Firmware & Hardware Security — Foundations

Concept cards — generated 2026-08-05

Contents

Track A

Module A0

Firmware anatomy, and the two firmware worlds

Why this matters

Someone hands you a 8 MB `.bin` from a vendor's download page. Before you can do anything useful with it you have to answer one question, and answering it wrong costs you an afternoon:

> **Is there a filesystem in here, or is this a single blob?**

The answer splits the entire field in two. Every tool, every methodology stage, and every technique that follows depends on which side you are on. This card is how you tell.

World one: Linux-based firmware

The router, the IP camera, the NAS. A general-purpose CPU with an MMU (Cortex-A, or MIPS), enough RAM to run a kernel, and an image built out of three recognisable parts:

The bootloader. Usually **U-Boot**. It initialises RAM and the clock, finds the kernel, and hands over. It is also a target in its own right: a U-Boot console on a serial port is frequently a full compromise, and its environment variables control what gets booted. Module E2 attacks it properly.

The kernel. Linux, usually old — a 2.6 or 3.x kernel in a device shipped last year is completely normal, and that gap is itself a finding. Often wrapped in a **uImage** header carrying the load address, entry point and a CRC.

The root filesystem. Where the interesting things live: the web interface, the CGI handlers, the service binaries, the configuration, and the credentials. Extracting it is the point of FSTM stage 4, and everything in Track B's static analysis happens inside it.

Filesystems you will meet

Read-only compressed filesystems dominate, because flash is small and the vendor does not intend you to write to it.

Filesystem	Magic	Notes
SquashFS	`hsqs` (LE) or `sqsh` (BE)	By far the most common. Read-only, compressed. `unsquashfs` extracts it.
JFFS2	`0x1985`	Journaling, designed for raw flash. Writable — often the overlay holding settings.
UBIFS	`UBI#`	Newer flash filesystem, on top of the UBI layer.
CramFS	`0x28cd3d45`	Older, read-only, largely displaced by SquashFS.

Two more signatures worth recognising, which are containers rather than filesystems:

Format	Magic	Notes
uImage	`0x27051956`	U-Boot's kernel wrapper. Header gives load/entry addresses.
TRX	`HDR0`	Broadcom-lineage router image container.

The `hsqs` / `sqsh` distinction is not trivia — it tells you the **endianness** of the target before you have looked at a single instruction, which is the setting that most reliably ruins a Ghidra session (A5).

The workflow
`binwalk` scans for these signatures, `binwalk -e` or `unsquashfs` extracts, and you are looking at a filesystem tree. That is FSTM stages 3 through 5, and Track B is built on it.

World two: RTOS and bare metal
The thermostat, the smart plug, the motor controller, the BLE tag. A microcontroller — Cortex-M, MSP430, RISC-V — with kilobytes of RAM, no MMU, and often no memory protection at all.

There is **no filesystem**. There is no libc as a separate object, no kernel as a separate object, and usually no symbols. The application, the RTOS (if any), and the drivers are all compiled and linked into **one flat blob** that gets written to flash and executed in place.

An RTOS here means FreeRTOS, Zephyr, VxWorks, ThreadX, embOS or similar — a scheduler and some primitives linked into your binary, not an operating system that owns the machine. Plenty of devices have no OS at all: a `main()` with an infinite loop and some interrupt handlers.

Where the methodology diverges. FSTM stage 4 is "extract the filesystem". There is no filesystem, so stage 4 does not apply as written, and stage 5 — static analysis of filesystem contents — has nothing to operate on. Instead of extracting a tree, you **reconstruct the memory map yourself**: find the base address, locate the vector table, define the memory regions, and work outward from the reset handler. That is Track C, and modules A2 and A5 are what make it possible.

Telling them apart
In practice this takes about thirty seconds.

It is the Linux world if: `binwalk` reports a SquashFS/JFFS2/CramFS/UBI signature, or a uImage/TRX container; the image is measured in megabytes; strings include recognisable paths like `/etc/passwd`, `/bin/busybox`, `/dev/mtdblock0`.

It is the RTOS/bare-metal world if: `binwalk` finds no filesystem anywhere; the image is measured in kilobytes to a couple of megabytes; the first words look like a vector table — an SRAM address followed by a run of odd, similar-looking code addresses (exactly the A2 pattern); strings include an RTOS name or scheduler symbols such as `xTaskCreate`.

Two cases worth expecting, so they do not read as a failure of your method:

- **Encrypted or unknown-compressed.** `binwalk` finds nothing and entropy is uniformly high across the whole image — around 7.9 bits per byte. That is not a blob; that is an image you cannot read yet. Encryption and compression look alike by entropy, and separating them is FSTM stage 3 (module B1).
- **Both at once.** A larger device may have a Cortex-A running Linux *and* a Cortex-M co-processor with its own blob, shipped in one update file. Finding a filesystem does not prove there is nothing else in the image.

What to carry forward

- Linux firmware = **bootloader + kernel + extractable filesystem**. The filesystem is the prize.
- SquashFS is the common case; `hsqs` versus `sqsh` also tells you **endianness**.
- RTOS/bare-metal = **one flat blob**, no filesystem, no symbols, no MMU.
- The split decides the methodology: **FSTM stages 4-5 do not apply to a blob**, and you reconstruct the memory map instead.
- High uniform entropy with no signatures means **encrypted or unknown-packed**, not bare metal.

Next: the three OWASP documents that structure the rest of this course, and how they divide the work between them.

FSTM, ISTG and ISVS: what each one is for

Why this matters

Three OWASP documents cover this field and their names are close enough to blur together. They are not alternatives and they do not overlap much. Each answers a different question, and knowing which to reach for saves you from the two failure modes of self-taught security work: wandering without a plan, and producing findings nobody can act on.

They also give your work a shape someone else recognises. "I ran some tools on a router" and "I completed an FSTM pass and these are the stage-5 findings" describe the same afternoon. Only one of them reads as an assessment.

The division of labour

Document	Answers	Shape
ISVS	*What must be true for this device to be secure?*	Requirements, by level
ISTG	*How do I test the device?*	Test cases, by component
FSTM	*How do I assess the firmware specifically?*	A nine-stage process

ISVS is the definition of secure. It is a list of requirements — "firmware updates are cryptographically signed and the signature is verified before installation" — organised into verification levels so you can say which bar a device is being held to. It contains no instructions. It is what your findings are measured *against*.

ISTG is the testing catalogue, organised by **component** rather than by process: the processing units, the memory, the firmware, the data-exchange services, and the internal, physical, wireless and user interfaces. It is a breadth document — its job is to make sure you did not forget the BLE stack because you were absorbed in the web interface. Track F is built on it.

FSTM is the depth document for one component. Firmware is where most of the interesting bugs live and where the workflow is most involved, so it gets its own nine-stage process. Track B is FSTM, stage by stage.

FSTM says explicitly that it complements ISTG. Use FSTM for the firmware; use ISTG so you remember the device has radios, pins and a UI as well.

The nine stages

Read these as a pipeline. Each stage consumes what the previous one produced, which is why the order is not arbitrary.

1. **Information gathering and reconnaissance.** FCC ID lookups, datasheets, the vendor's support pages, existing CVEs, identifying the SoC. Cheapest stage, and it decides whether the target is worth your time.
2. **Obtaining firmware.** Vendor downloads, intercepting an update check, capturing an OTA. If none of that works, you dump it off the chip — which is Track F, and why hardware exists in this course at all.
3. **Analyzing firmware.** What *is* this file? Signature scanning with `binwalk`, entropy analysis to separate encrypted from compressed from plain, header parsing.
4. **Extracting the filesystem.** `binwalk -e`, `unsquashfs`, firmware-mod-kit. Nested and obfuscated images live here. **This is the stage that does not apply to a blob** — the divergence from the previous card.
5. **Analyzing filesystem contents.** Static analysis of the tree: hardcoded credentials, keys and certificates, backdoor accounts, insecure configuration, SUID and world-writable binaries, outdated packages. Most findings on a typical consumer device come from this stage.
6. **Emulating firmware.** Bring it to life without the device — user-mode QEMU for a single binary, or full-system emulation with FirmAE for the whole stack.
7. **Dynamic analysis.** Exercise it while it runs: the web interface, the network services, fuzzing the inputs.
8. **Runtime analysis.** Attach a debugger, set breakpoints, watch memory and registers as something goes wrong.
9. **Binary exploitation.** Turn a crash into control. Everything in modules A2, A3 and A4 exists to make this stage possible.

Why the order is load-bearing

You cannot analyse a filesystem you have not extracted (5 needs 4). You cannot extract from an image you have not identified (4 needs 3). You cannot debug a running process without something running (8 needs 6). And you cannot write an exploit without a crash to work from (9 needs 8).

The failure this ordering prevents is a real and common one: jumping to stage 9 because exploitation is the interesting part, on an image whose endianness you never confirmed. That is also what makes stage order a fair thing to be tested on — not because the list is worth memorising, but because the dependencies are.

Stages are not all equally productive. Stage 5 finds most of the bugs on most consumer devices, and it is nearly free. Stage 9 is where the effort goes and where the impressive findings are. A good assessment does not skip 5 to get to 9 faster.

Where this course sits

- **Track B** is FSTM, one module per stage group.
- **Track C** is what you do instead of stages 4-5 when there is no filesystem.
- **Track F** is ISTG's hardware and wireless components.
- **ISVS** is the yardstick your capstone report in Track G measures findings against.

What to carry forward

- **ISVS = requirements. ISTG = tests, by component. FSTM = the firmware process, in nine stages.**
- The nine stages are a **pipeline**, and each depends on the one before it.
- Stage 4 is the divergence point between the two firmware worlds.
- Stage 5 is cheap and finds the most; stage 9 costs the most and proves the most.
- FSTM complements ISTG rather than replacing it — the device is more than its firmware.

Next: drill the signatures until identification is instant, then order the stages yourself.

Module A1

BusyBox, init and nvram: the embedded userland

Why this matters

You have extracted a root filesystem. It looks like Linux, and it is — but it is Linux built for 4 MB of flash, and almost every assumption from a desktop system is wrong in a way that matters.

More practically: **most router bugs are not in the kernel.** They are in the glue — a shell script that interpolates a configuration value, a CGI handler that passes a form field to `system()`, a service that trusts a setting because "only the admin can change it". You cannot find those bugs if the userland reads as noise.

BusyBox: every command is the same binary

Look in `/bin` on an extracted firmware and you will find dozens of commands that are all symlinks to one executable.

```
lrwxrwxrwx  ls -> ../bin/busybox
lrwxrwxrwx  cat -> ../bin/busybox
lrwxrwxrwx  sh -> ../bin/busybox
-rwxr-xr-x  busybox
```

BusyBox implements hundreds of standard utilities as *applets* inside a single binary, and dispatches on `argv[0]` — the name it was invoked as. One binary, one copy of the shared code, a fraction of the space.

Three consequences you will use:

The symlink list is a capability list. Before you have shell access, `ls -l bin sbin usr/bin` tells you exactly which commands exist on the device. No `wget`? No `nc`? That shapes what post-exploitation looks like, and it is free information from the extracted filesystem.

The applets are reduced. BusyBox implements the common options, not all of them. A script that works on your desktop may not work on the device, and that includes anything you drop on there yourself.

Its version is a finding. `strings bin/busybox | grep BusyBox` gives you a version, and BusyBox has a real CVE history. So does every other component: this is the "outdated packages" half of FSTM stage 5, and it is often the cheapest finding in the entire assessment.

Init: how the device gets from power-on to listening

Embedded init is simpler than a desktop's, and it is written down where you can read it.

BusyBox init reads `/etc/inittab`, which is short and mostly says "run this script":

```
::sysinit:/etc/init.d/rcS
::respawn:/sbin/getty -L ttyS0 115200 vt100
```

That second line is worth pausing on. It is a login prompt on the serial port, and it is why Track F starts with UART: a device that spawns `getty` on `ttyS0` will offer you a login prompt the moment you attach a USB-serial adapter to the right three pins.

`/etc/init.d/rcS` then runs the numbered scripts in `/etc/init.d/` or `/etc/rc.d/` in order: mount filesystems, configure the network, start the web server, start the vendor's daemons.

OpenWrt-derived devices use `procd` instead, with service definitions under `/etc/init.d/` in a slightly different shape, and `/etc/config/` for configuration. Same idea, different spelling. IoTGoat is OpenWrt-based, so this is the variant you will meet in Track B.

Reading these scripts is high-yield static analysis. They are plain text, they are short, and they tell you the complete list of what listens, what runs as root, and what gets started with arguments interpolated from configuration. Do this before you open a disassembler.

nvram: the configuration substrate, and why it is an attack surface

Desktop Linux keeps configuration in files. Embedded devices frequently cannot: the root filesystem is a read-only SquashFS. So settings live in **nvram** — a dedicated flash partition holding key/value pairs, read and written through a helper.

```
# nvram get http_passwd
# nvram set wan_hostname=router1
# nvram commit
```

Stored as `key=value` entries packed together, typically NUL-separated, which is why the partition dumps as one run of text with no line breaks.

Here is why it matters, and it is the central insight of this card:

> **nvram values reach command interpreters, and the device sets them from places an attacker can reach.**

If any path lets an attacker set an nvram value — a web form, a UPnP request, a poorly validated API, a DHCP option the device stores — then that attacker's data is flowing into code that runs as root. This is the most productive bug class on consumer routers, and it is why "who can write this key" is worth chasing for every key you find.

But be precise about the mechanism, because the obvious-looking version is not a bug. In a POSIX shell, the result of a parameter expansion undergoes word splitting and globbing — it is **not re-parsed for control operators**. So this is *not* injectable:

```
HOST=$(nvram get wan_hostname)
curl "https://example.net/update?host=$HOST"     # `;` in $HOST is just a character
```

Setting `wan_hostname` to `x; reboot` here passes a weird string to `curl` and nothing else. A great deal of training material gets this wrong, and writing it up as a finding is how you lose credibility with a vendor.

The mechanism has to hand the assembled string to something that *parses* it. Three that do:

```
eval $CMD                        # explicit re-parse -- the classic vendor pattern
sh -c "ping -c 4 $HOST"          # a new shell parses the whole string
`nvram get cmd_prefix` arg       # command substitution on stored data
```

…and in C, the equivalent is `system()`, which hands its argument to `/bin/sh` precisely so that it *will* be parsed. That is the next card.

Three things to look for in the extracted filesystem:

- **Defaults.** `etc/nvram.default`, `etc/defaults/`, or a compiled-in table. Vendor default credentials live here surprisingly often.
- **Writers.** Anything calling `nvram set` with a value derived from input.
- **Readers that interpolate.** Anything doing `$(nvram get ...)` inside a command, or passing the result to `system()` in C.

A worked example

From an extracted OpenWrt-derived router, `/etc/init.d/ddns_updater`:

```
#!/bin/sh
HOST=$(nvram get ddns_hostname)
USER=$(nvram get ddns_username)

start() {
    CMD="/usr/bin/curl -s https://api.example.net/update?host=$HOST&user=$USER"
    eval $CMD
}
```

Read it in order:

1. `ddns_hostname` comes from nvram, so the question is who can set it. The dynamic-DNS settings page is one obvious writer.
2. It is interpolated into `CMD`. On its own that is inert — a string is just a string.

3. **`eval` is the bug.** It re-parses the assembled string as shell source, so control operators inside `$HOST` become live. A value of `x; telnetd -l /bin/sh -p 4444;` runs `telnetd`.
4. The script is started by init, which runs as **root**.

That is a full command injection to root, found by reading three lines of shell. No disassembler, no debugger, no exploit primitive. This is what stage 5 productivity means, and it is why the module before the architecture modules is this one.

Now re-read step 3 and notice what carried it. Delete the `eval` and run `$CMD` directly and the bug disappears — the value would be word-split into arguments and handed to `curl` unparsed. The finding is not "user data reaches a command line". The finding is **"user data reaches something that parses it"**, and telling those apart is the difference between a report a vendor acts on and one they dismiss.

What else is in there

Quick orientation for the rest of the tree:

- `/etc/passwd` and `/etc/shadow` — hashes worth cracking, and accounts nobody documented.
- `/etc/config/`, `/etc/*.conf` — service configuration.
- `/www/`, `/web/`, `/htdocs/` — the web interface, and the CGI handlers behind it (next card).
- `/usr/sbin/`, `/bin/` — the vendor's own binaries. Anything not a BusyBox symlink is custom code, and custom code is where the interesting bugs are.
- `/dev/mtdblock*` — the raw flash partitions, including the one holding nvram.

Prioritise the non-BusyBox binaries. BusyBox is widely reviewed. The vendor's `httpd`, their `upnpd`, their proprietary daemon — that code has been read by far fewer people.

What to carry forward

- **BusyBox is one binary** dispatching on `argv[0]`; its symlink list is a capability list, and its version is a finding.
- **Read the init scripts first.** They are plain text and they enumerate everything that runs.
- A `getty` on `ttyS0` in inittab is a **serial login prompt** — the Track F entry point.
- **nvram is the attack surface.** Values get interpolated into shell commands, and the scripts doing it run as root.
- For every nvram key: **who can write it, and where is it read?**
- Custom vendor binaries deserve more attention than BusyBox.

Next: the web stack that sits on top of this, and the C idioms that turn a form field into a shell.

The embedded web stack, and the C idioms that break it

Why this matters

The web interface is the largest attack surface on a consumer device and the one an attacker reaches first. It is also, on embedded targets, written in C — not PHP, not Python, not a framework with a template engine that escapes things for you. C, compiled into the vendor's `httpd`, parsing HTTP by hand.

That combination is why the same two bugs appear on device after device: a form field reaches a shell, or a form field reaches a fixed-size buffer. This card is about recognising both from the source, so that when Track B puts you in front of a decompiled handler you already know what you are looking for.

The stack

Small, and there are only a few of them:

Server	Notes
BusyBox httpd	The applet. Small, CGI-capable, common on OpenWrt-derived devices.
boa, **mini_httpd**, **thttpd**	Tiny standalone servers. Old, frequently unpatched.
lighttpd	Larger and more capable; on devices with more flash.
A vendor's own `httpd`	Custom C. **The interesting case** — least reviewed, most bespoke parsing.

The last row deserves the emphasis. BusyBox and lighttpd have had many eyes on them. A vendor's proprietary `httpd`, shipped on one product line, has had very few.

How a request becomes a command

The **CGI** model is old and simple, and its simplicity is the problem. The server receives a request, sets environment variables, and either executes a program or dispatches to a handler compiled into itself.

```
POST /cgi-bin/ping.cgi HTTP/1.1
Content-Type: application/x-www-form-urlencoded

host=192.168.1.1&count=4
```

The server sets `REQUEST_METHOD`, `QUERY_STRING`, `CONTENT_LENGTH` and the rest, then hands the body to the handler. The handler pulls out `host`, and does something with it.

On a device, "something" is very often this:

```
char cmd[256];
char *host = get_param("host");

sprintf(cmd, "ping -c 4 %s", host);
system(cmd);
```

Two separate bugs in four lines, and it is worth separating them because they lead to different places:

Command injection. `host` is interpolated into a string handed to `system()`, which runs it through `/bin/sh`. A value of `8.8.8.8; telnetd -l /bin/sh -p 4444` runs both commands. The handler is a child of the web server, which on these devices runs as **root**. This is CWE-78, it needs no memory-corruption knowledge, and it is the highest-yield bug class on consumer firmware.

Buffer overflow. `cmd` is 256 bytes and `sprintf` writes as much as it is given. A long `host` overruns it. This is CWE-120, and turning it into control of execution is modules A4 and B5.

The same input reaches both. Which one you pursue depends on which is easier to weaponise — and on a device with no ASLR and no stack canary, that is often a real choice rather than a foregone one.

The dangerous calls

Two families. Keep them separate in your head, because they produce different findings.

Family 1 — writes that are not bounded

Function	What it fails to bound
`strcpy(dst, src)`	Copies until a NUL in `src`. Nothing limits the write.
`strcat(dst, src)`	Appends until a NUL. Nothing limits the total.
`sprintf(buf, fmt, ...)`	Formats without a size limit.
`gets(buf)`	Reads a line with no limit at all. Unusable safely.
`scanf("%s", buf)`	Unbounded `%s`. `%255s` would bound it.
`memcpy(dst, src, n)`	Bounded by `n` — **but only as far as `n` is trustworthy.**

`memcpy` is the one that catches people. It takes a length, so it looks safe, and it is safe exactly when `n` is derived from the destination's size rather than from the input's. An attacker-controlled `n` — a length field from a packet, a `Content-Length`, a value from a header — is a bounded write to an unbounded degree.

The `n`-taking variants (`strncpy`, `strncat`, `snprintf`) are better and are not automatically safe: `strncpy` does not NUL-terminate when the source fills the buffer, and `strncat`'s length argument is the number of bytes to *append*, not the size of the destination. Both are recurring off-by-one sources.

Family 2 — input that reaches a shell

Function	Note
`system(cmd)`	Runs `cmd` through `/bin/sh`. Metacharacters are live.
`popen(cmd, mode)`	Same, with a pipe.
`execl`, `execlp`, `execvp`	Safer — **no shell** unless you invoke one. `execlp("sh", ...)` is not.

The distinction is the shell. `system("ping -c 4 8.8.8.8; id")` runs `id`. `execv("/bin/ping", argv)` with `argv[1] = "8.8.8.8; id"` passes that whole string to `ping` as one argument, which fails harmlessly. **A vendor who used `execv` here has closed the bug**, and recognising that saves you from writing up a finding that is not one.

How to hunt this in an extracted filesystem

Cheap and mechanical, which is the point of stage 5:

1. `grep -rn "cgi-bin\|/www\|/htdocs"` — find the handlers.
2. For each vendor binary, list imports and look for the two families above.
3. Work **backwards** from each dangerous call to its arguments (the A5 xref walk), asking one question at a time: does this argument come from a request?
4. Read the shell scripts too. The equivalent bug in `sh` is a stored value reaching **`eval`** or **`sh -c`** — not merely being interpolated, since a plain expansion is not re-parsed (previous card). Grep for `eval` first; it is a short list and it is where the bugs are.

What a corrected version looks like

Worth knowing so you can tell a fixed device from a vulnerable one:

```
char *host = get_param("host");

if (!valid_ipv4(host))          /* validate against an allowlist  */
    return error("bad host");

char *argv[] = { "/bin/ping", "-c", "4", host, NULL };
execv(argv[0], argv);           /* no shell, no interpolation     */
```

Two changes carry it: input is validated against what is *allowed* rather than filtered for what is forbidden, and the shell is gone entirely. Blocklist filtering — stripping `;` and `|` — is the fix vendors reach for first, and it is routinely bypassed with backticks, `$()`, newlines, or an encoding the filter runs before decoding.

What to carry forward

- The web stack is **C**, often the vendor's own, and it is the least-reviewed code on the device.
- **Command injection and buffer overflow are separate findings** that frequently share an input.
- Family 1 fails to bound a **write**; family 2 hands input to a **shell**.

- `memcpy` is only as safe as `n` is trustworthy.
- `execv` without a shell is genuinely fixed. `system` and `popen` are not.
- The shell-script version of the same bug is unquoted `$(nvram get ...)` — and it runs as root.

Next: drill the sinks, order the boot, and find the injection in a real handler.

Module A2

ARM profiles, registers, and the AAPCS calling convention

Why this matters

ARM is the architecture you will meet most. It runs the phone in your pocket, the router's successor generation, and essentially the entire microcontroller world — which means it shows up in both of the two firmware worlds, on opposite sides of the split. Getting oriented starts with one question that decides everything downstream: **which ARM is this?**

After that, it is the same job as MIPS. Find the arguments, find the return value, find where the return address is stored, and work out whether you can reach it.

Three profiles, and why the letter matters

Modern ARM ships in three profiles, and the letter tells you what kind of target you are holding.

Profile	Memory protection	Typically runs	Your workflow
Cortex-A (Application)	full **MMU**	Linux	extract a filesystem — Track B
Cortex-R (Real-time)	MPU	RTOS, hard-real-time	rare in consumer teardowns
Cortex-M (Microcontroller)	**MPU at best**, often nothing	RTOS or bare metal	flat blob — Track C

The A-versus-M split maps exactly onto the two firmware worlds from A0. A Cortex-A image has a bootloader, a kernel and a root filesystem you can unpack. A Cortex-M image is a single blob with no filesystem, no libc, and usually no symbols — you reconstruct the memory map yourself.

That split also decides how hard exploitation is. Cortex-A gives you an MMU, so ASLR and NX are at least *possible*. On Cortex-M there is no MMU at all: an MPU if the vendor bothered, and frequently not even that. An overflow reaches control flow directly, with nothing in the way.

The registers

Sixteen general-purpose registers in the base 32-bit state, `r0` through `r15`. Three of them have architectural jobs and the rest are assigned roles by the **AAPCS**, the ARM equivalent of MIPS's O32.

Register	Alias	Role
`r0` – `r3`	`a1` – `a4`	**Arguments 1–4.** Further arguments go on the stack.
`r0` – `r1`		**Return value** (`r0` alone for anything word-sized).
`r4` – `r11`	`v1` – `v8`	**Callee-saved.** A function must restore these before returning.
`r12`	`IP`	Intra-procedure-call scratch. The linker may clobber it.
`r13`	**`SP`**	Stack pointer.
`r14`	**`LR`**	**Link register — the return address.**
`r15`	**`PC`**	Program counter. Architecturally readable and writable.

Three of those deserve more than a row.

`LR` is the exploitation lever. `BL target` ("branch with link") sets `LR` to the address of the instruction after the call, then branches. A leaf function can return with `BX LR` and never touch memory. But any function that itself calls something must preserve `LR` across that call — so it pushes `LR` to the stack in its prologue and pops it in the epilogue. **That pushed copy is what a stack overflow reaches.** It is precisely MIPS's saved `$ra`, under a different name.

`PC` is a general-purpose register, and this is unusual. On x86 you cannot write to `EIP` directly. On ARM you can: `POP {r4, pc}` is an ordinary, extremely common epilogue, and it loads the program counter straight off the stack. That is why ARM ROP gadgets so often end in a `POP` that includes `pc` — the architecture hands you the primitive.

Reading `PC` does not give you the current address. For historical pipeline reasons, reading `r15` yields the address of the current instruction **plus 8** in ARM state, and **plus 4** in Thumb state. Compilers rely on this for PC-relative addressing, and it will confuse you the first time you compute a literal-pool offset by hand.

A worked example

The same function as the MIPS card, so the two are directly comparable.

```
copy_name:
    PUSH  {r4, lr}          @ 1. save LR (and a callee-saved register) -- 8 bytes
    SUB   sp, sp, #32       @ 2. carve 32 bytes of local space

    MOV   r4, r0            @ 3. stash argument 1 (the source pointer)
```

```
    MOV   r0, sp            @ 4. arg 1 = &buffer, at the bottom of the frame
    MOV   r1, r4            @ 5. arg 2 = the source pointer
    BL    strcpy            @ 6. strcpy(buffer, src)

    ADD   sp, sp, #32       @ 7. tear down the local space
    POP   {r4, pc}          @ 8. restore r4 and return by loading PC directly
```

Read it as an attacker:

- Line 1 pushes `lr`, so the return address now lives **in memory**, above the locals.
- Line 2 puts 32 bytes of locals **below** it.
- Line 4 points the copy destination at the bottom of that 32-byte region.
- So the distance from the buffer to the saved `LR` is **32 bytes of locals, then 4 bytes of saved `r4`, then the saved `LR`** — 36 bytes of filler before you are writing the return address.
- Line 8 pops it straight into `PC`. You do not need a separate "jump to it" step; the epilogue is the jump.

