Reverse Engineering — Ghidra, x86-64 and RISC

Concept cards — generated 2026-08-05

Contents

Track R

Module R0

The first thirty minutes

An unknown binary is intimidating for one reason: there is no obvious place to start, so people start at the entry point and scroll. That is the worst available option, and this card is about the alternative.

The goal of the first thirty minutes is not to understand the program. It is to know enough to ask a specific question, because a specific question is what makes every later hour productive.

The loop

```
triage -> orient -> name -> type -> cross-reference
      -> hypothesise -> verify -> record
              |
              +--- and back to the specific question you now have
```

Analysis is note-taking under uncertainty. You will be wrong repeatedly, and the only thing that stops that being expensive is writing down what you believed and why.

What you can know before disassembling anything

Every one of these comes from the headers, in seconds, and each rules out a large amount of work:

Format and architecture. PE or ELF, 32- or 64-bit, x86 or ARM or MIPS — and for the RISC targets, **endianness**, which is the first irreversible decision you will make.

Subsystem. Native means a kernel driver, and that changes the ABI, the threat model and how you would ever run it.

Stripped or not. Names change the cost of everything downstream.

Imports. The capability list — the single fastest characterisation available. Filesystem, network, registry, kernel, crypto.

Exports. For a library, the attack surface, and a better starting point than the entry point.

Strings. Error messages, paths, protocol names, format strings. Run this sweep before reading a single instruction.

Mitigations. ASLR, NX, Control Flow Guard — which belong in an impact assessment later, and which you should read now while you are in the headers.

Build fingerprints. The Rich header on MSVC binaries, a `.comment` section on GCC-built ELF, and the debug directory's PDB path — which frequently leaks project names and build-machine layout even when no symbols are present.

The entry point is not `main`

```
MSVC:  mainCRTStartup -> __scrt_common_main_seh -> main
ELF:   _start -> __libc_start_main -> main
DLL:   DllMain, called for attach and detach
Driver: DriverEntry
```

Starting at the entry point of a C program means reading several hundred instructions of runtime initialisation before reaching anything the programmer wrote. It is the single most common way to waste a first hour.

Find `main` by its shape instead: on ELF it is the function whose address is passed to `__libc_start_main`; on MSVC it is what the startup path calls after initialisation. Or skip it entirely — for a library, the exports are the real entry points.

What survives stripping

Stripping removes names. It does not remove:

- **exports and imports** — the loader needs them;
- **`.pdata`** on 64-bit PE, which is a function table with frame sizes;
- **RTTI**, if the C++ was built with it, which gives class names and the inheritance graph;
- **relocations**, which mark every location holding an absolute address;
- **string references**, which are as good as they ever were.

A stripped binary is meaningfully harder than one with symbols and much easier than a blank page. Knowing what is still there stops you treating it as one.

Choose the question before the tool

> "Understand the binary" is not a plan. "Find the code that parses the network header" is.

A specific question tells you which sweep to run and when you are done. The question can be wrong — that is fine, and it is why the loop has a *hypothesise* step and a *verify* step as separate things.

Good opening questions, in rough order of how often they are the right one:

- What does this handle from outside — network, file, IPC, IOCTL?
- Where is the parser?
- What does this export, and what can a caller reach through it?
- What does it do with the values it parses?

The findings log

Four columns, and the third is the one people skip:

address	claim	**evidence**	confidence
`0x1400012a0`	parses the packet header	string “hdr too long” at `0x140012a08`; called from the recv loop	confirmed
`0x140001c40`	the retry path	shape only — has not been checked	guess

Without the evidence column you will re-derive the same conclusions next week. Without the confidence column you will not remember which conclusions you would defend — and that distinction is exactly what makes an analysis usable by anyone else, including you.

What to take away

1. **The headers answer most of the orientation questions** before you disassemble anything.
2. **The entry point is not `main`**, and for a library the exports are the real entry points.
3. **Stripping removes names, not structure** — exports, imports, `.pdata` and RTTI survive.
4. **Pick a specific question first.** It decides which sweep to run and when you are finished.
5. **Log the evidence and the confidence**, not just the conclusion.

Module R0M

Ghidra mechanics

The tool vocabulary the rest of the course speaks in. This is the shortest card in the pack on purpose — tool procedure is the least transferable thing here and the most likely to change under you.

Two things on this page are not procedure and will still be true in ten years: **the importer’s first decision is irreversible**, and **the decompiler is a model, not the program.**

The importer, and the decision you cannot take back

Loading a file asks you for a **language and compiler spec** — processor, endianness, bitness, and which calling convention to assume.

For a PE or a normal ELF this is read from the headers and is usually right. For a raw binary — a firmware dump, an extracted blob — **you are asserting it**, and there is nothing to check you.

> Get the processor or the endianness wrong and the bytes do not fail to decode. They decode into **plausible instructions that mean nothing**, and you can work for an hour before the wrongness becomes obvious.

MIPS ships big- and little-endian. ARM is usually little, not always. This is the first irreversible decision in any RISC analysis, and re-importing is cheap — far cheaper than the hour.

The **image base** matters for the same reason: it decides every address you write down. A raw blob loaded at 0 gives addresses that will not match anyone else's notes.

Auto-analysis, and how each analyzer fails

Auto-analysis is a set of independent passes, and knowing which one produced a result tells you how much to trust it:

- **Function Start Search** finds prologues. It misses functions that do not start conventionally, which is why `.pdata` is worth checking separately.
- **Decompiler Parameter ID** guesses signatures from which registers are read before being written. Reliable for ordinary functions; wrong for varargs, hand-written assembly, and anything where a float consumed an argument slot.
- **Demangler** turns decorated symbols into names. It tells you what the symbol says, which is a fact about the symbol and not about the code.
- **Aggressive Instruction Finder** disassembles bytes that nothing references. It recovers real code and also **manufactures functions out of data** — if a function looks like nonsense and nothing calls it, suspect this before suspecting the binary.

The decompiler is a model

Ghidra lifts instructions into **P-code**, an intermediate representation, and decompiles from that. This is why one decompiler serves every processor: the ARM and MIPS front ends produce the same IR, so all the analysis is shared.

It also means **the output is a reconstruction, not a translation.** The decompiler infers types, widths, signedness and signatures, and where it cannot it guesses. The artifact vocabulary is how it tells you it guessed:

What you see	What it means
`undefined4`, `undefined8`	a value of known SIZE but unestablished type
`undefined`	not even the size is settled
`uVar1`, `iVar2`	invented names; the `u`/`i` is the inferred signedness
`unaff_EBX`	a register read that the model cannot source - often a wrong signature
`in_stack_00000008`	a stack read past the known frame - usually a missed parameter
`CONCAT44(a, b)`	two 32-bit values joined into 64 bits
`SUB84(x, 0)`	the low 4 bytes of an 8-byte value
`code *`	a value the model believes is a function pointer
`halt_baddata`	the bytes here are not valid instructions

`CONCAT` and `SUB` are worth stopping for. They appear where values were joined or split in ways the type model could not follow — which is frequently exactly where a truncation lives.

Fixing a signature rewrites the function

Because the output is derived from a model, correcting one input re-derives all of it. Retype a parameter as a pointer to a structure you have defined and pointer arithmetic collapses into named field accesses across the whole function, in one action.

This is the highest-leverage thing the tool does, and it is why the data module insists on defining structures rather than reading offsets.

Quality differs per processor

The IR is shared; the front ends are not equally mature. **MIPS output shows delay-slot artifacts** — instructions appearing where semantics put them rather than where the bytes are — and **ARM output shows mode-switch artifacts**. Knowing which oddities belong to the processor and which belong to the program is its own skill, and the answer when they disagree is always the listing.

Changing the listing versus changing the database

Renaming, retyping, commenting and defining data all modify the **program database**, not the file. Nothing you do in Ghidra alters the binary.

That is worth internalising early because it makes everything reversible. `Undo` works across analysis passes. Being wrong is cheap, so being decisive is correct.

The keystrokes worth making reflexive

Key	Does
`L`	rename the thing under the cursor
`T`	retype it
`D`	disassemble at the cursor
`F`	create a function here
`;`	set an end-of-line comment
`G`	go to an address or symbol
`Ctrl+Shift+G`	go back to where you were

Everything else can be found in a menu. These six are used often enough that looking for them breaks your reading.

What to take away

1. **The importer's processor and endianness are asserted, not verified** — and wrong ones decode into plausible nonsense.
2. **Know which analyzer produced a result**, especially Aggressive Instruction Finder and Parameter ID.
3. **The decompiler is a model.** `undefined`, `CONCAT`, `unaff_` and `in_stack_` are it telling you where it guessed.
4. **Correcting a signature re-derives the whole function** — the highest-value single action available.
5. **You are editing a database, not the binary.** Undo works; be decisive.

Module R1

Reading x86-64: the subset that matters

You do not need to be able to *write* x86-64 to reverse it. You need to read a few dozen instruction shapes fluently and know what each one tells you about the C that produced it.

This card is about the reading, and about the two things the instruction stream tells you that the decompiler will sometimes get wrong:

> **The operand size tells you how wide a value is. The kind of extension tells you whether it is signed.**

Both are structural evidence. Both are recoverable from a single instruction. And when a decompiler shows you `undefined4` or an `int` that should be a `uint`, these are what you check it against.

Widening: three instructions, three different meanings

Here are three functions from one object file, compiled at `/O2`. Each is two or three bytes long, and the difference between them is the whole lesson.

Zero-extension is explicit

```
unsigned int widen_unsigned(const unsigned char *p) { return p[0]; }
```

```
widen_unsigned:
  movzx   eax, byte ptr [rcx]
  ret
```

`movzx` — **mov with zero extend**. One byte is loaded and the upper bits of `eax` are filled with zeros. The value came from an *unsigned* source, so its high bits carry no meaning and zero is the right filler.

Sign-extension is a different instruction

```
int widen_signed(const signed char *p) { return p[0]; }
```

```
widen_signed:
  movsx   eax, byte ptr [rcx]
  ret
```

`movsx` — **mov with sign extend**. Identical source shape, one letter different in the mnemonic, and the meaning inverts: the top bit of the byte is copied through the upper bits. A byte of `0xFF` becomes `0x000000FF` under `movzx` and `0xFFFFFFFF` under `movsx`.

This single letter is the most consequential thing on this page. It is the difference between a length of 255 and a length of −1, and a length of −1 that later gets treated as unsigned becomes 4,294,967,295. That thread runs from here to the bug-class module and never really stops.

Widening to 64 bits, with and without an instruction

```
unsigned long long widen_implicit(unsigned int n) { return n; }
```

```
widen_implicit:
  mov     eax, ecx
  ret
```

Look at what is *not* there. The function returns a 64-bit value, and the only instruction is a **32-bit** move. There is no `movzx`, no `and`, nothing that clears the upper half of `rax`.

It does not need one:

> **On x86-64, any write to a 32-bit register clears the upper 32 bits of the full register.** Writing `eax` zeroes the top half of `rax`, for free, always.

This is why 32-bit operations are everywhere in 64-bit code — they are shorter to encode and the zero-extension comes free. It is also a trap: a 32-bit computation that overflows silently produces a small `rax`, and nothing in the listing marks the truncation. Compare with the explicit signed case:

```
long long widen_sign_to_64(int n) { return n; }
```

```
widen_sign_to_64:
  movsxd  rax, ecx
  ret
```

`movsxd` — sign-extend a dword into a qword. **Sign extension needs an instruction; zero extension can be a side effect.** Whenever you see `movsxd` on a length or an index, stop: the value is signed, and signed values can be negative.

You see	Width in	Width out	Upper bits	The C
`movzx eax, byte ptr [..]`	1	4 (then 8)	zeros	`unsigned char` → `unsigned`
`movsx eax, byte ptr [..]`	1	4 (then 8)	copy of bit 7	`signed char` → `int`
`mov eax, ecx`	4	4 (then 8)	zeros, implicitly	`unsigned` → `unsigned long long`
`movsxd rax, ecx`	4	8	copy of bit 31	`int` → `long long`

Addressing: one grammar, used everywhere

Almost every memory reference on x86-64 fits a single template:

```
[ base + index*scale + displacement ]
```

`scale` is 1, 2, 4 or 8. Any part may be absent. Two real examples:

```
unsigned int   index_scaled  (const unsigned int   *base, size_t i) { return base[i +
3]; }
unsigned short index_halfword(const unsigned short *base, size_t i) { return
base[i];     }
```

```
index_scaled:
  mov     eax, dword ptr [rcx+rdx*4+12]
  ret

index_halfword:
```

```
  movzx   eax, word ptr [rcx+rdx*2]
  ret
```

The scale is the element size, so it tells you the type of the array without any declaration: scale 4 with a `dword` access is a 32-bit element; scale 2 with a `word` access is a 16-bit one. **The displacement is the offset into the object**, so `+12` here is `i + 3` scaled by four — the constant part of the index folded into the addressing mode.

That folding is why array indexing usually costs no extra instructions, and why a displacement is often a *field offset* rather than an array index. Which one it is depends on whether an index register is present.

`lea` computes an address without touching memory

```
unsigned long long arith_lea(unsigned long long a, unsigned long long b)
{
    return a * 4 + b + 9;
}
```

```
arith_lea:
  lea     rax, [rcx*4+9]
  add     rax, rdx
  ret
```

`lea` evaluates the bracket expression and stores the *result* rather than the memory it points at. The compiler uses it constantly as a multiply-and-add: one instruction here computes `a*4 + 9`.

