting and pressing `L` (e.g. `fw_key`), then annotate `main`'s call graph with the whole boot sequence.

Do not read the bare addresses in that table as arbitrary. `main` at `0x10000234` is `0x234` bytes into the image because the vector table and the SDK startup preamble occupy everything ahead of it, and the app's `main` is the same offset at `0x10020234`. Appendix A.2 derives this from first principles and proves it with a live GDB session against the board.

Step 24: Recover the "spec" from the image

- List exports/imports: the binary imports `cdns_usb_table`, stdio routines, and - from `.dynamic` or relocation - nothing surprising for an SDK binary.
- Note `binary_info` header: the SDK stores `program_name` (`0x0026a_bootloader`) and build date inside the image at `0x10000100`; this is exactly what `picotool info` prints and a useful fingerprint for hunting unknown binaries in the wild.

Part 16: Reverse Engineering the Application in Ghidra (Find the AES)

The app is the richer target. Repeating Part 15 step 22 with `build/0x0026a_app.elf` gives you the full symbol table, because we shipped the ELF. On a real malicious/hardened build you would import `0x0026a_app.bin` at `0x10020000` (base from `manifest.bin`) and rely on strings.

Step 25: The easy tour (symbols)

Symbol	Address	Observations
<code>_reset_handler</code>	<code>0x1002015c</code>	first code; assumes SP/VTOR set (our bootloader does)
<code>main</code>	<code>0x10020234</code>	the thin orchestrator from Part 8.5
<code>secret_print_banner</code>	<code>0x10020568</code>	four mbdctl calls; the AES gate

led_handle_ir	0x100204d8	flattened struct logic (see below)
---------------	------------	------------------------------------

Open `main` . Because everything was inlined and optimized, Ghidra shows a clean calling skeleton - the same one GDB showed in Part 14 Step 21.

Compare the addresses with the bootloader's from Step 23 and you will see the relocation for yourself: `_reset_handler` moved from `0x1000015c` to `0x1002015c` , and `main` from `0x10000234` to `0x10020234` . Every symbol is the bootloader's plus `0x20000` ; only the base changed. Appendix A.2 explains and proves why both images put `main` `0x234` bytes in.

Step 26: The "struct is a lie" lesson, revisited

Open `led_handle_ir` at `0x100204d8` . Ghidra's decompiler will *not* show you `simple_led_ctrl_t` fields by name (the type was flattened at compile time, WEEK 11 Part 10). Instead you see direct byte loads off the pointer and the constant compares `0xc` , `0x18` , `0x5e` - the NEC commands. The three `mccr` / `strb` groups are the `gpio_put` / `gpio_set_dir` calls inlined from the LED module. Same behavior, zero struct abstraction, single program counter.

Step 27: Find the four AES calls

Open `secret_print_banner` . The decompiler names the imported mbedtls calls directly:

```
mbedtls_aes_init(ctx)
mbedtls_aes_setkey_dec(ctx, key, 128)
mbedtls_aes_crypt_cbc(ctx, MBEDTLS_AES_DECRYPT, 32, iv, ct_buf, ct_buf)
mbedtls_aes_free(ctx)
```

The red referenced addresses are:

Data	Address (VA)	Bytes	ASCII
banner ciphertext	0x10024d78	ba ec 4a 19 ... 7b b2 e7 6e	(binary)
banner IV	0x10024d9c	41 70 70 53 65 63 72 65 74 49 56 31 32 33 34 35	AppSecretIV12345
app banner: %s	0x10024d68	61 70 70 20 62 61 6e 6e 65 72 3a 20 25 73 0a	app banner: %s\n
banner key	0x10024e6c	49 72 6f 6e 4e 65 74 42 6f 6f 74 4b 65 79 21 21	IronNetBootKey!!

Step 28: Decrypt it with what you found

Ghidra hands you plain values; OpenSSL decrypts for you - you do not need to recover the source:

```
$ printf 'baec4a1925861bb92dfd9f13f79b3e80cf12b698db9165be8fd7c3477bb2e76e' \
| xxd -r -p \
| openssl enc -d -aes-128-cbc \
-K 49726f6e4e6574426f6f744b65792121 \
```

```
-iv 41707053656372657449563132333435 -nopad
SECRET: IR LINK SECURED ONLINE
```

Because the key is 70 bytes from its ciphertext in the same ELF, "breaking" the app took: import, click, copy three hex constants, run one command. The whole exercise is about **finding key material**, never about cracking AES.