Compare that to the MIPS version: same shape, same lever, different spelling. `PUSH {r4, lr}` is `sw $ra, 36($sp)`. `POP {r4, pc}` is `lw $ra` followed by `jr $ra`. The offset arithmetic in module A4 is identical on both.

One difference worth noticing now: there is no instruction after `BL` on line 6 that secretly executes, and nothing after `POP {r4, pc}` on line 8 either. **ARM has no branch delay slot.** Where MIPS makes you correct every listing you read, ARM behaves the way the printed order suggests. That is one fewer trap here — and it is replaced by a different one, which the next card covers.

What to carry forward

- **Profile first.** Cortex-A means Linux and a filesystem; Cortex-M means a flat blob and no memory protection worth the name.
- `r0` – `r3` in, `r0` out, `LR` back. Same five-register answer as MIPS.
- **The saved `LR` on the stack is the target.** MIPS calls it `$ra`.
- `PC` is writable, and `POP {..., pc}` is a normal epilogue — which makes it a gift to ROP.
- Reading `PC` gives you current + 8 (ARM) or + 4 (Thumb).
- No delay slot. The listing means what it says.

Next: ARM and Thumb are two different instruction encodings in the same binary, and the low bit of an address decides which one you get.

ARM and Thumb, and the bit that chooses between them

Why this matters

MIPS's signature trap is the delay slot: an instruction runs when you did not expect it to. ARM has no delay slot, so that whole class of error disappears. In its place is a subtler one — a single bit, in an address, that decides how the processor will interpret the bytes it finds there.

Get it wrong in Ghidra and the disassembly is garbage. Get it wrong in a payload and the target faults instead of executing your code. It is the most common reason an otherwise correct ARM exploit does nothing, and it is invisible in a hex dump.

Two encodings, one binary

ARM processors execute two different instruction encodings:

- **ARM state** — fixed 32-bit instructions, 4-byte aligned. The full instruction set, including conditional execution on nearly every instruction.
- **Thumb state** — mostly 16-bit instructions, 2-byte aligned. Denser code, fewer registers reachable per instruction. **Thumb-2** extends it with a mix of 16- and 32-bit encodings and recovers most of what plain Thumb gave up, which is why modern code is overwhelmingly Thumb-2.

The same binary can contain both, and functions in one state can call functions in the other. That is called **interworking**, and the processor has to be told which state to enter on every such branch.

On Cortex-M there is no choice at all: it executes Thumb only. ARM state does not exist on those parts. That fact does not make the rule irrelevant — it makes violating it fatal, as below.

The rule

> **The low bit of a branch target address selects the state. Bit 0 set means Thumb; bit 0 clear means ARM.**

The bit is *not* part of the address. Thumb instructions are 2-byte aligned, so bit 0 of a real instruction address is always 0 and the encoding space is free. ARM reuses it as a state selector on the instructions that can change state — `BX`, `BLX`, and any `POP` or `LDR` that loads directly into `PC`.

This has an immediate, disorienting consequence:

> **A pointer to a Thumb function is the function's address plus one.**

If `handler` begins at `0x08001C4C`, then the function pointer value stored in a vector table, a jump table, or a struct is **`0x08001C4D`**. Every Thumb function pointer in the binary looks odd — literally, an odd number.

So when you are reading a firmware image and a table of what are obviously code addresses all end in an odd digit, that is not corruption. That is a Thumb vector table, and you are looking at it correctly.

To get the instruction address, clear bit 0. To build a working pointer, set it.

A worked example

A Cortex-M vector table, as raw words at the start of flash:

```
0x08000000:  0x20005000     <- initial stack pointer (MSP), not code
0x08000004:  0x08001C4D     <- reset vector
0x08000008:  0x08001D01     <- NMI handler
0x0800000C:  0x08001D2D     <- HardFault handler
```

Read it:

- The **first** word is not an address at all. It is the value the processor loads into `MSP` at reset. `0x20005000` sits in SRAM, which is the tell — Cortex-M SRAM conventionally starts at `0x20000000`.
- The **second** word is the reset vector: `0x08001C4D`. Bit 0 is set, so the target is Thumb. The actual first instruction of `Reset_Handler` is at **`0x08001C4C`**.
- Every subsequent entry is odd for the same reason.

If you tell Ghidra to disassemble at `0x08001C4D`, you will get nonsense, because you have asked it to start one byte into an instruction. Disassemble at `0x08001C4C` **in Thumb mode** and it resolves. That one-byte correction is the difference between a readable function and a screen of garbage, and it is the single most common mistake when loading a Cortex-M image in module C1.

Where it breaks exploits

You have overwritten a saved `LR` and you want execution to land on a Thumb function at `0x08001C4C`. Write that value and the processor takes bit 0 as clear, and therefore requests **ARM state**.

- On a **Cortex-A** part, it will try to decode your Thumb instructions as ARM. They are valid bytes, so it usually will not fault immediately — it will execute something arbitrary, which is worse than a crash because it is harder to diagnose.
- On a **Cortex-M** part, ARM state does not exist. The processor raises a **UsageFault with the INVSTATE bit set**, immediately. The symptom is a fault at the moment of transfer, with a fault address that looks correct — which reads like "my gadget address is wrong" when the address was right and only the bit was missing.

The fix is one character: write `0x08001C4D`, not `0x08001C4C`.

The mirror-image mistake is just as common. If you take a function pointer out of the binary and use it as an address — to compute a gadget offset, say — you must clear bit 0 first, or every offset you derive from it is off by one.

Reading ARM disassembly: three things that will surprise you

Beyond the state bit, three features make ARM listings read differently from MIPS.

Conditional execution. In ARM state, most instructions carry a condition code: `ADDEQ r0, r0, #8` adds only if the Z flag is set. There is no branch and no label — the instruction simply does nothing when the condition fails. A block of eight instructions may have four that ran and four that did not, with no control flow visible at all. Thumb-2 narrows this to the `IT` (if-then) block, which is easier to spot but means the same thing.

PC-relative addressing and the literal pool. ARM instructions cannot encode an arbitrary 32-bit constant, so the assembler places constants in a **literal pool** near the code and loads them with a PC-relative `LDR`. You will see `LDR r0, [pc, #0x1c]` and a block of data sitting in the middle of what is otherwise a function. That data is not code, and a disassembler that has not identified it will render it as instructions.

`PC` reads high. Reading `r15` gives the current instruction address **+ 8** in ARM state and **+ 4** in Thumb. When you compute a literal-pool target by hand and land 8 bytes short, this is why.

What to carry forward

- Bit 0 of a branch target selects state: **1** = **Thumb, 0** = **ARM**. It is not part of the address.
- A Thumb function pointer is **address + 1**. Odd-looking code addresses in a table are correct.
- Clear bit 0 to disassemble; set bit 0 to branch.
- On Cortex-M, a cleared bit is a **UsageFault (INVSTATE)**, not a wrong jump — and the fault address will look right.
- Conditional execution means instructions can be skipped with no branch in sight.
- ARM has **no delay slot**. The `LR` ↔ `$ra` lever is shared with MIPS; the Thumb bit ↔ delay slot are the two architectures' signature traps.

Next: drill the register roles, then trace a conditional sequence and diagnose a payload where the state bit was cleared.

Module A3

MIPS registers and the O32 calling convention

Why this matters

You are about to open a router binary in Ghidra. It will not have symbols. It will not have comments. What it will have is thirty-two registers being shuffled around, and the entire question of "what does this function do, and can I break it" reduces to knowing which of those registers carry meaning across a call boundary.

Concretely: when you find a `strcpy` in a MIPS binary, the thing you need to know immediately is *what is in `$a0` and `$a1`* — because those are the destination and the source, and whether the source is attacker-controlled is the whole bug. And when you overflow that destination, the thing you are trying to reach is the **saved `$ra`**, because that is what the function will jump to when it returns.

Registers are not bookkeeping here. They are the map.

The registers

MIPS has 32 general-purpose registers. They are architecturally interchangeable — the hardware does not care which you use — but software agrees on roles, and that agreement is the **O32 calling convention** (the one you will meet in essentially all embedded Linux MIPS firmware).

Symbolic	Number	Role
`$zero`	`$0`	Always reads as 0. Writes are discarded.
`$at`	`$1`	Assembler temporary. Reserved for pseudo-instruction expansion.
`$v0` – `$v1`	`$2` – `$3`	**Return values.** `$v0` is the one you will care about.
`$a0` – `$a3`	`$4` – `$7`	**Arguments 1 through 4.** Further arguments go on the stack.
`$t0` – `$t7`	`$8` – `$15`	Temporaries. **Caller-saved** — a callee may clobber them freely.
`$s0` – `$s7`	`$16` – `$23`	Saved. **Callee-saved** — a callee must restore them before returning.
`$t8` – `$t9`	`$24` – `$25`	More temporaries. `$t9` has a special job — see below.
`$k0` – `$k1`	`$26` – `$27`	Reserved for the kernel. Do not use.
`$gp`	`$28`	Global pointer.
`$sp`	`$29`	Stack pointer.
`$fp` / `$s8`	`$30`	Frame pointer, when one is used at all.
`$ra`	`$31`	**Return address.** Written by `jal`.

Two entries deserve more than a table row.

`$ra` is the exploitation lever. `jal target` does two things: it sets `$ra` to the address of the instruction to return to, and it jumps. At the end of the function, `jr $ra` jumps back. If the function stored `$ra` on the stack (any function that itself calls something must), and if you can overflow a stack buffer far enough to reach that stored copy, then you choose where the function returns. Everything in module A4 and module B5 is built on that sentence. On ARM the same lever is called `LR`; the mechanism is identical.

`$t9` holds the callee's own address in position-independent code. Shared libraries on MIPS are compiled PIC, and the convention is that a function is entered with its own address in `$t9`,

which it uses to compute where its globals live. So a call through a library looks like `jalr $t9`, not `jal some_absolute_address`. When you are hunting for a place to redirect execution, `$t9` is where the target address is already sitting — which is why so many MIPS exploitation writeups end with "control `$t9` and jump".

A worked example

Here is a complete leaf-ish function, the kind you will see hundreds of. It calls `strcpy` into a local buffer, which is the classic embedded bug in its natural habitat.

```
copy_name:
    addiu $sp, $sp, -40      # 1. carve 40 bytes of frame
    sw    $ra, 36($sp)       # 2. save the return address at the TOP of the frame
    sw    $s0, 32($sp)       # 3. save a callee-saved register we intend to use

    move  $s0, $a0           # 4. stash argument 1 (the source pointer)

    addiu $a0, $sp, 0        # 5. arg 1 = &buffer, which is the BOTTOM of the frame
    move  $a1, $s0           # 6. arg 2 = the source pointer
    jal   strcpy             # 7. strcpy(buffer, src)
    nop                      # 8. delay slot -- the next card explains this

    lw    $ra, 36($sp)       # 9. restore the return address
    lw    $s0, 32($sp)       # 10. restore the saved register
    jr    $ra                # 11. return
    addiu $sp, $sp, 40       # 12. delay slot: tear down the frame
```

Read it as an attacker.

- Line 1 tells you the frame is **40 bytes**.
- Line 2 tells you the saved `$ra` sits at **`$sp + 36`**.
- Line 5 tells you the destination buffer starts at **`$sp + 0`**.
- Therefore the distance from the start of the buffer to the saved return address is **36 bytes**. Write 36 bytes of filler and the next 4 bytes land on `$ra`.
- Line 7 copies into that buffer with `strcpy`, which stops at a NUL byte and checks nothing else. If the caller's string comes from the network, you control the return address.

That arithmetic — frame size, where `$ra` was stored, where the buffer starts — is the whole of the offset calculation you will do by hand in module A4 and automate in `fw-a4-find-offset`.

Notice also lines 7–8 and 11–12: **there is an instruction after each jump that is not a mistake.** That is the branch delay slot, and it is the subject of the next card. It is the single most common reason a MIPS listing does not mean what a first reading suggests.

What to carry forward

- `$a0` – `$a3` in, `$v0` out, `$ra` back. Those five registers answer most questions.
- Callee-saved (`$s*`) versus caller-saved (`$t*`) tells you which values survive a call — which tells you where a pointer you care about is likely to be stashed.
- **The saved `$ra` on the stack is the target.** ARM calls it `LR`. Same lever, same idea.
- `$t9` carries the callee's own address in PIC, and is a recurring target in MIPS exploitation.

Next: the branch delay slot, which changes what the listing above actually executes.

The branch delay slot

Why this matters
Every other rule in this module is bookkeeping you could look up. This one changes what a listing *means*, and it does so silently. A MIPS function you read correctly under x86 habits will do something different from what you concluded — and the difference will not announce itself. It will show up as an exploit that jumps to the right address with the wrong arguments, or a traced value that is off by exactly one instruction's worth of work.

It is the single biggest surprise MIPS holds for someone arriving from x86 or ARM, and it is worth over-learning.

The rule

> **The instruction immediately after a branch or jump always executes, before control transfers.**

That position — the one instruction slot following a branch or jump — is the **branch delay slot**.

It is not a quirk of the assembler and it is not dead code. It is architectural. MIPS is pipelined, and by the time the hardware has worked out whether a branch is taken and where it goes, the next instruction has already been fetched. Rather than discard that work, the architecture defines it as always executing. The compiler then treats the slot as free real estate and schedules something useful into it. When it cannot find anything useful, it emits `nop`.

So a MIPS listing has an execution order that differs from its printed order, and you have to apply the correction every time you read one.

A worked example
Take this fragment. Read it first the way you would read x86, then read it again.

```
    li    $t0, 5          # 1
    li    $t1, 10         # 2
    b     skip            # 3   unconditional branch
    addi  $t0, $t0, 3     # 4   <-- DELAY SLOT
    addi  $t0, $t0, 100   # 5
skip:
    add   $v0, $t0, $t1   # 6
```

The naive reading. "Line 3 branches to `skip`, so lines 4 and 5 are both skipped. `$t0` is 5, `$t1` is 10, so `$v0` is 15."

What actually happens:

Step	Instruction	Effect
1	`li $t0, 5`	`$t0` = 5
2	`li $t1, 10`	`$t1` = 10
3	`b skip`	branch is *issued*; control has not moved yet
4	`addi $t0, $t0, 3`	**executes** — it is the delay slot. `$t0` = 8
—		control now transfers to `skip`
5	`addi $t0, $t0, 100`	**skipped** — jumped over
6	`add $v0, $t0, $t1`	`$v0` = 8 + 10 = **18**

The answer is 18, not 15. Line 4 looked like it was on the not-taken path. It was not. Line 5, which looked identical in structure, genuinely was.

The distinction between lines 4 and 5 is *position*, and nothing else. One is adjacent to the branch and one is not.

The case that bites hardest: arguments

The example above costs you an arithmetic error. This one costs you an exploit.

```
lw    $a0, 0($s0)     # 1   $a0 = the string pointer
jal   strlen          # 2   call strlen
addi  $a0, $a0, 4     # 3   <-- DELAY SLOT
move  $s1, $v0        # 4   $s1 = the result
```

`jal` is a jump, so line 3 is in its delay slot and executes **before** `strlen` begins. By the time `strlen` reads `$a0`, it has already been advanced by 4. The function is called on `pointer + 4`, not on `pointer`.

Nothing in the listing says "argument fixup". It reads like a line that runs after the call returns. It does not.

Two related details worth carrying:

- **`jal` sets `$ra` to the instruction *after* the delay slot** — that is, `PC + 8`, not `PC + 4`. Execution resumes at line 4 on return, which is what you would want, but it means the saved return address is two instructions along from the `jal`, not one.
- **`jr $ra` has a delay slot too.** The frame teardown in the previous card (`addiu $sp, $sp, 40` printed after `jr $ra`) is the standard idiom: the stack pointer is restored on the way out, in the slot. It executes. The function does return correctly.

Where it breaks exploits

When you build a ROP chain or a ret2text jump on MIPS, every gadget that ends in a jump carries a delay slot, and **that slot is part of your gadget whether you planned for it or not**.

- A gadget ending `jr $t9` followed by `addiu $sp, $sp, 32` moves the stack pointer *every time you use it*. Your chain's offsets have to account for it.
- A stub that sets up `$a0` and then jumps to `system` will have whatever sits in the slot applied to `$a0` — after you set it, before `system` sees it.

The most common failure is not a crash. It is a call that happens with arguments you did not intend, which looks like the payload being wrong rather than the chain being wrong.

One exception, for completeness

The **branch-likely** instructions (`beql`, `bnel`, and friends) *annul* the delay slot when the branch is **not** taken: the slot executes only on the taken path. These are deprecated and you will meet them rarely, but if a listing's behaviour refuses to match the rule above, check the mnemonic for a trailing `l`.

What to carry forward

- The instruction after a branch or jump **always executes**, before the transfer.
- Read every MIPS listing twice: once printed, once with the slot correction applied.
- `jal` saves `PC + 8`. `jr $ra` has a slot as well.
- A gadget's delay slot is part of the gadget. Budget for its side effect.
- ARM has no delay slot. Its equivalent trap is the Thumb state bit in the low bit of an address — different mechanism, same category of silent, position-dependent error.

Next: drill the register roles until they are automatic, then trace sequences where the slot decides the answer.

Module A4

The stack frame, and how an overflow reaches the return address

Why this matters

Modules A2 and A3 taught you to read the architecture. Module B5 will ask you to exploit a router. This card is the bridge, and without it that jump is a cliff.

Everything here reduces to one arithmetic problem:

> **How many bytes from the start of this buffer to the saved return address?**

Get that number and you control where the function returns. The rest of exploitation is deciding what to put there.

Process memory, briefly

An embedded Linux process is laid out much like any other:

```
    high addresses
        +-------------------+
        |      stack        |  locals, saved registers, return addresses
        |         |         |  GROWS DOWNWARD
        |         v         |
        |                   |
        |         ^         |
        |         |         |
        |       heap        |  malloc
        +-------------------+
        |    bss / data     |  globals
        +-------------------+
        |      text         |  code
        +-------------------+
    low addresses
```

Two facts matter and the rest is detail:

- **The stack grows downward.** Each call moves the stack pointer to a *lower* address.
- **Buffers are written upward.** `strcpy` writes from the start of the buffer toward higher addresses.

Those two directions are opposite, and that opposition is the entire vulnerability. The buffer sits at a low address in the frame; the saved return address sits at a high one. Overflowing the buffer walks *up* into it.

The frame

Look at a MIPS prologue and read the numbers as a map.

```
handle_request:
    addiu $sp, $sp, -96      # 1. carve 96 bytes
    sw    $ra, 92($sp)       # 2. save the return address near the TOP
    sw    $s1, 88($sp)       # 3. save callee-saved registers below it
    sw    $s0, 84($sp)
    ...
    addiu $a0, $sp, 24       # 4. &buf -- the buffer starts 24 bytes up from $sp
    move  $a1, $s2
    jal   strcpy             # 5. strcpy(buf, attacker_string)
```

That gives you the layout:

```
    $sp + 96  +---------------------+  <- caller's frame above here
    $sp + 92  |   saved $ra         |  <- the target
    $sp + 88  |   saved $s1         |
    $sp + 84  |   saved $s0         |
              |   ... other locals  |
    $sp + 24  |   buf[]  starts     |  <- the write starts here
              |   ...               |
    $sp + 0   +---------------------+  <- $sp
```

The offset is 92 − 24 = 68. Write 68 bytes of filler and the next 4 bytes land on the saved `$ra`.

Three numbers, all of them read straight off the prologue:

Number	Where it comes from
Frame size, 96	`addiu $sp, $sp, -96`
Saved `$ra` at +92	`sw $ra, 92($sp)`
Buffer at +24	`addiu $a0, $sp, 24`

The frame size is a distractor for the offset calculation — it tells you how big the frame is, not where either landmark sits inside it. `92 - 24`, not `96 - 24`.

ARM is the same shape

```
handle_request:
    PUSH  {r4, r5, lr}      @ pushes 12 bytes: r4, r5, lr
    SUB   sp, sp, #84       @ 84 bytes of locals below them
    ADD   r0, sp, #16       @ &buf at sp+16
    BL    strcpy
```

After the `PUSH` and `SUB`, the saved `lr` sits at `sp + 84 + 8` = `sp + 92`, and the buffer at `sp + 16`. Offset **76**.

The `+8` is the two registers pushed below `lr` — `PUSH {r4, r5, lr}` stores in increasing register order at increasing addresses, so `lr` ends up highest. Miscounting the register list is the ARM-specific way to get this wrong.

Same lever, same arithmetic. `LR` ↔ `$ra`, as in A2 and A3.

Finding the offset without the source

You will usually not have a clean prologue to read — you will have a crash. The standard technique:

1. **Send a cyclic pattern** instead of a repeated character. A De Bruijn sequence has the property that every 4-byte window is unique: `Aa0Aa1Aa2Aa3Aa4...`
2. **Crash it and read the register.** The saved return address is loaded into `$ra` (or `pc`), so the crash reports a nonsense address like `0x62413761`.
3. **Find that value in the pattern.** Its position *is* the offset, because every window is unique.

This is what `fw-a4-find-offset` implements, and it converts one crash into an exact number with no guessing.

Note the byte order. `0x62413761` on a little-endian target is the bytes `61 37 41 62`, which is `"a7Ab"` — the pattern reads back reversed. Get this wrong and you search for a substring that is not there, conclude the technique failed, and go back to guessing.

What "control" actually means

When the function returns, `jr $ra` (or `POP {..., pc}`) jumps to whatever is in that slot. You chose it. That is the primitive, and it is the whole of what a stack overflow buys you.

What it does **not** buy you:

- It does not run your code. That needs somewhere to put code and permission to execute it — the subject of the next card.
- It does not survive a wrong offset. Off by one word and you overwrite a saved register instead, which usually crashes somewhere confusing and looks like the target being hardened.
- It does not necessarily give you a full 4 bytes. `strcpy` stops at a NUL, so an address containing a zero byte truncates the copy. On MIPS, addresses in the `0x004xxxxx` range have a leading zero byte, and since MIPS binaries are commonly big-endian that zero is written *first*. Which addresses are reachable at all is a real constraint, not a footnote.

What to carry forward

- The stack grows **down**; buffers are written **up**. That opposition is the bug.
- Offset = **(where `$ra` / `LR` was saved) − (where the buffer starts)**. Not the frame size.
- Read all three numbers off the prologue: `addiu $sp`, `sw $ra`, and the `&buf` computation.
- On ARM, count the `PUSH` register list carefully — `lr` sits above the others.
- No source? **Cyclic pattern → crash → look up the register value**, and mind the endianness when you search for it.
- Control of the return address is the primitive. Turning it into execution is the next card.

Next: what actually stops you from using that primitive, and which of those defences are genuinely present on an embedded target.

Mitigations, and code reuse when they are present

Why this matters

You control the return address. The obvious next step — put your shellcode in the buffer and return to it — has been dead on general-purpose systems for twenty years, and most tutorials teach the defences as though they are universal.

On embedded targets they very often are not. A consumer router binary with no NX, no ASLR and no stack canary is not a museum piece; it is Tuesday. The skill this card builds is not "how to defeat mitigations" — it is **checking which ones are actually there** before deciding how much work you have.

The four defences

NX / DEP — no execute. Memory is either writable or executable, not both. Your shellcode sits in the buffer, the buffer is on the stack, the stack is not executable, and returning to it faults. *Defeats: injected shellcode. Does not defeat: reusing code that is already there.*

ASLR — address space layout randomisation. The stack, heap and libraries land at different addresses each run, so a hardcoded address is wrong. *Defeats: hardcoded addresses. Does not defeat: anything you can compute at runtime from a leak, or anything at a fixed address.* Note that ASLR without **PIE** leaves the main executable at a fixed address — its code and its PLT are still hardcodable, which is frequently enough.

Stack canary. A random value written between the locals and the saved return address, checked before returning. A linear overflow has to cross it, and the check aborts. *Defeats: linear stack overflow. Does not defeat: a write primitive that skips the canary, or a leak of its value.*

RELRO. Makes the GOT read-only after linking, so an overwrite of a function pointer there is not available. *Defeats: GOT overwrite. Partial RELRO leaves it writable.*

The embedded reality

This is the part general material omits.

NX requires MMU support and a toolchain that marks the stack non-executable. Common on Cortex-A Linux; frequently absent on older MIPS builds. **On Cortex-M there is no MMU at all** — an MPU if the vendor configured one, which is often not the case.

ASLR needs kernel support and PIE binaries. Consumer devices run old kernels, vendors build non-PIE, and plenty of embedded systems have `randomize_va_space` set to `0`. The main executable is at a fixed address in most router binaries you will meet.

Canaries need `-fstack-protector`, which vendors optimising for size routinely leave off.

RELRO likewise defaults off in many older toolchains.

So check.

```
# One binary. This is the command that decides your whole approach.
checksec --file=./squashfs-root/usr/sbin/httpd

# Every executable in an extracted image, so you pick the soft target.
checksec --dir=./squashfs-root/usr/sbin/

# No checksec on the box? These three answer the same questions.
readelf -l  ./httpd | grep -A1 GNU_STACK      # RWE => stack is executable
readelf -d  ./httpd | grep -E 'BIND_NOW|RELRO'
readelf -h  ./httpd | grep Type               # EXEC = no PIE, DYN = PIE
readelf -s  ./httpd | grep -c __stack_chk_fail  # 0 => no canaries

# And the system-wide ASLR setting, from the running device:
cat /proc/sys/kernel/randomize_va_space       # 0 = off, 2 = full
```

The answer determines whether you are writing a ten-line exploit or a ROP chain. Assuming hardened is as wrong as assuming defenceless — and it is the more expensive mistake, because it sends you building machinery you did not need.

Code reuse

When NX is on — or when injection is impractical for other reasons — you stop supplying code and start reusing the target's.

ret2text. Return to a function already in the binary. If there is a `spawn_shell()` or a debug routine or anything that does something useful, jump straight to it. Cheapest technique available, and it exists more often than you would expect on devices with factory diagnostics left in.

ret2libc. Return into a library function with arguments you control — classically `system("/bin/sh")`. You need the address of `system` and a pointer to a command string, and on firmware the string is often already in the binary. No injected code, so NX is irrelevant.

ROP. When no single function does what you want, chain fragments — *gadgets* — each ending in a return. Each gadget does a small amount of work and returns into the next. This is the general technique and the most work; reach for it when ret2text and ret2libc do not fit.

On MIPS, code reuse is the default even without NX

Module A3 covered why, and it is worth restating because it surprises people: MIPS data and instruction caches are frequently **not coherent**. Your shellcode is written through the data path and lands in the D-cache. The processor fetches instructions through the I-cache, which was never invalidated, so it executes whatever stale bytes were there.

The result is an exploit that works sometimes, or after unrelated activity flushes a cache line, and looks like an unreliable offset. **ret2libc sidesteps it entirely** by reusing code that is already resident and already cached. That is why real MIPS router exploits lean on it even where NX is absent.

The calling-convention wrinkle

On x86, ret2libc arguments go on the stack, right where your overflow already writes. On **MIPS and ARM the first arguments are in registers** (`$a0`, `r0`), and an overflow writes memory, not registers. So you need a gadget that loads the argument register from the stack before reaching your target function — typically an epilogue that pops into the argument register, or one of the `lw $a0, N($sp)` sequences that appear in ordinary function tails.