Do not read `lea` as a memory access. It is arithmetic that happens to be spelled with brackets, and reading it as a load will send you looking for a pointer that does not exist.

Globals are reached relative to the instruction pointer

```
reach_global:
  mov     rax, qword ptr [rip + g_counter]
  add     rax, rcx
  mov     rcx, rax
  mov     qword ptr [rip + g_counter], rax
  jmp     sink
```

A global is addressed as an offset from the *current instruction*, not by an absolute address. That keeps the code position-independent, and it means the address you see in the listing was computed by the disassembler from the instruction's own location. It is also what makes globals easy to find by cross-reference: every access names the same target.

Note the last line: a `jmp` to another function rather than a `call`. That is a tail call — the frame is reused and the call-graph edge is easy to miss.

Comparison: where signedness becomes visible

`cmp` subtracts and throws the result away, keeping only the flags. `test` ANDs and does the same. The conditional jump that follows is what interprets them — and **the jump mnemonic tells you whether the operands were signed.**

Here are two functions whose C differs by exactly one word:

```c
int guard_unsigned(const unsigned char *buf, unsigned int len, unsigned int cap)
{
    if (len > cap) return -1;
    ...
}

int guard_signed(const unsigned char *buf, int len, int cap)
{
    if (len > cap) return -1;
    ...
}
```

```
guard_unsigned:                          guard_signed:
  sub     rsp, 0x28                        sub     rsp, 0x28
  cmp     edx, r8d                         cmp     edx, r8d
  jbe     .Lok                             jle     .Lok
  mov     eax, -1                          mov     eax, -1
  add     rsp, 0x28                        add     rsp, 0x28
  ret                                      ret
.Lok:                                    .Lok:
  movzx   ecx, byte ptr [rcx]              movzx   ecx, byte ptr [rcx]
  mov     eax, edx                         movsxd  rax, edx
  add     rcx, rax                         add     rcx, rax
  call    sink                             call    sink
  ...                                      ...
```

The listings are identical except in two places, and both places say the same thing.

`jbe` versus `jle`. `jbe` is *jump if below or equal* — an **unsigned** comparison. `jle` is *jump if less or equal* — a **signed** one. Same `cmp`, same operands, different interpretation of the flags.

`mov eax, edx` versus `movsxd rax, edx`. The unsigned version widens for free; the signed version needs an instruction, because the sign bit has to be propagated.

> **The branch mnemonic is the type declaration.** `jb` / `jbe` / `ja` / `jae` are unsigned; `jl` / `jle` / `jg` / `jge` are signed. When you are reconstructing a function's signature, the branch after a bound check tells you how the length was declared, and it is often the *only* thing that does.

And it is a bug-class tell. A signed length compared with `jle` passes the check when it is negative, and is then used as a size by code that treats it as unsigned. That is the shape of a very large number of real vulnerabilities.

`test reg, reg` is the null check

```c
int guard_null(const unsigned char *p) { if (!p) return -1; return p[0]; }
```

```
guard_null:
  mov     eax, -1
  test    rcx, rcx
  je      .Lout
  movzx   eax, byte ptr [rcx]
.Lout:
  ret
```

`test rcx, rcx` ANDs a register with itself, which changes nothing but sets the zero flag if the value is zero. It is the idiomatic *is this null / is this zero* test, and it is shorter than comparing against an immediate.

Note also that the compiler set the `-1` **before** the branch, so the error value is in place whether or not the branch is taken. Instructions appearing before the test they belong to is normal, and reading strictly top to bottom will mislead you.

What to take away

1. **Operand size is the type's width.** `byte`, `word`, `dword`, `qword`.
2. **Extension is the type's signedness.** `movzx` unsigned, `movsx` / `movsxd` signed, and a plain 32-bit `mov` zero-extends to 64 for free.
3. **Scale is the element size; displacement is the offset.**
4. **`lea` is arithmetic, not a load.**
5. **The branch mnemonic after a `cmp` says whether the comparison was signed** — and that is the single most useful one-instruction tell in the course.

The next module takes these and adds the calling convention, so that a function's parameters — not just its instructions — can be recovered.

Module R1M

The ABI: whose register is that?

You can read every instruction in a function and still not know what it takes. Arguments arrive in registers, and *which* registers is decided by the calling convention — a platform choice, not a machine one. Get it wrong and you will confidently mislabel every parameter.

This is also the module that decides whether you can trust a decompiled signature. Ghidra guesses parameters from how registers are used before they are written, and the guess is often right and sometimes badly wrong. Knowing the convention is how you tell the difference.

Two conventions, one machine

	Microsoft x64 (Windows)	**System V AMD64** (Linux, macOS, BSD)
Integer args	`rcx`, `rdx`, `r8`, `r9`	`rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`
Float args	`xmm0`–`xmm3`	`xmm0`–`xmm7`
Registers before spilling	4	6
Return	`rax` / `xmm0`	`rax` / `xmm0`
Caller reserves	32 bytes of **shadow space**	nothing (but a 128-byte **red zone** below rsp)
`this` pointer	`rcx`	`rdi`

Here is the same two-argument function compiled both ways. Nothing about the source changed:

```
; Microsoft x64                       ; System V AMD64
copy_c32:                             copy_c32:
  movups  xmm0, [rdx]                   movdqu  xmm0, [rsi]
  movups  [rcx], xmm0                   movups  [rdi], xmm0
```

Destination in `rcx` versus `rdi`; source in `rdx` versus `rsi`. **Read that listing under the wrong convention and you have swapped the source and the destination of a copy** — which inverts the direction of data flow in your notes and, in a bug hunt, points you at the wrong buffer.

The first question about any binary is which platform it targets. Everything below depends on the answer.

Shadow space, and where the fifth argument lives

Microsoft x64 requires the **caller** to reserve 32 bytes on the stack before the call, even when every argument fits in a register. Those four slots belong to the callee: it may spill `rcx`, `rdx`, `r8` and `r9` into them, and a debugger can find the arguments there.

The consequence is arithmetic you will do constantly. Take a function with six integer parameters:

```
six_ints:
  lea     rax, [rcx+rdx]              ; args 1 and 2
  add     rax, r8                     ; arg 3
  add     rax, r9                     ; arg 4
  add     rax, qword ptr [rsp+0x28]   ; arg 5
  add     rax, qword ptr [rsp+0x30]   ; arg 6
  ret
```

This is a leaf — no `push`, no `sub rsp` — so `rsp` still points at the return address. Counting up from there:

```
[rsp+0x00]   return address
[rsp+0x08]   shadow slot for arg 1   \
[rsp+0x10]   shadow slot for arg 2    |  32 bytes, reserved by the caller,
[rsp+0x18]   shadow slot for arg 3    |  whether or not anything is put here
[rsp+0x20]   shadow slot for arg 4   /
[rsp+0x28]   argument 5              <- the first stacked argument
[rsp+0x30]   argument 6
```

8 for the return address, plus 32 of shadow, puts the fifth argument at `[rsp+0x28]`. Whenever you see a read from around `rsp+0x28` in a function that has not moved `rsp`, suspect a fifth argument rather than a local.

And the count itself is information: **a read from the stacked-argument area proves the function takes more arguments than the register set holds.** Under Microsoft x64 that means more than four. Under System V the same evidence means more than six, because the boundary sits in a different place.

Floats: the two conventions genuinely disagree

This is the part that catches people, because the two rules produce different answers for the same source.

```
double mixed_args(unsigned int a, double b, unsigned int c, double d);
```

```
mixed_args:                         ; Microsoft x64
  xorps    xmm0, xmm0
  add      ecx, r8d                 ; a is in ecx (slot 1), c is in r8d (slot 3)
  cvtsi2sd xmm0, rcx
  addsd    xmm0, xmm1               ; b is in xmm1 (slot 2)
  addsd    xmm0, xmm3               ; d is in xmm3 (slot 4)
  ret
```

Under **Microsoft x64 the slot is positional**. Argument *n* uses either the *n*th integer register or the *n*th XMM register, and **the other one is skipped**. So `b` is argument 2 and lands in `xmm1`; `d` is argument 4 and lands in `xmm3`. `rdx` and `r9` are never touched — they were consumed by position, not by use.

Under **System V the two classes are counted independently**: the integers take `rdi` then `rsi`, and the doubles take `xmm0` then `xmm1`. Same source, and `d` would arrive in `xmm1` rather than `xmm3`.

> **A gap in the register sequence is not a missing argument — it may be a float that consumed the slot.** If you see `rcx` and `r8` used but never `rdx`, look for `xmm1` before concluding the signature has three parameters.

Reading a frame

```
with_locals:
  push    rbx                       ; save a non-volatile register
  sub     rsp, 0x40                 ; allocate 64 bytes
  lea     rax, [rcx+1]
  mov     qword ptr [rsp+0x20], rcx  ; scratch[0]
  mov     qword ptr [rsp+0x28], rax  ; scratch[1]
  mov     rbx, rcx                  ; keep n across the call
  ...
  call    sink
  lea     rcx, [rbx+rbx*2]
  mov     rdx, rbx
  add     rsp, 0x40
  pop     rbx
  jmp     combine                   ; tail call
```

Four things to read off this.

The frame size is the `sub`. 64 bytes. Its lower 32 bytes are the shadow space this function must reserve for the calls *it* makes; the locals live above that, which is why `scratch[0]` is at `+0x20` and not at `+0`.

A pushed non-volatile register means a value must survive a call. `rbx` is callee-saved, so the compiler chose it for `n`, saved the caller's copy on entry and restored it on exit. Volatile registers — `rax`, `rcx`, `rdx`, `r8`–`r11` — are destroyed by any call, so anything live across one must be in a non-volatile register or on the stack.

Argument offsets shift as the frame grows. With one push and a 64-byte allocation, the incoming argument's home slot is at `[rsp+0x50]`: 64 + 8 + 8. Compute this from the prologue rather than assuming, because every push moves it.

The exit is a `jmp`, not a `call`. The frame is torn down first and control transfers directly to `combine`, whose `ret` will return to *this* function's caller. A tail call, and a call-graph edge that is easy to miss.

Compare a genuine leaf:

```
leaf_noframe:
  mov     rax, rcx
  and     rcx, rdx
  xor     rax, rdx
  add     rax, rcx
  ret
```

No push, no `sub`, no prologue at all. Nothing needed saving and nothing needed storing. **Absence of a prologue is itself information:** this function makes no calls and has no spilled locals.

`.pdata`: the function table you are given for free

On 64-bit PE, exception handling requires the compiler to emit unwind data for almost every non-leaf function, in a `.pdata` section of `RUNTIME_FUNCTION` records:

```
DWORD  BeginAddress
DWORD  EndAddress
DWORD  UnwindInfoAddress   -> prologue length, frame register, saved registers
```

This is the most under-used artifact in Windows reverse engineering. It hands you **function boundaries** in a stripped binary, and the unwind info tells you the frame size and which registers were saved without reading the prologue at all. When auto-analysis misses a function, `.pdata` frequently already knows about it.

ELF has an equivalent in `.eh_frame`, which serves the same purpose and is present even in binaries built without exceptions.

When the decompiler's signature is wrong

Ghidra infers parameters from which registers are read before they are written. That heuristic fails predictably:

- **Varargs.** `printf`-style functions read a variable number of registers, so the guess is whatever this particular body happened to touch.
- **Hand-written or non-standard functions.** Anything not following the convention is invisible to a convention-based guess.
- **A float in the sequence.** Under Microsoft x64 a skipped integer register looks like a missing parameter, so a four-argument function can be reported as taking two.
- **Wrapper functions.** A function that forwards its arguments may never read some of them at all, so they vanish from the signature.

The check is cheap: count the argument registers the body reads, in order, and compare with the signature the decompiler shows. A read from the stacked argument area with no corresponding parameter is the loudest version of this — the function is using something the signature does not admit exists.

Fixing the signature is worth doing as soon as you spot it. The decompiler re-derives the whole function from it, so a corrected parameter list often turns several lines of `undefined` arithmetic into ordinary field accesses.

What to take away

1. **Establish the platform first.** `rcx`-first or `rdi`-first changes every parameter you name.
2. **32 bytes of shadow plus 8 for the return address puts the fifth Microsoft x64 argument at `[rsp+0x28]`** in a function that has not moved `rsp`.
3. **A skipped integer register may be a float that took the slot** — under Microsoft x64, position is what is consumed.

4. **The prologue tells you the frame size, what must survive a call, and whether this is a leaf.**
5. **A decompiled signature is a guess.** Count the registers the body actually reads and check it.

Module R1R

RISC for reading: AArch64, ARM, MIPS

You already know how to read x86-64. This card is about what changes — and how little of it is conceptual.

Everything from the x86 module still applies: operand size is width, extension is signedness, the comparison's interpretation lives in the branch. What changes is the spelling, plus **two genuine surprises** that will silently break a reading if you do not know them: the Thumb bit and the MIPS branch delay slot.

What a load/store machine changes

Computation happens only in registers. Nothing does arithmetic directly on memory. Every value that participates must first be loaded and afterwards stored, explicitly.

That sounds like a restriction. For a reader it is a gift: **every memory access is a separate, visible instruction.** On x86-64 an `add` can quietly touch memory; on a RISC machine it cannot. Data flow is easier to follow, not harder.

Instructions are fixed width — four bytes on AArch64, MIPS, and ARM in ARM mode. Instruction count is byte count divided by four, and a function's size tells you how much it does.

Endianness is a choice you make, not a fact you read. MIPS ships big-endian and little-endian; ARM is usually little but not always. Pick wrong in the importer and the bytes decode into plausible nonsense rather than an error, which is far worse than a crash. This is the first irreversible decision in any RISC analysis.