Step 29: Now repeat it on the stripped .bin (harder, real-world)

Delete the app from the project, import `0x0026a_app.bin` with **Language = ARM LE v8M** and **Image base = 0x10020000**. No symbols. Now your anchors are:

- The string `app banner: %s` (String search, `Ctrl+Shift+S`), which xrefs to the second half of `secret_print_banner`.
- The 16 bytes `AppSecretIV12345` nearby in `.rodata`.
- The constant `0x48564d411f...` = `AppSecretIV12345` pattern as 4 little endian words (this is your Ghidra jump-start trick: search for the hex of an ASCII IV).

Renaming data (`L`) and entering type views as you go will reconstruct the module structure from the stripped blob exactly as if a debugger had been kind.

Part 17: A Bootloader You Can Read (Reference Address Map)

Everything you should have collected so far, in one table. Builds are deterministic for the checked-in sources, so your numbers should match.

Read every address as `base + offset`. The bootloader is based at `0x10000000` and the app at `0x10020000`, so `main` is `0x10000234` in one and `0x10020234` in the other - the same `0x234` offset, relocated. Appendix A.2 derives that offset from the vector table and the SDK startup code and proves it on the board.

Bootloader (0x0026a_bootloader.elf)

Role	Symbol	Address
Reset vector (vector[1])	<code>_reset_handler</code>	<code>0x1000015c</code>
Initial SP (vector[0])	(data)	<code>0x20082000</code>
Boot sequence	<code>main</code>	<code>0x10000234</code>
Console + 2 s pause	<code>boot_init</code>	<code>0x10000278</code>
Failure idle loop	<code>boot_halt</code>	<code>0x10000294</code>
Success handoff	<code>boot_launch</code>	<code>0x100002ac</code>
Flash manifest read	<code>manifest_read</code>	<code>0x100002c8</code>
Magic/version check	<code>manifest_is_valid</code>	<code>0x100002fc</code>
mbedtls CMAC front-end	<code>aes128_cmac</code>	<code>0x1000031c</code>
Constant-time compare	<code>cmac_tags_equal</code>	<code>0x10000348</code>

VTOR/MSP/bx jump	image_start	0x10000394
CMAC verdict	verify_manifest	0x100003b0
Boot key (rodata)	fw_key	0x10006064

Application (0x0026a_app.elf)

Role	Symbol	Address
App reset vector	_reset_handler	0x1002015c
App entry point	main	0x10020234
NEC decoder init	ir_init	0x1002027c
NEC blocking decode	ir_getkey	0x100202a4
LED default config	led_default	0x10020498
LED GPIO setup	led_init	0x100204a8
IR -> LED dispatch	led_handle_ir	0x100204d8
AES banner decrypt	secret_print_banner	0x10020568
String app banner: %s	(rodata)	0x10024d68
Banner ciphertext	(rodata)	0x10024d78
Banner IV	(rodata)	0x10024d9c
Banner key	(rodata)	0x10024e6c
Initial SP (vector[0])	(data)	0x20082000

Part 18: Summary and Review

What You Built

+-----+	+-----+	+-----+
ROM	BOOTLOADER	APPLICATION
(silicon) -->	@0x10000000 -->	@0x10020000
+-----+	+-----+	+-----+

Enforcement point: `verify_manifest` recomputes `AES_CMAC(fw_key, app_base, app_size)` and compares to the stored tag. On match, `image_start` sets VTOR, loads SP, and branches to the app reset vector. On mismatch, `boot_halt` reports and idles. One flipped byte in the app and the whole chain refuses.

Key Lessons

1. **Linker scripts place your program.** `MEMORY /ORIGIN/LENGTH` + a changed `FLASH` line moved the app 128 KiB up. `__StackTop = 0x20082000` in the vector table is direct evidence.

2. **Modules keep firmware readable.** Both `main.c` files are thin; crypto, manifest, IR, LED, and banner live in one-responsibility `.c / .h` pairs, every prototype Doxygen-documented, nothing past 8 executable lines.
 3. **AES is everywhere in SDK firmware.** `pico_mbedtls` gives you a tuned AES for free. You used it both to authenticate (bootloader CMAC) and to hide a string (app CBC banner).
 4. **CBC with no padding** encrypts 32 bytes as two blocks; IV is read/updated in place - give `crypt_cbc` a mutable copy.
 5. **Static keys are secrets-in-name only.** SWD/GDB/Ghidra read the key from flash (`0x10006064`) in minutes. Real products store root keys in OTP and verify with the ROM (Part 19).
 6. **Reverse engineering = treasure hunt.** You never "break" AES; you find the key, IV, and ciphertext and press decrypt.
-

Part 19: Your Turn (Hacking Exercises)

Try these on your own board; each one is a small, safe extension.

Challenge 1 - Swap two LED pins without touching the source

The PINs are value-initialized data in `led_default` (the struct fields the compiler stored in `.rodata`). Find the address in the stripped app `.bin` (Ghidra: `led_default` xrefs the literals 16, 17, 18), patch the 16 and 18 literals, re-flash the *unsigned* image, and watch the bootloader reject it - then re-sign with `sign_firmware.py` and this time it boots with the hack.