This is the point at which even a "simple" MIPS ret2libc becomes a short ROP chain, and it is the first genuinely architecture-specific difficulty in exploitation. Module B5 works it on DVRF.

Choosing

A rough order, cheapest first:

1. **Is there anything in the binary that already does what I want?** → ret2text.
2. **Can I call one library function with controlled arguments?** → ret2libc.
3. **Neither?** → ROP.

And before any of them: **run checksec**, and confirm which defences are real on this target.

What to carry forward

- **NX** kills injected shellcode. **ASLR** kills hardcoded addresses. **Canaries** kill linear overflows. **RELRO** kills GOT overwrites.
- On embedded targets these are **frequently absent** — check rather than assume, in either direction.
- ASLR without PIE still leaves the **main binary at a fixed address**.
- **Code reuse** — ret2text, ret2libc, ROP — is immune to NX because you supply no code.
- On MIPS, prefer code reuse regardless of NX, because of **cache incoherency**.
- Register-based calling conventions mean MIPS/ARM ret2libc needs a gadget to load `$a0` / `r0`.

Next: drill the mitigations, compute a real offset, and diagnose an exploit that assumed the wrong thing.

Module A5

Loading a binary so Ghidra can help you

Why this matters

"I have used Ghidra" and "I can drive Ghidra against an unknown embedded binary" are different claims, and the gap between them is almost entirely front-loaded. Get the four load settings right and the tool does an enormous amount of work for you. Get any one of them wrong and you will spend an hour reading garbage, usually concluding that the binary is packed or encrypted when it is neither.

The good news is that each wrong setting has a **distinct symptom**. You do not have to guess. You can read the failure and know which knob you turned wrong, which is the difference between a two-minute fix and an afternoon.

The four settings

When you import a file, Ghidra asks — or infers, and you should check its inference:

1. Processor / language. Not just "ARM" but the variant: `ARM:LE:32:v7` versus `ARM:LE:32:Cortex` are different, and picking the generic ARM variant for a Cortex-M image means the analyzer expects ARM-state instructions on a part that only executes Thumb.

2. Endianness. MIPS ships big-endian and little-endian, both common in consumer routers. This is folded into the language string — `MIPS:BE:32:default` versus `MIPS:LE:32:default` — and it is the single most destructive setting to get wrong.

3. Base address. Where the image is mapped. An ELF carries this and Ghidra reads it. A **flat blob does not**, so Ghidra defaults to `0x0` and you must set it yourself — STM32 flash lives at `0x08000000`, not `0x0`. This is the defining problem of module C1.

4. Entry point. For an ELF, given. For a flat blob, you supply it — on Cortex-M, by reading the reset vector out of the vector table and clearing its Thumb bit, exactly as in A2.

Reading the failure

This table is worth memorising, because it converts an hour of confusion into a single correction.

Symptom	Cause
Uniform garbage from the very first byte	**Endianness** wrong
Plausible instructions, but every string and xref points at nothing	**Base address** wrong
Instructions decode as 32-bit ARM on an MCU image; nothing resolves	**Language variant** wrong (needs Cortex/Thumb)
Everything looks shifted by one byte	Disassembled at a **Thumb pointer** without clearing bit 0
Strings visible, but no functions created at all	No **entry point** — nothing seeded the disassembler
Functions are all `FUN_00401a20`, no library names	**Nothing is wrong.** The binary is stripped.

That last row matters as much as the others. Firmware binaries are almost always stripped, and newcomers routinely treat `FUN_` names as a loading failure and re-import repeatedly trying to make them go away. They are not going away. Recovering meaning from unnamed functions *is* the work.

Auto-analysis, and what to distrust

On import Ghidra offers to run its analyzers. Say yes — but know what they are guessing at.

The analyzers find function boundaries, follow calls, propagate types, and identify switch tables. On a well-formed ELF with symbols this is close to magic. On a stripped firmware blob it is inference, and inference is sometimes wrong:

- **Function boundaries** can be missed entirely in a flat blob, because nothing calls the function from code Ghidra has already found. If a region shows as undefined bytes, select it and disassemble manually.
- **Data misread as code.** ARM's literal pools sit *inside* functions (see A2). If the analyzer did not identify a pool, it renders constants as instructions. Sudden nonsense in the middle of an otherwise sane function usually means data.
- **Re-run after you fix something.** Analysis is not automatically retroactive. Correct a base address or define an entry point, then re-run analysis so the new information propagates.

Cross-references are the tool

If you learn one navigation habit, make it this one.

A **cross-reference** ("xref") records that one address refers to another. Ghidra builds them during analysis, and they answer the questions you actually have:

- *Who calls this function?* — xrefs to a function entry.
- *Where is this string used?* — xrefs to the string, which is how you find the code behind an error message you saw on the device.

- *What touches this global?* — xrefs to a data address, which is how you find the code behind an `nvram` key.

The core firmware workflow is xref-driven and runs **backwards from the dangerous thing**:

1. Find `strcpy`, `system`, `sprintf`, `memcpy` in the symbol tree or imports.
2. List xrefs to it — every call site.
3. For each call site, walk xrefs *up* to its callers, and keep walking.
4. Stop when you reach something attacker-reachable — a request handler, a parser, a socket read.

That backward walk from sink to source is the single most productive thing you do in Ghidra, and it is what the `fw-a5-xref-sink` task and the `fw-a5-xref-sinks` decoder automate.

Annotate as you go

An analysis you cannot hand to someone else is half an analysis, and an analysis you cannot resume in two weeks is worse. Ghidra gives you four durable annotations, and using them is a discipline rather than a feature:

- **Rename** functions and variables the moment you know what they are. `FUN_00401a20` becomes `parse_header`. Everything downstream reads better immediately, and renames propagate into the decompiler.
- **Retype** parameters and locals — the subject of the next card, and the one with the biggest payoff.
- **Comment** the reasoning, not the mechanics. `; length is attacker-controlled, from the Content-Length header` is worth ten comments saying what an instruction does.
- **Bookmark** anything you will want to come back to.

This is also what makes "documented application" a real claim rather than a line on a résumé. The artifact you produce in `fw-a5-analysis-writeup` is built from these annotations.

Scripting and headless mode

Two capabilities worth knowing exist before you need them:

- **The Script Manager** runs Java or Python scripts against the open program, with the full API: enumerate functions, walk references, read and patch bytes, apply types. Anything you find yourself doing more than five times is a script.
- **The headless analyzer** (`analyzeHeadless`) imports and analyses without the GUI, which is what you use to run the same analysis across forty firmware versions — and it is the basis of the diffing work in module E3.

```
# Import one binary, auto-analyse, keep the project for later GUI work.
# -processor is the same decision as the GUI's language field: get it wrong
# and you get a clean-looking disassembly of the wrong instruction set.
analyzeHeadless /path/to/projects MyProject \
  -import ./squashfs-root/usr/sbin/httpd \
  -processor MIPS:LE:32:default \
  -loader BinaryLoader -loader-baseAddr 0x00400000
```

```
# Run a script over every binary in an extracted filesystem, no GUI, no
# project kept. This is the shape that scales to a whole image.
analyzeHeadless /tmp/ghidra-proj Sweep \
  -import ./squashfs-root/usr/sbin/ \
  -recursive \
  -scriptPath ./scripts -postScript FindDangerousSinks.java \
  -deleteProject
```

`-deleteProject` matters more than it looks: a headless sweep over a firmware image creates a project per run, and without it you fill a disk with analysis you are never going to open.

What to carry forward

- Four settings: **language variant, endianness, base address, entry point.** Check Ghidra's inference rather than accepting it.
- Every misload has a **distinct symptom**. Read the failure, do not guess.
- `FUN_` names and no library symbols is **normal**, not a defect.
- Auto-analysis is inference. Re-run it after you correct anything.
- **Navigate by xrefs, backwards from the sink.** That is the firmware workflow.
- Annotate as you go; the annotations are the deliverable.

Next: why the decompiler's output is a model rather than the truth, and how correcting it changes what you can see.

The decompiler is a model, not the truth

Why this matters

The decompiler window is the most useful thing in Ghidra and the easiest to over-trust. It shows you C. The binary does not contain C — it contains instructions, and the C is a **reconstruction** built from assumptions the tool made because it had to make some.

Most of those assumptions are good. The ones that are wrong are wrong *quietly*: the output stays syntactically valid and plausible, and it simply describes a different program from the one on the chip. Analysts talk themselves into vulnerabilities that are not there, and past ones that are, on the strength of uncorrected pseudo-C.

The fix is not to read harder. It is to **correct the model until its output is honest**, which is a different and much more productive activity.

What the placeholders are telling you

Ghidra's default names are not noise. Each one is a specific admission of ignorance, and reading them as such is the first skill.

You see	It means
`undefined4`	"A 4-byte value. I could not infer what kind."
`undefined` / `undefined1`	A single byte of unknown type.
`param_1`, `param_2`	"Something arrives in this register or stack slot. I do not know what."
`local_c`	A stack local at offset `0xc`. The name **is** the offset.
`FUN_00401a20`	An unnamed function at that address.
`DAT_0800a1c4`	An unnamed data address.

`undefined4` in particular is worth internalising. It is not a type. It is a placeholder meaning *I know the size and nothing else*, and every one of them is an invitation.

The signature is where the leverage is

A function's signature — parameter types and return type — is the single highest-leverage thing to correct, because the decompiler propagates it. Fix the signature and the body often rewrites itself into something readable.

Here is a stripped function as the decompiler first presents it:

```
undefined4 FUN_0800a1c4(undefined4 param_1,undefined4 param_2)
{
  int local_c;

  local_c = 0;
  while (local_c < (int)param_2) {
    *(undefined1 *)(param_1 + local_c) = *(undefined1 *)(DAT_0800b100 + local_c);
    local_c = local_c + 1;
  }
  return local_c;
}
```

Readable, but working. Now read it as evidence rather than as code:

- `param_1` is only ever used as `*(undefined1 *)(param_1 + local_c)` on the **left** of an assignment. It is written through, one byte at a time. That is a **byte pointer**, and it is the destination.
- `param_2` is compared against the loop counter and is **never dereferenced**. That is a **count**, not a pointer.
- `local_c` counts up and is returned. The return value is a **count of bytes written**.

Retype `param_1` to `uint8_t *`, `param_2` to `size_t`, the return to `int`, and rename sensibly. The decompiler re-renders:

```
int copy_bytes(uint8_t *dst,size_t len)
{
  int i;

  for (i = 0; i < len; i = i + 1) {
    dst[i] = src_table[i];
  }
  return i;
}
```

Nothing about the binary changed. What changed is that the second version can be **read for bugs** — `dst[i]` with a caller-supplied `len` and no bound on `dst` is a question you can now actually ask, and the pointer-arithmetic version was hiding it in plain sight.

Structures are the same move, one level up

The single biggest readability win on firmware is defining structures.

Uncorrected, a peripheral or a header access looks like this:

```
*(undefined4 *)(param_1 + 0xc) = 0x2000;
if ((*(uint *)(param_1 + 0x18) & 0x80) != 0) { ... }
```

Those magic offsets are fields. Define a struct with `0xc` and `0x18` named, apply it to `param_1`, and you get:

```
uart->BRR = 0x2000;
if ((uart->SR & UART_SR_TXE) != 0) { ... }
```

This is exactly what **SVD-Loader** automates for microcontroller peripherals in module C2 — it reads a CMSIS-SVD file and generates these structures for you, for 650+ parts. Doing it by hand once first is what makes the automated version legible when you get there.

Where the model breaks in ways worth knowing

Four failure modes recur on embedded targets:

Calling convention. Ghidra assumes the platform default. Firmware is full of hand-written assembly, interrupt handlers, and functions reached by indirect jump, where the convention is whatever the author wanted. If arguments look implausible, check whether the convention is right before concluding the caller is doing something strange.

Stack-frame confusion. A function that adjusts `SP` dynamically, or an interrupt handler with a hardware-pushed frame, can confuse local-variable recovery. Locals appear at odd offsets or overlap.

Data as code. ARM literal pools sit inside functions (A2). If a pool was not identified, the decompiler is decompiling constants.

Missing xrefs on indirect calls. A call through a function pointer — a jump table, a vtable, an RTOS task registration — has no static target, so "who calls this" comes back empty for functions that are called constantly. An empty xref list is not proof a function is dead.

The habit

Work in a loop, and treat it as the actual job rather than preparation for it:

1. Read the body for **evidence** — how is each parameter used?
2. **Retype and rename** from that evidence.
3. Let the decompiler re-render.
4. Read again. The new rendering usually exposes the next thing to correct.
5. **Comment the reasoning**, not the mechanics, so the conclusion survives you.

Stop when the output stops changing. That is when the model matches the program, and it is the point at which a vulnerability claim is worth making.

What to carry forward

- The decompiler output is a **reconstruction under assumptions**, not the program.
- `undefined4` means "4 bytes, unknown kind" — a question, not an answer.
- **Correct the signature first**; it propagates further than anything else.
- Dereferenced means pointer; compared-against-a-counter means count. Read usage as evidence.
- Structs turn magic offsets into field names. SVD-Loader does this automatically in C2.
- An empty xref list can mean **indirect call**, not dead code.
- Re-render, re-read, repeat. Stop when the output stabilises.

Next: drill the load-failure symptoms, reconstruct a corrected declaration, and diagnose an analysis that went wrong because nobody checked all the callers.

Track B

Module B1

Recon and acquisition: knowing the target before you have the file

Why this matters

FSTM's first two stages are the cheapest in the methodology and the ones most likely to be skipped, because neither produces a finding. What they produce is a decision: **is this target worth your week?**

A device with downloadable firmware, a documented SoC, an OpenWrt port and four existing CVEs is a good first target. A device with an encrypted image, an undocumented SoC and no serial header is a research project. Both are legitimate. Knowing which one you picked before day three is the point.

Stage 1: reconnaissance

Free sources, roughly in order of value.

The FCC ID. Any device sold in the US carries one, usually on the label. Search it and you frequently get **internal photographs** — the board, top and bottom, at high resolution, with the chip markings legible. Sometimes a full test report with block diagrams. This is a teardown you

did not have to do, and it tells you the SoC, the flash part, and whether there are unpopulated header pads.

The SoC part number. Once you can read the main chip, you have the architecture — and therefore the endianness, the Ghidra language variant, and what "normal" looks like. A Broadcom BCM47xx is MIPS. A Realtek RTL83xx is MIPS. A Qualcomm IPQ40xx is ARM. Search the part number plus "datasheet"; even a marketing brief usually gives you the core.

The flash part number. Tells you the capacity — which bounds how much firmware there is — and the package. An SOIC-8 SPI flash is clip-attachable (Track F); a BGA is not, without rework.

Community documentation. The OpenWrt Table of Hardware, the router wikis, and the firmware-modding forums have often already documented the SoC, the flash layout, the serial pinout and the bootloader unlock procedure. Checking this first is not cheating; it is not repeating work.

Existing CVEs. Search the vendor and the model. This is the single best predictor of whether you will find something: a vendor with a history of unauthenticated command injection has a codebase and a review culture, and both persist. It also gives you the n-day targets for module E3.

The support site. Firmware downloads, release notes ("fixed a security issue in the web interface" is an invitation), and the changelog history that tells you which versions to diff.

Once a file is in hand, the first five minutes are always the same four commands:

```
# 1. What IS it? Frequently "data" — which is itself the answer: no container.
file fw.bin
sha256sum fw.bin | tee fw.bin.sha256      # record it now; you will need it in E3

# 2. What is inside, and at which offsets?
binwalk fw.bin

# 3. Is it compressed or encrypted? Flat-high entropy across the WHOLE file
#    means encrypted; high in one region and low elsewhere means compressed.
binwalk -E fw.bin

# 4. What does it say about itself?
strings -n 8 fw.bin | grep -iE 'linux|version|busybox|u-boot|copyright' | head -40
```

Run all four before deciding anything. `file` reporting `data` and `binwalk` reporting nothing is not a dead end — it is a specific finding that sends you to entropy, and entropy tells you whether you are looking at encryption (Track F, get the plaintext off the chip) or at a container `binwalk` does not have a signature for (read the header yourself).

Stage 2: obtaining firmware

Four routes. Try them in this order, because the cost differences are enormous.

1. Download it. The vendor's support page. Free, instant, and works more often than people expect. Grab *several* versions if they are available — version pairs are what module E3's diffing needs, and old releases vanish from support pages.

2. Intercept the update check. The device asks a server whether a newer image exists. Put yourself in the middle — DNS redirect, proxy, or just watch the traffic — and you learn the update URL, which frequently serves images that are not linked from the support page at all. Also tells you immediately whether updates are fetched over plain HTTP, which is a finding on its own.

3. Capture an OTA. Trigger an update from the device's UI and capture what it downloads. Slower, but it yields exactly the image the device accepts, including any wrapper the offline download lacks.

4. Dump it off the chip. UART, SWD, a SOIC-8 clip on the SPI flash, or chip-off. This is Track F, it needs hardware, and it is the only route that works when the vendor publishes nothing. It is also the only route that gets you the *provisioned* state — the nvram partition with real credentials in it, rather than factory defaults.

That last distinction is worth holding onto. A downloaded image contains what the vendor ships. A flash dump contains what your device has become — and the difference is where the interesting configuration lives.

What to record

Keep a target sheet from the start. It costs nothing and it is what makes an assessment reproducible:

- Vendor, model, hardware revision, region
- Firmware version and where you got it, with a SHA-256
- SoC, architecture, endianness, flash size
- Known CVEs and their versions
- Whether serial or JTAG pads are documented

The hash matters more than it looks. Vendors silently re-upload images under the same version string, and a finding that says "in firmware 1.0.4" without a hash is not reproducible.

Deciding to proceed

Reasonable stopping points, all of which are legitimate:

- **The image is encrypted and there is no obvious decryption path.** Uniform ~7.9 bits per byte entropy with no recognisable header. Sometimes the bootloader decrypts it and the key is in flash — but that is Track F, and it is a different project from the one you started.
- **No firmware is published and you have no hardware.** Nothing to analyse.
- **The device is an unmodified reference design** with an SDK everyone shipped. Interesting for breadth, and the bugs may already be public across a dozen brands.

And the signals to keep going: **downloadable image, documented architecture, existing CVE history, an OpenWrt port.** That combination is why consumer routers are the standard first target — not because they are the most interesting devices, but because every one of these stages resolves quickly on them.

Ethics, briefly

You may analyse firmware and devices **you own**. Downloading a vendor's publicly-posted image is ordinarily fine; intercepting update traffic on a network you control is fine; attacking a device you do not own is not, whatever its security posture. Module E3 covers coordinated disclosure and the research exemptions properly. Read it before you touch anything you did not buy.

What to carry forward

- Stages 1 and 2 produce a **decision**, not a finding, and skipping them costs days later.
- **FCC ID internal photos** are a free teardown; the SoC part number is your architecture.
- Acquisition order: **download → intercept → OTA capture → dump the chip.**
- Grab **multiple versions** while they exist; E3 needs pairs.
- A **flash dump gives you provisioned state**; a download gives you factory defaults.
- Record a **SHA-256** with every image, or your findings are not reproducible.

Next: what the file actually is — signatures, entropy, and headers.

Reading the image: entropy, signatures and headers

Why this matters

You have a file. Before anything else you need to answer three questions, and they have to be answered in this order:

1. **What is in it, and where?** — signature scanning
2. **Which parts are compressed, encrypted, or plain?** — entropy analysis
3. **What do the containers say about themselves?** — header parsing

Getting this wrong is not a subtle failure. It is the difference between "this firmware is encrypted, I'm stuck" and "the header says the payload starts at offset 64".

Signature scanning

`binwalk` walks the file looking for known magic values and reports what it finds and where. It is the first command you run and it usually resolves the whole structure:

```
DECIMAL     HEXADECIMAL   DESCRIPTION
--------------------------------------------------------------------------------
0           0x0           TRX firmware header, little endian, image size: 8003584
bytes
32          0x20          LZMA compressed data
1310752     0x140020      Squashfs filesystem, little endian, version 4.0, size:
6689324 bytes
```

Read it as a map: a TRX container at 0, a compressed kernel at 32, a SquashFS root filesystem at 0x140020. That is the whole layout from one command, and stage 4 now knows exactly where to cut.

Two habits worth forming:

- **Believe the offsets, question the descriptions.** `binwalk` matches short magic sequences, and short sequences occur by chance. A "PNG image" reported in the middle of a kernel is

almost certainly four coincidental bytes. Corroborate with entropy and with whether the thing extracts.
- **An empty report is information.** No signatures anywhere means either whole-image encryption or a format binwalk does not know — and entropy tells you which.

Entropy

Shannon entropy measures how unpredictable the bytes are, in **bits per byte**, on a scale from 0 to 8. Computed over a sliding window it gives you a profile across the image, and the profile has a characteristic shape for each kind of content:

Entropy (bits/byte)	What it usually is
~0	Padding — a run of one value, often `0x00` or `0xFF`
1–4	Plain text, configuration, ASCII strings
4–6	Executable code — structured, repetitive opcodes
6–7.5	Already-compressed data, or code with embedded data
~7.9–8.0, flat	**Compressed or encrypted**

The last row carries the important limitation:

> **Entropy cannot distinguish encryption from compression.** Both produce near-maximum, uniform-looking output, because both are in the business of removing redundancy.

What *does* distinguish them is **structure at the boundaries**:

- **Compressed** data is usually introduced by a header its decompressor recognises — LZMA, gzip, XZ. `binwalk` finds those. It also usually sits *inside* a region, with recognisable material before and after.
- **Encrypted** data typically has no header, starts at a suspiciously round offset, and runs to the end of the image or to an exact block boundary. A whole file at flat 7.95 with nothing recognisable anywhere is the encrypted case.

And a third tell, which is the most useful one: **the device has to decrypt it too.** If the image is encrypted, something in the bootloader or the update routine holds the key. That is a lead, not a wall — though following it usually means Track F.

Reading a profile

A typical Linux firmware image, described as a profile:

```
offset 0x000000 - 0x000040   entropy 3.1   <- header, ASCII version strings
offset 0x000040 - 0x140000   entropy 7.94  <- compressed kernel
offset 0x140000 - 0x7A0000   entropy 7.91  <- compressed SquashFS
offset 0x7A0000 - 0x800000   entropy 0.02  <- padding to the erase block
```

Four regions, and every one of them is actionable. The low-entropy head is a header worth parsing. The two high-entropy regions are compressed, not encrypted, because `binwalk` found LZMA and SquashFS magic at their starts. The zero-entropy tail is padding, which tells you the real image is smaller than the file — and where the flash erase-block boundary falls.

Contrast that with the encrypted case:

```
offset 0x000000 - 0x800000   entropy 7.95  <- uniform, no signatures anywhere
```

One region, no structure, no header. That is stage 3 telling you to go find the decryption routine.

Headers

Containers describe themselves, and the fields are useful rather than decorative.

uImage — U-Boot's kernel wrapper, 64 bytes, **big-endian regardless of the target's endianness**:

Offset	Size	Field
0	4	magic, `0x27051956`
4	4	header CRC
8	4	timestamp
12	4	payload size
16	4	**load address**
20	4	**entry point**
24	4	payload CRC
28	1	OS
29	1	architecture
30	1	image type
31	1	compression
32	32	image name, NUL-padded

The load address and entry point are the two that matter later: they are exactly what you type into Ghidra when you load the kernel, so parsing this header is what stops the kernel being another flat blob with a guessed base.

The big-endian detail catches people. uImage headers are always big-endian — a design decision from U-Boot — so a little-endian MIPS device still has a big-endian header in front of its little-endian kernel.

TRX — a Broadcom-lineage container, `HDR0`, little-endian, carrying offsets to each partition inside it. Its whole purpose is to tell you where the kernel ends and the rootfs begins.

Vendor headers are frequently neither. A proprietary wrapper with a version string, a checksum and a length is common, and reverse-engineering it is ordinary work: find the length field by correlating it with the file size, find the checksum by changing a byte and seeing what the device rejects.

Checksums are not signatures

Both headers above carry a CRC. A CRC detects corruption; it does not detect tampering, because anyone can recompute it. If a vendor's update routine checks only a CRC, modified firmware installs — which is a finding, and it is module E2's territory.

Worth keeping straight now: **CRC is integrity against accident. A signature is integrity against an adversary.** Reporting a CRC as though it were a signature, or the reverse, undermines the rest of the report.

What to carry forward

- Order: **signatures → entropy → headers.** Each informs the next.
- Believe `binwalk`'s **offsets**, corroborate its **descriptions**.
- Entropy is **bits per byte, 0 to 8**; ~7.9 flat means compressed *or* encrypted.
- **Entropy cannot tell those apart** — boundary structure and the presence of a decompressor header can.
- If it is encrypted, **the device decrypts it too**; find that routine.
- **uImage headers are big-endian always**, and their load address and entry point are what you give Ghidra.
- **A CRC is not a signature.**

Next: drill the entropy readings, order the identification workflow, and write the two tools — a sliding-window entropy scan and a uImage parser.

Module B2

Extraction: getting the filesystem out

Why this matters

Stage 4 is the shortest stage in FSTM because on a well-behaved image it is genuinely one command:

```
$ binwalk -e firmware.bin
```

You get a directory tree and stage 5 begins. That happens often enough that the interesting content of this card is the other cases — and the more interesting failure, which is an extraction that *appears* to work and quietly gives you half the filesystem.

The happy path, and what it actually did

`binwalk -e` walks the signature list from stage 3, and for each recognised region invokes the matching extractor: `unsquashfs` for SquashFS, `jefferson` for JFFS2, `ubireader` for UBI, a decompressor for LZMA or gzip. The result lands in `_firmware.bin.extracted/`.

Two things worth knowing about that output:

- **The directory names are offsets.** `140020.squashfs` came from `0x140020`. That correspondence is how you tie an extracted tree back to a location in the image, which matters when you later want to modify and repack it.
- **Extraction is recursive.** binwalk re-scans what it extracted, so a compressed blob inside a filesystem inside a container comes out in one pass. Useful, and occasionally it recurses into something enormous — the `--depth` limit exists for that.

Doing it by hand

You will need to cut manually more often than the tutorials suggest — when binwalk lacks an extractor, when the vendor's container confuses it, or when you want exactly one region.

```
$ dd if=firmware.bin of=rootfs.squashfs bs=1 skip=1310752
$ unsquashfs rootfs.squashfs
```

The offset comes from stage 3. `bs=1` is slow on a large image but unambiguous, which is the right trade when you are doing it once.

This is why the `fw-b2-squashfs-offset` task exists: the whole manual path depends on knowing where the filesystem starts, and finding that yourself — rather than trusting a signature scan that may have matched coincidentally — is the skill.

When it does not work

No extractor for the type. binwalk identifies the filesystem and cannot unpack it. Install the tool it wants: `jefferson` for JFFS2, `ubi_reader` for UBI, `cramfsswap` for some CramFS variants. A "cannot extract" message is a missing dependency far more often than an exotic format.

A vendor container binwalk does not know. The image starts with a proprietary header and binwalk finds nothing until well past it. Parse the header yourself (stage 3, module B1), find the payload offset, cut, and re-scan the payload. Vendor headers are usually simple — magic, version, length, checksum — and correlating the length field with the file size identifies it quickly.

Non-standard compression. Some vendors ship SquashFS with a patched LZMA that stock `unsquashfs` rejects. The `firmware-mod-kit` bundles the older `unsquashfs` builds that handle these. The symptom is characteristic: the superblock parses, the file count looks right, and decompression fails.