The same four functions, three machines

Every listing below is real compiler output for one C source file, built for x86-64, AArch64 and MIPS.

Widening a byte

```c
unsigned int widen_unsigned(const unsigned char *p) { return p[0]; }
int          widen_signed  (const signed char   *p) { return p[0]; }
```

```
x86-64 (MSVC)              AArch64 (GCC)          MIPS (GCC)
  movzx eax, byte [rcx]      ldrb  w0, [x0]         lbu  $2, 0($4)
  movsx eax, byte [rcx]      ldrsb w0, [x0]         lb   $2, 0($4)
```

The same distinction, spelled three ways. x86 uses two widening instructions; AArch64 and MIPS fold the widening into the load itself — `ldrb` / `ldrsb`, `lbu` / `lb`. The `s` and the `u` are the whole difference, and they are exactly as consequential as `movzx` versus `movsx`.

Learn the pattern rather than the mnemonics: **any machine that can load a narrow value into a wide register must say how to fill the rest, and that choice is the signedness of the source type.**

Widening 32 to 64

```
unsigned long long widen_implicit  (unsigned int n) { return n; }
long long          widen_sign_to_64(int n)          { return n; }
```

```
x86-64 (MSVC)      AArch64 (GCC)
  mov   eax, ecx     uxtw  x0, w0
  movsxd rax, ecx    sxtw  x0, w0
```

`uxtw` and `sxtw` are *unsigned* and *signed extend word* — the direct twins of the x86 pair. Note that AArch64 spends an instruction on the unsigned case where MSVC did not: writing `w0` does zero-extend into `x0`, but the ABI does not promise that the caller left the upper half clean, so the compiler makes it explicit.

Where MIPS diverges completely. This build targets the 32-bit O32 ABI, so a 64-bit value does not fit in a register at all — it occupies a *pair*:

```
widen_sign_to_64:
  sra   $2, $4, 31        ; high word = the sign bit, replicated 32 times
  jr    $31
  move  $3, $4            ; low word = the original value
```

`sra` shifts right *arithmetically*, dragging the sign bit down, so `$2` becomes all-ones for a negative input and all-zeros for a positive one. A 64-bit quantity split across two registers is normal on 32-bit targets, and a decompiler that has not been told the return type will show you two unrelated values.

Indexing an array

```
unsigned int index_scaled(const unsigned int *base, size_t i) { return base[i + 3]; }
```

```
x86-64                                AArch64                           MIPS
  mov eax, [rcx+rdx*4+12]               add x1, x1, 3                     sll  $7, $7,
2
                                        ldr w0, [x0, x1, lsl 2]           addu $4, $4,
$7
                                                                          jr   $31
                                                                          lw   $2,

12($4)
```

One instruction, two, then four — and the reason is how much addressing each machine folds in.

x86-64 encodes base, scaled index *and* displacement in a single operand. AArch64 has a shifted-register form, `[x0, x1, lsl 2]`, where `lsl 2` is the multiply-by-four — but no displacement alongside it, so the `+3` becomes its own `add`. MIPS has **only** `displacement(base)`: no index register and no scale, so the multiply becomes an explicit shift and the index becomes an explicit add.

The `lsl 2` and the `sll ..., 2` are the scale, and the scale is still the element size. Same reading, more instructions.

Surprise one: the MIPS branch delay slot

Look again at that MIPS listing:

```
  jr    $31          ; return
  lw    $2, 12($4)   ; ...load the return value?
```

Read top to bottom and the function returns before computing what it returns.

It does not. **On MIPS the instruction after a branch or jump always executes**, before the branch takes effect. That position is the *delay slot*, and the compiler fills it with useful work whenever it can.

This is not a curiosity. Consider:

```
widen_unsigned:
  jr    $31
  lbu   $2, 0($4)
```

A two-instruction function where the *second* line produces the result of the *first*. Any reading that treats source order as execution order gets this exactly backwards.

When there is nothing useful to put there, the slot is filled with a `nop`, and you can see the cost:

```
guard_null:
  beq   $4, $0, .L25
  nop                  ; wasted slot - nothing safe to hoist here
  jr    $31
  lbu   $2, 0($4)
```

> **The rule: on MIPS, always read one instruction past a branch before deciding what the branch does.** A guard can look present in reading order and be bypassed in execution order, and that is a real source of bugs, not just of misreadings.

Ghidra models this correctly, but its decompiled output will sometimes place the delay-slot instruction where the *semantics* put it rather than where the bytes are — so the listing and the decompilation legitimately disagree about line order. Both are right.

Surprise two: the Thumb bit

ARM (32-bit) has two instruction encodings — ARM and Thumb — and switches between them at run time. **The low bit of a function address encodes which one to use.** An address ending in 1 does not mean "one byte past the function"; it means "this function is Thumb, and the real address is the address with that bit cleared."

Ghidra models the mode as a context register, `TMode`. Set it wrong and the bytes decode into valid-looking instructions that make no structural sense — no sensible prologue, branches into the middle of nothing. If a function looks like garbage but the bytes are clearly code, check the mode before anything else.

Comparison and branching

Neither surprise affects the thing you actually came for: **the comparison still tells you the signedness.**

```
int guard_signed(const unsigned char *buf, int len, int cap)
{
    if (len > cap) return -1;
    ...
}
```

```
x86-64                AArch64                   MIPS
  cmp edx, r8d          cmp w1, w2                slt  $6, $6, $5
  jle .Lok              bgt .L18                  bne  $6, $0, .L18
```

x86 and AArch64 both set flags and branch on them; only the mnemonic family differs, and `bgt` is as signed as `jg`.

MIPS has no flags register at all. It cannot branch on a comparison, so it *materialises* the comparison into a register: `slt` sets its destination to 1 if the second operand is less than the third, and 0 otherwise. Then `bne` branches on that register being non-zero.

And the signedness lives in the `slt`:

	zero-extend byte	sign-extend byte	signed compare	unsigned compare
x86-64	`movzx`	`movsx`	`jl` / `jle`	`jb` / `jbe`
AArch64	`ldrb`	`ldrsb`	`blt` / `bgt`	`blo` / `bhi`
MIPS	`lbu`	`lb`	`slt`	`sltu`

> **`slt` versus `sltu` is the MIPS signedness tell**, and it is exactly as load-bearing as `jle` versus `jbe`. One letter, and a negative length either passes the bound check or does not.

The corroborating evidence transfers too. The x86 build sign-extended the length at its use site with `movsxd`; AArch64 wrote `add x0, x0, w1, sxtw` — a sign-extending register operand folded into the add; MIPS emitted `sra $4, $5, 31` to build the sign-extension high word. Three spellings, one fact: **that length is signed.**

Registers, briefly

Full argument conventions are the next module. For reading, the essentials:

- **AArch64:** `x0`–`x30` with 32-bit views `w0`–`w30`. **`x30` is the link register** — the return address lives in a register, not on the stack. `x29` is the frame pointer, `sp` is separate.
- **MIPS:** `$0` is hardwired to zero, which is why `beq $4, $0` is the null test and `move $2, $0` is how you return zero. **`$31` (`$ra`) is the return address**, which is why `jr $31` is a return. `$4`–`$7` are the first arguments; `$2`–`$3` are returns.
- **ARM32:** `r0`–`r15`, where `r15` *is* the program counter — writing to it is a branch. `r14` is the link register.

The lever, across all three: x86 keeps the return address on the stack; ARM and MIPS keep it in a register (`x30`/`lr`, `$31`/`$ra`) and spill it only when the function makes calls of its own. A leaf that never spills it is identifiable at a glance — and where the return address lives is the first thing to establish about any frame.

What to take away

1. **Every concept from the x86 module transfers.** Width, signedness, scale, the branch carrying the interpretation.
2. **Widening is folded into the load** on AArch64 and MIPS — `ldrb`/`ldrsb`, `lbu`/`lb`.
3. **MIPS materialises comparisons** into a register with `slt`/`sltu`, because it has no flags.
4. **Read one instruction past every MIPS branch.** The delay slot always runs.
5. **Check the ARM mode before believing a listing** that looks like noise.
6. **The return address is in a register**, which the next module builds on.

Module R1S

RISC ABIs: where the return address lives

One fact organises this whole module:

x86-64 puts the return address on the stack. AArch64 and MIPS put it in a register.

`call` pushes; `bl` and `jal` write a register. Everything else — why leaf functions have no prologue, what a prologue is actually saving, how you spot a function that calls something — follows from that one difference.

The argument registers

	Integer args	Float args	Return	Return address	`this`
Microsoft x64	`rcx` `rdx` `r8` `r9`	`xmm0` – `xmm3`	`rax`	on the stack	`rcx`
System V AMD64	`rdi` `rsi` `rdx` `rcx` `r8` `r9`	`xmm0` – `xmm7`	`rax`	on the stack	`rdi`
AAPCS64	`x0` – `x7`	`v0` – `v7`	`x0`	**`x30` (lr)**	`x0`
MIPS O32	`$4` – `$7`	`$f12`, `$f14`	`$2`, `$3`	**`$31` (ra)**	`$4`
MIPS N64	`$4` – `$11`	`$f12` – `$f19`	`$2`	**`$31`**	`$4`

AAPCS64 gives you eight integer argument registers — more than either x86 convention — so genuinely stacked arguments are rare. When you do see a read from above the frame on AArch64, it is a real ninth argument and worth noticing.

MIPS O32 has only four, and it reserves a **16-byte argument save area** at the bottom of every frame for them. The register/stack boundary therefore sits in a different place on each of the four conventions, which is why "the fifth argument" is not a portable idea.

The AArch64 prologue, and how to read a frame off it

```
guard_signed:
  cmp   w1, w2
  bgt   .L18                    ; early-out path: no frame needed
  stp   x29, x30, [sp, -16]!    ; <- the canonical prologue
  mov   x29, sp
  ldrb  w0, [x0]
  add   x0, x0, w1, sxtw
  bl    sink
  mov   w0, 0
  ldp   x29, x30, [sp], 16      ; <- the canonical epilogue
  ret
```

`stp x29, x30, [sp, -16]!` is the shape to recognise on sight. It does three things at once:

- **stores a pair** — the frame pointer `x29` and the link register `x30`;
- **at `sp - 16`**, i.e. below the current stack pointer;
- **with `!`**, pre-index writeback, which also *sets* `sp` to that address.

So **the immediate is the frame size**, and it is negative because the stack grows down. Here: 16 bytes. `x29` lands at `[sp+0]` and `x30` at `[sp+8]`.

The epilogue mirrors it. `ldp x29, x30, [sp], 16` is *post*-index: load the pair from `[sp]`, then add 16 to `sp`. The brackets close before the immediate, which is how you tell post-index from pre-index at a glance.

```
  [sp, -16]!    pre-index    adjust sp, THEN access     -> a prologue
  [sp], 16      post-index   access, THEN adjust sp     -> an epilogue
```

Why `x30` is saved at all: this function calls `sink`, and `bl` overwrites `x30` with its own return address. The incoming one would be destroyed, so it has to be spilled. That is the entire reason the frame exists.

A bigger frame, from another function in the same pack:

```
copy_var:
  stp   x29, x30, [sp, -32]!    ; frame is 32
  uxtw  x2, w2
  mov   x29, sp
  stp   x19, x20, [sp, 16]      ; save two callee-saved registers
  ...
  ldp   x19, x20, [sp, 16]
  ldp   x29, x30, [sp], 32
  b     sink                    ; tail call
```

`x19`–`x28` are callee-saved, so a value living across a call goes in one and is spilled on entry. **Count the saved registers and you know how many values had to survive calls.** And note the exit: `b`, not `bl`. A tail call — the frame is gone before control transfers, and `sink`'s `ret` will return to *this* function's caller.

Now a leaf:

```
guard_null:
  cbz   x0, .L25
  ldrb  w0, [x0]
  ret
.L25:
  mov   w0, -1
  ret
```

No prologue whatsoever. It calls nothing, so `x30` is never clobbered and never needs saving; it has no locals, so nothing needs allocating. **On a RISC machine, a leaf can be genuinely prologue-free in a way an x86 function usually cannot** — because the return address was already in a register rather than on the stack.

> **The read: if the link register is spilled, this function calls something. If it is not, this is a leaf.** One glance at the first instruction.

The MIPS frame, and two things that look strange

```
guard_signed:
  slt   $6, $6, $5
  bne   $6, $0, .L18
  lw    $25, %call16(sink)($28)   ; delay slot
  addiu $sp, $sp, -32             ; frame is 32 bytes
  sw    $31, 28($sp)              ; save the return address
  lbu   $2, 0($4)
  sra   $4, $5, 31
  ...
  jalr  $25                       ; call through $25
  addu  $4, $2, $4                ; delay slot
  move  $2, $0
  lw    $31, 28($sp)              ; restore
  jr    $31
  addiu $sp, $sp, 32              ; delay slot: tear the frame down
```

The frame is the `addiu` immediate, negated: 32 bytes. Its bottom 16 bytes are the O32 argument save area, reserved whether or not anything is written there — the direct analogue of Microsoft x64's shadow space, and the reason a MIPS O32 frame is bigger than its locals suggest. `$31` is spilled at `+28` because this function calls `sink`; a leaf would not spill it, exactly as on AArch64.

Note the teardown sitting in the delay slot of `jr $31`. The stack pointer is restored *after* the return jump is issued and before it takes effect. Normal, and it means the last instruction of a MIPS function is frequently the frame teardown rather than the return.