Challenge 2 - Change the boot key (both sides)

Edit the 16 bytes in `verify.c` , rebuild only the bootloader, re-sign the app with `--key` matching, flash all three, and boot. Then sign the app with the OLD key and confirm rejection. You have now installed your own signing authority on your own board.

Challenge 3 - Make the banner say something else

Change the plaintext (must stay 32 bytes for the no-padding CBC), re-encrypt with `openssl` , update `banner_ct` in `secret.c` , rebuild, re-sign, flash. Your Ghidra notes from Part 16 now decode the new banner id to differentiate it.

Challenge 4 - Think through the upgrade path

The manifest has a `version` field that is currently ignored after a magic check. Sketch how `boot_launch` could validate a counter ("only downgrade a fixed number of times") - the classic anti-rollback feature that separates consumer OTA from device management-grade OTA. You should be able to name the flash sectors you would need to reserve to make that safe (hint: state must survive app updates, so it cannot live in the app image).

Challenge 5 - The honest conversation about the key

Write one paragraph each for a product manager: (1) why AES-CMAC with a flash key stops a script-kiddie flash overwrite, and (2) why it does not stop a determined hardware attacker, and (3) which real mechanism (OTP key, ROM verifier with `secp256k1` , anti-dumping measurements) closes that hole. If you can explain all three, you understand secure boot at a professional level.

Key Takeaways

- Authenticated boot = each stage refuses to start the next until a secret join the image to a tag proves it is the right bytes.
- AES-128-CMAC (RFC 4493) is a keyed CBC-MAC with two subkeys; unforgeable without the key, verifiable with a few lines of mbedtls.
- `manifest.bin` is a 4096-byte sector at `0x1001F000` ; a 16-byte tag is the only secret-bearing field.
- The app lives at `0x10020000` because `memmap_app.ld` says so; the vector table's `0x20082000` stack pointer proves it.
- The SDK bundles Mbed TLS; `mbedtls_aes_*` decrypts at runtime with zero third-party code.
- OpenOCD `program <file> <addr> verify reset exit` writes each partition; flash read-back `verify` is not authentication.
- GDB `hbreak / stepi / x/16bx` on a live Pico 2 shows you key bytes and the exact `bx` that enters the app.
- Ghidra on the ELF names the AES calls; Ghidra on the .bin finds them via the constant table and IV string.
- A key dumped from flash is not a secret. OTP + ROM verification is.

Glossary

Term	Definition
Boot chain	Ordered stages where each stage authenticates the next.
XIP	Execute-in-place: code runs straight from flash memory-mapped at <code>0x10000000</code> .
CMAC	Keyed message-authentication code based on a block cipher (NIST CMAC / RFC 4493).
CBC-MAC	MAC built by chaining AES-CBC blocks; needs the final-block fix CMAC adds.
Tag	The short authenticator output (16 bytes here) proving message integrity.
Manifest	Small signed record describing an image: address, size, tag.
VTOR	Vector Table Offset Register; tells the CPU where the vector table is.
MagnumOpus issue	A willful misunderstanding source: see Part 18.
OTP	One-time-programmable memory; writes once, reads forever, cannot be re-flashed.
RO/RW/HAL	read only / read write / hardware abstraction layer, the three concerns a linker script separates.

Additional Resources

RFC and Standard References

- RFC 4493 - The AES-CMAC Algorithm (the subkey derivation and four vectors used here)

- NIST SP 800-38B - Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication

In the Repo

- `tools/sign_firmware.py` - the pure-Python signer/verifier (read the header comment for a CMAC walkthrough)
- `common/include/mbedtls_config.h` - the minimal SDK mbedtls config (AES_C, CBC, CMAC_C)

Pico SDK Internals (open these while you read Part 6)

- `~/pico-sdk/sdk/2.2.0/src/rp2_common/pico crt0/rp2350/memmap_default.ld` - the SDK linker script we copied
- `~/pico-sdk/sdk/2.2.0/src/rp2_common/pico_standard_link/` - crumbs for flash region generation
- `~/pico-sdk/sdk/2.2.0/src/rp2_common/pico crt0/crt0.S` - `_reset_handler` , `_entry_point` , and the vector table layout

Tooling Used in This Lab

- OpenOCD from the SDK (`~/pico-sdk/openocd/0.12.0+dev`)
 - GDB / objdump / nm from the ARM GNU toolchain (`~/pico-sdk/toolchain/14_2_Rel1`)
 - Ghidra (import both ELF and raw .bin; see Parts 15-16)
 - `openssl enc -aes-128-cbc -nopad` for host-side banner cryptography
-