Encryption. Nothing to cut. Back to stage 3 and the search for the decryption routine.

The failure that looks like success

This is the one to internalise, because nothing warns you:

> **A partial extraction produces a real directory tree with real files in it.**

If `unsquashfs` hits an error partway, or binwalk's recursion stopped early, or the image was truncated on download, you get a tree that looks entirely plausible and is missing things. You then grep it, find nothing, and conclude the device is clean.

Sanity-check every extraction against expectations:

- **Is there a `/bin/busybox`?** Almost every embedded Linux root filesystem has one. Its absence means you have a fragment, not a root.
- **Are `/etc`, `/bin`, `/sbin`, `/usr`, `/www` all present?** A root filesystem missing `/etc` is not a root filesystem.
- **Does the extracted size roughly match the header's stated size?** An order of magnitude short is a truncated extraction.
- **Did the tool report errors?** Read the output rather than only checking that a directory appeared.
- **Are symlinks intact?** Extracting as an unprivileged user can mangle device nodes and ownership; that is usually harmless for analysis but worth knowing when something looks odd.

The `/bin/busybox` check is the cheapest and catches most of it.

Multiple filesystems

Devices frequently carry more than one, and they are not interchangeable:

- **The root filesystem** — read-only SquashFS, the vendor's code. Where the bugs are.
- **An overlay** — writable JFFS2 or UBIFS holding configuration and anything the user changed. On a downloaded image this is usually empty or absent; on a **flash dump** it is populated, and that is where provisioned credentials live (module B1).
- **A recovery or factory partition** — a second copy, sometimes an older version. Worth extracting: an older version is a free diff candidate for module E3.

Extract all of them. The overlay is small and it is where the interesting configuration is.

Repacking, briefly

Extraction's inverse matters for module E2. If you can modify the filesystem, repack the image and have the device accept it, you have demonstrated that the update mechanism does not verify authenticity — which is a finding regardless of what you changed. `firmware-mod-kit` exists largely for this, and the checksum you must recompute is the one from B1's header parsing.

What to carry forward

- `binwalk -e` handles the common case; **extracted directory names are offsets**.
- Manual extraction is `dd` at the offset plus the right unpacker, and it needs the **offset from stage 3**.
- "Cannot extract" is usually a **missing dependency**, not an exotic format.
- **A partial extraction looks exactly like a successful one.** Check for `/bin/busybox`, `/etc`, and a plausible total size, and read the tool's errors.
- **Extract every filesystem**, especially the writable overlay — that is where provisioned configuration lives.

Next: what to look for once the tree is in front of you.

Static analysis: what to hunt in the extracted tree

Why this matters
This is the highest-yield stage in the methodology. On a typical consumer device, stage 5 produces more findings per hour than everything else combined, and it needs no debugger, no emulator and no exploitation skill — just a filesystem and an ordered list of places to look.

It is also the stage people skip to get to the interesting part. Module A0 made that point; this card is the actual checklist.

The order to work in
Cheapest and highest-yield first.

1. Run `firmwalker`. A shell script that greps for the obvious: password files, private keys, certificates, URLs, `nvram` calls, dangerous binaries. Seconds to run, and it gives you a floor to work up from. It is not a substitute for looking; it is a way to not miss the obvious while you do.

2. Read the startup scripts. `/etc/init.d/`, `/etc/rc.d/`, `/etc/inittab`. Plain text, short, and they enumerate everything the device runs as root (module A1). Grep them for `eval`, `sh -c` and backticks before anything else.

3. Credentials. `/etc/passwd` and `/etc/shadow`. Look for accounts with a hash rather than `*` or `!` in the password field, accounts with UID 0 that are not `root`, and accounts with a login shell that should not have one. Then crack what you find — embedded hashes are frequently DES or MD5-crypt and fall in seconds.

4. Keys and certificates. `*.pem`, `*.key`, `*.crt`, `id_rsa`, and anything with `BEGIN PRIVATE KEY` in it. A private key shipped in firmware is the same key on every device of that model, which is what makes it a finding rather than a curiosity.

5. Configuration defaults. `/etc/nvram.default`, `/etc/defaults/`, `/etc/config/`. Default credentials, enabled-by-default services, debug flags left on.

6. The web root. `/www`, `/web`, `/htdocs`. The CGI handlers, and any file that should not be served — a backup, a `.git` directory, a config file with credentials in it.

7. Binary permissions. SUID and SGID binaries (`find . -perm -4000`), world-writable files, and anything writable that is also executed by a root-owned script.

8. Versions. Every third-party component with a version string, checked against known CVEs. BusyBox, dropbear, openssl, the web server, the kernel. This is where an automated tool earns its place.

Grep-craft
Some patterns that pay off, and what each is actually looking for:

```
grep -rIn --exclude-dir=proc -E \
  '(password|passwd|pwd|secret|token|api[_-]?key)[[:space:]]*[=:]' .
grep -rIn -E 'BEGIN (RSA |EC |OPENSSH )?PRIVATE KEY' .
grep -rIn -E '\b(eval|sh -c)\b' etc/ usr/
grep -rIn -E 'nvram[ _](get|set)' .
grep -rIn -E 'https?://[a-zA-Z0-9.-]+' etc/ usr/sbin/
```

`-I` skips binary files, which is what keeps the output readable. Drop it deliberately when you *want* strings out of binaries, and pair it with `strings` for anything interesting.

The URL grep is underrated: hardcoded update servers, telemetry endpoints and vendor APIs all turn up, and each one is both a finding candidate and a lead for module B1′s update-interception route.

Is it actually reachable?
This is the discipline that separates a finding from a line of grep output. For every secret you find, ask three questions:

Is it used? A key in `/etc/ssl/private/` referenced by nothing may be a leftover from the vendor's build. Grep for its filename across the tree. If nothing references it, say so in the writeup — it is still a finding, with a lower severity and an honest caveat.

Is it reachable by an attacker? A root password hash matters if `dropbear` is enabled and listening. If SSH is compiled out and the only login is a serial console, the same hash is a much smaller finding — and Track F is what makes it a big one again.

Is it the same on every device? This is the multiplier. A key baked into the firmware image is shared by every unit of that model; a key generated at first boot is not. The difference is usually visible in the startup scripts: if something calls `ssh-keygen` on first boot, the shipped key is a placeholder.

Answering these is what makes a report actionable instead of a list.

Automation, and its limits
EMBA does the heavy version-and-CVE work: it builds an inventory of components, matches them against CVE databases, checks configurations, and produces a substantial report. On a large image it finds things no manual pass would.

It also produces false positives in volume, and the reasons are structural:

- **Version strings lie.** A vendor backports a patch without bumping the version, so a matched CVE may already be fixed. Or they bump the version and change nothing.
- **Reachability is not assessed.** A CVE in a library that is present but never called is not a vulnerability on this device.
- **Config checks are generic.** A "weak permission" flagged on a file nothing executes is noise.

So treat automated output as **leads to verify**, never as findings to report. A report that forwards an EMBA dump unverified is worse than a short report of three confirmed issues — it transfers the verification work to the reader, who has less context than you.

What good output looks like

A stage-5 finding that is worth writing up has: the file and line, what the thing is, whether it is referenced and by what, whether an attacker can reach the path that uses it, whether it is shared across devices, and — where you could not establish something — a plain statement that you could not.

That last part is not hedging. "The key is present and I could find no code referencing it" is a more useful sentence than either "unused key found" or "hardcoded private key allows impersonation", because it says exactly what is known.

What to carry forward

- Stage 5 is the **highest-yield stage**; work it before reaching for a debugger.
- Order: **firmwalker → startup scripts → credentials → keys → defaults → web root → permissions → versions.**
- `grep -rIn` with `-I` to skip binaries; drop `-I` deliberately and pair with `strings`.
- For every secret: **is it used, is it reachable, is it shared across devices?**
- **EMBA output is leads, not findings.** Version strings lie and reachability is unassessed.
- Say plainly what you could not establish. It is more useful than a confident guess.

Next: drill the findings, order the extraction, and write the two tools — a SquashFS locator and a credential scanner that does not drown you in false positives.

Module B3

User-mode emulation: running one foreign binary

Why this matters

Everything so far has been static. You have read the code; you have not watched it run. Stage 6 is where that changes, and it is the gate on stages 7 and 8 — you cannot fuzz, debug or exercise a service that will not start.

The cheapest form is user-mode emulation: run **one binary** on your x86 laptop as though it were a MIPS or ARM process. No kernel, no boot, no device. Just the program.

The mechanism

`qemu-user` translates the target's instructions to yours and forwards its syscalls to your kernel. A MIPS `httpd` makes a MIPS `open()` syscall; QEMU catches it and makes an x86 `open()` on its behalf.

That means user-mode emulation gives you the **CPU and the syscall interface**, and nothing else. It is enough for a startling amount of firmware, because most vendor binaries are ordinary Linux programs that read files, bind sockets and parse input.

```
$ file usr/sbin/httpd
ELF 32-bit MSB executable, MIPS, MIPS32 ... dynamically linked
```

```
$ qemu-mips usr/sbin/httpd
/lib/ld-uClibc.so.0: No such file or directory
```

That error is the normal first result, and it is the whole practical content of this card.

The library problem
The binary is dynamically linked against libraries that exist in the **extracted filesystem**, not on your host. QEMU looked in your `/lib` and found nothing.

Two ways to fix it, and the difference matters:

`-L` or `QEMU_LD_PREFIX` points QEMU at the extracted root:

```
$ qemu-mips -L ./squashfs-root usr/sbin/httpd
```

Paths get resolved against that prefix. Simplest, and it works when the binary only needs libraries.

`chroot` with a static QEMU is the fuller version:

```
$ cp $(which qemu-mips-static) ./squashfs-root/
$ sudo chroot ./squashfs-root ./qemu-mips-static /usr/sbin/httpd
```

Copying a **statically linked** QEMU inside the extracted root is what makes this work — a dynamic one would need its own host libraries, which are not in there. Now the binary sees the firmware's filesystem as `/`, so absolute paths in its configuration resolve correctly.

Reach for chroot when the binary reads its own config files, writes to `/var`, or execs helpers. Reach for `-L` when it just needs libc.

What it cannot reproduce
User-mode emulation gives you the CPU and syscalls. Everything else is missing, and each gap has a characteristic symptom:

nvram. The binary calls the vendor's `nvram_get` and receives nothing, so it takes a configuration-missing path or crashes on a NULL. The fix is to supply the values — module A1 gave you the format, and `fw-b3-nvram-stub` builds the responder.

Hardware devices. Opens of `/dev/mtd0`, `/dev/gpio`, a proprietary driver node — none exist. Some can be faked with a regular file; some cannot.

Kernel modules. Vendor kernel modules are not loaded and cannot be. A binary depending on a custom `ioctl` will fail at that call.

The network environment. The binary may expect interfaces named `br0` or `eth0.1` that your host does not have.

Other processes. A daemon that talks to another vendor process over a socket or shared memory has nobody to talk to.

When enough of these bite at once, single-binary emulation stops paying and you move up to full-system — the next card.

Getting useful work out of it

Once it runs, the payoff is immediate:

- **Exercise the parser directly.** Feed it input on stdin, as an argument, or over a socket, and watch what it does. That is stage 7.
- **Attach a debugger.** `qemu-mips -g 1234` waits for `gdb-multiarch` to connect. That is stage 8, and it is the same setup module B4 uses.
- **Fuzz it.** AFL++ QEMU mode is this, instrumented — module E1.

The reason to fight for a running binary is that all three of those become available at once.

What to carry forward

- `qemu-user` gives you **the CPU and the syscall interface**; nothing else.
- The first error is always **missing libraries** — fix with `-L` / `QEMU_LD_PREFIX`, or `chroot` with a **statically linked** QEMU inside the extracted root.
- Missing **nvram, device nodes, kernel modules, interfaces and sibling processes** are the four things that break it, and each has a distinct symptom.
- A running binary unlocks **stage 7, stage 8 and fuzzing** simultaneously.

Next: when one binary is not enough, and how full-system emulation fails.

Full-system emulation, and why it fails

Why this matters

Single-binary emulation runs the web server. It does not run the *device*: no init, no nvram, no network interfaces, no sibling daemons, no startup scripts configuring any of it.

For a lot of firmware that is fine. For the rest, you need the whole thing booted — and booting a router image on a laptop is genuinely hard, which is why a tool exists to do it.

FirmAE

FirmAE (the maintained successor to **Firmadyne**) takes a firmware image and tries to boot it under QEMU system emulation with a generic kernel, then reports whether the web interface came up.

Its useful property is that it **automates the workarounds** rather than requiring you to know them. Firmadyne's original research found that most images fail to boot for a small number of recurring reasons, and FirmAE's "arbitration" applies fixes for each. Understanding what those fixes are is what lets you diagnose the cases it still cannot handle.

A run proceeds in stages, and knowing them is how you read a failure:

1. **Extract** the filesystem from the image.
2. **Identify the architecture** — MIPS or ARM, and the endianness.
3. **Infer the network configuration** — which interface, which address, by watching what the firmware tries to configure during a first boot.
4. **Boot** with an inferred configuration and check whether the web interface responds.

A failure at stage 1 is an extraction problem you already know how to solve (module B2). A failure at stage 3 or 4 is emulation-specific, and that is what this card is about.

```
# FirmAE, from its checkout. The flags select which of the four stages to run.
sudo ./download.sh && sudo ./install.sh     # once, per machine
sudo ./init.sh                              # start the postgres FirmAE needs

sudo ./run.sh -c <brand> ./images/fw.bin    # check: boot and probe the web UI
sudo ./run.sh -a <brand> ./images/fw.bin    # analyse: boot and leave it running
sudo ./run.sh -d <brand> ./images/fw.bin    # debug: boot with gdbserver attached

# Where a run leaves its evidence. Read these before concluding anything.
cat  ./scratch/<iid>/qemu.final.serial.log  # the boot log — start here
cat  ./scratch/<iid>/*.log                  # arbitration decisions
ls   ./scratch/<iid>/image/                 # the rebuilt filesystem it booted
```

`<iid>` is the numeric image id FirmAE prints as it starts; everything it learned about your image lives under that directory. The serial log is the single most useful artifact in the tree — a failure at stage 3 or 4 is almost always legible in its last twenty lines.

If FirmAE cannot boot it at all and you only need one binary exercised, drop to user-mode QEMU instead of fighting the whole system:

```
# Run one extracted binary with the image's own libraries.
qemu-mipsel-static -L ./squashfs-root ./squashfs-root/usr/sbin/httpd

# Same, but wait for a debugger on :1234 before the first instruction.
qemu-mipsel-static -L ./squashfs-root -g 1234 ./squashfs-root/usr/sbin/httpd
```

Why images fail to boot

nvram. The single biggest cause. Startup scripts read configuration through the vendor's nvram helper, get nothing, and either take a failure path or crash. FirmAE ships a library that intercepts those calls and answers them from a default set — and when a specific key is missing, the fix is to add it. Module A1 gave you the format; `fw-b3-nvram-stub` builds the responder.

Missing hardware. The firmware opens `/dev/mtd0` to read a partition, or talks to a switch chip over a proprietary interface, and the device node is not there. Often survivable — a regular file at the right path is sometimes enough.

Kernel version mismatch. The image expects a 2.6 vendor kernel with custom patches; the emulator provides a generic one. Vendor kernel modules will not load. If the daemon you care about depends on one, this is where you stop.

Network interface names. Startup scripts configure `br0`, `eth0.1`, `vlan1`. If those do not exist the configuration fails silently and nothing binds. This is what stage 3 exists to infer.

A watchdog. The firmware kicks a hardware watchdog that is not there, so the system reboots every few seconds. Disabling it is a standard first move.

Reading a failed run

FirmAE tells you which stage it reached. Match it to the cause:

Symptom	Where to look
Extraction failed	Module B2 — a nested or unsupported format
Architecture not identified	Check the binaries yourself with `readelf` (module A1)
Boots, no interface comes up	Network inference failed — check the startup scripts for interface names
Boots, web server not listening	The daemon crashed. Get its console output; nvram is the usual cause
Boots and immediately reboots	Watchdog
Kernel panic early	Kernel mismatch. Often terminal

The habit worth forming: **read the boot console before changing anything**. Almost every one of these announces itself in the log, and guessing at fixes without reading it is how an afternoon disappears.

When to stop

Full-system emulation is not always worth it, and knowing when to abandon it is a real skill:

- **If the bug you want is in one binary's parser**, go back to user-mode. It is faster, more reliable, and it is what the fuzzer wants anyway (module E1).
- **If the daemon needs a vendor kernel module**, the emulator cannot help.
- **If you have the physical device**, running the real thing beats emulating it. That is Track F, and a device with a UART console gives you the boot log emulation was approximating.

Full-system emulation earns its cost when you need the **whole stack** — the web interface talking to the daemons talking to the configuration — because that is where the interesting authentication and authorisation logic lives.

What it unlocks

A booted image gives you the device's own web interface on a local address. From there:

- **Stage 7** — exercise it. The web-app skills from topic #1 transfer directly, and this is where they pay.
- **Stage 8** — attach a debugger to a process inside the emulated system.
- **N-day reproduction** — module E3′s work needs a running target to demonstrate against.

What to carry forward

- Full-system emulation runs **the device**, not one binary; FirmAE automates the workarounds.
- The stages are **extract → identify arch → infer network → boot**, and a failure names its stage.

- The recurring causes are **nvram, missing devices, kernel mismatch, interface names, watchdog**.
- **Read the boot console first.** Most failures announce themselves.
- Go back to **user-mode** if the bug is in one parser; stop entirely if a vendor kernel module is required.

Next: drill the failure symptoms, order a user-mode setup, and build the nvram stub that fixes the most common one.

Module B4

Dynamic analysis: exercising what you emulated

Why this matters
Stage 6 got something running. Stage 7 is where you push on it — and where the candidates you found by reading (module A1, module B2) become confirmed findings with a reproduction attached.

The techniques here are largely borrowed. Testing a device's web interface is testing a web application, and topic #1's skills apply directly. What is firmware-specific is **which surfaces exist** and **which ones nobody has ever tested**.

The surfaces
A consumer device typically exposes more than its web UI, and the difference in scrutiny between them is where the findings are.

The web interface. Well-trodden, and still productive: the vendor's own CGI handlers are custom C, and the input validation is usually thinner than the login page suggests.

The API behind the UI. Modern devices drive their interface from JSON or XML endpoints. Those endpoints frequently accept more than the UI sends — extra parameters, additional actions, different verbs — because they were written to serve a family of products.

UPnP / SSDP. Enabled by default on a great many routers, unauthenticated by design, and almost never tested. `AddPortMapping` handlers have a long CVE history.

Custom TCP/UDP services. Look at what listens. Vendors ship management daemons, cloud-tunnel clients and discovery protocols on nonstandard ports. These are the least-reviewed code on the device.

Telnet / SSH / debug consoles. Sometimes present, sometimes enabled by an undocumented request, sometimes reachable only after a magic packet — all of which the startup scripts told you about in module A1.

The prioritisation rule is the same as everywhere else in this course: **start with what receives untrusted input and has been read by the fewest people.** That is almost never the login page.

Working from static to dynamic
The productive loop runs in one direction:

1. **Static analysis gave you a candidate** — a `system()` reachable from a form field, a `memcpy` with a length from the request, an `eval` on an nvram value.
2. **Find the request that reaches it.** Which endpoint, which parameter. The CGI handler's `get_param` calls name them.
3. **Send it and observe.** A sleep is the classic proof for injection: if `; sleep 10` makes the response take ten seconds, the command ran.
4. **Escalate to something demonstrable** — a file written, a connection out, a shell.

Step 3 matters more than it looks. **A time delay is a demonstration; a guess is not.** It is also non-destructive, which is what you want on a device you may still need working.

```
# The injection probe. Ten seconds of latency is the proof; nothing is written.
time curl -s -X POST 'http://192.168.0.1/cgi-bin/ping.cgi' \
  --data-urlencode 'host=127.0.0.1; sleep 10'

# The overflow probe. Length first, payload later.
curl -s -X POST 'http://192.168.0.1/cgi-bin/ping.cgi' \
  --data-urlencode "host=$(python3 -c 'print("A"*2000)')"
```

Attaching a debugger to the process, once you know which request reaches the handler:

```
# On the target (or inside the emulated image):
gdbserver 0.0.0.0:1234 --attach $(pidof httpd)

# On your machine. `set sysroot` is what makes libc symbols resolve.
gdb-multiarch ./squashfs-root/usr/sbin/httpd
  (gdb) set architecture mips
  (gdb) set sysroot ./squashfs-root
  (gdb) target remote 192.168.0.1:1234
  (gdb) continue                    # now send the request

# At the crash, the three things worth recording before anything else:
  (gdb) info registers ra sp a0     # MIPS  — did $ra come from your bytes?
  (gdb) info registers lr sp r0     # ARM
  (gdb) x/64wx $sp
  (gdb) info proc mappings          # the map module B4's parser consumes
```

`set sysroot` before `target remote`, not after — gdb resolves the shared-library list at attach time, and a sysroot set afterwards leaves every libc frame as a bare address.

Practical points that are firmware-specific

Authentication is often theatre. Devices ship with an admin account, and many endpoints do not check the session at all — they were written assuming the UI would only call them after login. Testing every endpoint unauthenticated is cheap and finds a surprising amount.

Watch the console. Under emulation you have the boot console; on real hardware you have UART (Track F). A crash message, a stack trace or a "config error" line as you send a request tells you far more than the HTTP response does.

Rate limits and lockouts usually do not exist. Which makes credential and parameter fuzzing practical in ways it is not against a web service.

The device may brick. On real hardware, be careful with anything that writes flash — an update endpoint, a factory-reset handler, an nvram commit. Under emulation, none of this matters and you can be as reckless as you like, which is a reason to do the destructive testing there first.

What stage 7 produces

A stage-7 finding is a stage-5 candidate with a reproduction attached: the request that triggers it, the observed effect, and the conditions required — authenticated or not, which model, which firmware version.

That last part is what makes it a report rather than an anecdote, and it is the same discipline module E3 applies to n-day reproduction.

What to carry forward

- The techniques are **borrowed from web testing**; what is firmware-specific is which surfaces exist.
- **UPnP, custom daemons and the API behind the UI** are less reviewed than the login page.
- Work **static candidate → the request that reaches it → observe → escalate**.
- **A time delay is a demonstration.** A guess is not.
- Authentication is frequently unenforced per-endpoint; test everything unauthenticated.
- **Watch the console**, not just the HTTP response.
- Do the destructive testing **under emulation**, not on hardware you need.

Next: when observing the response is not enough, and you attach a debugger.

Runtime analysis: gdb-multiarch on a foreign binary

Why this matters

Stage 7 tells you the daemon crashed. Stage 8 tells you **why**, and — far more importantly — **whether the crash is worth anything**.

That distinction is the whole value of this stage. A crash where the program counter came from your input is a path to control. A null-pointer dereference is a denial of service you will never turn into anything. They look identical from outside the process, and they take about ten seconds to tell apart with a debugger attached.

Attaching

`qemu-user` speaks the GDB remote protocol. Start the target with `-g` and it waits:

```
$ qemu-mips -L ./squashfs-root -g 1234 usr/sbin/httpd
```

Then, from another terminal:

```
$ gdb-multiarch ./squashfs-root/usr/sbin/httpd
(gdb) set architecture mips
```

```
(gdb) target remote localhost:1234
(gdb) continue
```

`gdb-multiarch` is ordinary GDB built with every architecture compiled in — the plain `gdb` on your host only knows x86 and will connect and then print nonsense. **Set the architecture before connecting**, because GDB will otherwise try to infer it and can guess wrong on a stripped binary. That single mistake produces a session where every register value looks like garbage, and it reads like a broken target rather than a misconfigured debugger.

For full-system emulation the mechanism differs — you attach `gdbserver` inside the emulated system, or use QEMU's own `-s -S` — but the debugger side is identical.

Reading a crash

When the target faults, the register dump answers the question. On MIPS:

```
(gdb) info registers
       ra   0x41414141   1094795585
       sp   0x7fff2c30   2147474992
       pc   0x41414140   1094795584
```

Read it in this order:

`pc` first. If it holds your data — `0x41414141`, or a value from your cyclic pattern — the program counter came from your input. That is control of execution, and it is the finding.

`ra` next. On MIPS this is where control was *about* to come from. `$ra` holding your bytes means the saved return address was overwritten, which is the module A4 primitive arriving from a real target.

The faulting address. A read or write at `0x00000000` is a null dereference — file it and move on. A fault at `0x41414149` is a pointer built from your input, which is an arbitrary-read primitive and much more interesting.

`sp`. Tells you where the stack is, which is what you need to compute the offset.

Then take the value out of `$ra` and hand it to `fw-a4-find-offset`. The cyclic pattern turns it into an exact byte count, and you have gone from "it crashes" to "byte 68 controls the return address" in about a minute.

The commands that carry the work

A small set does most of it:

Command	What it gives you
`info registers`	The crash state. First thing, every time.
`x/32wx $sp`	The stack around the crash — find your pattern in it
`break *0x00401a20`	Stop before the interesting call, not after the crash
`x/s $a0`	What was actually passed as the first argument
`bt`	A backtrace, when the frame is intact enough to walk
`stepi` / `nexti`	Single instructions — how you watch a copy overrun in real time

`x/s $a0` deserves the emphasis. Breaking on a `strcpy` and printing its arguments answers "is this input attacker-controlled" directly, which is the question static analysis left open in module A5's xref walk.

Breakpoints beat crashes

The instinct is to crash the target and look at the wreckage. Often better: **break before the dangerous call and watch what arrives**.

Set a breakpoint on the `strcpy` you found statically, send a normal request, and print the arguments. You learn the destination buffer's address, the source, and the length — without any crash at all. Then send a long one and single-step through the copy watching the saved return address get overwritten.

That is a more informative and more reproducible experiment than a stack trace after the fact, and it works on bugs that do not crash cleanly.

What this stage produces

A stage-8 result is a crash with an interpretation: which register was controlled, at what offset, and what that gives an attacker. That is the input module B5 needs — you cannot write an exploit against "it crashes".

It is also exactly what a fuzzing campaign hands you in bulk (module E1), which is why the triage skill here and there is the same skill.

What to carry forward

- **`gdb-multiarch`**, and **set the architecture before connecting** — a wrong guess reads like a broken target.
- `qemu-user -g <port>` makes the target wait for you.
- Read the crash in order: **`pc`**, **`ra`**, **the faulting address**, **`sp`**.
- **`pc` or `$ra` holding your bytes is the finding.** A fault at zero is a DoS.

- Feed the faulting value to the **cyclic-pattern tool** for an exact offset.
- **Break before the dangerous call** rather than only examining the crash.

Next: drill the commands, read a crash transcript, and build the tool that says which mapping an address belongs to.

Module B5

The firmware vulnerability classes, ranked by what they cost you

Why this matters
FSTM stage 9 is titled "binary exploitation", and the honest framing is that **most firmware findings never get there**. The classes below are ordered by what they cost to turn into a working result, and the cheapest ones dominate real assessments.

That ordering is a working tool, not a disclaimer. When you have a device and a week, you spend the first day on the classes that need no exploitation at all, and the rest on memory corruption only if the first day came up empty or the target demands it.

The classes

Command injection — cheapest, highest yield
Attacker input reaches something that *parses* it: `system()`, `popen()`, `eval`, `sh -c`. No memory-corruption knowledge, no offsets, no architecture. Send `; telnetd -l /bin/sh` and you have a shell as root.