`$28` and `$25`: what position-independent code looks like

```
  lw    $25, %call16(sink)($28)
  ...
  jalr  $25
```

Two register conventions carry this, and neither is guessable:

- **`$28` is `$gp`, the global pointer.** In PIC, globals and imported functions are reached through a table indexed off `$gp` rather than by absolute address. `%call16(sink)($28)` is "load the address of `sink` from the GOT". If you see `$28` used as a base, you are looking at PIC.
- **`$25` is `$t9`, and the ABI requires it to hold the callee's address on entry** to any PIC function. The callee uses `$t9` to compute its own `$gp`. That is why an indirect `jalr $25` is the *normal* shape of a MIPS call to an external function — it is not an obfuscated or virtual call.

This trips people constantly. On x86 an indirect call through a register suggests a function pointer, a vtable, or something worth investigating. On MIPS PIC it is how ordinary calls work. Reading every `jalr $25` as a mystery target will waste a lot of time; the symbol is right there in the `%call16` relocation.

The lever, across all four conventions

	Return address	Spilled when	Frame size read from
x86-64	on the stack, pushed by `call`	always there	`sub rsp, N`
AAPCS64	`x30`	the function calls something	the `stp` immediate
MIPS O32/N64	`$31`	the function calls something	the `addiu` immediate

And the two reserved regions that exist for the same reason and are easy to confuse:

- **Microsoft x64 shadow space** — 32 bytes, reserved *by the caller*, above the return address.
- **MIPS O32 argument save area** — 16 bytes, at the *bottom* of the callee's own frame.
- **System V and AAPCS64 reserve neither.** System V instead has a 128-byte *red zone* below `sp` that a leaf may use without allocating anything — which is why a System V leaf can write to `[rsp-8]` with no prologue and be perfectly correct.

What Ghidra gets right and wrong per processor

The parameter heuristic is the same everywhere — which registers are read before they are written — but its failure modes differ:

- **AArch64:** generally good. Eight argument registers mean fewer stacked arguments to miss. Watch for `w` versus `x` reads: a parameter used only through its 32-bit view may be typed too narrow.
- **MIPS:** watch the delay slot. An argument set up in a delay slot is positioned after the call in source order, and a reader (human or heuristic) scanning for argument setup *before* the call can miss it entirely.
- **MIPS PIC:** `$25` and `$28` are ABI plumbing, not parameters. They should never appear in a recovered signature; if they do, the signature is wrong.

The check is the same as on x86: count the argument registers the body reads, in order, and compare against the signature shown.

What to take away

1. **The return address is in a register** on both RISC targets — `x30`, `$31`.
2. **`stp x29, x30, [sp, -N]!` is the AArch64 prologue and `N` is the frame size.** Pre-index writes back, post-index does not: `[sp,-16]!` opens a frame, `[sp],16` closes one.
3. **A spilled link register means this function calls something.** No spill means leaf.
4. **MIPS O32 reserves 16 bytes of argument save area** at the bottom of the frame, the way Microsoft x64 reserves 32 above the return address.
5. **`jalr $25` after a `%call16(...)($28)` load is an ordinary PIC call**, not an indirect jump worth chasing.

Module R2

PE and ELF as maps

A file format is not trivia to memorise. It is the map that answers three questions you will ask about every binary:

- **Where does this address live in the file?**
- **What can this program do?**
- **What does it offer to others?**

Everything below is read off one real DLL, built for this module. The header dumps are in `notes/`.

The section table, and why one number is not enough

Sections are how a file's bytes get mapped into memory, and they are mapped at different alignments than they are stored. Here is a real one:

Section	Virtual address	Raw offset	difference
`.text`	0x1000	0x400	0xC00
`.rdata`	0xE000	0xD400	0xC00
`.data`	0x18000	0x16E00	**0x1200**
`.pdata`	0x1A000	0x17A00	**0x2600**
`.reloc`	0x1C000	0x18C00	**0x3400**

The difference is not constant. Sections are aligned to 0x1000 in memory and 0x200 in the file, so every section slips by a different amount. This is the single most common mistake in header work: computing one delta from `.text` and applying it everywhere.

RVA to file offset

An **RVA** is an offset from the image base — what a disassembler shows you minus `ImageBase`. To find the bytes on disk:

1. Find the section whose virtual range contains the RVA.
2. `file_offset = RVA - section.VirtualAddress + section.PointerToRawData`

For RVA `0x18500`: it falls in `.data` (0x18000 to 0x19C6F), so `0x18500 - 0x18000 + 0x16E00` = **0x17300**.

Two cases that bite: an RVA in a section whose virtual size exceeds its raw size is **not in the file at all** — that is `.bss`-style uninitialised data, which exists only once loaded. And an RVA below the first section is in the headers themselves.

The optional header, read as a capability statement

From the same file:

```
magic                20B          PE32+  (64-bit)
image base           180000000    the DLL default
entry point RVA      13FC
section alignment    1000
file alignment       200
subsystem            2            Windows GUI
DLL characteristics  160          High Entropy VA | Dynamic Base | NX
```

Each line is a fact you would otherwise have guessed:

- **`magic`** distinguishes PE32 from PE32+, which tells you the pointer size before you decode a single instruction.
- **`image base`** is what a disassembler adds to every RVA. `0x140000000` is the default for an EXE, `0x180000000` for a DLL — recognising them saves you looking it up.
- **`subsystem`** is where kernel drivers announce themselves: subsystem 1 is *native*, and a native PE importing `ntoskrnl.exe` is a driver.
- **`DLL characteristics`** is the mitigation summary. Dynamic Base means ASLR, NX means a non-executable stack, and their **absence** is as informative as their presence.

Imports are a capability list

```
KERNEL32.dll
    CreateFileA
    OutputDebugStringA
    CloseHandle
    ...
```

Before reading any code, this file touches the filesystem and writes debug output. That is a capability list, and it is the fastest characterisation of an unknown binary available.

Mechanically: the import directory names each DLL and points at two parallel arrays — the **Import Name Table**, which survives in the file, and the **Import Address Table**, which the loader overwrites with real addresses. Calls go through the IAT, which is why they appear as `call qword ptr [rip + something]` rather than as direct calls, and why Ghidra labels the target `__imp_CreateFileA`.

Two variations worth recognising. An import **by ordinal** has a number instead of a name — common in older or deliberately obscured binaries, and it means you need the exporting DLL to know what was called. And **delay-loaded** imports are resolved on first use rather than at load, through a different directory.

Exports are the offered surface

```
ordinal  RVA       name
   1     00001020  session_banner
   2     00001030  session_close
   3     00001050  session_open
   4     000010B0  session_write
```

For a DLL this is the attack surface: everything another program can call directly. Start here, not at the entry point.

A **forwarder** is an export whose "address" is actually a string naming another DLL's function — the loader chases it for you, and it looks like an export that points outside the image.

The sections you get for free

`.pdata` is the one most people ignore. On 64-bit PE it holds a `RUNTIME_FUNCTION` per non-leaf function — begin address, end address, and unwind information giving the prologue length and saved registers. **In a stripped binary this is a function table you were handed**, frequently including functions auto-analysis missed.

`.reloc` lists every address that needs fixing if the image loads somewhere other than its preferred base. As a side effect it marks every location holding an absolute address, which is a map of the pointers in the file.

The debug directory carries the PDB path and a GUID. Even with no symbols present, the recorded path often leaks build machine layout and project names, and the GUID is what a symbol server matches on.

The Rich header is an undocumented MSVC block recording the toolchain build numbers. It fingerprints the compiler with more precision than anything else in the file, which makes it useful for grouping related binaries.

ELF, briefly

The same jobs, different names — and one structural difference that matters.

ELF has **two** tables over the same bytes: **section headers** describe the file for the linker, and **program headers** describe the segments the loader maps. Only the program headers matter at run time, and **a stripped ELF may have no section headers at all** while remaining perfectly loadable. If a tool shows you nothing, look at the program headers.

- `PT_LOAD` segments are what actually gets mapped.
- `.dynamic` with its `DT_NEEDED` entries is the shared-library list — the ELF equivalent of the import DLL names.
- Calls to imported functions go through a **PLT** stub which reads a **GOT** entry. Same indirection as the IAT, different spelling.
- `.init_array` holds constructors that run before `main`, which is where initialisation you did not expect will be.
- **PIE** is the ELF norm now, so the load base is 0 in the file and everything is offsets — which is why an ELF address in your notes needs the base recorded beside it.

Two special cases worth knowing

ARM64EC on Windows: a PE whose machine field says ARM64 but whose functions follow an x64-shaped calling convention so they can interoperate with emulated x64 code. The machine field alone will mislead you about the ABI.

Kernel-mode PE: native subsystem, imports from `ntoskrnl.exe` and `hal.dll`, an entry point called `DriverEntry`, and `INIT` and `PAGE` sections — `INIT` is discarded after initialisation and `PAGE` may be swapped out. Recognising these tells you the ABI, the threat model and the debugging story all at once.

What to take away

1. **Compute the RVA-to-offset delta per section.** It is not a constant.
2. **The optional header tells you pointer size, load base, subsystem and mitigations** before you read any code.
3. **Imports are capabilities; exports are attack surface.**
4. **`.pdata` is a free function table** in a stripped 64-bit PE.
5. **A stripped ELF may have no section headers** — use the program headers.

Module R3

Naming: making analysis cumulative

Four hours of reading and four hours of progress look identical while you are doing them. The difference shows up the next morning, and it is entirely a matter of what you wrote down.

A name is a compressed hypothesis. Each one you apply makes the next function cheaper to read, because you no longer have to re-derive what `FUN_140001a20` was for. That compounding is the whole return on this work.

A name is a claim

Every name asserts something, and the assertion should be one you can defend by pointing at evidence.

Compare:

```
FUN_1400010d0 -> free_session        ; asserts a PURPOSE
FUN_1400010d0 -> call_on_close_cb    ; describes the MECHANISM
```

for a function whose body is:

```
mov     rax, qword ptr [rcx+24]
test    rax, rax
je      .Lout
rex_jmp rax
```

The second is defensible — you can point at the jump. The first is a guess about intent wearing the clothes of an observation. Nothing here frees anything: no allocator, no write of a null back over the field.

The cost is not the wrong name; it is everything built on top of it. A reader who later sees `free_session` called twice will hunt a double-free that does not exist. And they will not doubt the name, because you wrote it.

> When you want to record an intuition, write it as a comment marked as a hypothesis. Names are for what the evidence supports.

Where names come from

In rough order of strength:

A string the function references. Error messages and format strings are the strongest single source, and the reason the string sweep is the first thing you run.

An import it calls. A function calling `CreateFileW` then `ReadFile` is doing file I/O whatever else it does.

A constant it compares against. Magic numbers, error codes, protocol identifiers — a comparison against `0x50450000` is a PE check.

RTTI, exports, or a PDB. Names you were given rather than derived.

The caller's context. A function called only from a loop that walks a list is doing something per-element.

The shape of the body alone. Weakest, and where guessing creeps in.

Names you are given are still claims

- **A PDB from a symbol server** is trustworthy — the vendor built it.
- **A FunctionID or signature-database match is a probability.** It says this looks like a known library function. Similar code gets matched wrongly, and a wrong library name is worse than none because it stops you reading.
- **A demangled C++ symbol is a fact about the symbol**, not about the code. The function may not do what its name says.

Claimed versus confirmed

Adopt a convention and use it consistently. The specific convention matters less than that an unmarked name means something definite:

```
parse_header          confirmed - the evidence is in the plate comment
q_parse_header        claimed   - a hypothesis, not yet checked
```

An unmarked name is a promise. If everything is unmarked, then nothing is, and in a week you will not remember which names you would defend.

The prefix is also a worklist. Every `q_` is something to confirm or retract, and retracting is normal — a hypothesis that survives contact with more code is worth more than one you never tested.

Comments carry the evidence

Three kinds, three jobs:

The plate comment states the contract *and its evidence*:

```
; Reads a length from the header and copies that many bytes into a caller
; buffer. Returns bytes copied, or -1.
; Evidence: string "hdr too long" at 140012a08; called from the parse loop
; at 140001c40 with the socket buffer.
```

The end-of-line comment is the one-line why, at the point of confusion:

```
cmp     eax, 0x50450000     ; "PE\0\0" - this is a PE header check
```

The pre-comment carries the hypothesis under test:

```
; HYPOTHESIS: this is the retry path. Not confirmed - I have not found the
; caller that sets the flag at +0x30.
```

The evidence line is the part people skip and the part that pays. In a week you will not remember why you named something, and re-deriving it costs more than writing it once did.

The unknown count as a progress metric

Count what is still `FUN_`, `DAT_` and `undefined`. It goes down as you work, it is honest, and it is the only measure of progress that does not flatter you.

Two things it tells you that a feeling does not: whether an hour actually achieved anything, and when to stop — because the count falling slowly means the remaining unknowns are the hard ones, and grinding them in order is worse than picking the ones your current question needs.

When to rename, and when not to

Rename immediately when the evidence is in front of you. The moment you work out that a function parses a header is the cheapest moment to record it.

Do not rename to something you would have to defend with “it felt like it”. Leave the `FUN_` and add a comment. An unnamed function is an honest statement that you have not established what it does; a wrongly named one is a false statement that other people — including you — will rely on.

Revisit names when contradicted. New evidence that conflicts with a name means the name was wrong, and the correction costs less than the confusion.

What to take away

1. **A name is a claim.** Name the mechanism you can point at, not the purpose you suspect.
2. **Mark claimed versus confirmed**, and treat unmarked as a promise.
3. **The plate comment carries the evidence**, because you will not remember it.
4. **Symbols you were given are claims too** — especially signature-database matches.
5. **Count the unknowns.** It is the only honest progress metric available.