Real-World Implications

What You've Learned in This Course

Your Pico 2 is now a small secure-boot platform: ROM -> bootloader -> manifest -> app, with an authenticated handoff at every boundary you control. The same skills invert perfectly for defense: when you audit a device, this is the mental model you bring - *what authenticates the next stage, where is the key, and what happens if it is not there?*

The Power and Responsibility

- The **power**: you can now build boot chains, move firmware around the address space at will, extract keys, and re-sign images. That is the exact toolbox used both to protect smart locks and to compromise them.
- The **responsibility**: only ever apply these techniques to devices you own 100% or are explicitly authorized to test. The moment you run `B00T FAILED` on someone else's device, you have crossed into criminal tampering and possibly a safety-critical product.
- The **engineering honesty**: a flash key is a speed bump, not a wall. Any product story that claims "AES secure boot" must be asked, *where is the key, and who can read it?* If the answer is "in the firmware," the whiteboard does not add up.

Keep Learning

- Swap the CMAC for the RP2350 ROM `secp256k1` verifier to get real asymmetric trust (flash dumps stop being useful).
- Use `pico_set_uf2_...` on a real board to ship an OTA payload that your own bootloader authenticates before the boot stage writes it.
- Read how real device IDs (RP2350's unique serial) can be mixed into the key schedule so an image signed for one board will not run on another.

Now go flash your own CA. You control the manifest, the key, and the gate that decides what runs on your hardware.

Appendix A: End-to-End Reverse Engineering with GDB and Ghidra

Parts 14-16 teach the two tools one at a time. This appendix is the consolidated, start-to-finish version: a single adversary workflow that starts with nothing but the shipped artifacts and a board, and ends with a forged image the bootloader accepts. Every command is meant to be run in order. It is the "give me the whole hunt" version - keep a terminal and a Ghidra window open side by side.

For short paths throughout, set:

```
$ export W=0x0026a_custom-bootloader
$ export GDB=~/.pico-sdk/toolchain/14_2_Rel1/bin/arm-none-eabi-gdb
$ export PCT=~/.pico-sdk/picotool/2.3.1/picotool/picotool
```

A.1 What the attacker starts with

Assume you were handed the release folder and a Debug Probe, with no source and no README. Each artifact leaks a specific thing:

Artifact	What it gives you
0x0026a_bootloader.elf	Full symbols for the gate: function names + addresses
0x0026a_bootloader.bin	The same code with no symbols (the real-world case)
manifest.bin	The on-flash spec: magic, base, size, and the tag
0x0026a_app.bin	The signed payload; its .rodata hides the AES key
0x0026a_full.uf2	The one-drag installer (bootloader + manifest + app)

The trust boundary is a single comparison: a freshly computed AES-128-CMAC of the app bytes versus the 16-byte `tag` in the manifest. Everything below is about recovering the inputs to that comparison, then supplying your own.

A.2 The image layout: from the vector table to main

Read a Pico image at the byte level and know, with nothing assumed, why `main` is at `0x234` and why the relocated app's `main` is at `0x10020234`. Every number below is confirmed on the live board in A.2.5.

A.2.1 The rule: an address is a base plus an offset

An image is linked at a base address: `0x10000000` for the bootloader, `0x10020000` for the app. The linker places byte 0 of the program at the base, so every address is `base + offset`:

```
0x10000234 = 0x10000000 + 0x234
```

`0x234` is hexadecimal (564 decimal). It is not a whole address and it is not "the 234th byte"; it is the distance from the start of the image.

A.2.2 The first bytes are a vector table, not main

The Cortex-M33 does not reset into `main` . The hardware requires a vector table at the image base and reads two words from it to start: word 0 is the initial stack pointer, word 1 is the reset handler. The RP2350 has 16 system vectors plus 52 IRQs, so the table is 68 words = 272 bytes = `0x110` . Live dump from this board:

```
(gdb) x/16wx 0x10000000
0x10000000 <__vectors>:    0x20082000 0x1000015d 0x1000011b 0x1000011d
0x10000010 <__vectors+16>: 0x10000111 0x10000111 0x10000111 0x10000111
```

Word	Value	Meaning
0	0x20082000	initial stack pointer (top of RP2350 SRAM)
1	0x1000015d	reset handler = <code>_reset_handler</code> (0x1000015c) + Thumb bit
2	0x1000011b	NMI
3	0x1000011d	HardFault
4	0x10000111	MemManage
...	...	remaining system vectors, then IRQ0..IRQ51

Word 1 ends in `...5d` , not `...5c` , because Cortex-M code is Thumb code: the hardware uses that low bit and branches to `0x1000015c` . GDB resolves it:

```
(gdb) info symbol 0x1000015d
_reset_handler + 1 in section .text
```

Two entry numbers exist and are not the same:

- the reset vector (table word 1) = `_reset_handler` at `0x1000015c` ;
- the ELF entry point (`readelf -h`) = `0x1000014d` = `_entry_point` at `0x1000014c` , an SDK trampoline that reloads SP and branches to `_reset_handler` .