In decompiled code: a `sprintf` or `snprintf` building a string from a parameter, immediately followed by `system(cmd)`. Module A1 taught the sinks; module A5′s xref walk finds them.

The distinction that matters: reaching a command line is not enough. Something must re-parse the assembled string. In shell that means `eval`, `sh -c` or backticks — a plain expansion is not re-parsed. In C, `system()` hands its argument to `/bin/sh` precisely so that it will be.

Hardcoded credentials and keys — free
No exploitation at all. You read them out of the filesystem in stage 5. A backdoor account, a shipped TLS private key, a shared HMAC secret.

The severity question is reachability and sharing: is the account's service enabled, and is the key the same on every device of that model? Module B2 covered both.

Authentication bypass — cheap
An endpoint that does not check the session, a token that is a predictable serial number, a "local network only" check made on a header the client controls. Common because the vendor assumed the UI would only call the endpoint after login.

In decompiled code: a handler with no call to whatever the session-check function is. Find one endpoint that *does* check, learn the function's name, then find every caller that skips it.

Path traversal — cheap
`../` in a filename parameter, reaching `/etc/shadow` or the nvram partition. Also the write direction, which is worse: a file upload with a controlled path overwrites a startup script and gives you code execution at next boot.

Insecure update mechanism — moderate
The device accepts firmware it should not: no signature, a CRC treated as authentication, a signature verified *after* writing, or a rollback to a vulnerable version. Module E2 covers this properly. It is moderate cost because you must build a modified image and confirm it installs.

Memory corruption — most expensive, and the reason this track exists
A stack overflow, a heap overflow, an integer overflow feeding an allocation. Needs the architecture (A2, A3), the frame arithmetic (A4), a debugger (B4) and usually a chain (below).

Expensive to weaponise, and it is the class that gets you into code the other classes cannot reach: a bug in a binary network protocol, in a parser reached before authentication, in the kernel. It is also what the role you are training for is largely hiring for, which is why the course invests in it despite being honest about the cost.

Which to pursue
A rough decision procedure:

1. **Did stage 5 find credentials or keys?** Report them. Free.
2. **Is there a `system()` reachable from a request?** Pursue it. Cheapest path to root.
3. **Does any endpoint skip the session check?** Cheap, and it widens what the other classes can reach.
4. **Do you need pre-auth code execution on a binary protocol?** Now memory corruption, and now the rest of this module.

The mistake this ordering prevents is spending three days on a stack overflow in a post-authentication CGI handler while an unauthenticated command injection sits in the UPnP daemon. Both are findings; one cost an afternoon.

Reading them in decompiled output
Module A5 taught you to correct the decompiler until its output is honest. These are the shapes you are correcting it in order to see:

Class	What it looks like
Command injection	string built from a parameter, then `system` / `popen`
Auth bypass	a handler with no session check, next to one that has it
Path traversal	a parameter concatenated into a path with no canonicalisation
Stack overflow	`strcpy` / `sprintf` / `memcpy` into a fixed local, length from input
Heap overflow	`malloc` with a computed size, then a copy that exceeds it
Integer overflow	a length arithmetic that can wrap before reaching `malloc`

The last two are worth watching for specifically because they are easy to miss when you are scanning for `strcpy`: the dangerous operation is the *arithmetic*, and the copy that follows may look perfectly bounded.

What to carry forward

- Ordered by cost: **credentials → command injection → auth bypass → traversal → update → memory corruption.**
- **Most firmware findings never reach stage 9**, and that is a fact about the field rather than a shortcoming.
- Command injection needs something that **parses** the string, not merely a command line.
- Memory corruption is the expensive class and the one that reaches **pre-auth binary protocols**.
- Watch for **integer overflow before an allocation** — the copy afterwards can look bounded.

Next: what makes MIPS exploitation specifically different, and the DVRF pwnables.

MIPS exploitation in practice: DVRF and the chain

Why this matters

Every piece is already in place. Module A3 gave you `$ra` and the delay slot. A4 gave you the offset arithmetic and the mitigation reality. B4 gave you the crash and the debugger. This card assembles them, and names the one thing that makes a MIPS exploit look different from every tutorial you have read.

The target

DVRF — Damn Vulnerable Router Firmware — is the IoTGoat of exploitation: a deliberately vulnerable MIPS firmware whose `pwnable/` directory contains a graded series of binaries. It is the standard practice ground, and its difficulty curve is deliberate:

- **`stack_bof_01`** — a plain overflow. Control `$ra`, return to a function already in the binary.

- **`stack_bof_02`** — the same with less room, forcing you to be precise.
- **The ROP challenges** — no convenient function to return to, so you chain.

Work them in order. The temptation is to skip to ROP because it sounds like the real thing; `stack_bof_01` is where you find out whether your offset tooling, your endianness handling and your delivery actually work, and finding that out on the easy target is the entire point.

Run it under `qemu-mipsel` with the B3 setup and `-g` for the debugger, exactly as B4 described.

The pieces you already have

1. **Find the offset.** Cyclic pattern, crash, read `$ra`, feed it to `fw-a4-find-offset`. Mind the endianness — DVRF is little-endian MIPS, so the register value byte-reverses into the pattern.
2. **Check the mitigations.** `checksec` on the binary. On DVRF and on most real router binaries the answer is "essentially none", which means fixed addresses you can hardcode.
3. **Pick a target.** Something already in the binary (ret2text) or in libc (ret2libc).
4. **Build the payload.** `fw-a4-build-chain`, which refuses addresses containing a NUL when the delivery is `strcpy` — a constraint that bites constantly on MIPS.

The one thing that is different

Here is the step no x86 tutorial has, and it is the reason a "simple" MIPS ret2libc is already a chain:

> **MIPS passes the first arguments in registers. An overflow writes memory.**

On x86, `system("/bin/sh")` reads its argument from the stack — exactly where your overflow already wrote. You overwrite the return address, append the argument, done.

On MIPS, `system` reads its argument from **`$a0`**, and no amount of stack writing sets a register. So you need an intermediate step: a **gadget** that loads `$a0` from the stack and then returns.

Those gadgets are ordinary function epilogues. A function that took a pointer argument, stashed it in a saved register, and restores it on the way out gives you exactly the sequence:

```
lw   $ra, 0x24($sp)
lw   $a0, 0x18($sp)     # <- the useful instruction
jr   $ra
addiu $sp, $sp, 0x28    # <- delay slot: the stack moves
```

Your payload therefore has three parts rather than two: filler to the offset, the gadget's address, and then — at the displacements *that gadget expects* — the string pointer and the address of `system`.

Two details from module A3 come due here:

The delay slot is part of the gadget. That `addiu $sp, $sp, 0x28` executes. Your next chain link is read from the *moved* stack pointer, and if you laid the payload out ignoring it, the chain reads garbage.

`$t9` matters for PIC calls. Position-independent code expects a function's own address in `$t9` on entry, because it computes its globals from there. Calling a libc function via `jr $t9` rather

than jumping straight to it is often what makes a ret2libc actually work rather than fault inside the callee.

Cache coherency, again
Module A3 said it and it decides your approach here: MIPS D-cache and I-cache are frequently not coherent. Bytes you write as data may not be visible as instructions, so injected shellcode works intermittently and reads as an unreliable offset.

Use code that is already resident. ret2text and ret2libc sidestep the problem entirely, which is why real MIPS router exploits lean on them even on targets with no NX at all.

Finding gadgets
`ropper` and `ROPgadget` both support MIPS. Two habits:

- **Search the binary and its libraries.** A non-PIE binary's own gadgets are at fixed addresses; libc's are fixed too when there is no ASLR, which on these targets is usual.
- **Read the delay slot of every candidate.** A tool that lists `jr $ra` as a gadget boundary may not show you the instruction after it, and that instruction runs.

What "done" looks like
A working DVRF solve is: a payload of a known length, an offset you derived rather than guessed, addresses you read out of the binary, and a shell. Then write it up — `fw-b5-writeup-exploit` is the module's deliverable, and it is the artifact that makes this portfolio-legible rather than a private win.

What to carry forward
- **DVRF in order**: `stack_bof_01` before the ROP challenges, because the easy one is where your tooling gets debugged.
- The pieces are already yours: **offset, mitigations, target, payload builder**.
- **MIPS passes arguments in registers**, so even a simple ret2libc needs a gadget to load `$a0`.
- **The delay slot is part of the gadget** — usually moving `$sp`, which shifts everything after it.
- `$t9` carries the callee's own address in PIC; setting it is often what makes the call work.
- **Prefer resident code** — cache incoherency makes injected shellcode unreliable.

Next: read a decompiled handler, order the exploitation, build a chain, and write it up.

Track C

Module C1

The RTOS landscape, and recognising which one you have

Why this matters
Module A0 split firmware into two worlds. Tracks A and B worked the Linux one. This is the other, and it is bigger than it looks: the thermostat, the smart plug, the BLE tag, the motor controller, the medical sensor. Kilobytes of RAM, no MMU, and no filesystem to extract.

Your first question is which RTOS — or whether there is one at all — because it decides what structures exist to find and what "the attack surface" even means.

What an RTOS is here

Not an operating system in the sense that Linux is. There is no kernel that owns the machine, no process isolation, no separate userspace. An RTOS on a microcontroller is **a scheduler and some primitives, compiled and linked into your application**.

The consequence is the one to hold onto:

> **Every task runs in the same flat address space, at the same privilege, with no isolation between them.**

A buffer overflow in the BLE handling task can corrupt the network task's stack, because they are both just memory. There is no boundary to cross. Module C3 is built on that fact.

The landscape

RTOS	Where you meet it	Fingerprint
FreeRTOS	The default for hobbyist and much commercial Cortex-M; AWS-flavoured variants on connected devices	`xTaskCreate`, `vTaskDelay`, `pxCurrentTCB`, the string `FreeRTOS`
Zephyr	Newer designs, Nordic BLE parts, anything Linux-Foundation-adjacent	`k_thread_create`, `z_` prefixed symbols, a devicetree-derived config blob
VxWorks	Industrial, aerospace, medical, telecom infrastructure. Old and long-lived	`taskSpawn`, a symbol table often left in the image, the string `VxWorks`
ThreadX / Azure RTOS	Consumer devices, storage controllers, some Wi-Fi modules	`tx_thread_create`, `_tx_` prefixed symbols
Micrium µC/OS	Industrial and safety-certified products	`OSTaskCreate`, `OS_TCB`
SEGGER embOS	Where a J-Link toolchain is already in use	`OS_` prefixed symbols, the string `embOS`
Mbed OS	Arm-ecosystem prototypes and some products	`osThreadNew`, CMSIS-RTOS naming
NuttX / RIOT / Contiki	Research, IoT, some drones	POSIX-ish naming in NuttX; `RIOT` and `Contiki` strings
Bare metal	Simple devices, and more of them than you would expect	**No scheduler at all** — a `main()` with an infinite loop and interrupt handlers

Bare metal is not an oversight in that table. A great many shipping devices have no RTOS: `main` initialises the peripherals, enters `while (1)`, and everything else happens in interrupt handlers. If you cannot find a scheduler, consider that there may not be one — and that this makes the firmware *simpler* to analyse, not harder.

Fingerprinting a stripped blob

You will not have symbols. You will have bytes. Three approaches, cheapest first.

1. Strings. Run `strings` and read it. RTOS vendors leave their name in the binary far more often than you would expect — version banners, assertion messages, the `configASSERT` filename strings that FreeRTOS emits with `__FILE__`. Assertion text is especially good: it frequently contains full source paths like `../FreeRTOS/Source/tasks.c`, which gives you the RTOS *and* its version *and* the build layout.

```
# Name the kernel. Assertion paths are the highest-signal hit here.
strings -n 6 firmware.bin | grep -iE 'freertos|vxworks|threadx|ucos|zephyr|rtems'
strings -n 6 firmware.bin | grep -E '/(Source|kernel|src)/[a-z_]+\.c'

# Version banners and build stamps.
strings -n 6 firmware.bin | grep -iE 'v[0-9]+\.[0-9]+|built|compiled|GCC'

# Offsets too, so you can go straight to the reference in the disassembler.
strings -t x -n 6 firmware.bin | grep -i freertos | head
```

2. Distinctive constants and structures. Each kernel has recognisable internals. FreeRTOS's task control block has a characteristic layout beginning with a stack pointer; its ready-list array is sized by `configMAX_PRIORITIES`. Finding one instance of the structure gives you every task in the system.

3. Helper scripts. The **VxWorks Symbol Table Finder** locates the symbol table VxWorks images frequently ship with — which hands you names for everything, turning a stripped blob into a labelled one. **FindCrypt** locates crypto constants. **Leaf Blower** identifies standard library functions by their behaviour. Module C2 covers these properly.

Enumerating the tasks

Once you know it is FreeRTOS, the highest-value early move is enumerating the tasks, because a task list is the closest thing this world has to a process list.

Find the calls to `xTaskCreate` — usually clustered in an initialisation function — and read the arguments. Each call gives you the entry function, the stack depth, a priority, and often a name string. That is your map of what the firmware actually does: a network task, a sensor task, a watchdog task, a shell task.

The one you care about is whichever handles external input. That is where module C3′s memory-corruption work goes.

What the RTOS does not give you

Worth being explicit, because it shapes every exploitation decision in Track C:

- **No MMU.** No virtual memory, no per-task address spaces, no ASLR. An MPU exists on many Cortex-M parts and is frequently left unconfigured.
- **No privilege separation in practice.** Cortex-M has thread and handler modes and a privileged bit; most RTOS builds run everything privileged because it is simpler.
- **No stack guards by default.** FreeRTOS has an optional stack-overflow check that runs *at context switch*, which detects an overflow after the damage rather than preventing it.
- **Static allocation is common**, so a heap bug is less likely and a fixed-size-buffer bug is more so.

The reason this world produced **URGENT/11**, **Ripple20**, **BadAlloc** and **AMNESIA:33** is that a memory-safety bug in a network stack here reaches control flow directly, on millions of devices that will never be patched. Module C3 studies those.

What to carry forward

- An RTOS here is **a scheduler linked into your application**, not an OS that owns the machine.
- **One flat address space, no isolation between tasks.** That is the security model.
- Fingerprint from **strings first** — assertion messages carry source paths, RTOS name and version.
- `xTaskCreate` **sites give you the task list**, which is your map of the firmware.
- If there is no scheduler, suspect **bare metal** rather than a failed search.
- **No MMU, no ASLR, an unconfigured MPU at best.** Overflows reach control flow directly.

Next: how to make a flat blob into something a disassembler can resolve — deriving the base address rather than guessing it.

Reconstructing the memory map of a flat blob

Why this matters

An ELF tells the disassembler where it belongs. A flat blob dumped off flash tells you nothing: no header, no sections, no entry point. Load it at the default `0x0` and you get instructions that decode but cross-references that point at nothing — module A5′s second symptom, and the one people misdiagnose as "the binary is packed".

The base address is not something you look up. It is something you **derive**, and this card is how.

The vector table is your landmark

A Cortex-M image begins with its vector table, and that is a gift, because it is the one structure whose layout is fixed by the architecture rather than by the vendor.

```
offset 0x00   initial stack pointer (MSP)      <- DATA, an SRAM address
offset 0x04   reset vector                     <- code, Thumb bit set
offset 0x08   NMI handler
offset 0x0C   HardFault handler
offset 0x10   MemManage handler
...
```

Three things fall out immediately:

The first word is not code. It is the value loaded into `MSP` at reset, and it points into SRAM. On Cortex-M, SRAM conventionally starts at `0x20000000`, so a first word like `0x20005000` is instantly recognisable — and it confirms you are looking at a Cortex-M vector table rather than at arbitrary bytes.

Every subsequent entry is a code address with the Thumb bit set. They look odd — literally, odd numbers — for the reason module A2 covered. A run of similar-looking odd values near the start of a blob is a vector table, read correctly.

Unused vectors are zero. Real tables have gaps. A block of `0x00000000` entries between populated ones is normal and is itself a confirmation.

Deriving the base

The relationship you need is the one every loader uses:

base = virtual address − file offset

Any point where you know both halves gives you the base. Two reliable ways to find such a point:

From the reset vector. The reset vector holds `Reset_Handler`'s virtual address. Find where `Reset_Handler` actually *is* in the file — its first instruction — and subtract. In practice you find it by disassembling at a few candidate bases until the instruction stream at the reset vector resolves into a plausible prologue; or, faster, by noticing that the reset vector's low bits are its file offset once the base is aligned.

From the clustering. All vector entries point inside the image, so they all lie in `[base, base + image_size)`. If the entries cluster in `0x08001C00`–`0x08001E00` and the image is 256 KB, the base is at or just below `0x08001C00` and aligned to a sensible boundary. For an STM32 that is `0x08000000`; for other families it differs, and the SVD file or the datasheet settles it.

The `fw-c1-derive-base` task does the subtraction version, because it is exact rather than inferential.

Common base addresses

Worth recognising, though never worth assuming:

Family	Flash base	Notes
STM32	`0x08000000`	Also aliased at `0x00000000` on many parts
Nordic nRF5x	`0x00000000`	Flash genuinely at zero
NXP Kinetis / LPC	`0x00000000`	
TI CC13xx/26xx	`0x00000000`	
Atmel/Microchip SAM	`0x00400000`	
SRAM (most Cortex-M)	`0x20000000`	The architectural convention

The STM32 alias is worth understanding rather than memorising. The part maps flash at `0x08000000` *and* mirrors it at `0x00000000` so the core can fetch its vector table from address zero at reset. Both addresses reach the same bytes. If cross-references in your analysis are split between the two, define both regions and the split resolves.

The regions you must define

Analysis resolves only if the addresses the code touches are inside something you declared. Minimum set:

- **Flash**, at the derived base, sized to the image. This is where the blob goes.
- **The flash alias**, if the part has one. Same bytes, second address.
- **SRAM**, typically at `0x20000000`. Nothing is *in* it — it is uninitialised at rest — but the code writes to it constantly, and an undeclared region means every one of those references is dropped rather than recorded.

- **Peripheral space**, typically `0x40000000` upward. Same reasoning, and module C2's SVD-Loader is what turns it from a blank region into named registers.

The SRAM point is the one people skip. It contains no data, so declaring it feels pointless. What it buys you is that every global variable access becomes a cross-referenced location rather than a dangling number, which is how you find the buffer a task writes into.

Symptoms and what they mean
An extension of module A5's table, specific to blobs:

Symptom	Cause
First word looks like an SRAM address; rest are odd	**You have a Cortex-M vector table.** Proceed.
Instructions decode, no cross-references resolve	Base address wrong
Everything shifted one byte	Disassembled at a Thumb pointer without clearing bit 0
Strings present, zero functions	No entry point defined — nothing seeded the disassembler
Data references dangle at `0x2000xxxx`	SRAM region not defined
Data references dangle at `0x4000xxxx`	Peripheral region not defined
References split between `0x0800xxxx` and `0x0000xxxx`	Flash alias not defined

What to carry forward
- A flat blob has **no metadata**; the base is derived, not looked up.
- **The vector table is the landmark**: first word is an SRAM address and is data; the rest are odd code addresses; gaps are zero.
- **base = virtual address − file offset.** Any point where you know both gives you the base.
- Define **flash, its alias, SRAM, and peripheral space** — undeclared regions silently drop references.
- Recognise the symptoms rather than re-importing hopefully.

Next: drill the RTOS fingerprints, derive a base by hand, and write a vector-table parser.

Module C2

MMIO and SVD: turning magic addresses into named registers

Why this matters
You have a correctly-based Cortex-M blob and clean disassembly. It is full of this:

```
*(uint32_t *)0x40004408 = 0x2000;
if ((*(uint32_t *)0x40004400 & 0x80) != 0) { ... }
*(uint32_t *)0x40004404 = uVar1;
```

Nothing is wrong with the analysis. That *is* what the code does — and it is unreadable, because every one of those numbers is a hardware register whose name you do not know.

This card is how those become names. It is the single largest readability win available in Track C, and it takes about two minutes once you know the move.

Memory-mapped I/O

Microcontrollers have no I/O instructions. Peripherals are wired into the address space, so talking to hardware is reading and writing ordinary addresses:

```
0x20000000 +   SRAM
0x40000000 +   peripherals    <- the interesting region
0x08000000 +   flash
0xE0000000 +   Cortex-M core peripherals (NVIC, SysTick, SCB)
```

A write to `0x40004408` is not a variable assignment. It is a command to a UART. Reading `0x40004400` is not loading a value; it is *asking the hardware a question*, and the same read twice can give different answers.

Two consequences for analysis:

- **These accesses are volatile.** A loop that reads the same address repeatedly is not dead code; it is polling. That is the shape of `while (!(UART->SR & TXE));` — waiting for a hardware flag.
- **The addresses are stable per family, not per device.** Every STM32F1 puts USART1 at `0x40013800`. So the numbers are lookupable, which is what makes the next section possible.

SVD files

CMSIS-SVD is Arm's format for describing a part's peripherals: an XML file listing every peripheral, its base address, every register at its offset, and every named bit field.

```
<peripheral>
  <name>USART1</name>
  <baseAddress>0x40013800</baseAddress>
  <registers>
    <register><name>SR</name><addressOffset>0x00</addressOffset>
      <fields><field><name>TXE</name><bitOffset>7</bitOffset></field></fields>
    </register>
    <register><name>DR</name><addressOffset>0x04</addressOffset></register>
    <register><name>BRR</name><addressOffset>0x08</addressOffset></register>
  </registers>
</peripheral>
```

Vendors publish these because compilers need them, and the `cmsis-svd` repository collects them for **hundreds of families across every major vendor**. Which means the mapping from magic number to name already exists, for free, for almost any part you will meet.

The lookup itself is arithmetic: find the peripheral whose base is the largest one at or below your address, subtract to get an offset, and find the register at that offset. That is exactly what `fw-c2-svd-lookup` implements.

SVD-Loader

SVD-Loader is a Ghidra script that reads an SVD file and does two things:

1. **Creates memory regions** for every peripheral block, so accesses stop dangling — the missing `0x4000xxxx` region from module C1's symptom table, filled in automatically.
2. **Generates a struct per peripheral and applies it**, so the decompiler renders field accesses by name.

The before and after is the whole argument:

```
/* before */
*(uint32_t *)0x40004408 = 0x2000;
if ((*(uint32_t *)0x40004400 & 0x80) != 0) { ... }

/* after */
USART2->BRR = 0x2000;
if ((USART2->SR & USART_SR_TXE) != 0) { ... }
```

The second version tells you this function configures a baud rate and then waits for the transmit buffer to empty. The first version tells you nothing without an afternoon and a reference manual.

This is the same move as retyping a signature in module A5 — correcting the model until the output is honest — applied to hardware instead of to function arguments. And as there, it needs the correct part: an SVD for the wrong family produces confidently wrong names, which is worse than numbers.

Identifying the part

SVD-Loader needs to know which SVD to use, and a flat blob does not say. Evidence, in order of reliability:

- **The chip marking**, if you have the device or an FCC internal photo (module B1).
- **Strings.** Vendor SDK paths and part numbers turn up constantly — `STM32F103xB`, `nRF52840`, an HAL source path.
- **The vector table's length.** The number of populated interrupt slots past index 15 is family-specific, and a table with 68 entries narrows the field considerably.
- **Which peripheral addresses appear.** If the code touches `0x40013800` and `0x40010800`, you can work backwards from the SVD collection to find which families place peripherals there.
- **The flash and SRAM bases** you already derived in C1.

Corroborate at least two before committing, and sanity-check afterwards: if the renamed code reads sensibly — a UART init that configures a baud rate, a GPIO setup that sets a mode — the SVD is right. If USART registers are being written with values that make no sense, it is not.

Which peripherals matter

For finding the attack surface, not all are equal:

- **UART / USART** — a serial console, often a debug shell. Also the Track F entry point.
- **USB, Ethernet, radio (BLE, 802.15.4, Wi-Fi)** — the externally reachable input. This is where module C3′s memory-corruption work goes.
- **SPI / I2C** — external flash, EEPROM, sensors. Where keys and configuration are stored.
- **Flash controller** — self-programming, which is the OTA update path and module E2′s territory.
- **Crypto accelerator / RNG** — if present, whether it is actually used is a finding either way.
- **Watchdog** — matters practically: it will reset the part while you are debugging, and disabling it is often the first thing you do.
- **GPIO, timers, ADC** — usually the device's function rather than its attack surface.

Start at whichever peripheral receives external data, and work outward from there.

What to carry forward

- Peripherals live **in the address space**; a magic `0x4000xxxx` access is hardware, not a variable.
- MMIO reads are **volatile** — a tight polling loop is waiting on a flag, not dead code.
- **SVD files already exist** for hundreds of families in the `cmsis-svd` repo.
- **SVD-Loader** creates the peripheral regions and applies register structs, turning numbers into `USART2->SR & USART_SR_TXE`.
- **Identify the part from two independent signals**, and sanity-check the renamed output — a wrong SVD is worse than no SVD.
- Prioritise the peripheral that **receives external data**.

Next: recovering the RTOS's own structures, and the helper scripts that label a stripped blob for you.

Recovering the RTOS's own structures

Why this matters

Module C1 identified the RTOS. The previous card made the hardware legible. This one recovers the kernel's own bookkeeping — and it is what turns "this is FreeRTOS" into "this firmware runs five tasks, here is what each does, and this one reads the radio."

That list is the map. Without it you are reading functions in isolation with no sense of which ones matter.

The task control block

Every RTOS keeps a per-task structure. FreeRTOS calls it the TCB, and its shape is stable enough to recognise:

```
typedef struct tskTaskControlBlock {
    volatile StackType_t  *pxTopOfStack;    /* FIRST field, always */
    ListItem_t            xStateListItem;
    ListItem_t            xEventListItem;
    UBaseType_t           uxPriority;
```

```
    StackType_t           *pxStack;         /* base of the task's stack */
    char                  pcTaskName[16];   /* configMAX_TASK_NAME_LEN */
    ...
} tskTCB;
```

Two features make it findable in a blob:

`pxTopOfStack` is first, and the ABI depends on it. The context-switch assembly loads the stack pointer from offset 0 of the current TCB, so it cannot move. A structure beginning with a pointer into SRAM is the signature.

`pcTaskName` is an inline character array. So a TCB contains a readable ASCII name *inside* the structure rather than a pointer to one. That is the practical way in: find the task-name strings in SRAM or the initialised-data region, and the bytes around them are TCBs.

Work backwards from a name to the structure start and you have the priority, the stack base, and the top-of-stack pointer for that task. Apply the struct in Ghidra once and every other TCB becomes readable for free.

Two ways to get the task list

Statically, from the creation sites. Find the `xTaskCreate` calls — usually clustered in one initialisation function — and read their arguments: entry function, name, stack depth, parameter, priority, handle. This is the cheaper route and it is usually enough.

From memory, via `pxCurrentTCB` and the ready lists. The scheduler keeps an array of ready lists indexed by priority, plus delayed and suspended lists. Walk them and you enumerate every task that exists, including ones created dynamically at runtime that no static call site reveals. This matters when a task is created in response to a connection or a command — which is exactly the kind of task that handles attacker input.

Do the static pass first. Reach for the lists when the static picture looks incomplete: a scheduler configured for eight priorities and only three creation sites is a hint that something is being created elsewhere.