Module R4

Cross-references: querying instead of scrolling

A binary you scroll through is a binary you are reading in the order the linker happened to lay it out. Cross-references let you read it in the order the *questions* suggest instead, and that difference is most of what separates an afternoon of progress from an afternoon of reading.

This module is short, and it is the method the bug-hunting module depends on.

The three sweeps that pay

String → cross-reference. The fastest orientation there is. Error messages, format strings, registry paths, protocol names — each one is a label on a function whose purpose you now know without reading a single instruction. Start here on an unfamiliar binary, every time.

Import → cross-reference. The bug-hunting sweep. Take a sink — `memcpy`, `sprintf`, `RtlCopyMemory`, `ExAllocatePoolWithTag`, `system` — and list every call site. You are not looking for a bug yet, you are building the short list of places one could be.

Global → cross-reference. Find who *writes* a global and you have found the parser, the initialiser, or the configuration loader. Find who reads it and you have the consumers. For a field of a structure, the same query answers "what sets this?", which is how a struct recovery gets its semantics rather than just its offsets.

The counts are informative on their own. A function called from two hundred places is a utility and rarely interesting. A function called from exactly one is a step in a procedure, and its caller is the context you need.

Reachability, and why the answer is usually wrong

The question you actually want answered is:

> **Can attacker-controlled input get from an entry point to this sink?**

A cross-reference query walks the call graph and answers it. The answer is frequently wrong, and always wrong in the same direction: **it under-reports.**

The call graph is built from `call` instructions. Anything that transfers control without one is an edge that was never recorded. There are four common ways that happens, and this pack has compiled examples of three:

Tail calls

```
mov     ecx, edx
jmp     handler_a
```

A `jmp` to a function. The frame is already gone and the callee's `ret` returns to *this* function's caller. Perfectly ordinary, and not a call instruction, so not an edge.

In the switch-dispatch listing from the idioms module, **all seven cases end this way** — seven missing edges in one small function. Ask "who calls `handler_a`" and the dispatcher does not appear.

Calls through a function pointer

```
mov     rax, qword ptr [rcx+24]
test    rax, rax
je      .Lout
rex_jmp rax
```

The target is a value loaded from memory, so there is nothing for a cross-reference to point at. Until you type that field and find what gets stored into it, the edge does not exist and the call graph simply stops here.

This is why the data-recovery module treats a function-pointer field as its highest-value find: typing it is what creates the edge.

Virtual dispatch

```
mov     rax, [rcx]
call    qword ptr [rax+0x10]
```

Same problem with an extra level. The target comes from a vtable, so the edge exists only once you have recovered the class and the slot. Every virtual call in an un-analysed C++ binary is a hole in the graph.

Guarded indirect calls

On Windows with Control Flow Guard, indirect calls are routed through `__guard_dispatch_icall_fptr`. The graph then shows a great many functions calling one thunk, and the thunk calling nothing — technically accurate and useless. The real targets are still in the registers.

The one that looks indirect and is not

```
lw      $25, %call16(sink)($28)
jalr    $25
```

MIPS position-independent code. Every ordinary call to an external function looks like this, because the ABI requires the callee's address in `$t9`. **The target is named in the relocation** — there is nothing to chase. Treating each one as a mystery is a standard way to lose an afternoon.

> The rule: **a cross-reference tells you what it recorded, not what happens.** When a sweep reports a sink as unreachable, the next question is which of these five shapes lies on the path you cannot see.

Working a sink backwards

The productive direction is usually from the sink, not from the entry point:

1. **List the sink's call sites.** Import → cross-reference.
2. **For each, look at the size or index argument.** Constant? Not interesting. Loaded from a structure, a parsed field, a parameter? Keep it.
3. **Walk callers upwards** until you reach something attacker-facing — an IOCTL dispatcher, a parser entry, a message handler, an exported function.
4. **At each step, check for the shapes above.** A step that appears to have no callers has usually been reached by one of them.
5. **Record the path**, because you will lose it otherwise: source, path, sink, and the precondition that has to hold.

Source-first — starting from a parser and walking down — is the same work in the other direction, and it is better when the entry points are few and the sinks are many.

Creating an edge by hand

When you *prove* an indirect target — you find the store into the function pointer, you recover the vtable slot — record it in the database as a reference. Two reasons, and the second is the one people skip.

It makes the next query correct rather than making you remember an exception. And it is a note to the version of you that comes back in a week, who will otherwise re-derive the same edge from scratch.

What to take away

1. **Three sweeps**: string for orientation, import for bug hunting, global for ownership.
2. **Cross-reference counts are triage.** Two hundred callers is a utility; one is a step.
3. **The call graph under-reports.** Tail calls, function pointers, virtual dispatch and CFG thunks are all real edges that were never recorded.
4. **A MIPS `jalr $25` after a `%call16` load is an ordinary call**, named in the relocation.
5. **Record edges you prove.** An exception you remember is an exception you will forget.

Module R5

Data: giving the decompiler something to work with

The decompiler's output is only as good as the types you give it. Left alone it shows you `*(int *)(param_1 + 0x18)` — technically correct, unreadable, and identical for a timeout, a length and a function pointer. Define the structure and the same line becomes `session->on_close`.

This is the highest-leverage work in reverse engineering, and it is almost entirely mechanical: **collect the accesses, read the offsets and widths off them, lay them out.**

Every access is a row of evidence

A field access carries three facts, and each comes from a different part of the instruction:

```
movsx   eax, word ptr [rcx+4]
        ^^^^^^^^^^^^  ^^^^^^
        |             |
        |             +-- the displacement is the OFFSET
        +---------------- the operand size is the WIDTH
^^^^^
+-- the extension is the SIGNEDNESS
```

Collect those from every instruction that touches the pointer and you have a layout. Nothing about it requires insight — it requires being systematic.

Width comes from the operand size

`byte ptr` is 1, `word ptr` is 2, `dword ptr` is 4, `qword ptr` is 8. On a RISC target the same information is in the mnemonic: `ldrb` / `ldrh` / `ldr`, `lbu` / `lhu` / `lw`.

Signedness comes from the extension — when there is one

```
movsx   eax, word ptr [rcx+4]      ; +4 is SIGNED
movzx   eax, word ptr [rcx+6]      ; +6 is UNSIGNED
```

Two adjacent fields, the same width, and **the only difference between them is one letter in the mnemonic.** A recovered declaration that types both the same way has thrown away information the binary preserved.

But the evidence is not always there:

```
mov     eax, dword ptr [rcx+12]    ; signedness: unknown
```

A four-byte field loaded into a four-byte register needs no extension, so none is emitted, so nothing records whether it was signed. **The honest entry is "unknown"**, and knowing which fields you genuinely cannot type is part of doing this well.

Arrays announce themselves two different ways

An ordinary field sits at a fixed offset. An array needs an *index*, and the index has to be scaled by the element size — which hands you that size for free.

Indexed:

```
  mov     eax, dword ptr [rcx+rax*4+32]
                           ^     ^^
                           |     +-- the array starts here
                           +-------- the scale IS the element size
```

Walked:

```
  add     rcx, 48          ; base of the member
.Lloop:
  cmp     word ptr [rcx], 0
  je      .Ldone
  add     rcx, 2           ; the increment IS the element size
```

Same fact, completely different shape. A pointer walk has no index register and no scale at all, so if you are only looking for the first pattern you will read an array as a scalar field.

And the element count comes from whatever bounds the loop — `cmp eax, 8` above bounds it at eight. Without such a bound, the count is not observable, which is worth noticing for its own reasons.

Gaps are padding, and padding is not free

```
  movzx   edx, byte ptr [rcx+8]     ; a one-byte field at +8
  mov     eax, dword ptr [rcx+12]   ; the next field is at +12
```

One byte at +8 leaves +9, +10 and +11 untouched. That is not a missing field — it is **alignment padding**: a four-byte member must sit on a four-byte boundary, and +9 is not one. The compiler inserted three bytes to get there.

Two things follow, and the second is a bug class.

You cannot see padding directly. It has no access, so it appears in your layout only as arithmetic that does not add up. Infer it from the gap and the alignment requirement of the field that follows.

Padding is never initialised. No code writes it, so it holds whatever was in that memory before. A structure copied out wholesale — to a caller, across a trust boundary, into an output buffer — carries those bytes with it. That is an information disclosure with no bug at the source level at all: the C looks perfectly innocent, and the leak lives in the gap.

Sizing the whole thing

The size is the highest byte touched, rounded up to the alignment of the widest member:

```
last member starts at 48, elements are 2 bytes, there are 8 of them
  -> it spans 48..63, so the extent is 64
the widest member is 8 bytes, so the size is a multiple of 8
  -> 64 already is; no trailing padding
```

Two mistakes to avoid: stopping at the last member's *start* offset rather than its end, and forgetting the trailing round-up when the last member is narrow.

Function-pointer fields are the highest-value find

```
mov     rax, qword ptr [rcx+24]
test    rax, rax
je      .Lout
rex_jmp rax
```

What makes this a code pointer is not how it was loaded — an eight-byte load looks like any other — but **what the code does with the value: it jumps to it.**

Until this field is typed, that jump is a dead end. There is no cross-reference from it to anything, so the call graph simply stops. Type the field as a function pointer and the edge appears, and every caller that stores something into it becomes a lead worth following.

The null test before the jump is evidence too: the field is optional, so callers are expected to leave it zero sometimes.

Strings: the one Ghidra will not show you

A stride-2 walk with a null test is **UTF-16**, which is the default string encoding across the Windows API.

Ghidra's default string search looks for ASCII runs. UTF-16 text is ASCII bytes interleaved with zeros, so it fails that search and **your Defined Strings window will not list it.** A binary that looks like it contains no interesting strings frequently contains plenty, in the encoding you did not search for.

Also worth recognising: length-prefixed forms such as `UNICODE_STRING`, where a 16-bit length and a 16-bit capacity precede a pointer — three fields that are one logical object.

Where to stop

A structure 80% named beats one 100% guessed.

Define what the evidence supports. Leave the rest as offsets — an offset is an honest statement that you have not established the type yet, and it stays visible as work. A wrong type looks finished, and everything built on top of it inherits the error silently.

And expect incompleteness: a structure is rarely fully visible from one group of callers. Fields nothing in front of you touches simply will not appear, and a layout that claims to be complete is claiming more than it can support.

The procedure

1. **Collect every access** through the pointer: offset, width, direction.
2. **Note the extension** on each, and mark the ones that do not have one.
3. **Look for index scales and pointer increments** — those are arrays, and the scale or the increment is the element size.
4. **Find the gaps** and account for them as alignment padding.
5. **Define the structure**, retype the parameter, and re-read the function.
6. **Follow the code pointers** you just typed — they are new call-graph edges.

Step 5 is what pays. The decompiler re-derives the whole function from the type, so a defined structure often turns several lines of opaque pointer arithmetic into named field accesses in one action.

Module R6

The constructor tells you where the vtable is

A virtual call is a dead end until you know which function it reaches. You will see this shape constantly:

```
mov     rax, qword ptr [rcx]
call    qword ptr [rax+0x10]
```

Two loads and a call, with no symbol anywhere. Cross-references give you nothing, because the target is computed at runtime. If you cannot resolve it, every class in the binary stays a black box and the call graph has a hole in it exactly where the interesting work happens.

You can resolve it. The information is all present, and the way in is almost always the constructor.

The object model, in three facts

1. The vptr sits at offset 0. An object with virtual functions begins with a pointer to its virtual function table. Fields follow it. So `[this]` is the vptr, and `[this + something]` is data.

2. A vtable is an array of code pointers, in read-only data. Slot *n* lives at `vtable + n * pointer_size`. On x86-64 and AArch64 that is 8 bytes per slot; on 32-bit targets it is 4.

3. The constructor writes the vptr. This is the part that turns the model into a technique. Before a constructor runs, the memory is just bytes. The constructor's job includes storing the address of the class's vtable into `[this]` — so **a store of a read-only-data address into offset 0 of a pointer is the definitive signature of a constructor, and the address it stores is the vtable you were looking for.**

That store is the oracle. Find it and you have the class.

Reading a call site

Here is a real one. MSVC 19.44, x86-64, `/O2 /GR`, from a function that takes a `Stream *` and calls a virtual method on it:

```
?call_read@@YAHPEAVStream@@PEAEI@Z:
  mov     rax, qword ptr [rcx]        ; rax = the vptr, loaded from [this]
  rex_jmp qword ptr [rax+8]           ; call slot 1 (8 / 8 = 1), tail-called
```

Two steps, and each is worth naming.

`mov rax, qword ptr [rcx]` — `rcx` is `this` under the Microsoft x64 convention, so this dereferences the object at offset 0. That is fact 1: it is loading the vptr.

`rex_jmp qword ptr [rax+8]` — an indirect call through the table. The displacement is 8, the pointer size is 8, so this is **slot 1**. The `rex_jmp` rather than a `call` is a tail call: the compiler reused the frame, which is efficient and costs you a call-graph edge.

> **Slot index = displacement / pointer size.** A displacement of 0 is slot 0, 8 is slot 1, 16 is slot 2, and so on. Getting this backwards - reading the displacement itself as the index - is the most common early mistake, and it silently shifts your whole slot table.

Reading the table

The vtable for `Stream` in the same object file:

```
??_7Stream@@6B@:
    DQ  FLAT:??_R4Stream@@6B@                ; the complete-object locator
    DQ  FLAT:??_EStream@@UEAAPEAXI@Z         ; slot 0: deleting destructor
    DQ  FLAT:_purecall                       ; slot 1
    DQ  FLAT:_purecall                       ; slot 2
    DQ  FLAT:_purecall                       ; slot 3
```

Four things to take from this.