Both converge on `_reset_handler` . Neither is `main` .

A.2.3 The rest of the preamble and the arithmetic

After the table, this fixed preamble runs before `main` :

Address	Symbol	Purpose
0x10000110	<code>__default_isrs_start</code>	weak ISR defaults (<code>isr_irq0..isr_irq51</code>)
0x10000124	<code>binary_info_header</code>	name/build date shown by <code>picotool info</code>
0x10000138	<code>embedded_block</code> (IMAGE_DEF)	RP2350 metadata block the ROM reads
0x1000014c	<code>_entry_point</code>	trampoline
0x10000150	<code>_enter_vtable_in_r0</code>	msr MSP then bx to the reset handler
0x1000015c	<code>_reset_handler</code>	reset entry
0x10000164	<code>hold_non_core0_in_bootrom</code>	parks core 1

0x10000180	bss_fill_loop / data_cpy	zero .bss, copy .data
0x100001e4	_init	runtime / constructors
0x10000234	main	your code

The sizes add up exactly:

```

vector table      0x110   (68 words)
default ISRs     0x014
binary_info_header 0x014
embedded IMAGE_DEF 0x014
-----
fixed preamble   0x14c   ends at 0x1000014c
reset handler + crt0 0x0e8   fills 0x1000014c .. 0x10000234
-----
main              0x234

```

So `main` is at `0x10000000 + 0x234` : 564 bytes in, because the hardware tables and the C runtime occupy the first 564 bytes. This is the SDK layout for every image; it is not specific to a bootloader.

The live disassembly of the trampoline and reset handler shows the handoff:

```

(gdb) disassemble 0x1000014c,+56
0x1000014c <_entry_point+0>:      mov.w r0, #0
0x10000150 <_enter_vtable_in_r0+0>: ldr    r1, [pc, #120]
...
0x10000154 <_enter_vtable_in_r0+4>: ldmia r0!, {r1, r2}
0x10000156 <_enter_vtable_in_r0+6>: msr    MSP, r1
0x1000015a <_enter_vtable_in_r0+10>: bx    r2
0x1000015c <_reset_handler+0>:      mov.w r0, #3489660928
...
0x10000172 <hold_non_core0_in_bootrom+14>: bl    0x1000019a <data_cpy>
...
0x10000180 <bss_fill_loop+0>:      stmia r1!, {r0}

```

`_enter_vtable_in_r0` loads the initial stack pointer into MSP (`msr MSP, r1`) and branches to the reset handler (`bx r2`); `_reset_handler` then parks core 1, copies `.data` , zeroes `.bss` , runs `_init` , and calls `main` .

A.2.4 Relocation: the same layout, base shifted

The app is the same image linked at `0x10020000` , so every startup symbol shifts by `0x20000` :

Symbol	Bootloader	App
__VECTOR_TABLE	0x10000000	0x10020000
__vectors_end	0x10000110	0x10020110
_entry_point	0x1000014c	0x1002014c
_reset_handler	0x1000015c	0x1002015c

main	0x10000234	0x10020234
------	------------	------------

main is at offset 0x234 in both. Relocation changed the base only, which is why the bootloader can jump to an app linked at a different address.

A.2.5 Reproduce and verify

With the probe attached and ./debug-server.sh running, this command proves every claim above. The output is verbatim from this lab's board:

```
$ ~/.pico-sdk/toolchain/14_2_Rel1/bin/arm-none-eabi-gdb -batch \
  -ex "file 0x0026a_custom-bootloader/build/0x0026a_bootloader.elf" \
  -ex "target extended-remote :3333" \
  -ex "source pico-gdb-memory.gdb" \
  -ex "monitor halt" \
  -ex "x/16wx 0x10000000" \
  -ex "info symbol *(unsigned int*)0x10000004" \
  -ex "info address main" \
  -ex "x/4wx 0x10020000"

0x10000000 <__vectors>:    0x20082000 0x1000015d 0x1000011b 0x1000011d
0x10000010 <__vectors+16>: 0x10000111 0x10000111 0x10000111 0x10000111
...
_reset_handler + 1 in section .text
Symbol "main" is a function at address 0x10000234.
0x10020000: 0x20082000 0x1002015d 0x1002011b 0x1002011d
```

What each line proves:

- 0x10000000 = 0x20082000 then 0x1000015d : the image begins with [initial SP][reset handler] .
- info symbol resolves the reset word to _reset_handler .
- info address main = 0x10000234 , 564 bytes in.
- 0x10020000 repeats the same shape shifted by 0x20000 ; the app's main is 0x10020234 .
- Attaching mid-run showed the PC at 0x100202cc , inside the app region: the bootloader had already handed off.