What to record per task

For each task, four things decide whether you care about it:

Field	Why it matters
Entry function	Where to start reading
Stack depth	A small stack plus a large local buffer is an overflow waiting to happen
Priority	A high-priority task that blocks starves everything below it
Name	Vendor names are honest: `wifi_rx`, `cmd_parse`, `ota_task`

The stack depth is the one people skip and it is directly exploitable. FreeRTOS allocates each task's stack from the heap or a static buffer at the size given in `xTaskCreate`, and there is no guard page. A task with a 512-byte stack and a 256-byte local buffer has very little room for error — and overflowing past the end of one task's stack lands in whatever the allocator put next, which is frequently another task's stack or its TCB.

Overwriting a neighbouring TCB's `pxTopOfStack` is a control-flow primitive: the next context switch loads the stack pointer from it. Module C3 works that.

Helper scripts

Three worth knowing, in descending order of payoff:

VxWorks symbol table recovery. VxWorks images frequently ship their symbol table — a name-to-address list the RTOS uses for its own shell. Recovering it converts a stripped blob into a fully labelled one. Nothing else in this track comes close for effort spent, and it is the first thing to try the moment a VxWorks fingerprint appears.

FindCrypt. Locates cryptographic constants — S-boxes, round constants, curve parameters — by signature. It tells you which primitives are present, which is a finding either way: a device doing its own AES in software when the part has a hardware accelerator is worth a look, and a device with no crypto constants at all is not doing any.

Leaf Blower. Identifies standard library functions in stripped binaries by their behaviour rather than their names — `memcpy`, `strlen`, `strcpy` and friends. Useful precisely because those are the sinks: knowing which unnamed function is `strcpy` turns the A5 xref walk into something you can actually run on a blob.

Sanity-checking what you recovered

Structure recovery goes wrong quietly, so check it:

- **Do the task names read like names?** If applying your TCB struct produces `pcTaskName = "\x00\x14\xa8 "`, the offset is wrong.
- **Do the stack pointers point into SRAM?** `pxTopOfStack` and `pxStack` must both land in the RAM region. A pointer into flash means you are looking at the wrong structure.
- **Do the priorities fall below `configMAX_PRIORITIES`?** A priority of 0x20544553 is not a priority.
- **Does the count match the creation sites?** If you find six TCBs and four `xTaskCreate` calls, either two are created dynamically or two of your TCBs are not TCBs.

Each of these is one glance, and each catches a whole class of misapplied-struct error.

What to carry forward

- The **task list is the map**; without it you are reading functions with no priority.
- A FreeRTOS **TCB begins with `pxTopOfStack`** and contains **`pcTaskName` inline** — find the names, and the structures are around them.
- **Static creation sites first**, ready lists when the picture looks incomplete.
- Record entry, **stack depth**, priority, name. Small stack plus large buffer is the finding.

- Overwriting a neighbouring TCB's `pxTopOfStack` is a **control-flow primitive** (module C3).
- **VxWorks symbol recovery** is the biggest single win; FindCrypt and Leaf Blower label primitives and sinks.
- Sanity-check recovered structures: names readable, stack pointers in SRAM, priorities in range, count matching.

Next: locate a task's fields in a memory dump, then the module check.

Module C3

Memory corruption with nothing in the way

Why this matters

Module A4 taught stack overflows against embedded Linux, where the defences are inconsistent but at least *possible*. This is the same bug class in an environment where most of them do not exist at all, and where the ones that do are usually switched off.

The practical effect is that the distance from "I found an overflow" to "I control execution" is much shorter here than anywhere else in this course. That is worth knowing precisely rather than vaguely, because it changes which findings are worth your time.

The defences, and what is actually there

Defence	On a Cortex-M RTOS build
ASLR	Does not exist. No MMU, no relocation, addresses fixed at link time.
NX / DEP	An MPU can mark regions non-executable — **if** the vendor configured it. Frequently they did not.
Stack canaries	`-fstack-protector` is available and rarely enabled on size-constrained builds.
Guard pages	Do not exist. There is no MMU to fault on.
Heap hardening	Nothing comparable to a modern libc allocator.
Task isolation	**None by default.** All tasks share one address space, all privileged.

Every address you need is fixed and knowable from the image you already loaded in C1. There is no leak to chase, no offset to defeat, no canary to read. You are back in the position that general-purpose exploitation left behind around 2005 — and this is shipping hardware.

The one that surprises people is **stack overflow detection**. FreeRTOS offers it, and it is worth understanding what it does: `configCHECK_FOR_STACK_OVERFLOW` compares the stack pointer or a watermark pattern **at context switch**. So it detects an overflow *after* it has happened, on the

next scheduler tick, and calls a hook. It is a debugging aid, not a control — by the time it fires, the damage is done and your payload may already have executed.

Where the overflow lands

This is what makes RTOS exploitation different, so it is worth being concrete.

Each task gets a stack allocated at creation — from the heap with `xTaskCreate`, or from a static buffer with `xTaskCreateStatic`. The kernel places them wherever the allocator does. There is no guard page between them, so what follows a task's stack in memory is simply whatever was allocated next.

```
higher addresses
  +---------------------------+
  |  TCB for task B           |   <- pxTopOfStack at its base
  +---------------------------+
  |  stack for task B         |
  +---------------------------+
  |  TCB for task A           |
  +---------------------------+
  |  stack for task A         |   <- your overflow starts here and grows UP
  +---------------------------+
lower addresses
```

Overflow task A's stack far enough and you are writing into its own TCB, then task B's stack, then task B's TCB. Three distinct targets, all reachable from one bug:

1. The saved return address on your own stack. The A4 primitive, unchanged. Cheapest and usually enough.

2. A neighbouring TCB's `pxTopOfStack`. This is the RTOS-specific one. That field sits at offset 0 of the TCB, and the context-switch assembly loads the stack pointer *directly from it*. Overwrite it and the next time the scheduler switches to that task, it restores a register frame from an address you chose — including the PC. You do not need a return in your own task at all; the scheduler does the work.

3. Function pointers in kernel or application structures. Timer callbacks, queue handlers, registered event callbacks. All ordinary writable memory here.

The second one is worth internalising because it has no analogue in the Linux world: the *scheduler* becomes your gadget.

Finding the reachable bug

The C2 task list is what makes this tractable. For each task you recovered:

- **Which one reads external input?** Radio, USB, Ethernet, UART. Start there — everything else is reachable only through it.
- **What is its declared stack depth?** `xTaskCreate` takes it in words, not bytes, which is a recurring source of vendor error in its own right. A 128-word stack is 512 bytes.
- **What local buffers does its entry function declare?** A 256-byte buffer inside a 512-byte stack leaves very little margin.

- **Is there a bounded copy into it?** The A1 sink list applies unchanged: `strcpy`, `sprintf`, `memcpy` with a length from the packet.

That combination — external input, small stack, unbounded copy — is the finding, and it is findable statically from what modules C1 and C2 already gave you.

The landmark research

Four bodies of work worth reading, because they are this card at industrial scale and because they are what a hiring conversation will expect you to have heard of:

URGENT/11 (2019) — eleven vulnerabilities in the IPnet TCP/IP stack used by VxWorks. Several were remote code execution reachable from unauthenticated network traffic, on devices including medical equipment, industrial controllers and firewalls. The lesson is reach: a bug in a shared network stack is a bug in every product that licensed it.

Ripple20 (2020) — nineteen vulnerabilities in the Treck TCP/IP stack, embedded across an enormous and largely unmapped supply chain. Many affected vendors did not know they shipped Treck, because it arrived inside a module they bought from someone else. The lesson is that the SBOM problem is a security problem.

BadAlloc (2021) — memory-allocation bugs spanning more than twenty RTOSes and SDKs. Integer overflows in `malloc`-equivalents where a large requested size wraps and returns a small buffer. The lesson is that the same bug pattern recurs independently across implementations that never shared code.

AMNESIA:33 (2020) — thirty-three vulnerabilities across four open-source TCP/IP stacks used throughout embedded products. Same shape as Ripple20, in code anyone could have read.

The common thread: **a memory-safety bug in an embedded network stack reaches control flow directly, on millions of devices, most of which will never be patched.** That is the honest argument for why this track matters, and it is worth being able to make it in one sentence.

What to carry forward

- **No ASLR, no guard pages, usually no NX and no canary.** Addresses are fixed and knowable.
- FreeRTOS stack-overflow checking fires **at context switch**, after the fact — a debugging aid.
- One address space, no isolation: an overflow reaches **your return address, a neighbouring TCB, and function pointers**.
- **`pxTopOfStack` at TCB offset 0 is loaded directly by the context switch** — overwrite it and the scheduler transfers control for you.
- Find the bug from the task list: **external input + small stack + unbounded copy**.
- **URGENT/11, Ripple20, BadAlloc, AMNESIA:33** — shared stacks, unmapped supply chains, and devices that never get patched.

Next: drill the primitives, work a microcorruption level, and write up what you found.

Where to practise this without a device

Why this matters

Track B had IoTGoat and DVRF — deliberately vulnerable targets, maintained, documented, free. This track has almost nothing equivalent, and that is a real gap rather than an oversight on anyone's part.

The reason is structural. A vulnerable Linux firmware image is a filesystem you can build once and hand to anyone. A vulnerable RTOS image is compiled *for a specific microcontroller*, and running it means either owning that part or emulating it faithfully enough that its peripherals behave. Both are friction that IoTGoat does not have.

So the practice routes are different, and knowing them is the difference between this track being theory and being something you have actually done.

microcorruption

Start here. Free, browser-based, no installation, no hardware.

It presents a series of electronic lock firmwares for the **MSP430** — a real 16-bit microcontroller architecture — with a built-in disassembler, debugger and memory view. Each level is a lock you have to open, and the difficulty ramps deliberately: the early ones are password comparisons, the later ones are stack overflows, format strings and ROP.

What makes it the right recommendation for this module specifically:

- **It is genuinely microcontroller exploitation**, not x86 wearing a costume. Fixed addresses, no ASLR, no canaries, a flat address space — the environment the previous card described.
- **The debugger is right there.** You set breakpoints and watch memory as your input lands, which is the feedback loop that makes overflow mechanics click.
- **It is free and it will still be there next year.** No dependency on hardware you have to buy or an emulator you have to configure.

MSP430 is not ARM, and that matters less than it sounds. The register set and encoding differ; the *reasoning* — find the buffer, find the saved return address, compute the distance, control the value — is identical to A4 and transfers directly.

Work it in order. If you do one thing from this module, do the first six levels.

Build your own vulnerable target

The other route, and the one that teaches more about RTOS internals specifically: take an official example and break it on purpose.

The shape:

1. Start from a **FreeRTOS or Zephyr example** that already builds and runs. FreeRTOS's Posix/Linux demo and Zephyr's `native_sim` board both run as ordinary host processes, so you need no hardware at all.
2. **Add a task with a bug you chose** — a command handler with a `strcpy` into a fixed local buffer, a length field taken from input and passed to `memcpy`, a parser with an off-by-one.

3. **Give it a modest stack.** 128 words is realistic and leaves the margin real firmware has.
4. Build it, then **analyse your own binary as though you had never seen it** — load it, recover the task list (C2), find the sink, compute the offset.
5. Exploit it, then instrument: watch the neighbouring TCB, and see whether `pxTopOfStack` is reachable from your buffer.

Two things make this worth the setup cost. You control the difficulty, so you can build exactly the bug you want to understand. And you see both sides — the source that produced the binary, and the binary as an attacker meets it — which is the fastest way to build the intuition for what source patterns look like after compilation.

The obvious objection is that knowing where you put the bug makes finding it meaningless. That is true for the *finding* and not for the *exploiting*, which is the part this module is about. If you want the finding to be honest too, write the vulnerable task, leave it a fortnight, and come back to it.

Real firmware

Once the above are comfortable: pull a real image. Vendors publish firmware for BLE modules, sensor hubs and development boards, and much of it is unencrypted Cortex-M. Load it with C1, name its peripherals with C2, find its input-handling task, and read.

You will usually not find an exploitable bug on your first image, and that is a normal outcome rather than a failure — it is what an honest null result looks like, and Track B's module B2 made the same point about static analysis.

What about the hardware route

Track F's Black Pill target and the SWD flash dump are where this module reconnects to physical hardware: dump a real MCU's flash, and you have a genuine flat blob for the C1 workflow with provisioning your own build would never have. That is the eventual destination, and none of the above requires it.

What to carry forward

- **Deliberately vulnerable RTOS targets are scarce**, and the reason is that an RTOS image is built for a specific part.
- **microcorruption is the free zero-hardware ramp** — real MSP430, real debugger, deliberate difficulty curve. Do the first six levels.
- **Build your own** from a FreeRTOS or Zephyr host-runnable example: add the bug, then analyse your own binary as a stranger would.
- **Real vendor firmware** is the third step, and finding nothing on your first image is a normal result.
- The reasoning transfers across architectures even when the encoding does not.

Next: drill the primitives, submit a microcorruption solve, and write it up.

Track E

Module E1

Fuzzing as a discipline: corpus, coverage, and what it cannot find

Why this matters

Fuzzing has a reputation for being simple — throw garbage at a program until it falls over. That description fits a 1990 fuzzer and nothing since, and believing it produces campaigns that run for a week and find nothing.

Modern fuzzing is a **search**, guided by feedback, over inputs that are already nearly valid. The skill is not running the tool. It is choosing what to feed it, wiring it to the right entry point, and knowing what it will never find so you spend your other hours elsewhere.

Coverage feedback is the whole idea

A blind random fuzzer generating bytes against a firmware parser will spend its entire life being rejected by the first length check. The input space is astronomically larger than the interesting part of it.

Coverage-guided fuzzing changes the search into something that can actually make progress:

1. Take an input from the queue and mutate it — flip bits, splice, insert magic values.
2. Run the target with **instrumentation** that records which edges of the control-flow graph were executed.
3. If this input reached an edge no previous input reached, **keep it** and add it to the queue.
4. Repeat.

That single retention rule is what makes it work. The fuzzer has no model of the format, but it accumulates one: an input that got past the length check is kept because it reached new code, so every later mutation starts from something that already passes. Depth is discovered incrementally rather than guessed.

This is also why **instrumentation is not optional**. Without coverage feedback the queue never grows in a useful direction and you are back to random bytes. On a binary you cannot recompile — which is every firmware binary — that instrumentation has to come from emulation, which is the next card.

The seed corpus decides the campaign

If there is one lever that separates a productive campaign from a wasted one, it is the seeds.

Seeds should be valid inputs that already reach the code you care about. For a firmware parser, that means real firmware images. For a config parser, real config files. For a network service, captured packets.

Rules worth following:

- **Small.** A fuzzer mutates the whole input; a 4 MB seed wastes most of its mutations on regions nothing reads. Prefer many small seeds to one large one.

- **Diverse.** Coverage of different *shapes*, not many copies of one shape. Two seeds that exercise the same branch are one seed.
- **Valid.** A corpus of invalid files teaches the fuzzer only how the reject path works.
- **Minimised.** Tools exist to trim a corpus to the smallest subset preserving total coverage. Do it before starting; every redundant seed costs execution time forever.

An empty or careless corpus is the most common reason a campaign finds nothing, and it looks identical to "there are no bugs here".

Speed is a first-class concern

Fuzzing throughput is executions per second, and it multiplies directly into how much of the search space you cover. A campaign at 2,000 exec/s explores in an hour what a campaign at 20 exec/s covers in four days.

This is why emulation-based fuzzing of firmware is fiddly: naive full-system emulation might give you tens of executions per second, and the work of building a tighter harness — emulating one function rather than booting an image — is what buys back two orders of magnitude. That trade is the subject of the next card.

```
# Source available: instrument at compile time. Always prefer this.
CC=afl-clang-fast CXX=afl-clang-fast++ ./configure && make

# Binary only, cross-architecture — the firmware case. -Q is QEMU mode.
afl-fuzz -Q -i seeds/ -o findings/ -- ./parse_config @@

# Persistent + deferred forkserver, once the harness is yours. This is where
# the order-of-magnitude lives, not in the mutator.
AFL_LLVM_INSTRUMENT=PCGUARD afl-fuzz -i seeds/ -o findings/ -- ./harness @@
```

`@@` is the input filename; drop it and AFL++ feeds stdin instead. Read the throughput on the status screen before tuning anything else — under ~100 exec/s the campaign is not going to find much regardless of how good the corpus is, and the fix is the harness, not more cores.

```
# Corpus hygiene, before and during. Both pay for themselves in an hour.
afl-cmin -i raw_seeds/ -o seeds/ -- ./parse_config @@      # drop redundant seeds
afl-tmin -i findings/crashes/id:000000* -o min.bin -- ./parse_config @@
```

What fuzzing finds, and what it does not

Finds readily: memory corruption (stack and heap overflows, use-after-free), assertion failures, unhandled exceptions, infinite loops and hangs, integer overflows that lead to bad allocations, and parser confusion in anything that takes structured binary input. On embedded targets with no ASLR and no canaries, a crash is frequently a straightforward path to control.

Will not find, no matter how long you run it:

- **Command injection.** Nothing crashes. A fuzzer sees a normal exit and moves on. Module A1′s bug class is invisible here.

- **Hardcoded credentials and keys.** Not a crash. Static analysis finds these in seconds (FSTM stage 5).
- **Authentication and authorisation flaws.** Reaching an endpoint you should not is not a memory-safety event.
- **Logic bugs generally.** Wrong behaviour is not detectable without a specification, and the fuzzer has none.
- **Anything behind a check it cannot satisfy by mutation** — a checksum, a signature, a magic constant it never guesses. This is real and it is the reason harnesses often *patch out* the checksum verification before fuzzing, then re-add a correct checksum when reproducing.

The practical consequence: fuzzing is one technique in a methodology, not a replacement for it. A firmware assessment that fuzzes and does nothing else will miss the findings that were free.

Crashes are not findings

A campaign that runs overnight hands you a directory of crashing inputs, and the number is usually larger than the number of bugs. One bug reached by many mutation paths produces many crash files.

The work after the campaign:

- **Deduplicate.** Group crashes by where they actually happened — a normalised stack signature, not the raw addresses, which differ between runs. `fw-el-dedup-crashes` implements this.
- **Minimise.** Shrink each crashing input until every remaining byte is load-bearing. A 4 KB crashing input tells you little; a 12-byte one often tells you the whole bug. `fw-el-minimize` implements this.
- **Triage.** Decide which are exploitable and which are null-pointer dereferences that will never be more than a denial of service. Module A4′s offset arithmetic is what answers this.

Triage is where most of the value is, and it is the part that transfers directly from any prior crash-analysis work — the technique is the same whether the crashing binary is a Windows kernel driver or a MIPS router daemon. What is embedded-specific is everything before it: getting the target to run under instrumentation at all.

What to carry forward

- **Coverage feedback** is what makes fuzzing a search rather than a lottery.
- **The seed corpus decides the campaign** — small, diverse, valid, minimised.
- **Throughput multiplies coverage.** A tighter harness is worth more than a longer run.
- Fuzzing finds **memory safety**; it is blind to injection, credentials, authz and logic.
- A checksum the fuzzer cannot satisfy is a wall — patch it out to fuzz, restore it to reproduce.
- **Crashes are not findings.** Dedupe, minimise, then triage.

Next: how you instrument a binary you cannot recompile, and how to build a harness around it.

AFL++ QEMU mode, and building a harness around firmware

Why this matters

The previous card said coverage feedback is what makes fuzzing work. That feedback normally comes from compiler instrumentation — you rebuild the target with `afl-cc` and every branch reports itself.

You cannot rebuild a firmware binary. You have a stripped MIPS ELF pulled out of a SquashFS image, no source, no toolchain, and a target architecture that is not the one you are sitting at.

This card is how you fuzz it anyway.

Instrumentation without recompilation

AFL++'s QEMU mode runs the target under user-mode QEMU and instruments **basic blocks as they are translated**. QEMU already has to decode the target's instructions to emulate them; the mode hooks that translation step and emits the same coverage signal the compiler would have.

Three things follow, and all three matter in practice:

- **It is architecture-agnostic.** MIPS, ARM, PowerPC — whatever `qemu-user` supports. This is the whole reason it is the standard tool for firmware.
- **It is slower than compiled instrumentation**, often by an order of magnitude. Throughput is coverage (previous card), so everything below is about winning that back.
- **It sees only what executes.** Code behind an unsatisfiable check is not merely uncovered, it is invisible.

The `qemu-user` piece is the same mechanism as FSTM stage 6′s single-binary emulation, which Track B covers. If you can run the binary under `qemu-mipsel` at all, you are most of the way to fuzzing it.

The harness is the design decision

A harness is the small program that takes the fuzzer's input and delivers it to the code you want tested. Getting it right is most of the skill, and the choice is a trade between **fidelity** and **speed**.

Whole-system. Boot the full firmware image under emulation and drive it over the network. Highest fidelity, lowest throughput — tens of executions per second at best. Reasonable when the bug you want needs the whole stack up.

Single binary. Run just the vendor's `httpd` under `qemu-mipsel` with the input on stdin or as a file. Much faster, and usually enough. This is the common case.

Single function. Emulate from one function's entry to its return, feeding the input straight into its arguments — with **Unicorn** for raw CPU emulation or **Qiling** when you also need syscalls. Thousands of executions per second, and it requires you to know exactly what state that function expects.

The rule: **the narrowest harness that still reaches the target code.** Every layer you emulate that the bug does not need is throughput you are paying for nothing.

Persistent mode is the other multiplier. Instead of forking a fresh process per input, the harness loops inside one process, resetting state between iterations. AFL++ documents this as worth 10-20x. It is also the easiest thing to get subtly wrong, because state that is not reset carries between iterations and makes crashes unreproducible.

```
/* The single-binary harness, in the shape you will write it most often.
 * Build:  afl-clang-fast -o harness harness.c target.c   */
#include <stdint.h>
#include <string.h>

__AFL_FUZZ_INIT();

extern int parse_config(const uint8_t *buf, size_t len);   /* the target */

int main(void)
{
    __AFL_INIT();                       /* deferred forkserver: everything
                                         * above this line runs ONCE */
    uint8_t *buf = __AFL_FUZZ_TESTCASE_BUF;

    while (__AFL_LOOP(10000)) {         /* persistent mode: 10k per process */
        size_t len = __AFL_FUZZ_TESTCASE_LEN;
        if (len < 4) continue;          /* cheap reject, not a crash */

        reset_global_state();           /* <-- the line people omit */
        parse_config(buf, len);
    }
    return 0;
}
```

`reset_global_state()` is the whole difference between a harness that works and one that produces crashes nobody can reproduce. Anything the target mutates across a call — a static buffer, a parsed-config singleton, an open file descriptor, an allocator arena — has to go back to its starting state inside the loop. A crash that only fires on the four-hundredth iteration is almost always this, and it is indistinguishable from a real bug until you try to replay it.

Test the reset before you trust the campaign:

```
# Same input, many times, in one process. Any crash here is a reset bug.
for i in $(seq 1 500); do cat seeds/valid.bin; done | ./harness

# And confirm a crash actually replays from a fresh process.
./harness < findings/crashes/id:000000*
```

Getting the input in

Firmware code rarely reads from stdin. Some plumbing is always required, and the options are:

- **File path.** AFL++ passes a filename via `@@`; if the target already takes a path, you are done.
- **Stdin.** The default, if the target reads it.

- **A wrapper.** A small C program that reads the input and calls the target function directly, compiled for the target architecture with a cross-toolchain (module A1).
- **Programmatic.** With Qiling or Unicorn, write the bytes into emulated memory, set the argument registers per the calling convention (A2/A3), and start execution at the function.

That last one is where the architecture modules stop being background knowledge. Setting `$a0` to point at your buffer and `$a1` to its length *is* the harness.

Obstacles you will actually hit

Checksums and signatures. The target validates the input before parsing it, and the fuzzer never guesses a valid checksum, so every input dies at the same branch. The standard answer is to **patch out the check** for the campaign — NOP the comparison or force the branch — and then recompute a correct checksum when you reproduce a crash against the unmodified binary. Forgetting the second half produces "bugs" the real device rejects.

Missing peripherals. Firmware reads a hardware register that does not exist under emulation and hangs or faults immediately. Hook the address and return a plausible value; module C2′s peripheral work is what tells you which value is plausible.

nvram and configuration. The binary calls a vendor helper to read settings and gets nothing. Stub the function, or pre-populate the store. `fw-a1-parse-nvram` gives you the format.

Non-determinism. Timestamps, PIDs, addresses and random values make the same input crash differently, which wrecks both the coverage signal and your deduplication. Pin what you can.

Knowing whether the harness works

Before letting a campaign run overnight, check three things. Each has a characteristic failure that looks like "no bugs here":

1. **Does the coverage map grow?** If the fuzzer reports the same edge count after ten minutes as after ten seconds, nothing is being explored — usually the input never reaches the parser.
2. **Do the seeds produce different coverage from each other?** If every seed maps to the same edges, your corpus has one shape and the fuzzer has nothing to work with.
3. **Does a deliberately malformed input crash it?** Break the target on purpose — hand it an input you *know* overflows — and confirm the harness reports the crash. A harness that cannot report a crash you engineered will not report one it finds.

That third check is the one people skip, and it is the one that catches a harness which runs beautifully and is wired to nothing.

What to carry forward

- **QEMU mode instruments at translation time**, so a binary you cannot rebuild still gives coverage — at an order-of-magnitude speed cost.
- Choose **the narrowest harness that reaches the code**; persistent mode buys back 10-20x.
- Getting the input in is plumbing, and at the function level it is **setting argument registers** per the calling convention.
- **Patch out checksums to fuzz; restore them to reproduce.**

- Stub missing peripherals and nvram, and pin sources of non-determinism.
- **Verify the harness with an engineered crash** before trusting a quiet campaign.

Next: drill crash signatures, order a campaign, diagnose a harness that finds nothing, and build the two tools that turn a crash directory into findings.

Module E2

The boot chain, and U-Boot as an attack surface

Everything you have done so far started at the filesystem. Module A1 traced how a request reaches a CGI handler; Track B unpacked the image the handler lives in. All of that assumes the device already booted.

This module works on what happens before that — and on the question the whole of secure boot exists to answer: **when this device loads code, what makes it trust that code?**

The answer is frequently "nothing", and the second-frequently "something, but only one link of it".

The chain

A modern embedded Linux device boots in roughly four stages. The exact names differ per SoC; the shape does not.

Stage	Lives in	Loads	Typically verifies
Boot ROM	mask ROM on the die	the SPL	the SPL, if fuses say so
SPL (secondary program loader)	first sectors of flash	U-Boot proper	rarely
U-Boot	flash	the kernel + DTB	only with verified boot configured
Kernel	flash, or a FIT image	init, then userland	only with dm-verity or IMA

Read that column on the right again. Each stage *can* verify the next, and each one is an independent decision made by whoever built the device. A chain is only as strong as its weakest link, and unlike cryptography that is not a slogan here — it is literally how the mechanism works. If U-Boot does not verify the kernel, then verifying the SPL bought nothing: you replace the kernel and the verified SPL loads a verified U-Boot which loads your kernel without complaint.

The boot ROM is the one link you cannot replace. It is mask ROM, fixed at fabrication. That is why it is the **root of trust**: not because it is more carefully written, but because it is the only part an attacker with flash write access cannot change. Everything above it is trustworthy only insofar as the ROM's verification decision propagates upward.