The symbol does not point at the first entry. `??_R4Stream@@6B@` is the complete-object locator, and under MSVC it sits at **-8** from where the `??_7` symbol points. Slot 0 begins after it. Under the Itanium ABI - which GCC and Clang use on Linux, and which AArch64 and MIPS also use - the layout differs: `_ZTV` has *two* words before slot 0, an offset-to-top and a typeinfo pointer. **Same concept, different negative offsets, and an off-by-one or off-by-two in your slot table if you assume the wrong one.**

Slot 0 is the destructor. MSVC puts the deleting destructor first. The `??_E` prefix marks the *vector* deleting destructor, which takes a flag saying whether to free the memory. When your slot 1 turns out to be the class's first real method, this is why.

`_purecall` means the method is pure virtual. `Stream` declares `read`, `write` and `at_end` as `= 0`, so its own table has stubs that abort if called. A derived class overwrites those slots with real functions. Seeing `_purecall` tells you that you are looking at an abstract base and that the interesting implementations are elsewhere.

The order is the declaration order. Slots follow the order the virtual functions were declared in the class, which means the slot table you recover is a recovery of the *source*, not just of the binary.

RTTI hands you the names

When a binary is built with RTTI enabled - MSVC's `/GR`, which is the default - you do not have to guess class names at all. The same object file contains:

```
??_R0 ...   TypeDescriptor          - carries the decorated name, ".?AVStream@@"
??_R1 ...   BaseClassDescriptor     - one per base, with its offset
??_R2 ...   BaseClassArray          - the list of them
??_R3 ...   ClassHierarchyDescriptor - attributes, and how many bases
??_R4 ...   CompleteObjectLocator   - the thing at vtable[-8]
```

Follow `??_R4` from the vtable to `??_R3` to `??_R2` to the `??_R1` entries and you have the class name *and its inheritance graph*, for free, without reading a single instruction. The Itanium equivalents are `_ZTV` (vtable), `_ZTI` (typeinfo) and `_ZTS` (the name string), and they work the same way.

This is the highest-value five minutes in C++ reverse engineering. Check for RTTI before you start naming anything by hand.

Mangled names are a recognition skill

You do not need to decode these in your head - `undname` and any demangler will do it instantly. What you need is to recognise *what kind of thing* a symbol is, at a glance:

Shape	ABI	What it is
`??_7Cls@@6B@`	MSVC	a vtable
`??0Cls@@QEAA@XZ`	MSVC	a constructor
`??1Cls@@UEAA@XZ`	MSVC	a destructor
`??_R4...`	MSVC	RTTI complete-object locator
`_ZTV3Cls`	Itanium	a vtable
`_ZN3ClsC1Ev`	Itanium	a constructor
`_ZThn16_N3Cls3fooEv`	Itanium	an adjustor thunk, `this` shifted by 16

The `_ZT` family is data; `_ZN` is a function. The `??_` family is data; `??0` and `??1` are functions.

When a class has two bases

An object that inherits from two classes with virtual functions carries **two vptrs**, not one - the second at whatever offset its second base subobject starts at. Its constructor therefore performs *two* vtable stores rather than one:

```
lea     rax, offset FLAT:??_7Cls@@6BFirstBase@@
mov     qword ptr [rcx], rax          ; vptr for the first base, at offset 0
lea     rax, offset FLAT:??_7Cls@@6BSecondBase@@
mov     qword ptr [rcx+8], rax        ; vptr for the second base, at offset 8
```

Two consequences. Fields do not start at offset 8 any more - they start after *both* vptrs. And a pointer to the second base does not equal a pointer to the object; converting between them shifts by the subobject offset, which is what the `_ZThn`-style adjustor thunks exist to do.

If you reconstruct a multiple-inheritance class assuming a single vptr, every field offset after the first will be wrong by exactly one pointer, and the decompilation will look *almost* right - which is worse than looking wrong.

The procedure

1. **Look for RTTI first.** If it is there, you get names and the hierarchy for nothing.
2. **Find a constructor** - a function that stores a read-only-data address into `[this]`. The address is the vtable.
3. **Define the vtable as a structure of function pointers.** Its length is bounded by how far the entries stay plausible code addresses.
4. **Define the class structure**, vptr first, typed to that vtable.
5. **Retype `this`** on the methods, and re-read them. Field accesses that were `[rcx+0x18]` become named members.
6. **Convert each call site's displacement to a slot index** and connect it to the entry in the table.

Step 3 has to precede step 5. Retyping `this` before the vtable structure exists gives the decompiler nothing to resolve the indirect calls against, and you will do the work twice.

Where to stop. A class with its vtable recovered and half its fields named is more useful than one where every field has a confident guess attached. Mark what the evidence supports, and leave the rest as offsets.

What comes next

The tasks after this card work on a different class from the same binary: one that inherits from two bases, so it carries two vptrs and its fields begin further into the object than you might expect. You will convert call displacements into slot indices, work out how large the object is from what its constructor writes, reconstruct its declaration, and finally draw the layout with the evidence for each part of it.

Module R7

The copy that is no longer a call

You are reading a function. Somewhere in it, data moves from one buffer to another. If you are lucky you will see `call memcpy` and know exactly what happened. Most of the time you will not — the copy is *there*, but it has been dissolved into the surrounding instructions, and it looks like nothing in particular.

This matters for one reason. **A copy is where a buffer overflow lives.** If you cannot see the copy, you cannot ask the only question that matters about it: *what bounds the count?* An analyst who scrolls past four `movups` instructions without registering "that was a 32-byte copy" has scrolled past the bug.

So the skill is not memorising instruction patterns. It is this:

> **Find the (destination, source, count) triple and the loop bound.**

That question has an answer on every machine, in every compiler, at every optimisation level — including the ones you have never seen. The patterns below are worth knowing because they make the triple *fast* to spot. The method is what makes it possible at all.

One C function, six answers

Every listing on this page came from the same source file, compiled six ways. The build record is in this task's `notes/` — source, compiler banners, exact flags, and the SHA-256 of every object, so you can rebuild and check.

```c
void copy_c32(unsigned char *dst, const unsigned char *src)
{
    memcpy(dst, src, 32);
    sink(dst, 32);
}
```

That is the whole function. Watch what happens to it.

x86-64, MSVC 19.44, `/O2 /GS`

```
copy_c32:
  movups  xmm0, XMMWORD PTR [rdx]         ; load  src[0..15]
  movups  XMMWORD PTR [rcx], xmm0         ; store dst[0..15]
  movups  xmm1, XMMWORD PTR [rdx+16]      ; load  src[16..31]
  mov     edx, 32                         ; sink's second argument
  movups  XMMWORD PTR [rcx+16], xmm1      ; store dst[16..31]
  jmp     sink
```

No `memcpy`. No loop. The compiler knew the count was 32, so it moved 32 bytes with two 16-byte SSE loads and two 16-byte stores, and then **tail-called** `sink` with a `jmp` instead of a `call` — which is a separate idiom worth noticing, because the call graph just lost an edge.

Read the triple off it: destination `rcx`, source `rdx`, count 32 — and the count is a *constant*, visible in the displacements `[rdx+16]` and `[rcx+16]` and confirmed by `mov edx, 32`.

AArch64, GCC 14.2, `-O2`

```
copy_c32:
        mov     x2, x1
        mov     w1, 32
        ldp     q31, q30, [x2]          ; load  32 bytes into TWO 128-bit regs
        stp     q31, q30, [x0]          ; store 32 bytes from those regs
        b       sink
```

Same decomposition, different registers. `ldp` / `stp` are *load pair* and *store pair*: one instruction moves two registers. With 128-bit `q` registers that is 32 bytes in a single instruction, so the entire copy is two instructions.

`b sink` rather than `bl sink` — the same tail call as the x86 `jmp`.

MIPS (big-endian), GCC 14.2, `-O2`

```
copy_c32:
        addiu   $9,$5,32
        move    $2,$4
$L5:
        lwl     $8,0($5)                ; unaligned load, left half
        lwl     $7,4($5)
        lwl     $6,8($5)
        lwl     $3,12($5)
        lwr     $8,3($5)                ; unaligned load, right half
        lwr     $7,7($5)
        lwr     $6,11($5)
        lwr     $3,15($5)
        addiu   $5,$5,16
        swl     $8,0($2)                ; unaligned store, left half
        swr     $8,3($2)
        swl     $7,4($2)
        swr     $7,7($2)
        swl     $6,8($2)
        swr     $6,11($2)
        swl     $3,12($2)
        swr     $3,15($2)
        bne     $5,$9,$L5               ; loop while src != src_end
        addiu   $2,$2,16                ; DELAY SLOT — runs before the branch
```

Twenty instructions and a loop, for the copy that AArch64 did in two.

MIPS has no 128-bit registers and no block-copy instruction, and the compiler cannot prove the pointers are aligned — so it reaches for `lwl` / `lwr`, the unaligned-word load pair, and its `swl` / `swr` store twins. Four words in, four words out, sixteen bytes per iteration.

And note the last line. `addiu $2,$2,16` sits *after* the `bne`, but it executes *before* the branch is taken. That is the MIPS **branch delay slot**: the instruction following any branch or jump always runs. Read MIPS in source order and you will conclude the pointer is never advanced. It is.

Same triple, three shapes

	x86-64 / MSVC	AArch64 / GCC	MIPS / GCC
destination	`rcx`	`x0`	`$4` (⟶ `$2`)
source	`rdx`	`x1` (⟶ `x2`)	`$5`
count	32, constant	32, constant	32, constant
instructions	4	2	~20 in a loop
widest move	16 bytes (`movups`)	32 bytes (`stp q,q`)	4 bytes (`lwl` / `lwr`)

This is the lesson the rest of the module is built on. None of these shapes is "the memcpy idiom". Each is one compiler's answer to *how do I move 32 bytes on this machine*, and the answer changes with the register file, the alignment guarantees, and the instruction set. The count was a constant in all three, and that is what you were actually reading.

What changes the answer

The count stops being a constant

```
memcpy(dst, src, n);        /* n arrives from the caller */
```

MSVC 19.44 emits, simply:

```
  mov     r8d, r8d                      ; zero-extend the 32-bit count
  call    memcpy
```

The compiler cannot size the move, so it defers to the C runtime. **A visible `call memcpy` is not a failure to inline — it is evidence that the count was not known at compile time**, which is exactly the situation in which the count might be attacker-controlled. The call is a signal, not an absence of one.

AArch64 does the same thing, with one instruction worth naming:

```
        uxtw    x2, w2                  ; zero-extend w2 into x2
        bl      memcpy
```

`uxtw` is "unsigned extend word". A 32-bit count is being widened to 64 bits to become the `size_t` argument. Whenever you see a width change on a length, stop and ask whether it is signed or unsigned — `uxtw` here, but `sxtw` would mean the count is *signed*, and a signed length that goes negative becomes an enormous unsigned size. That thread is picked up properly in the bug-class module; for now, just register that the width change is information.

The size is a constant, but not a multiple of the register width

Change the 32 to a 40 and MSVC produces:

```
  movups  xmm0, XMMWORD PTR [rdx]         ; bytes 0..15
  movups  XMMWORD PTR [rcx], xmm0
  movups  xmm1, XMMWORD PTR [rdx+16]      ; bytes 16..31
  movups  XMMWORD PTR [rcx+16], xmm1
  movsd   xmm0, QWORD PTR [rdx+32]        ; bytes 32..39 — the 8-byte tail
  movsd   QWORD PTR [rcx+32], xmm0
```

Two 16-byte moves and an 8-byte tail: 16 + 16 + 8 = 40. AArch64 decomposes it identically — `ldp q31, q30` for the first 32 bytes, then `ldr x2` / `str x2` for the last 8.

The displacements are the count. The largest displacement plus the width of the move at that displacement gives you the total, and you never had to see a number 40 anywhere. Here: 32 + 8 = 40.

You wrote the loop yourself

```
for (i = 0; i < n; i++)
    dst[i] = src[i];
```

GCC on AArch64 gives you exactly what you asked for:

```
.L8:
        ldrb    w4, [x5, x3]            ; load one byte
        strb    w4, [x0, x3]            ; store one byte
        add     x3, x3, 1
        cmp     x3, x6
        bne     .L8
```

MSVC on x86-64 does something else entirely. It **recognises** the loop, and emits a size check, a `call memcpy` for the bulk, and a scalar remainder loop for the leftover:

```
  cmp     ebx, 16                       ; worth calling memcpy?
  ...
  call    memcpy                        ; the bulk
  ...                                   ; then a byte-at-a-time remainder
```

Two compilers, one source, opposite decisions. **A `call memcpy` in the disassembly does not mean the programmer wrote `memcpy`**, and a byte loop does not mean they wrote a loop. The idiom tells you what the *compiler* decided, and you are always one step removed from the source.

Somebody asked for `rep movsb` on purpose

```
  mov     rdi, rax                      ; destination
  mov     rsi, rdx                      ; source
  mov     ecx, r8d                      ; count
  rep movsb
```

This is the shape most people picture when they think "inlined copy" — and modern MSVC will not produce it for an ordinary `memcpy` at any optimisation level. Getting it into this listing required calling the `__movsb` intrinsic explicitly.

You will still meet it constantly, in two places: **Windows kernel drivers**, where `RtlCopyMemory` and friends inline to it, and **older binaries**, built by toolchains that made a different choice. So it is worth knowing cold — as an artifact of a particular era and a particular kind of code, not as "what a `memcpy` looks like".