A.2.6 Two gotchas

- 0x234 is hex; 0x234 = 564 decimal.
- No boot2 blob: __boot2_start__ == __boot2_end__ in both images, so the image begins at the vector table (the RP2350 boot ROM handles flash setup).

A.3 Static reconnaissance (before any disassembler)

Never open Ghidra first. Two minutes of file / strings / xxd frames the whole investigation:

```
$ file $W/0x0026a_bootloader.elf $W/0x0026a_bootloader.bin $W/0x0026a_app.bin
$ $PCT info -a $W/0x0026a_full.uf2
$ arm-none-eabi-size $W/build/0x0026a_bootloader.elf $W/build/0x0026a_app.elf
$ shasum -a 256 $W/0x0026a_app.bin
$ strings -n 6 $W/0x0026a_app.bin | sort -u | head -40
$ xxd -l 64 $W/manifest.bin
```

What each answers:

- `file` says ARM little-endian, and `picotool info` names the family (`rp2350-arm-s`) and the target - so you know the ISA and the load address.
- `arm-none-eabi-size` prints the ELF `text / data / bss` totals, and `wc -c` on `0x0026a_app.bin` reports 21656 bytes = `0x5498` , which is exactly `app_size` in the manifest.
Two artifacts agree, so the flash map is confirmed without a single line of source.
- `shasum` gives you a fingerprint to prove a re-signed image actually changed.
- `strings` shows `app banner: %s , %s` printf formats, and NUL-split fragments. Note them as search anchors for Ghidra.
- `xxd manifest.bin` ends the "what is the format" question immediately:

```
00000000: 5746 4c42 0100 0000 0000 0210 9854 0000  WFLB.....T..  
00000010: ef04 1ec5 92f0 c567 27da 2651 d373 947b  ....g'&Q.s.{
```

Decode it as five little-endian words plus a blob:

Offset	Bytes	Meaning
0x00	57 46 4c 42	magic WFLB (0x424C4657 little-endian)
0x04	01 00 00 00	version 1
0x08	00 00 02 10	app base 0x10020000
0x0C	98 54 00 00	app size 0x5498 (21656)
0x10	16 bytes	ef04...947b, the AES-CMAC tag
0x20	ff x 4064	erased sector padding

You have the manifest spec before opening a decompiler. `binwalk -E` (if installed) confirms a low-entropy, uncompressed image - none of it is packed.

A.4 Ghidra on the bootloader (the easy path)

1. **File > New Project** (non-shared), then **File > Import File** and pick `$W/build/0x0026a_bootloader.elf` .
2. Accept the detected language `ARM:LE:32:v8` (Cortex-M33). Click **Analyze** and take the defaults.
3. In **Symbol Tree > Functions**, confirm the boot sequence by address:

Function	Address	Reads as
<code>main</code>	<code>0x10000234</code>	five BLS, no logic of its own
<code>boot_init</code>	<code>0x10000278</code>	<code>stdio_init_all</code> , a wait, one puts
<code>manifest_read</code>	<code>0x100002c8</code>	memcpy of 32 bytes from <code>0x1001F000</code>
<code>manifest_is_valid</code>	<code>0x100002fc</code>	two compares: magic and version
<code>verify_manifest</code>	<code>0x100003b0</code>	the 16-byte key push + CMAC + tag compare
<code>aes128_cmac</code>	<code>0x1000031c</code>	<code>mbedtlsls cipher_info_from_type</code> + <code>cipher_cmac</code>

cmac_tags_equal	0x10000348	constant-time 16-byte compare
image_start	0x10000394	VTOR / MSP / bx launch block
boot_halt	0x10000294	the B00T FAILED dead end

Double-click `verify_manifest`. The decompiler will show a stack copy of the manifest and a call that hands `fw_key` to `aes128_cmac`, then a 16-byte compare with the tag. That is the entire security model on one screen.

A.5 Recovering the key from the image (static)

Inside `verify_manifest`, note the address pushed as the CMAC key. Follow it in the **Data** panel to `0x10006064`, or just ask the binary:

```
$ arm-none-eabi-nm -n $W/build/0x0026a_bootloader.elf | grep fw_key
10006064 r fw_key
```

Rename the data (`L`) to `fw_key`, then read it as bytes in Ghidra - it is `49 72 6f 6e 4e 65 74 42 6f 6f 74 4b 65 79 21 21`, which is ASCII `IronNetBootKey!!`. That is the AES-128-CMAC key. There was no cryptography to break: the key was in `.rodata` the whole time.