U-Boot is a debugger that ships in production

U-Boot's job is to bring up DRAM, find a kernel, and jump to it. To do that it carries a command interpreter with, among other things:

- `md` / `mw` — read and write arbitrary physical memory
- `sf read` / `nand read` — read raw flash to memory
- `tftpboot` — pull an image over the network into memory
- `bootm` / `booti` — jump to a loaded image
- `setenv` / `saveenv` — change and persist the environment
- `printenv` — dump it

That set is a complete firmware-extraction and code-execution toolkit. It is not a vulnerability; it is the intended feature set of a bootloader, and it is exactly what makes an accessible console prompt equivalent to full compromise. Getting one is generally the *end* of the hardware phase, not a step in it:

```
# At the U-Boot prompt. Boot a kernel you supplied, over the network.
tftpboot 0x80800000 my.uImage
bootm 0x80800000
```

runs your kernel. Or, without even that:

```
# Read the whole SPI flash into DRAM. Sizes and the load address come from
# `printenv` and the SoC's memory map -- 0x80800000 is the usual MIPS scratch.
sf probe 0
sf read 0x80800000 0x0 0x800000        # addr_in_dram  offset_in_flash  length
```

and you have the flash contents in memory to dump over serial — which is Track F's answer to a device whose firmware is not downloadable.

The full extraction, in the order you would actually run it:

```
printenv                               # capture EVERYTHING first, especially
                                       # bootargs and mtdparts
sf probe 0                             # bring up the SPI controller
sf read 0x80800000 0x0 0x800000        # flash -> DRAM

# With a network on the bench, this is minutes rather than hours:
setenv ipaddr 192.168.1.10
setenv serverip 192.168.1.1
tftpput 0x80800000 0x800000 dump.bin

# Without one, dump over the serial console and reassemble the hex on the host.
md.b 0x80800000 0x800000
```

And the one-line root shell, which needs no dump at all:

```
setenv bootargs "${bootargs} init=/bin/sh"
boot                                   # NOT saveenv -- keep it non-persistent
```

Leave `saveenv` out unless you mean it. Writing the environment persists your change across reboots on a device you may need to hand back in working order, and on a unit with a signed kernel but an unsigned environment it is also the exact modification a vendor would call tampering.

How the prompt is reached is a hardware question (Track F: find UART, get a console, interrupt autoboot). What it gets you is this card's point.

`bootargs`, and why it matters even when the console is locked

The most valuable environment variable is `bootargs` — the kernel command line. A representative one:

```
console=ttyS0,115200 root=/dev/mtdblock2 ro init=/sbin/init mtdparts=...
```

Three things a VR engineer reads out of it immediately:

- **`root=`** — which flash partition holds the rootfs, so which partition you care about in the image you just downloaded.
- **`mtdparts=`** — the full flash layout, names and offsets. This is the map that tells you where the bootloader ends and the kernel begins, and it is frequently how you resolve an offset that `binwalk` reported ambiguously.
- **`init=`** — the first userland program. Change it to `/bin/sh` and you have a root shell with no authentication, because you have replaced the thing that would have asked for one.

That last one is the single most common bootloader-level attack, and it needs no exploit at all: `setenv bootargs "... init=/bin/sh" ; boot`. Any path that lets you influence `bootargs` — a console, a writable environment partition, an update package that includes one — is a path to root.

Which is why devices that take this seriously either sign the environment, store it inside the signed image rather than in its own writable partition, or strip the console entirely in production builds.

What to record, per link

When you assess a boot chain, the deliverable is not "secure boot: yes/no". It is one row per link:

Question	Why it decides something
Is verification *implemented* at this link?	Absent verification is the common case; find it before assuming a bypass is needed.
Is it *enabled* — fuses blown, key programmed?	A build with verified-boot code compiled in and fuses unblown verifies nothing. This is the single most common "secure boot" failure, and it looks fully secure in the source.
What key, and where does it live?	A key in a writable partition is not a root of trust. A key in OTP is.
What happens on failure?	Halt, or fall back to an unsigned path? A fallback is the bypass.
Is the environment covered?	Signing the kernel and leaving `bootargs` writable leaves `init=` open.

The fourth and fifth rows are where real findings come from. Nobody ships a device whose verification is missing entirely and calls it verified; they ship one whose verification is present, enabled, correct — and has a recovery mode that skips it.

Where this goes

- **Track F** covers reaching the console physically: UART identification, level shifting, interrupting autoboot, and glitching the ROM when the console is genuinely gone.
- **E3** covers the update path — because the other way code gets loaded onto a device is an update, and an update that is not signed is an unauthenticated firmware load with a friendlier interface.
- The **module quiz** and `fw-e2-verify-image` turn the table above into something you execute rather than recall.

Signed updates, and the seven ways they fail

An update is an authorised firmware load. It is the one path a vendor deliberately builds for putting new code on a shipped device — which makes it, from a VR perspective, the most interesting code on the device. If the boot chain is verified and the update path is not, the update path is the boot chain.

You will assess more update mechanisms than boot ROMs, because updates are software you can read.

First: what each primitive actually proves

The recurring root cause is a mechanism that provides integrity being deployed where authenticity was needed. Three things get called "verification":

Primitive	Proves	Against whom
Checksum (CRC32, MD5, SHA-256 alone)	the bytes did not change *by accident*	transmission errors
MAC (HMAC)	the bytes came from someone holding the shared key	outsiders — but every device holds the key
Signature (RSA, ECDSA, Ed25519)	the bytes came from whoever holds the *private* key	everyone, including someone who fully owns a device

Only the third survives an attacker who has extracted the device's flash — and you should assume that attacker, because that is what Track B did in an afternoon.

The distinction has a concrete consequence. A device that HMACs its updates with a key baked into the firmware is not signing anything: pull the key out of the image, and you can author updates the device accepts. That is not a broken implementation of signing, it is a correct implementation of the wrong primitive — and it is common, because HMAC is easier and the code reads exactly like signature verification.

In the code, look for: which key material is loaded, and from where. A verifier that reads its key out of the same flash the attacker can write is verifying nothing, regardless of the algorithm's strength.

The seven failure modes
Ranked by how often they show up:

1. No verification at all
The update is a checksum'd blob. The checksum is CRC32, and it is there for transmission integrity. Anyone who can reach the update endpoint can load code.

Tell: the parser and the applier are the same function; there is no key anywhere in the binary.

2. Verification of the wrong thing
The signature covers the header, or the manifest, or the first block — but not the payload. Or it covers a *hash the package supplies* rather than a hash the verifier computes. The second is subtle and reads as correct:

```
computed = sha256(payload)
if computed != header.digest: fail          # integrity check
if !verify(pubkey, header.digest, header.sig): fail   # signature over the digest
```

That is actually fine. This is not:

```
if !verify(pubkey, header.digest, header.sig): fail   # signature checks out
apply(payload)                                        # payload never hashed
```

The digest is signed, the payload is never compared against it, and you can attach any payload to a genuine signed header.

Tell: find every byte range the signature covers, and diff it against the byte ranges that get used. Anything used but not covered is the finding.

3. Verify-then-use across a boundary (TOCTTOU)
Verification reads the package from a staging location; application re-reads it. Between the two, something else can change it — another thread, a second upload, a symlink, an unmounted-and-remounted volume. The classic embedded shape is verify-in-`/tmp`, apply-from-`/tmp`, with the upload endpoint still live.

Tell: verification and application take a *path*, not a buffer. If both name a file, ask who else can write it in between.

4. Rollback
Signature checks out — because it is the vendor's real signature on the vendor's real firmware from three years ago, with the bug you are trying to reach. No signature scheme prevents this; it needs a monotonic version counter that the device refuses to go below, stored where the update cannot rewrite it.

Tell: is there a version comparison at all, and is the stored version in OTP, in a protected partition, or in the same writable blob the update supplies?

5. Failure that is not fatal
Verification fails, the function logs and returns non-zero, and the caller ignores the return value or falls through to a recovery path that skips verification. Recovery modes are where this lives: the device must be able to un-brick itself, so the un-bricking path takes unsigned images, and that path is reachable in normal operation.

Tell: every caller of the verify function. One of them frequently does not check.

6. Parse before verify
The header is parsed — lengths read, sizes allocated, fields dereferenced — before anything is authenticated. Now your memory-corruption surface is reachable pre-authentication, and the signature never mattered because you never needed a valid one. This is the failure mode where E2 and Track B meet: the update parser is an unauthenticated attack surface even on a device with perfect signing.

Tell: the order of operations in the entry function. Any `malloc(header.len)` above the verify call is the finding.

7. Transport-only trust
The update is fetched over HTTPS and not signed, so authenticity depends entirely on the TLS session. Then you find that certificate verification is disabled — which in embedded is the norm, because the device has no reliable clock and expired-certificate errors generate support calls. DNS redirection, and the device installs your firmware.

Tell: `CURLOPT_SSL_VERIFYPEER 0`, `--no-check-certificate`, an empty CA bundle, or a plain `http://` update URL.

The ordering that makes it correct

Every one of failure modes 2, 3, 5 and 6 is an *ordering* defect rather than a cryptographic one. The safe sequence is short:

1. Read the package into memory you control, wholly, before inspecting it.
2. Locate the signature by a **fixed** offset — not one the package supplies.
3. Verify the signature over **every byte** that will subsequently be used.
4. On failure, stop. Not log-and-continue; stop.
5. Only now parse the header, and bound every length against the size you already read.
6. Check the version against a stored counter the package cannot write.
7. Apply from the **same buffer** you verified, never re-read from disk.

Read that list against any verifier and the defects announce themselves. Steps 3 and 7 are the two that get skipped in production code, and step 5's position — *after* step 4 — is the one that gets reordered for convenience, because parsing the header first is how you find out where things are.

That tension is real: you frequently do need a field from the header to know where the signature is. The resolution is that the *signature location* must be fixed by format, not by content — a trailer at a known offset from the end, or a fixed-size header prefix. A format that stores its signature offset inside the region the signature covers has a chicken-and-egg problem it cannot solve, and implementations solve it by trusting the field.

Writing it up

Rank findings by what an attacker needs. Absent signing (mode 1) is remote code execution for anyone who can reach the endpoint. Rollback (mode 4) needs a genuine old image, which is usually public. A TOCTTOU (mode 3) needs a race and local access. Same CWE family, three very different reports.

And separate *root cause* from *mechanism* in the recommendation: "sign the update" is not the fix for mode 2, mode 3, mode 5 or mode 6, all of which occur in systems that already sign. The fix is the ordering above.

Module E3

N-day reproduction, and how to pick a target

Everything before this module was scaffolded. The binaries were chosen to contain what you were looking for, the emulation was known to work, and the answer existed. This module removes the scaffolding while keeping one thing: the guarantee that a bug is there.

That guarantee is the entire value. When your reproduction fails — and the first several will — the question is never "is there a bug here?" It is always "what did I do wrong?", which is a question

you can answer. Hunting 0-day removes both the scaffolding and the guarantee at once, so a week of nothing is indistinguishable from a week of looking in the wrong place.

Reproduce n-days until the mechanics stop costing you attention. Then hunt.

Target selection is most of the outcome

The instinct is to pick an interesting device. The correct criterion is **availability**, and it is worth being mechanical about it. Score a candidate on five things before spending a day on it:

Criterion	Why it decides
Firmware is downloadable	No download means Track F, a UART, and a week of hardware work before you have started. Vendor support pages, and the archive sites that mirror them, are the first stop.
The vulnerable version is still downloadable	The one people miss. Vendors pull old images. If only the patched version is available you have no vulnerable target, and the CVE is unreproducible without hardware.
The patched version is ALSO downloadable	Two versions is what makes the diffing module possible, and a diff frequently shows you the bug faster than reading the advisory does.
The advisory has technical content	"Command injection in the web interface" is a month of work. "Unauthenticated command injection in `/cgi-bin/ping.cgi` via the `host` parameter" is an afternoon. Read the references field, not the summary.
It emulates	MIPS or ARM Linux with a normal init and a web server: FirmAE will probably boot it. A proprietary RTOS on a chip QEMU does not model is a different project.

A target scoring well on all five is worth more than an interesting one scoring well on three. Deliberately choose an easy first target — an old router with a detailed advisory and both images available — because the first reproduction is teaching you the workflow rather than the bug.

Where to look: NVD filtered by vendor and date, vendors with a long history of consumer networking gear, and the CVE lists that accompany published router-security research. Firmware archive sites hold versions vendors removed.

The reproduction sequence

Run it in this order, and confirm each step before the next. The discipline is the point: a reproduction that fails at step 7 with steps 1 through 6 unverified gives you nothing to work with.

1. **Get both images.** Vulnerable and patched. Record versions and checksums — your writeup needs them and so does anyone repeating your work.
2. **Extract, and confirm the extraction.** Track B, unchanged. Confirm the filesystem mounted, not that `binwalk` printed something.
3. **Read the advisory closely, then stop reading.** Take the affected component, the parameter, and the version range. Resist reading a full writeup — you will use it later to check yourself, and it is worth far more as a check than as a guide.
4. **Locate the component in the image.** The CGI, the daemon, the handler. If the advisory named a URL, find what serves it. This is A1 and A5 work.
5. **Emulate.** FirmAE or a user-mode invocation of the single binary. Confirm the service actually responds before you try to break it.
6. **Reach the vulnerable code path with benign input.** Send a request that works. Watch it in the debugger, or add a log line, or set a breakpoint — whatever proves you are reaching the function. Skipping this is the single most common reason reproductions stall: you spend two days on a payload for a handler your requests never enter.
7. **Trigger.** Now send the malicious input.
8. **Confirm you got what the CVE describes**, not something adjacent. A crash is not the CVE. A crash in the function the patch touched, reached by the input the advisory names, is.
9. **Diff against the patch** and confirm the patch would have prevented what you did.

Step 8 deserves emphasis. It is entirely possible to crash a router daemon by accident — the code is not robust — and to write that up as the CVE. The patched image is what disambiguates: if your input still works against the patched build, you reproduced something else. That check costs ten minutes and it is what separates a reproduction from a coincidence.

The first three steps, as commands — because getting the provenance right at the start is what makes step 9 possible at the end:

```
# 1. Both images, and record what they are. Do this BEFORE any analysis.
mkdir -p vuln patched && cd vuln
wget https://<vendor>/firmware/DIR-XXX_1.03.12.bin
cd ../patched && wget https://<vendor>/firmware/DIR-XXX_1.03.14.bin
cd .. && sha256sum vuln/*.bin patched/*.bin | tee images.sha256

# 2. Extract, and CONFIRM it rather than trusting binwalk's exit code.
binwalk -eM --directory=vuln/x  vuln/*.bin
binwalk -eM --directory=patched/x patched/*.bin
find vuln/x -name 'squashfs-root' -maxdepth 4      # did a filesystem appear?
ls vuln/x/*/squashfs-root/bin/busybox              # can you see a real binary?

# 4. Locate the component the advisory named.
```

```
grep -rl 'ping.cgi' vuln/x/*/squashfs-root/ 2>/dev/null
find vuln/x -path '*cgi-bin*' -o -name 'httpd' | head
```

Keep `images.sha256` next to the writeup. Firmware archives hold several builds under similar names, and a reproduction against an unidentified image cannot be checked by anyone — including you, three weeks later.

What a reproduction proves

It proves: you can go from a public advisory and a vendor firmware image to a working trigger, using extraction, static analysis, emulation and dynamic analysis. That is the day-one skill of the role, and it is legible to an interviewer in a way that "I understand buffer overflows" is not.

It does not prove you can find a bug nobody has found. That is the next step, and the module's diffing half is the bridge: the code around a patch is written by the same people who wrote the bug, frequently in the same week, and frequently containing a sibling of it. Reproducing the n-day puts you in exactly the right neighbourhood to hunt.

That is the practical reason to reproduce rather than to read: you end up holding a working environment, a working trigger, and detailed knowledge of one component's code — the three things a 0-day hunt needs and the three things reading a writeup does not give you.

Ethics, briefly and non-negotiably

Test devices you own, on a network you control. That is not legal caution dressed up as advice; it is the boundary between research and an offence.

- **Own the target.** Buy the router. They are cheap on the second-hand market, and owning it removes every question.
- **Isolate it.** Its own network segment, not routing anything you care about.
- **Do not touch other people's devices**, including devices reachable on the internet running the same firmware. A vulnerability scanner pointed at strangers' routers is unauthorised access regardless of intent.
- **Disclose coordinated.** Vendor first, with a reasonable deadline. Publish after a fix, or after the deadline with the vendor informed.
- **Know the exemptions rather than relying on them.** The DMCA's security-research exemption and the CFAA's post-*Van Buren* narrowing are real and are narrower than people assume. They protect research on devices you lawfully acquired; they do not protect testing against systems you do not own.

The disclosure drill in this module walks specific scenarios. The rule above covers most of them.

Patch diffing: reading a fix to find the bug

A vendor publishes a patched firmware image. Somewhere in it, one function changed in a way that removes a vulnerability. The advisory, if there is one, says "improved input validation".

Patch diffing is recovering the bug from the fix. It is how n-days become reproducible when the advisory is uninformative, it is how variant hunting starts, and it is one of the few genuinely mechanical routes from "a vulnerability exists here" to "here is the line".

It is also the technique with the worst signal-to-noise ratio in this course, and most of the skill is in the filtering.

The problem: everything changed

Diff two firmware versions and you get thousands of differing functions. Almost none of them are the fix. Between two releases a vendor also:

- rebuilds with a different toolchain, changing register allocation and inlining everywhere
- bumps a version string, shifting every subsequent address
- adds an unrelated feature, moving the layout
- rebases a vendored library

Raw byte diffing is useless against this — a one-byte version-string change shifts every address after it and makes the whole image look different. What makes diffing work is **function-level structural matching**: comparing functions by their control-flow graph shape, their call relationships and their constant references rather than by their bytes.

That is what BinDiff, Ghidra's Version Tracking, and ghidriff all do. They differ in interface and in how they combine correlators; the idea is the same.

Matching, and the two numbers

The tool pairs functions across the two binaries, then reports two scores per pair. They are different and conflating them wastes time.

- **Similarity** — how alike the two matched functions are. Low similarity on a matched pair means the function changed.
- **Confidence** — how sure the tool is that these two functions correspond at all. Low confidence means the *match* is doubtful, independent of the content.

The pair you want is **high confidence, low similarity**: the tool is sure these are the same function, and it changed. That combination is your candidate list.

Low confidence and low similarity means the tool has probably paired unrelated functions, which happens constantly in stripped binaries and produces the majority of the noise. High confidence and high similarity is the bulk of the image: matched, unchanged, ignore.

Unmatched functions on either side are a separate bucket and are worth a look: a function present only in the patched build may be a newly added validator, which points at what needed validating.

Ranking candidates

Even filtered, a real diff leaves dozens of changed functions. Rank them, and read in rank order rather than in address order.

Strong signals — read these first:

- A **new bounds check**: a comparison added before a copy, a length clamped, a loop condition tightened. This is the single most common shape of a real fix, and it points directly at the operation that was unbounded.
- A **size argument appearing**: `strcpy` becoming `strncpy`, `sprintf` becoming `snprintf`, a `memcpy` gaining a `min()`. Unambiguous, and the destination size tells you the buffer.
- A **new call to a validator or sanitiser**, especially one added on one path and not others — the others are your variant-hunting list.
- **Reachability narrowing**: an authentication check added ahead of a handler, or a path deleted entirely.
- The function is **reachable from an input parser** that A5′s cross-referencing put on your list already.

Weak signals — deprioritise:

- Only the epilogue or register allocation differs. Toolchain noise.
- Only a string constant or a version number changed.
- The change is confined to logging or error text.
- The function is a leaf with no path from any input.

The advisory, if it has content, narrows this enormously — a named component or parameter turns "rank forty candidates" into "check these three". Read the advisory before the diff and the diff before the writeups.

Reading a fix backwards

Once you have the changed function, invert the fix. The pattern is mechanical:

The fix added	Therefore, before the fix
`if (len > sizeof(buf)) return -1;`	`len` could exceed the buffer, and the copy below it was the bug
`strncpy(dst, src, sizeof dst)` replacing `strcpy`	`src` was unbounded and attacker-influenced
A call to `sanitize()` before `system()`	the argument reached a shell unfiltered
An authentication check at function entry	the handler was reachable unauthenticated
`snprintf` replacing `sprintf`	a format-driven write past the destination

Then answer the two questions the reproduction needs:

1. **What input reaches this?** Trace backwards from the changed function to a parser, to a socket, to a request. If nothing external reaches it, the fix may be a robustness change rather than a security one — vendors patch both, and the diff does not label them.
2. **What does the added check constrain?** That constraint is the boundary you must cross in the vulnerable build, and it is frequently a literal number you can read straight out of the patched code. A patch that clamps a length to 256 tells you the buffer is 256 bytes without your having to find it.

The part worth more than the n-day

The patched function is not the end. It is a location.

The code around a fix was written by the same developers, usually in the same week, frequently by copy-and-paste. If `ping_handler` had an unbounded copy, look at `traceroute_handler` — and expect to find the same copy, unpatched, because the vendor fixed the function named in the report rather than the pattern.

Variant analysis is this made systematic:

1. From the fix, extract the *pattern* — the unbounded operation, the missing check, the unsanitised sink.
2. Search the whole image for that pattern, not that function. Grep the decompilation, script Ghidra, or run the cross-reference tooling from A5 against the sink.
3. Filter to instances reachable from input.
4. Check each against the patched build. Any instance still present in the patched image and reachable from input is a **0-day**.

That last line is the point of the module. An n-day reproduction demonstrates capability; a variant found next door to it is original work — and it is the realistic first 0-day for most people entering this field, because the search space is small, the pattern is known, and someone has already proved the developers make that mistake.

Scaling it

For a single interesting pair, Ghidra's Version Tracking in the GUI is fine and the visual side-by-side is worth the setup.

For many pairs — every binary in a firmware image, or a vendor's whole release history — you want a command-line diffing tool that emits structured output you can filter and store. That is where ghidriff sits, and it is how a diff becomes a pipeline rather than an afternoon.

```
pip install ghidriff                    # needs GHIDRA_INSTALL_DIR set

# One pair. Markdown for reading, JSON for the triage script from this module.
ghidriff ./v1.03.12/usr/sbin/httpd ./v1.03.14/usr/sbin/httpd \
  --engine VersionTrackingDiff \
  --output-path ./diffs/ --format markdown --format json
```

```
# Record what you diffed, in the same breath. A diff of an image against
# itself produces a clean, plausible, empty result.
sha256sum ./v1.03.12/usr/sbin/httpd ./v1.03.14/usr/sbin/httpd \
  | tee ./diffs/inputs.sha256

# Every binary that exists in both trees, paired by path.
cd ./v1.03.12 && find . -type f -perm -u+x | while read -r f; do
  [ -f "../v1.03.14/$f" ] || continue
  ghidriff "$f" "../v1.03.14/$f" --output-path "../diffs/$(dirname "$f")" --format
json
done
```

The JSON is what `fw-e3-rank-candidates` consumes: it carries a confidence and a similarity per matched function, which is the pair of numbers the whole triage turns on.

Two cautions from practice, both of which cost real time:

- **Diff caches are frequently keyed by filename.** Diffing two different binaries that happen to share a name can silently return the first diff's results. If two diffs come back with identical output, check that before believing it.
- **Confirm you diffed what you think you diffed.** Record the checksums of both inputs alongside the output. A diff of an image against itself produces a clean, plausible, empty result.

Disclosure, and the boundary you do not cross

This card is short and none of it is optional. The difference between security research and a federal offence is not technique — the same commands, on two different devices, land on opposite sides of it.

This is professional practice as it is generally understood, not legal advice. For anything with real exposure, talk to a lawyer.

The boundary

You may do anything to a device you own, on a network you control.

Buy the router. Second-hand consumer networking gear is cheap, and owning it removes essentially every question — you may extract its firmware, decompile it, patch it, brick it, and publish what you find.

You may not touch a device you do not own without written authorisation.

This is where people get into trouble, usually without meaning to, and usually through one of these:

- **Scanning the internet for other instances.** You reproduced a bug on your own router; there are forty thousand of the same model exposed. Confirming the bug against even one of them is unauthorised access to someone else's computer. Shodan is fine to read; sending your trigger to a host on it is not.

- **"Just checking" a friend's or an employer's device.** Verbal permission from someone who does not own the device is not authorisation. Written permission from someone who does is.
- **The vendor's cloud service.** The router is yours; the API it talks to is not. A firmware bug reachable only through the vendor's servers needs the vendor's programme, not your curiosity.
- **Testing during an engagement, outside scope.** A signed contract authorises what it names and nothing else. Scope creep is not a paperwork problem.

The asymmetry is worth internalising: the effort to stay inside the boundary is buying a device and reading a scope document. The cost of leaving it is a criminal charge, and the intent behind it does not feature.

The legal shape, roughly

Two statutes come up in this work in the US, and both are more limited than their reputations suggest.

CFAA — unauthorised access to a computer. *Van Buren* (2021) narrowed "exceeds authorised access" to a gates-up-or-down question: whether you accessed areas you were not permitted to, rather than whether you violated a terms-of-use document. That helps researchers materially. It does not help at all with a device you have no permission to access, which was never the ambiguous case.

DOJ's 2022 charging policy says good-faith security research should not be charged. It is charging policy, not statute: it does not bind state prosecutors, it does not prevent a civil suit, and it defines "good faith" in a way that excludes anything that harms the system or uses the findings for anything other than improving security.

DMCA §1201 — circumventing access controls. The triennial exemption for good-faith security research covers circumventing protections on a device you **lawfully acquired**, in an environment designed to avoid harm. This is the provision that makes bypassing signed-firmware checks on your own router defensible. It is renewed every three years, has conditions, and rests on lawful acquisition.

Outside the US, comparable statutes exist with materially different research carve-outs — several jurisdictions have none. If you are not in the US, look up yours rather than assuming this transfers.

Coordinated disclosure, in practice

Find the contact. In descending order of usefulness: a `security.txt` on the vendor's site, a published PSIRT address, an established bug-bounty programme, `security@vendor.com`, then general support. Embedded vendors frequently have none of these, and the first message is often to a support queue that has never seen a vulnerability report.

The first message is short. That you have found a security issue, that you intend to disclose responsibly, the affected product and version, and a request for a secure channel. Do not put the exploit in an unencrypted email to a support address — it will end up in a ticketing system.

The report contains the affected versions, reproduction steps another engineer can follow, the impact, and your disclosure timeline. It should be good enough that their engineer can confirm it without talking to you, because frequently they cannot.

Set a deadline in the first message. 90 days is the widely accepted default. Stating it up front converts an awkward later conversation into a term everyone agreed to.

When the vendor does not respond — which for consumer embedded is the common case — escalate rather than giving up or going straight to publication. CERT/CC and CISA both coordinate on researchers' behalf, they have contacts you do not, and a report from them lands differently than a report from an individual. That is the correct next step after two or three unanswered attempts.

Publish after a fix ships, or after the deadline with the vendor informed. If the deadline passes with no fix, saying so factually is part of the record.

Request a CVE. MITRE directly, or through a CNA — many vendors are their own CNA, and CISA acts as CNA of last resort. A CVE with your name on it is the durable artifact.

Two edge cases worth pre-deciding

Active exploitation in the wild. The calculus changes: users need mitigation advice now, and the attackers already have the details. Coordinate with CISA rather than sitting on it or publishing unilaterally.

End-of-life products. The vendor will not fix it, so a 90-day clock achieves nothing. Disclosure here is about telling users to replace the device, and it is still worth doing — an unfixable bug in shipped hardware is exactly the thing buyers have no other way to learn about.

Why this is in the course rather than in a footnote

Two practical reasons, beyond the obvious one.