Note the register discipline: `rep movsb` has no operands. Destination is always `rdi`, source always `rsi`, count always `rcx`. The three `mov` instructions above it *are* the argument setup, and they are the triple.

Reading the triple: the checklist

When you hit something that might be a copy:

1. **Which register or stack slot is written?** That is the destination, and it is the one whose size you need to bound.
2. **Which is read?** The source. If it traces back to a parameter, an I/O buffer or a parsed field, it is attacker-influenced.
3. **What is the count, and is it constant?** Constant means read the displacements. Non-constant means find where the value came from.
4. **What bounds the loop?** A `cmp` against a length, a pointer compared to an end address, a `rep` with `rcx` — or nothing at all.
5. **If the count is not constant, where is it checked?** This is the question the whole module exists to make you ask. An inlined copy tells you exactly where the bounds check *should* be, and its absence is not something the listing announces. You have to notice that nothing is there.

Step 5 is the one that finds bugs. Steps 1 to 4 are what make step 5 possible in under a minute.

Where this goes next

The drill that follows this card gives you fragments and asks only *what operation is this, and is the count constant* — ten seconds each, until the shapes are reflexes rather than deductions.

After that you will read a complete function that takes a length from its caller and copies that many bytes into a fixed 64-byte stack buffer, and you will be asked where the outcome was decided. It is not the line the copy happens on.

The same reading carries into the next module, where `memcpy` is replaced by a switch statement, a division and a stack cookie — three more constructs that mean something different on each of these three machines, for the same reasons.

Module R7M

Idioms II: the constructs that hide the program's shape

The copy module was about a library call that stopped being a call. This one is about four constructs that stop looking like what the programmer wrote:

- a **switch** becomes a table and a bounds check;
- a **division** becomes a multiplication;
- a **ternary** becomes no branch at all;
- a **call** becomes a jump.

Each changes what the control-flow graph looks like, and two of them are directly load-bearing for finding bugs.

Switch: the bounds check the compiler wrote

```
switch (op) {
case 0: return handler_a(arg);
...
case 6: return handler_c(arg + 1);
default: return -1;
}
```

```
dispatch:
  cmp     ecx, 6
  ja      .Ldefault          ; <- a real bounds check
  movsxd  rax, ecx
  lea     r8, [rip + .Ltable]
  jmp     rcx                ; <- indirect dispatch through the table
.Lcase0:
  mov     ecx, edx
  jmp     handler_a
  ...
.Ltable:
  dd  .Lcase0, .Lcase1, .Lcase2, .Lcase3, .Lcase4, .Lcase5, .Lcase6
```

Three things to read here, in order of how much they matter.

`cmp ecx, 6` / `ja` is a bounds check, and the compiler wrote it. `ja` is the *unsigned* jump-if-above, so anything larger than 6 — including a value that would be negative read as signed — goes to the default. The index is then scaled and used to load a code pointer with no further validation. **That compare is the only thing making the indirect jump safe**, which is why its absence is a finding rather than an oddity.

Nothing in the encoding knows how long the table is. `jmp rcx` will jump wherever `rcx` points.

The count is the bound plus one. `cmp ecx, 6` with `ja` admits 0 through 6 — seven cases — and the table has seven entries. Reading the immediate as the count gives six and drops the last case, which in real code is usually the most recently added one.

Every case ends in `jmp`, not `call`. These are tail calls, and each is a call-graph edge that cross-references will not record. Ask "who calls `handler_a`" and this function does not appear, though it reaches it on two different cases.

Not every switch becomes a table. Sparse or small case sets compile to a chain of comparisons or a bit test instead, and MSVC also emits a two-level form with a byte index table when the cases are spread out. The table is what you get when the cases are dense.

Division: the compiler does not divide

Integer division is slow, so a compiler dividing by a *constant* does not use the divide instruction. It multiplies by a fixed-point reciprocal.

```
unsigned int div_unsigned(unsigned int n) { return n / 1000003u; }
```

```
  mov     eax, 0xC6F4545
  mul     ecx               ; 64-bit product; high half lands in edx
  sub     ecx, edx
  shr     ecx, 1
  lea     eax, [rdx+rcx]
  shr     eax, 19
```

A big odd constant, a multiply whose **high half** is kept, and a right shift. That shape is a division, and the divisor is not written anywhere in it.

The extra `sub` / `shr` / `lea` is a correction: the exact reciprocal did not fit in 32 bits, so the compiler uses a slightly-too-small magic number and adds back the error. You do not need to derive the algebra to read the code — you need to recognise the shape and know the divisor is recoverable.

Recovering it: evaluate, do not derive. The published closed form (`d ≈ 2^(32+s) / M`) is approximate for this variant, and here it gives **1000002** — off by one from the true divisor. The reliable method is to model the instruction sequence and find the smallest input for which it yields 1. That value is the divisor. It is also how you would sanity-check any answer you got from the formula.

Modulo is division plus a multiply back

```
return n % 953u;
```

```
  mov     eax, 0x1131289
  mul     ecx
  shr     edx, 2
  imul    eax, edx, 953     ; <- the divisor, in plain sight
  sub     ecx, eax          ; n - (n/d)*d
```

The same reciprocal trick computes the quotient, then multiplies it back by the divisor and subtracts. Convenient for a reader: **the divisor appears as an immediate**, so a modulo is usually easier to read than the division inside it.

Signedness shows up here too

```
int div_signed(int n) { return n / 1000003; }
```

```
  mov     eax, 0x8637A2A3
  imul    ecx               ; SIGNED multiply
  add     edx, ecx
  sar     edx, 19           ; ARITHMETIC shift
  mov     eax, edx
  shr     eax, 31           ; extract the sign bit
  add     eax, edx          ; and add it back
```

Three tells at once: `imul` instead of `mul`, `sar` instead of `shr`, and a sign correction on the end. The correction exists because C truncates toward zero while an arithmetic shift rounds toward negative infinity; adding the sign bit back fixes the difference for negative inputs.

Even a power of two needs it:

```
int div_pow2_signed(int n) { return n / 64; }
```

```
  mov     eax, ecx
  cdq                       ; edx = all ones if negative, else zero
  and     edx, 63           ; so the bias is 63 or 0
  add     eax, edx
  sar     eax, 6
```

An unsigned division by 64 is one instruction, `shr eax, 6`. The signed version takes five. **So a bare `shr` says unsigned, and a `sar` with a bias before it says signed** — the R1 thread again, now in arithmetic.

The same C, three machines — and a result worth being surprised by

The obvious prediction is that this idiom is an x86 artifact. x86-64 has no fast integer divide, so it multiplies instead; AArch64 has `udiv` and `sdiv`, MIPS has `divu`, so surely those emit a division and the trick disappears.

That prediction is wrong. Here is the same function, same source, compiled for all three:

```
x86-64 (MSVC)               AArch64 (GCC)                MIPS (GCC)
  mov   eax, 0xC6F4545        mov   w1, 17733              li    $2, 0xC6F0000
                              movk  w1, 0xc6f, lsl 16      addiu $2, $2, 17733
  mul   ecx                   umull x1, w0, w1             multu $4, $2
                              lsr   x1, x1, 32             mfhi  $2
  sub   ecx, edx              sub   w0, w0, w1             subu  $4, $4, $2
  shr   ecx, 1                                             srl   $4, $4, 1
  lea   eax, [rdx+rcx]        add   w0, w1, w0, lsr 1      addu  $2, $2, $4
  shr   eax, 19               lsr   w0, w0, 19             srl   $2, $2, 19
```

Identical magic constant. Identical shift. `0xC6F4545` and 19 on all three — AArch64 builds it with `mov` + `movk` and MIPS with `li` + `addiu` because neither can carry a 32-bit immediate in one instruction, but the value is the same. Neither `udiv` nor `divu` appears anywhere.

The reason is that a multiply-and-shift beats a hardware divide on all three machines. The magic constant is derived from arithmetic — from the divisor and the word size — not from the instruction set, so **it does not change when the machine does.**

So the lesson is the opposite of the tidy one, and more useful:

> **The magic constant and the shift are portable; only the spelling changes.** Recognise the shape — a large odd constant, the high half of a product, a shift — and you can read this idiom on a machine whose assembly you have never seen. Recover the divisor once and the answer holds everywhere.

The corollary is the habit worth keeping: **check what the compiler actually did rather than what the instruction set made available.** An ISA offering a divide instruction does not mean the compiler used it, and the reasoning that predicted otherwise here was perfectly sensible and still wrong.

*(Two more contradictions from the same builds, worth knowing before you generalise from the x86 listings on this page. GCC compiled the seven-case switch to a **comparison tree** on AArch64 — `cmp w2,3 / beq / bgt / cmp w2,1 / beq ...` — not a jump table, so the bound-check-plus-table shape above is one option among several rather than the compiled form of a switch. And AArch64's signed power-of-two division uses `cmp` + `csel` to select the bias where x86 used `cdq` + `and`. Same correction, different mechanism.)*

Branchless selection

```
int select_max(int a, int b) { return a > b ? a : b; }
```

```
  cmp     ecx, edx
  cmovg   edx, ecx
  mov     eax, edx
```

`cmovg` moves only if the comparison said greater. Both values were computed already; the choice costs no branch.

The consequence is structural rather than semantic: **there is no branch, so there is no basic-block boundary.** A control-flow graph shows straight-line code where the source had a decision, and a path-counting argument that assumes branches equal decisions will undercount. `setcc`, which materialises a condition as 0 or 1, has the same effect.

The stack cookie, read as evidence

```
with_array:
  push    rbx
  sub     rsp, 0x70
  mov     rax, qword ptr [rip + __security_cookie]
```

```
xor     rax, rsp                        ; mix with the frame address
mov     qword ptr [rsp+0x60], rax
...
mov     rcx, qword ptr [rsp+0x60]
xor     rcx, rsp
call    __security_check_cookie
```

Treat this as information first and as a mitigation second.

A cookie means the frame contains a local array. MSVC's `/GS` instruments functions with buffers, so seeing this tells you a buffer is here before you have found it.

Its position tells you where the buffer region ends. The cookie sits between the locals and the saved return address, so the distance from a buffer's start to the cookie is how far a forward overflow runs before it is detected.

Mixing with `rsp` is why the value differs per frame and cannot simply be replayed.

And what it does not tell you: nothing about whether a bug exists. `/GS` changes the *impact* of an overflow — a crash on return rather than a hijacked return address — without changing the fact that one is there.

Tail calls, and the hole they leave

```
mov     ecx, edx
jmp     handler_a
```

The frame is torn down first and control transfers with `jmp`. The callee's `ret` returns to *this* function's caller.

For a reader this is a **missing call-graph edge**. Every case in the dispatch example above ends this way — seven lost edges in one small function. Any reachability question answered from call cross-references alone will be wrong by however many tail calls lie on the path, and this is one of the blind spots the cross-reference module names.

The tell is a `jmp` to a *function* rather than to a label, usually right after a frame teardown and with arguments freshly set up.

What to take away

1. **Find the bound check on every jump table.** It is the compiler's own, and its absence is a finding that nothing on the page marks.
2. **The case count is the bound plus one** with an unsigned compare.
3. **A magic multiply is a division**, and the divisor is recovered by evaluating the sequence rather than trusting a formula.
4. **`sar`, `imul` and a sign correction mean signed**; a bare `shr` means unsigned.
5. **A cookie is evidence a buffer exists**, and it marks where the buffer region ends.
6. **A `jmp` to a function is a call your cross-references did not record.**

Module R8

Bug classes, and what they look like compiled

This is the module the rest of the course was building toward. You can read a listing, recover types, follow cross-references. Now: what does a *defect* look like?

Six classes, each with the C that produces it and the shape it compiles to. Every listing here is real output from a source file written to contain exactly one bug per function.

The rule this module is built on

> **A bug you can only see in the decompilation is a hypothesis until you have seen it in the listing.**

The decompiler is a model. It infers types, widths and signedness, and when it infers wrong it will show you code that is subtly not what runs — a `uint` where the machine used a signed compare, a parameter it did not notice, a jump table it failed to recover. Those inferences are exactly the things bug classes turn on.

So the decompilation is where you *find* candidates, and the listing is where you *confirm* them.

---

1. Missing bounds check

```c
memcpy(dst, src, n);        /* n never compared with sizeof(dst) */
```

```
copy_vuln:
  push    rbx
  sub     rsp, 0x70
  ...cookie setup...
  mov     ebx, edx
  mov     r8d, edx            ; the caller's length, straight to the size argument
  mov     rdx, rcx
  lea     rcx, [rsp+0x20]     ; a 0x40 local
  call    memcpy
```

The signature is an **absence**: no `cmp` against the destination size anywhere before the call. Search the whole function for a comparison and there is none.

The cookie is evidence, not protection — `/GS` instruments functions with local arrays, so its presence told you a buffer was here before you found it.

2. Signed length

```c
if (n > DST) return -1;     /* n is int, so this is a SIGNED compare */
memcpy(dst, src, (unsigned)n);
```

The check passes for `n = -1`, and the cast then makes it enormous. Compiled, the entire difference from the correct version is one mnemonic:

```
  signed_vuln:  cmp edx, 64 / cmovg  edx, eax     ; g = signed greater
  signed_fixed: cmp edx, 64 / cmova  edx, eax     ; a = unsigned above
```

One letter. `jg` / `jl` / `cmovg` are the signed family; `ja` / `jb` / `cmova` the unsigned one. A length compared with the signed form is a length that can be negative, and this is the single most productive tell in modern binary vulnerability research.