Cross-check the finding without the board. The signer implements AES and CMAC in pure Python, so the four official RFC 4493 vectors are a self-test you can read and reproduce (the `Key = / Tag =` block in Part 3 is your checklist). If your recovered key reproduces the shipped tag, you have proven both the key and the algorithm:

```
$ python3 $W/tools/sign_firmware.py $W/0x0026a_app.bin \
  --verify $W/manifest.bin \
  --key 49726f6e4e6574426f6f744b65792121
stored   : ef041ec592f0c56727da2651d373947b
computed : ef041ec592f0c56727da2651d373947b
result   : MATCH
```

A.6 Ghidra on the application (the AES gate)

Repeat A.4 with `$W/build/0x0026a_app.elf`. The app is the richer target.

Symbol	Address	Observations
<code>_reset_handler</code>	<code>0x1002015c</code>	first code; trusts the SP/VTOR the bootloader set
<code>main</code>	<code>0x10020234</code>	thin orchestrator
<code>ir_init / ir_getkey</code>	<code>0x1002027c / 0x100202a4</code>	NEC receiver setup
<code>led_handle_ir</code>	<code>0x100204d8</code>	flattened struct logic; compares <code>0xc</code> , <code>0x18</code> , <code>0x5e</code>
<code>secret_print_banner</code>	<code>0x10020568</code>	four mbedtls calls; the banner decrypt

Open `secret_print_banner`. The decompiler names the imported calls directly:

```
mbedtls_aes_init(ctx)
mbedtls_aes_setkey_dec(ctx, key, 128)
```

```
mbdctl_aes_crypt_cbc(ctx, MBEDTLS_AES_DECRYPT, 32, iv, ct, ct)
mbdctl_aes_free(ctx)
```

The red data references give you every constant you need:

Data	Address (VA)	Bytes	ASCII
banner format	0x10024d68	61 70 70 20 62 61 6e 6e 65 72 3a 20 25 73 0a 00	app banner: %s\n
banner ciphertext	0x10024d78	ba ec 4a 19 ... 7b b2 e7 6e (32 bytes)	(binary)
banner IV	0x10024d9c	41 70 70 53...31 32 33 34 35	AppSecretIV12345
banner key	0x10024e6c	49 72 6f 6e...21 21	IronNetBootKey!!

Note the last row: the AES key is identical to the CMAC key. One compromise, the whole chain falls - the classic "one key, two jobs" smell.

A.7 Decrypt on the host

Ciphertext, IV, and key are now just three hex strings. OpenSSL finishes the job; you never needed the source:

```
$ printf 'baec4a1925861bb92dfd9f13f79b3e80cf12b698db9165be8fd7c3477bb2e76e' \
| xxd -r -p \
| openssl enc -d -aes-128-cbc \
-K 49726f6e4e6574426f6f744b65792121 \
-iv 41707053656372657449563132333435 -nopad
SECRET: IR LINK SECURED ONLINE
```

Breaking the application took: import, click, copy three hex constants, run one command. AES was never attacked - the key schedule was simply lying in `.rodata`.

A.8 Forging an image the bootloader accepts

The recovery is only interesting if you can act on it. Because you own the key, you can sign anything:

1. Tamper the app (any byte, e.g. flip one in `0x0026a_app.bin`).
2. Re-sign the tampered image, producing a new manifest, with the recovered key:

```
$ python3 $W/tools/sign_firmware.py $W/0x0026a_app.bin \
--out $W/manifest.bin \
--key 49726f6e4e6574426f6f744b65792121 \
--app-base 0x10020000
$ python3 $W/tools/sign_firmware.py $W/0x0026a_app.bin \
--verify $W/manifest.bin --key 49726f6e4e6574426f6f744b65792121
```

3. Flash the tampered app at `0x10020000` and the new manifest at `0x1001F000` (Part 11). The bootloader computes the same CMAC over the tampered bytes, the tag matches, and the rogue

image runs. That is the whole "secure boot" defeated with a key you read out of the image it was protecting.

A.9 Live GDB: watching the boot gate

Now do the same hunt at runtime. The repo ships `debug-server.sh` (owns the Probe, leaves the target running on `127.0.0.1:3333`) and `pico-gdb-memory.gdb` (confines GDB reads to valid flash/RAM so register previews cannot fault SWD).