Every VR role you will apply for either asks about this or assumes it. A candidate who reproduces bugs competently and is vague about authorisation boundaries is a liability, and interviewers screen for it.

And a portfolio is a public record of what you did to what. A writeup that starts "I bought this router, isolated it on its own segment, and reported the finding to the vendor on this date" is doing work for you. One that does not say where the target came from raises a question the reader will answer for themselves.

Module E4

Symbolic execution, and when it beats fuzzing

Why this matters

Module E1 built a fuzzer campaign: generate inputs, run them, keep the ones that reach new code. It is enormously effective and it has one structural blind spot.

A fuzzer cannot get past a check it cannot guess. If the parser wants a four-byte magic value, random mutation will find it eventually — 2^{32} tries eventually. If it wants a correct CRC, or a password, or a value derived from the input by a transform, the fuzzer never arrives. The coverage map stops growing and you conclude the code behind the check is uninteresting, when in fact you never saw it.

Symbolic execution attacks exactly that. Instead of guessing inputs and watching what happens, it runs the program with the input left **unknown** and asks a solver what the input would have to be to reach a given point.

The idea

Replace the input with a **symbolic variable** — not `0x41414141` but "a 32-bit value, call it `x`". Then execute.

When execution reaches a branch on `x`, it cannot decide which way to go, because `x` has no value. So it takes **both**, and records what must be true on each side:

```
if (x > 100)     ->  path A carries the constraint  x > 100
    ...          ->  path B carries the constraint  x <= 100
```

Every path accumulates a list of constraints — the conditions that had to hold for execution to have come this way. That list is the **path constraint**.

When you find a path that reaches somewhere interesting, you hand its constraints to a **solver** (angr uses Z3 underneath) and ask: is there an assignment of `x` satisfying all of these? If yes, the solver gives you one — and that value is an input that drives the real program down that exact path.

You did not guess it. You derived it.

What that buys you

Checks stop being walls. A magic value, a checksum, a password comparison, a serial-number transform — these are just constraints. The solver inverts them. This is the canonical use, and it is why "solve a crackme with angr" is the standard first exercise.

You can ask directed questions. Not "what happens if I fuzz this for nine hours" but "give me an input that reaches line 0x4011a0" or "give me an input where this length field exceeds the buffer". Fuzzing is a search; symbolic execution is a query.

It proves absence, narrowly. If the solver reports a path's constraints are unsatisfiable, that path is genuinely unreachable. A fuzzer can never tell you that.

What it costs: path explosion

Every branch on symbolic data doubles the number of paths. A loop whose bound is symbolic produces a new path per iteration. Real programs branch constantly.

The result is that naive symbolic execution on a non-trivial binary exhausts memory long before it reaches anything interesting. This is not a tuning problem; it is the fundamental limit of the technique, and it is why symbolic execution has not replaced fuzzing.

Three practical mitigations, all of which mean *doing less symbolic execution*:

- **Start close to the target.** Do not begin at `main` and hope. Begin at the function that handles the input, with a state you construct yourself.
- **Hook expensive functions.** Replace `memcpy`, `printf`, crypto routines and anything with loops over symbolic lengths with a summary that returns the right answer without exploring.
- **Prune aggressively.** Tell the explorer which addresses to `avoid` so paths through error handlers die immediately rather than being carried along.

Choosing between them

Situation	Reach for
Unknown input format, want memory-safety bugs	**Fuzzing**
A specific check you cannot get past	**Symbolic execution**
"What input reaches this line?"	**Symbolic execution**
Broad exploration of a big parser	**Fuzzing**
A crackme, licence check, or key derivation	**Symbolic execution**
Deep code behind a loop with a symbolic bound	**Neither, directly** — hook the loop, or fuzz past it

The productive combination is **both**: use symbolic execution to generate an input that gets past the check, then hand that input to the fuzzer as a seed and let it explore the region behind it. The techniques are complementary in exactly the place each is weakest.

The shape of an angr solve

Every worked example you will read has this structure, and recognising it makes them all legible:

1. **Load** the binary into a project.
2. **Build a state** — either at the entry point, or a *blank state* at the address you care about, with registers and memory set up by hand.
3. **Make the input symbolic** — a bitvector of the right width, written into a register, a buffer, or stdin.
4. **Hook** anything you do not want explored.
5. **Explore**, with a `find` address and usually an `avoid` set.
6. **Solve** the found state's constraints for the symbolic input.

Steps 2 and 3 are where the architecture modules stop being background. Constructing a blank state means knowing which register holds the first argument (A2, A3) and where the stack should point — the same knowledge that built the harness in E1 and the payload in A4.

What to carry forward

- Run the program with the input **unknown**, and let a solver tell you what it must have been.
- A **path constraint** is the list of conditions that had to hold to come this way.
- Symbolic execution **defeats checks fuzzing cannot guess**, and can prove a path unreachable.
- **Path explosion** is the fundamental cost. Every mitigation amounts to exploring less.
- **Start close, hook aggressively, avoid early.**
- The best use is often to **generate a seed** that unblocks a fuzzer.

Next: the emulation frameworks — Unicorn, Qiling, Renode — and choosing the right one.

Unicorn, Qiling, Renode, QEMU: picking the right layer

Why this matters

Four tools, overlapping names, and every tutorial insists its one is the right one. They are not competitors — they sit at **different layers**, and picking the wrong layer is the most common way this work stalls.

Emulate too little and the code faults on the first thing you did not provide. Emulate too much and you spend a week booting an image to test one function. The whole skill is choosing the narrowest layer that still reproduces the behaviour you care about.

The ladder

Layer	Tool	Emulates	You supply
Function	**Unicorn**	the CPU, and nothing else	memory map, registers, every stub
Binary	**Qiling**	CPU + syscalls + a filesystem view	a rootfs, and hooks for the awkward parts
Board	**Renode**	CPU + modelled peripherals + timers	a platform description
System	**QEMU**	a whole machine	a kernel and a disk, or the firmware image

Read it top to bottom as increasing fidelity and decreasing speed.

Unicorn is a CPU and a flat address space. It has no concept of a process, a syscall, a file or a heap. You map memory, write your code and data into it, set the registers, and say "execute from here to there". If the code calls `malloc`, nothing happens — there is no `malloc` — so you hook the address and supply a return value yourself.

That sounds punitive and it is the point: nothing is emulated that you did not ask for, so it is extremely fast and completely predictable. Ideal for one function.

Qiling is built on Unicorn and adds the operating system. It emulates Linux syscalls, gives the binary a rootfs to see, and lets a dynamically-linked MIPS `httpd` from an extracted firmware image run on your x86 laptop. When a firmware binary "just needs to run", this is usually the answer.

Renode is a different shape: a full-board simulator for microcontrollers, with **modelled peripherals** — UARTs, timers, GPIO, sensors. It can run unmodified Cortex-M firmware, including the RTOS, and it can simulate several nodes on a network at once. This is the Track C tool, and it is the only one on this list that answers "what does this firmware do when the ADC reads 0x300".

QEMU you have already met. User-mode (`qemu-mipsel`) runs a single foreign binary and is what AFL++'s QEMU mode is built on; system-mode boots a whole machine and is what FirmAE drives in Track B.

The same job at each layer, so the trade is concrete rather than abstract:

```
# UNICORN -- one function, nothing else. You own every byte of the address space.
from unicorn import *
from unicorn.mips_const import *

CODE, STACK = 0x400000, 0x7f000000
mu = Uc(UC_ARCH_MIPS, UC_MODE_MIPS32 + UC_MODE_LITTLE_ENDIAN)
mu.mem_map(CODE, 0x10000)
mu.mem_map(STACK, 0x10000)
mu.mem_write(CODE, open("func.bin", "rb").read())
mu.reg_write(UC_MIPS_REG_SP, STACK + 0x8000)
mu.reg_write(UC_MIPS_REG_A0, CODE + 0x2000)      # arg 0: pointer to input

def stub_malloc(uc, addr, size, ud):             # there is no malloc; supply one
    uc.reg_write(UC_MIPS_REG_V0, 0x600000)
mu.hook_add(UC_HOOK_CODE, stub_malloc, begin=0x400abc, end=0x400abc)

mu.emu_start(CODE, CODE + 0x120)                 # run exactly this range
print(hex(mu.reg_read(UC_MIPS_REG_V0)))
```

```
# QILING -- the whole binary, with the image's own rootfs and libraries.
from qiling import Qiling
ql = Qiling(["./squashfs-root/usr/sbin/httpd"], "./squashfs-root", verbose=1)
ql.run()
```

```
# RENODE -- a modelled board, for Cortex-M firmware that touches peripherals.
renode
  (monitor) mach create "target"
  (monitor) machine LoadPlatformDescription @platforms/cpus/stm32f4.repl
  (monitor) sysbus LoadBinary @firmware.bin 0x08000000
  (monitor) sysbus.cpu PC 0x08000190      # reset handler, Thumb bit cleared
```

```
  (monitor) showAnalyzer sysbus.usart2    # watch a peripheral as it runs
  (monitor) start
```

```
# QEMU -- user mode for one binary, system mode for a whole machine.
qemu-mipsel-static -L ./squashfs-root ./squashfs-root/usr/sbin/httpd
qemu-system-mips -M malta -kernel vmlinux -hda rootfs.ext2 -nographic \
  -append "root=/dev/sda console=ttyS0"
```

Note what shrinks as you climb: the Unicorn block is thirteen lines and every one of them is a decision you made, while the Qiling block is three and the decisions are Qiling's. That is the whole trade — control against setup cost — and it is why the answer to "which one" is decided by how much of the environment your target actually touches.

Choosing

Ask what the code needs, and stop climbing as soon as you have it.

- **A pure function** — a parser, a checksum, a key derivation, a decompressor. It touches its arguments and returns. → **Unicorn.** Thousands of executions per second.
- **A binary that opens files, uses sockets, calls libc. → Qiling.**
- **Firmware that talks to hardware** — reads a peripheral register, waits on a timer, drives a GPIO. → **Renode.**
- **A whole Linux firmware image with its web stack. → QEMU system mode**, via FirmAE.

The rule from E1 restated: **the narrowest layer that still reaches the behaviour.** Every layer you add is throughput you pay for continuously and setup you pay for once.

What goes wrong at the function layer

Unicorn's austerity is where beginners lose a day, and the failures are worth recognising by their symptom because none of them announce themselves clearly.

No stack. You mapped code and data and forgot that the function's prologue writes to the stack immediately. Symptom: a fault on the very first instruction that touches memory. Fix: map a stack region and point `SP` at somewhere sensible *inside* it — not at its base, since the stack grows down and there must be room below.

Arguments in the wrong place. You wrote the input into memory but did not set the argument register to point at it. The function reads whatever `$a0` / `r0` happened to contain. Symptom: it runs and returns nonsense rather than faulting. This is where A2 and A3 are load-bearing — the calling convention *is* the interface.

Unmapped memory. The function touches a global, a jump table or a literal pool you did not map. Symptom: a fault at an address that looks like it belongs to the binary. Fix: map the whole image at its correct base rather than only the function's bytes.

Unhooked calls. The function calls `malloc`, `strlen`, or a vendor helper. There is no libc, so execution runs off into unmapped memory. Fix: hook the address and emulate the effect.

No stop condition. You started execution and never said where to stop, so it runs past the `return` into whatever follows. Always give an explicit end address — usually the return address you planted on the stack.

That last pair is worth stating as a habit: **you must plant a return address you control and stop there.** It is the same saved-return-address slot from module A4, used constructively for once.

Where symbolic execution fits

angr uses this same idea — the *blank state*. Rather than starting at `main`, you create a state at the function's address, set up memory and registers exactly as this card describes, and explore from there. The setup work is identical; only the execution semantics differ, concrete versus symbolic.

So the harness skill transfers directly. If you can build a Unicorn harness for a function, you can build an angr blank state for it, and vice versa.

What to carry forward

- Four layers: **function (Unicorn), binary (Qiling), board (Renode), system (QEMU).**
- Choose **the narrowest that reaches the behaviour** — fidelity costs throughput and setup.
- Unicorn gives you a CPU and nothing else: **map the stack, set the argument registers, hook the calls, and give an explicit stop address.**
- A harness that returns nonsense without faulting usually means the **arguments** were wrong.
- angr's blank state is the **same setup work**, so the two skills are one skill.

Next: drill the tool choice, order an angr solve, and build a solver that prunes infeasible paths.

Track F

Module F1

The four interfaces, and what each one gets you

You reach for hardware for exactly one reason: the firmware is not downloadable. Everything in Tracks A through E ran off an image you could fetch. When there is no image — a niche vendor, a discontinued product, an industrial device with no public support page — the bytes are on the board and you go and get them.

That is the honest framing. Hardware work is a means of acquisition and a means of instrumentation, not a discipline you enter because it is more advanced.

Read this before you buy anything. The point of a hardware-free module about hardware is that the expensive mistakes are all decisions you make before you touch the board.

The four, ranked by what they cost you

Interface	Gets you	Gear	Risk to device
UART	a console — often a root shell or a U-Boot prompt	~$5 USB-UART adapter	very low
SPI flash	the entire image, in-circuit or off-chip	~$8 CH341A + SOIC-8 clip	low in-circuit, real off-chip
JTAG / SWD	halt, single-step, read memory, dump internal flash	~$10 (Pi) to ~$40 (Tigard)	low, unless fuses are involved
I2C	EEPROM contents — config, keys, calibration	shares the SPI gear	low

Start at the top. A UART console on a Linux router is frequently the whole job: `cat /dev/mtd0 > /tmp/x` and a transfer, or `sf read` from U-Boot, and you have the image without touching a flash chip.

UART — the first thing you look for

Three or four pads in a row, near the SoC, frequently labelled, frequently populated with nothing. You need three of them: **TX**, **RX**, **GND**.

Finding them, in order of effort:

1. **Look.** Silkscreen saying `TX`, `RX`, `GND`, `J1`, `CON2`. Vendors label debug headers far more often than you would expect, because their own engineers needed them.
2. **Continuity to ground.** With the board off, a multimeter in continuity mode between a candidate pad and a known ground (a shield can, a large plane, the barrel jack's outer contact). The one that beeps is GND.
3. **Voltage on boot.** Power the board with a multimeter on a candidate pad. TX sits at the idle level (3.3 V or 1.8 V) and drops as bursts of data go out at boot — you will see the reading flicker. RX idles high and does not move. A pad that sits at 0 V is ground or unused.
4. **Logic analyser.** Ten dollars, and it turns the above from inference into a picture. Capture through the boot, and the pin with framed serial traffic on it is TX.

Determining baud. 115200 is the overwhelming default; 57600 and 9600 are the common others. If the console prints legible text at 115200, you are done. If it prints consistent garbage, you have the right pin and the wrong rate — work down the standard list. If it prints *nothing*, you likely have the wrong pin or a swapped connection.

```
# Opening the console. Any one of these; all three are on most distros.
picocom -b 115200 /dev/ttyUSB0          # ctrl-a ctrl-x to quit
```

```
screen /dev/ttyUSB0 115200              # ctrl-a k to quit
minicom -D /dev/ttyUSB0 -b 115200

# Which device did the adapter enumerate as?
dmesg | tail -5                         # look for ttyUSB0 / ttyACM0
ls -l /dev/ttyUSB* /dev/ttyACM*

# Capture the boot log to a file -- you will want to re-read it.
picocom -b 115200 /dev/ttyUSB0 --logfile boot.log

# Wrong rate? Work down the list, in this order.
for b in 115200 57600 38400 19200 9600; do
  echo "== $b"
  timeout 5 picocom -b "$b" -q /dev/ttyUSB0
done
```

If your user cannot open the port, you are in the wrong group rather than on the wrong pin: `sudo usermod -aG dialout $USER`, then log out and back in.

Two facts worth knowing before your first attempt:

Voltage matters and 5 V will damage a 3.3 V device. Check the adapter is set to the board's level. Most embedded boards are 3.3 V; some are 1.8 V, and a 3.3 V adapter on a 1.8 V line is a bad day. Measure before you connect.

TX goes to RX. The adapter's TX connects to the board's RX and vice versa. Getting it backwards produces silence rather than damage, and it is the single most common reason a first attempt shows nothing. If you see nothing, swap them before you conclude anything.

GND connects first. Always.

What a console actually gets you

Three outcomes, in descending frequency:

A root shell, no password. Common on consumer gear. The serial console is how the vendor's own engineers debug, and it frequently ships enabled.

A login prompt. Now you need credentials — which is what an image would have given you, so this is circular unless you take the SPI route.

A U-Boot prompt, if you interrupt autoboot with a keypress in the first couple of seconds. This is the best of the three, and module E2 covered why: `sf read` and a transfer gets you the flash contents, `printenv` gets you the layout, and `setenv bootargs ... init=/bin/sh` gets you a root shell on a device whose Linux console is locked.

Vendors who have thought about this disable the console, remove the header, or set `bootdelay=0`. Vendors who have thought about it *harder* leave a console that only prints and does not accept input — worth testing rather than assuming.

SPI flash — when the console does not cooperate

The image lives on a flash chip, usually an 8-pin SOIC package a few millimetres across, marked with something like `W25Q64` (Winbond, 64 Mbit = 8 MB) or `MX25L12835F` (Macronix, 128 Mbit = 16 MB). Read the marking with a phone camera and search it — the datasheet gives you the capacity and the pinout, and the capacity tells you how big your dump should be, which is how you know you got all of it.

Two ways in:

In-circuit, with a SOIC-8 clip: attach the clip, power the flash chip from the programmer, and read with `flashrom`. No soldering, no desoldering, reversible. The complication is that the SoC is on the same bus and can interfere; holding it in reset frequently fixes it, and some boards simply will not read in-circuit.

Off-chip: desolder the flash, read it in a socket, solder it back. Always works, needs hot air and practice, and gives you a real chance of destroying the target. Practise on a board you do not care about first.

The dump is a flat blob with no header, and it goes straight into the Track B workflow — `binwalk`, extract, mount. Take **at least two dumps and compare them**: a clip with one poor contact yields a corrupt image that looks entirely plausible, and finding that out after a day of analysis is worse than the two minutes it costs to check.

```
# 1. Does the programmer see the chip at all? Do this before anything else --
#    if the part is not detected, nothing below will help.
flashrom -p ch341a_spi

# 2. Two dumps, then diff. This is the step people skip.
flashrom -p ch341a_spi -r dump1.bin
flashrom -p ch341a_spi -r dump2.bin
cmp dump1.bin dump2.bin && echo "IDENTICAL - trust it" || echo "DIFFER - re-seat"

# 3. Sanity-check the size against the part number BEFORE analysing.
#    W25Q64 is 64 Mbit = 8 MiB = 8388608 bytes. Anything else means a
#    mis-detected chip or a truncated read.
ls -l dump1.bin && sha256sum dump1.bin

# 4. Straight into the Track B workflow.
binwalk dump1.bin
```

If the SoC fights you for the bus, hold it in reset while reading — most boards expose a reset pad or a pin you can pull low:

```
flashrom -p ch341a_spi -r dump.bin --flash-name    # confirm the part first
flashrom -p linux_spi:dev=/dev/spidev0.0,spispeed=1000 -r dump.bin   # Pi, slow clock
```

Dropping the SPI clock is the single most effective fix for a flaky in-circuit read; 1 MHz reads succeed on boards where the default rate returns garbage.

JTAG / SWD — for MCUs and for debugging

JTAG is the general debug interface — typically TDI, TDO, TCK, TMS, and often TRST plus reset. **SWD** is ARM's two-wire replacement (SWDIO, SWCLK) and is what you will find on almost anything Cortex-M.

It gets you what UART and SPI cannot: **halt the core, read and write memory and registers, single-step, and dump internal flash.** On an MCU whose firmware lives inside the chip rather than on an external SPI part, SWD plus OpenOCD is the only route to the image — and that image is exactly the flat blob Track C works on.

Finding the pins is the hard part on an unlabelled board, which is what a JTAGulator automates and what a logic analyser plus patience does for free.

Expect it to be disabled on production hardware. Read-out protection — RDP on STM32, the debug-disable fuses elsewhere — is the standard hardening, and it is the boundary where the next module's techniques begin.

```
# Connect. The two -f files are "which adapter" and "which target".
openocd -f interface/stlink.cfg -f target/stm32f1x.cfg
openocd -f interface/ftdi/tigard-swd.cfg -f target/stm32f4x.cfg

# In a second terminal: talk to it, halt the core, dump internal flash.
telnet localhost 4444
  > halt
  > flash banks                     # confirm base address and size first
  > dump_image fw.bin 0x08000000 0x100000
  > resume

# One-shot, no interactive session:
openocd -f interface/stlink.cfg -f target/stm32f1x.cfg \
  -c "init; halt; dump_image fw.bin 0x08000000 0x100000; exit"

# Read-out protection check -- do this BEFORE assuming a dump failed.
openocd -f interface/stlink.cfg -f target/stm32f1x.cfg \
  -c "init; stm32f1x options_read 0; exit"
```

A dump that comes back all `0x00` or all `0xFF` is the RDP signature, not a wiring fault. Check the option bytes before re-seating anything — and note that clearing RDP mass-erases the part, which destroys the very image you came for.

Choosing what to buy

Buy in the order you will use things:

Tier 1, about $25. A USB-UART adapter with a selectable voltage, an 8-channel logic analyser, and basic tools — jeweller's screwdrivers, a spudger, a multimeter. This is enough to find and use UART, which is enough for most consumer Linux devices.

Tier 2, about $50. A Tigard (FT2232H-based, labelled SPI/JTAG/SWD/UART, works with OpenOCD and flashrom out of the box) or a cheaper Bus Pirate, plus a CH341A and a SOIC-8 clip for flash. Now you can dump chips and speak SWD.

Later, if you get there. A JTAGulator for pin discovery, and a soldering iron with hot air for chip-off work.

Targets. An STM32 "Black Pill" dev board is a few dollars and is the safe teaching target for UART, SWD and OpenOCD — documented, replaceable, and impossible to embarrass yourself with. A cheap second-hand consumer router is the real workflow: pick one with documented UART headers or OpenWrt support, and buy two so the first mistake is not the end of the exercise.

Before you touch anything

Own the device. Everything in the disclosure card applies unchanged, and physical access does not soften it. A device on your bench that belongs to your employer is not yours.

Photograph the board before and after, both sides, in focus. You will want to know where a wire went, and a photograph of the intact board is the only reliable record of what you changed.

Expect to destroy one. Budget for it rather than being careful enough to avoid it — being careful enough is what a second board teaches you.

Track G

Module G1

Running an assessment: scope, time, and what you owe the reader

Every module before this one gave you a bounded problem. This one gives you a device and a deadline.

That is the actual job. A firmware VR engineer is rarely handed a function and asked to find the bug in it; they are handed an image and two weeks, and the first decision — how to spend the two weeks — determines the outcome more than any technique in the course.

Budget the window before you start

The nine FSTM stages are not equally expensive and are not equally productive. A workable split for a two-week image-only assessment:

Stages	Days	What “done” means
1–2 recon, obtain	0.5	Both images in hand, versions and checksums recorded, the product’s role and exposure understood
3–4 analyse, extract	1	Filesystem mounted, layout characterised, nothing left unidentified
5 static analysis	3	Every network-reachable entry point enumerated; the dangerous-sink list built and cross-referenced
6 emulation	1.5	The target service running and responding, or a written decision that it will not and why
7–8 dynamic, runtime	2	Reaching handlers with instrumented input, one confirmed crash or injection
9 exploitation	2	One finding driven to demonstrated impact
report	2	Written, evidenced, reviewed once by you a day later

Two things about that table matter more than its exact numbers.

Stage 5 is the largest block, and it is where findings come from. The temptation is to rush it to reach the interesting work. The interesting work is downstream of it: exploitation without a target is nothing to do.

Emulation is time-boxed, and the box is enforced. Emulation is the single biggest time sink in firmware work — it can absorb a whole engagement and produce a booting router and no findings. When the box runs out, write down what failed and move to static analysis of the same component. A report saying “the web service could not be emulated; the handler was reviewed statically and the following defect identified” is a good report. A report saying “we spent nine days on FirmAE” is not.

Enumerate the surface before you go deep

The failure mode of a first assessment is depth-first: find something interesting in the first hour and spend the fortnight on it. Sometimes that pays. Usually you finish the engagement having examined 3% of the device and having missed the unauthenticated endpoint next door.

Go **broad first, then deep**:

1. **Enumerate every entry point.** Network services and their ports, CGI handlers and their URLs, the update mechanism, IPC and local sockets, any parser reachable from a file the device accepts.
2. **Mark which are pre-authentication.** This is the single most valuable column in your notes. A stack overflow behind an admin login and an unbounded copy in the update parser are the same defect and not remotely the same finding.
3. **Cross-reference the dangerous sinks** — the A5 tooling — and intersect the result with the reachable set.
4. **Now** pick your targets, ranked by reachability first and technique cost second.

Keep the enumeration in the report even where nothing was found. A reader needs to know what was looked at; a list of examined-and-clean components is what separates "we found three things" from "we found the three things that were there".

Ranking findings

Rank by **impact × reachability**, and never by how hard the work was. Reviewers consistently overweight the finding that cost them the most effort, and that is exactly backwards from the reader's interest.

A workable order:

1. Unauthenticated remote code execution
2. Unauthenticated information disclosure of credentials or keys
3. Authenticated RCE (and say what the authentication is worth — a hardcoded default password makes this row 1)
4. Unauthenticated denial of service on a device whose function is safety- or availability-critical
5. Missing hardening: no secure boot, no signed updates, debug interfaces live
6. Authenticated lower-severity issues
7. Informational: outdated components with no demonstrated reachable defect

Row 7 deserves care. "This image contains BusyBox 1.19 which has 14 CVEs" is not a finding unless you show the vulnerable applet is present and reachable. Version-scanner output pasted into a report is the fastest way to lose a reader's trust in the rest of it.

What you owe the reader

A report is not a diary. Each finding carries, at minimum:

- **What it is**, in one sentence a non-specialist can act on.

- **Where**, precisely: binary, function, offset or line, parameter.
- **How to reproduce it**, well enough that their engineer needs nothing from you.
- **Evidence** — the crash, the response, the shell. Not a description of it.
- **Impact**, argued rather than asserted, and stated in terms of what an attacker gains rather than what class the bug belongs to.
- **Root cause**, separate from the technique used to reach it.
- **Remediation** that addresses the root cause. "Sanitise input" is not remediation; "bound the copy on line 214 to `sizeof dst`, and reject requests whose `Content-Length` exceeds the buffer before parsing" is.

And two sections people leave out, both of which make the rest more credible:

What was not tested. Components out of scope, interfaces you had no hardware for, paths that could not be emulated. Its absence invites the reader to assume you covered everything, and the first time they find something you did not look at, everything else you wrote becomes suspect.

What was tested and found clean. Negative results are results. They tell the vendor where their engineering worked, and they tell the next assessor where not to spend a week.

Being wrong in public

You will report something that turns out not to be exploitable. Everyone does. The way to survive it is to have been precise about what you demonstrated:

> "The copy is unbounded and reachable with a 200-byte parameter, which reliably crashes the daemon. I did not establish control of the program counter; the frame layout suggests the saved return address is reachable at offset 268, but I was unable to confirm this under emulation."

That paragraph is honest, useful, and cannot be wrong. Compare:

> "Unauthenticated remote code execution."

which is a claim you did not demonstrate and which, if it fails to reproduce, costs you the reader's belief in every other finding in the document.

State what you showed. State what you inferred. Keep the line between them visible, and the report survives being checked.