3. Integer overflow in a size computation

```
unsigned total = count * size;      /* wraps */
return alloc(total);
```

```
alloc_vuln:
  imul    ecx, edx
  jmp     alloc
```

Two instructions, and the bug is that nothing checks the multiply. A wrapped product gives a small allocation, and whatever fills it afterwards uses the *unwrapped* count.

The fixed version is unmistakable by contrast — it divides `0xFFFFFFFF` by one operand and compares against the other, so a `div` sitting in front of an allocation is very often an overflow guard.

Where to look: any multiply whose result becomes an allocation size or a copy length, with no comparison between them.

4. Uninitialised padding

```
struct Record { uint32_t id; uint8_t kind; /* 3 bytes padding */ uint32_t value; };
out->id = id; out->kind = kind; out->value = value;
```

```
leak_vuln:                          leak_fixed:
                                      xor     eax, eax
                                      mov     qword ptr [rcx], rax   ; <- the fix
  mov     dword ptr [rcx], edx        mov     dword ptr [rcx], edx
  mov     byte ptr [rcx+4], r8b       mov     byte ptr [rcx+4], r8b
  mov     dword ptr [rcx+8], r9d      mov     dword ptr [rcx+8], r9d
```

Every field is written and the structure is still not fully initialised: the three bytes between `+5` and `+7` are padding, nothing writes them, and they carry whatever was in that memory before. Copy the structure across a trust boundary and those bytes go with it.

The signature is arithmetic, not an instruction. Sum the widths of the stores and compare with the structure's size. A shortfall is padding, and padding that is later copied out is a disclosure.

The fix is a single zeroing store, which is why an added `memset` in a patch so often means an information leak was fixed.

5. Double fetch

```
if (*user > DST) return -1;                /* read once  */
memcpy(dst, user + 1, *user);              /* read again */
```

```
fetch_vuln:                          fetch_fixed:
  mov     eax, dword ptr [rcx]         mov     eax, dword ptr [rcx]
  cmp     eax, 64                      cmp     eax, 64
  jbe     .Lok                         jbe     .Lok
  ...                                  ...
.Lok:                                .Lok:
  mov     r8d, dword ptr [rcx]         mov     r8, rax        ; <- reuses it
  lea     rdx, [rcx+4]                 lea     rdx, [rcx+4]
  call    memcpy                       call    memcpy
```

Two loads from the same address, one before the check and one after. If the memory is shared — a user-mode buffer a driver is reading, another thread's data — the value can change in between, so the check validates one value and the copy uses another.

The signature is exactly that: the same address loaded twice around a validation. The fix is deleting the second load.

6. Off-by-one

A bound that admits one value too many: `jbe` where `jb` was meant, `<=` where `<` was meant. One instruction, and usually one byte in the encoding.

These are found by reading the comparison against the array's real size rather than by any distinctive shape — which is why recovering the size in the data module matters here.

How the decompiler misleads

Worth knowing specifically, because these are the failure modes that hide the classes above:

- **A wrong signature** hides a parameter, so a length appears to come from nowhere.
- **A missed sign extension** shows a `uint` where the machine used a signed compare — hiding class 2 entirely.
- `undefined4` / `undefined8` mean the width was not established, so truncation is invisible.
- `CONCAT` **and** `SUB` appear when values were joined or split in ways the type model could not follow — often exactly where a truncation lives.
- **An unrecovered jump table** leaves half the function unreachable in the decompilation, so whole paths simply do not appear.

(This card shows the disassembly side of each class. The paired decompiled forms are not in this pack yet — producing them needs Ghidra output for these exact objects, and nothing here grades a decompiled listing.)

Writing it down

A finding is: **source, path, sink, precondition, class.**

- *source* — where the attacker-influenced value enters
- *path* — how it reaches the sink, including the edges cross-references missed
- *sink* — the operation that goes wrong
- *precondition* — what has to be true for it to trigger
- *class* — which of the above it is

And then stop. Whether it is exploitable is a different question and a different job; this course ends at identification, deliberately.

What to take away

1. **Confirm in the listing what you suspect in the decompilation.**
2. **Missing checks have no instruction.** You find them by looking for something that should be there.
3. **Signed comparison on a length is the highest-yield single tell.**
4. **Sum the stores against the structure size** to find uninitialised padding.
5. **The same address loaded twice around a check** is a double fetch.

One bug, and how many copies of it are in the binary

Every listing below is real compiler output. The build record — source, compiler banners, exact flags, object SHA-256 — is in `notes/build.txt`.

R8′s rule is that a bug you can only see in the decompilation is a hypothesis until you have seen it in the listing. This card adds the question that comes immediately after: **you have seen it in the listing — but is that the copy anyone can reach?**

The source

```
void vuln(const char *input) {       // line 8
    char buf[16];                    // line 9
    strcpy(buf, input);              // line 10
}                                    // line 11

int main(int argc, char **argv) {    // line 13
    if (argc > 1) vuln(argv[1]);     // line 14
    return 0;                        // line 15
}                                    // line 16
```

One bug: an unbounded `strcpy` into a 16-byte frame buffer. Note the line numbers. They are about to do real work.

What the compiler is obliged to emit

`vuln` has **external linkage**. Nothing in this file requires that — but the compiler cannot know whether some other translation unit calls it, so it must emit a standalone, callable copy. GCC does exactly that at `-O2`:

```
vuln:
        sub     rsp, 40
```

```
        mov     rsi, rdi
        mov     rax, QWORD PTR fs:40          ; canary, read from TLS
        mov     QWORD PTR [rsp+24], rax
        xor     eax, eax
        mov     rdi, rsp                      ; buf is rsp+0
        call    strcpy                        ; the bug
        mov     rax, QWORD PTR [rsp+24]
        sub     rax, QWORD PTR fs:40
        jne     .L5
        add     rsp, 40
        ret
.L5:
        call    __stack_chk_fail
```

Intact, and easy to find. Two habits worth taking from it before we go further:

- The canary here is a **TLS read** (`fs:40`) checked with `sub` / `jne`. MSVC instead reads a **global** `__security_cookie` and XORs it with `rsp`, then calls `__security_check_cookie`. Same purpose, different shapes — and the shape tells you the toolchain before anything else does.
- `buf` is at `rsp+0`, the canary at `rsp+24`. That is **24 bytes** across a 16-byte array: eight bytes of alignment padding sit between them. The distance from a buffer to its cookie is a per-toolchain fact, never `sizeof(buf)` by assumption. A decompiler reporting a 24-byte array here is reporting the *gap to the next thing it recovered*, not the declaration.

The copy you were not looking for

At `-O2` the compiler may also **inline** `vuln` into every caller. When it does, the binary contains the same overflow more than once: once standalone, once per call site.

That would be a nuisance and nothing more, except for which one is *live*. In this program `main` is the only caller. If `vuln` was inlined there, then:

- the copy inside `main` is **reachable** — it runs on `argv[1]`;
- the standalone `vuln` is **dead** — nothing calls it, and it survives only because external linkage obliged the compiler to emit it.

A finding that names the standalone `vuln` is a finding about code that never executes. It will be perfectly accurate about the bug class, cite a real instruction, and be useless.

How to tell inlined code from code that was always there

You do not have to guess. A `/FAsc` listing from MSVC — and `-g` line tables generally — interleaves the C source lines that produced each instruction. **Inlined code keeps the line number of the function it came from.** So a function whose own source spans lines 13–16 will suddenly show an annotation from line 10, and that discontinuity is the tell:

```
; 14   :     if (argc > 1) vuln(argv[1]);      <- this function's own source
  cmp     ecx, 1
  jle     SHORT $LN6@main
```

```
; 10   :     strcpy(buf, input);              <- line 10 is not in lines 13-16
  ...                                           this came from somewhere else
```

Without source annotation you use the same reasoning on weaker evidence: a frame that allocates a buffer the function's own C never declares, a `/GS` cookie in a function with no visible local array, or two byte-identical sequences at different addresses.

And sometimes the bug is not there at all

The other direction is worse, because it is silent. An optimiser is allowed to delete a vulnerability outright when it can prove the results are unused. If the copy gets expanded inline, and the destination is a local that is never read afterwards, dead-code elimination takes the whole thing — buffer, copy, and the guard around it.

When that happens the source is still vulnerable and the shipped binary is not. The same source, at the same optimisation level, can produce an intact bug on one target and nothing at all on another, decided by whether the copy became a call the optimiser must respect or inline code it can reason about.

This is why **the listing is the evidence and the source is only the hypothesis** — the rule runs in both directions.

What a finding has to say

R8 records a finding as *(source, path, sink, precondition, class)*. Reachability is what the *path* element is for, and this card is the reason it is not optional:

1. **Which instance.** Give the address of the copy that runs, not the first one you found. If several are reachable, that is several findings.
2. **Reached from where.** Name the caller and the input that gets there.
3. **Confirmed present in this build.** Not "the source does an unbounded strcpy" — *this object, at this address, does.*

The discipline R8 ends on still applies: identify, and stop. Naming the reachable instance is part of identification, not a step past it.

Try it yourself

`notes/build.txt` has the exact commands and object hashes. Compile the source above for two different targets at `/O2` and read both listings. The build record's "WHAT EACH BUILD ACTUALLY EMITTED" section records what happened here and disagrees with what most people predict.

Module R9

Patch diffing: the patch tells you where the bug was

Everything so far has been about finding a bug in a binary that nobody has told you anything about. This module is the opposite situation, and it is the one the industry actually runs on:

Somebody already found the bug and fixed it. The fix is in the file. Read it backwards.

That is why n-day reproduction is the highest-yield exercise in vulnerability research. You are not searching — you are being shown where to look, and the only work left is understanding what you were shown.

It is also this course's exit skill, and where it stops.

What a security fix actually looks like

The expectation is a rewritten function. The reality, over and over, is two or three instructions.

Here are three real fixes, each the complete difference between a vulnerable function and its patched twin:

```
cmovg   edx, eax        ->   cmova   edx, eax
```

One mnemonic. Signed comparison became unsigned, so a negative length no longer passes the bound check. That is the entire patch.

```
                            xor     eax, eax
                            mov     qword ptr [rcx], rax
mov  dword ptr [rcx], edx   mov     dword ptr [rcx], edx
```

One added store, zeroing the structure before its fields are written. That is an information-disclosure fix: the bytes being zeroed are alignment padding that nothing else writes.

```
mov     r8d, dword ptr [rcx]   ->   mov     r8, rax
```

A load *deleted*. The vulnerable version read the user's length a second time after checking it; the fix reuses the value it already validated. A double fetch, closed by not fetching twice.

None of these would stand out in a diff that also contains a compiler upgrade's worth of layout churn. That is the actual difficulty of this work.

The fix signatures worth knowing

What appeared, or vanished	What was probably fixed
A `cmp` and a conditional branch before a copy	missing bounds check
A signed branch became unsigned (`jg` → `ja`, `cmovg` → `cmova`)	signed length
`movsx` / `movsxd` became `movzx`, or `ldrsw` became `ldr w`	sign confusion on a width change
A `div` or a comparison added before a multiply	integer overflow in a size
An added `memset`, or a zeroing store before field writes	uninitialised memory or padding disclosure
A second load of the same address disappeared	double fetch
A bound changed by one, or `jbe` became `jb`	off-by-one
An allocation size expression changed	overflow, or an off-by-one in a size
A dispatch table entry changed	a handler that should not have been reachable

Each is small. That is the point: **learn to see the small change, because the small change is the fix.**

Getting both sides, and matching them

You need the same binary before and after. In practice: a vendor update package, a symbol server, or two versions of a firmware image.

Then the matching problem. Two builds of the same source rarely produce comparable output, because everything unrelated has also moved:

- **Inlining decisions change**, so a function present in one build is distributed across three in the other.
- **A compiler upgrade** changes instruction selection everywhere at once.
- **Profile-guided layout** reorders functions between builds, producing thousands of differences that mean nothing.

A diff tool scores similarity and pairs functions up. It will pair some things wrongly, and it will report a great many changes that are pure noise.

> **Filter by symbol first.** If you have names — from a PDB, from exports, from a symbol server — start from the handful of functions whose names suggest they handle the thing the advisory mentions, and read those by hand. Trusting a similarity score across a whole binary is how a day disappears.

`ghidriff` and BinDiff both do the pairing. Neither does the reading.

The silent fix

Most fixes have no advisory. A vendor finds a bug internally, fixes it in the next release, and says nothing — so a diff between two ordinary versions frequently contains a security fix that no CVE describes.

This is the reason to diff releases that were not announced as security updates, and it is also a reason to be careful: **a change is not a security fix just because you found it in a diff.** Most changes are features and refactors. The question to ask of each one is what class of defect it would have prevented, and if the answer is "none", move on.

Writing it up, and stopping

The output is: **function, class, what changed, precondition.**

- *function* — which one, in which version
- *class* — what kind of defect the change prevents
- *what changed* — the specific instructions, not a paraphrase
- *precondition* — what an attacker would need to reach it

And that is where this course ends. Turning a patch diff into a working trigger is the next skill and a different one, with its own risks and its own ethics; identifying what was fixed is a complete and useful piece of work on its own.

What to take away

1. **The patch names the bug.** Reading backwards from a fix is far cheaper than searching.
2. **Fixes are small** — often one mnemonic, one added store, one deleted load.
3. **Most diff output is noise.** Filter by symbol, then read by hand.
4. **A change is not automatically a security fix.** Ask what class it prevents.
5. **Stop at identification.**