Terminal 1:

```
$ ./debug-server.sh
```

Terminal 2:

```
$ $GDB $W/build/0x0026a_bootloader.elf
(gdb) target extended-remote :3333
(gdb) source pico-gdb-memory.gdb
(gdb) monitor halt
```

Read the manifest straight out of flash and confirm the A.3 decode at runtime:

```
(gdb) x/4wx 0x1001F000
0x1001f000: 0x424c4657 0x00000001 0x10020000 0x00005498
(gdb) x/16bx 0x1001F010
0x1001f010: 0xef 0x04 0x1e 0xc5 0x92 0xf0 0xc5 0x67 ...
```

Break on the gate, reset, and continue:

```
(gdb) hbreak *0x100003b0
(gdb) monitor reset halt
(gdb) continue
```

When it stops, dump the key argument the CMAC call will receive - the live version of A.5:

```
(gdb) x/16bx 0x10006064
0x10060064: 0x49 0x72 0x6f 0x6e 0x4e 0x65 0x74 0x42 ...
```

`0x49 0x72 0x6f 0x6e ...` is `IronNetBootKey!!` again. Then watch the verdict:

```
(gdb) finish
(gdb) info registers r0
```

`r0 == 1` means the tags matched and `boot_launch` is next. Set a breakpoint at the launch tail and single-step the handoff:

```
(gdb) break *0x100003ae
(gdb) continue
(gdb) info registers r3
(gdb) x/2wx 0x10020000
(gdb) stepi
```

`r3` equals `vectors[1]` , and after `stepi` the PC is in the app's `_reset_handler` with SP already set to `0x20082000` . The gate is behind you.

A.10 Live GDB: the application's AES

Reattach to the app ELF (the core is already running it, so no reset is strictly needed, but a reset proves the chain end to end):

```
(gdb) file $W/build/0x0026a_app.elf
(gdb) hbreak *0x10020568
(gdb) monitor reset halt
(gdb) continue
```

You are now at `secret_print_banner` , before the banner exists in RAM. Disassemble and read the three constants the same way Ghidra showed them:

```
(gdb) disassemble 0x10020568,+120
(gdb) x/16bx 0x10024e6c      # banner key  -> IronNetBootKey!!
(gdb) x/16bx 0x10024d9c      # banner IV   -> AppSecretIV12345
(gdb) x/32bx 0x10024d78      # banner ciphertext
```

Single-step the four `mbedtls` calls and confirm the argument registers line up with A.6 (`r0` = `ctx` , `r3` = `iv` , and the ciphertext pointer on the stack). By the time `secret_print_banner` returns, the plaintext `SECRET: IR LINK SECURED ONLINE` has been printed - the same bytes OpenSSL produced on the host, now observed live on the silicon.

A.11 The stripped binary (no symbols, the real case)

Everything above leaned on symbols. A shipped image usually has none. Delete the app from the Ghidra project and import `$W/0x0026a_app.bin` with **Language = ARM:LE:32:v8** and **Image base = 0x10020000**. Now reconstruct the anchors you already know from the easy pass:

- **Strings first.** `Ctrl+Shift+S` , search `app banner: %s` . Its only xref is the second half of `secret_print_banner` - you have found the function blind.
- **Then its neighbours.** The IV `AppSecretIV12345` sits 0x24 bytes later in `.rodata` ; the key `IronNetBootKey!!` sits at `0x10024e6c` . Search the raw bytes (`Search > For Bytes , 41 70 70 53 ...`) to jump straight to them.
- **Then the calls.** The `BL` targets into the `mbedtls` block (`mbedtls_aes_setkey_dec` at `0x10024088` , `mbedtls_aes_crypt_cbc` at `0x10024994`) confirm the four-call shape even without names.
- **Then the manifest magic.** In the bootloader `.bin` , search for `57 46 4c 42` (`WFLB`); its only compare is `manifest_is_valid` , which anchors the whole boot sequence the same way `app banner: %s` anchored the app.

Renaming data (`L`), applying types, and setting a couple of function signatures rebuilds the module structure from the raw blob - no debugger, no source, no strings beyond a `printf` format.

A.12 Cheat sheet

Task	Command / address
Start GDB server	<code>./debug-server.sh</code> then target <code>extended-remote :3333</code>

Confine memory reads	<code>source pico-gdb-memory.gdb</code>
Stop / reset target	<code>monitor halt / monitor reset halt</code>
Break on the gate	<code>hbreak *0x100003b0 (verify_manifest)</code>
Manifest in flash	<code>x/4wx 0x1001F000</code>
CMAC key in flash	<code>x/16bx 0x10006064 -> IronNetBootKey!!</code>
App banner gate	<code>hbreak *0x10020568 (secret_print_banner)</code>
Banner key / IV / ciphertext	<code>0x10024e6c / 0x10024d9c / 0x10024d78</code>
Re-sign an image	<code>python3 tools/sign_firmware.py <bin> --out manifest.bin \</code>
	<code>--key <hex> --app-base 0x10020000</code>
Verify a manifest	<code>python3 tools/sign_firmware.py <bin> --verify manifest.bin</code>

The lesson of the appendix is the lesson of the lab: the adversary never breaks AES. They read the manifest spec from a sector, lift the key from `.rodata`, and re-sign. Design your next boot chain so that the key is not in the image you are trusting it to protect.