Vulnerability Research — classes, signatures and mitigations

Concept cards — generated 2026-08-05

Contents

Track V

Module M01-mitigations

M — Mitigations: what each one changes, and where to look for it

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

The question this module answers

Not *"which mitigations exist"* — you can look that up. The question is:

> **Given a binary, how do you tell which ones are on?**

And the answer splits in two, which is the part that catches people.

Visible in the instructions	Visible only in headers or at link time
stack guard — cookie stored, checked on exit	**NX / DEP** — a header bit, a segment flag
CFG — indirect calls routed through a dispatcher	**ASLR** — `/DYNAMICBASE` bit
CET — `endbr64` landing pads	**PIE** — `ET_DYN` vs `ET_EXEC`, PLT use
FORTIFY — `__*_chk` calls carrying a size	**RELRO** — a `GNU_RELRO` segment

A reader who only greps the disassembly will conclude a binary has no mitigations when four of the seven were never going to appear there. That is the mistake this module exists to prevent.

It is not a theoretical worry. Compiling this course's V1 source with `-fPIE` produced a **byte-identical object** to the non-PIE build — same SHA-256. The function reads no globals and makes one library call, so position independence changed nothing the compiler emitted. It changes what the *linker* does (`call strcpy` becomes `call strcpy@PLT`) and what the header says.

Backward edge: the stack guard

Covered in full in V1. One line to keep: it protects the **return address**, by putting a value in front of it that a forward overflow must destroy first. It changes impact, not existence.

Forward edge: CFG and CET

Both constrain where an *indirect* call or jump is allowed to land. Neither has anything to do with the return address.

CFG — Windows, `/guard:cf`

An indirect call through a function pointer, unguarded:

```
mov     rax, qword ptr [rdi]      ; load the function pointer
...
call    rax                       ; go wherever it points
```

The same call site with `/guard:cf`:

```
mov     rax, qword ptr [rdi]
...
call    qword ptr [__guard_dispatch_icall_fptr]
```

The target still arrives in `rax`; what changed is that control goes through a dispatcher which checks `rax` against a table of addresses the binary declared as valid indirect-call targets. A pointer that has been overwritten with attacker bytes is not in that table, and the process dies.

Two things follow, and both matter when reading real binaries:

- CFG is only visible where there *is* an indirect call. A CFG-enabled binary whose function you happen to be reading has no function pointers in it looks exactly like a CFG-disabled one. **Absence of the dispatcher is not evidence the mitigation is off.**
- The check is on the *target*, not on how the pointer got its value. A use-after-free that lands on a valid target is not stopped.

CET — Linux, `-fcf-protection=full`

```
vuln:
        endbr64                   ; the landing pad
        sub     rsp, 24
        mov     rsi, rdi
        mov     rdi, rsp
        call    strcpy
        add     rsp, 24
        ret
```

One extra instruction, at the top of the function. Hardware enforces that an indirect branch may only land on an `endbr64`; anything else faults.

Unlike CFG, `endbr64` is emitted at function entry **whether or not the function is ever called indirectly**, which makes it a reliable per-binary tell. See a few `endbr64` and the build had CET on.

CET also has a backward-edge half — a shadow stack — which is a runtime and hardware property and leaves no mark in the instruction stream at all.

FORTIFY — Linux, `-D_FORTIFY_SOURCE=2`

Plain:

```
        call    strcpy
```

Fortified:

```
        mov     edx, 16
        call    __strcpy_chk
```

The compiler already knew the destination was sixteen bytes. FORTIFY makes it say so: `__strcpy_chk` takes the real size and aborts if the copy would exceed it.

Three things to be precise about:

1. **It only works where the size is known at compile time.** A copy into a heap buffer of runtime size gets the ordinary `strcpy` back, with no warning that the protection did not apply.
2. **It converts, it does not fix.** An overflow becomes a controlled abort. The source is unchanged and still wrong.
3. **MSVC has no analogue.** There is no `/D_FORTIFY_SOURCE`, no `__strcpy_chk` family, nothing that passes the destination size to the runtime. So a `_chk` call is not only a mitigation tell, it is a **platform tell** — you are looking at glibc.

The third category: mitigations that are in neither place

The split above has two columns because both of them are about **a file**. There is a third group that is in neither, and it is the group that decides what a kernel-mode bug is worth:

Visible in the instructions	Visible in headers / at link time	**Visible in neither**
stack guard, CFG, CET, FORTIFY	NX/DEP, ASLR, PIE, RELRO	**SMEP, SMAP, KPTI, HVCI, kASLR**

These are **CPU state and OS policy**, not properties of a binary. SMEP is a bit in `CR4`. Whether it is set is decided by the kernel at boot, on the machine the target is running on — so **two identical copies of the same driver, byte for byte, can have completely different exploitability** depending on the host.

You cannot grep for them. You ask the system.

What each one actually stops

Name	What it prevents	Why it matters to a memory-corruption bug
SMEP	the kernel executing a page marked user-accessible	kills the classic "point a kernel function pointer at my user-mode shellcode". Forces code reuse *inside* kernel memory instead.
SMAP	the kernel *reading or writing* user pages outside sanctioned windows	kills the follow-on: staging a fake structure or a ROP chain in user memory and having the kernel dereference it
kASLR	knowing where the kernel and its drivers are loaded	turns a write primitive into a write-somewhere primitive; creates a *demand for an information leak*
KPTI	user page tables containing kernel mappings	closes the Meltdown-class read side channel that made kASLR cheap to defeat
HVCI	unsigned or writable-and-executable kernel pages	removes "allocate kernel memory and write code into it" entirely; the hypervisor enforces it, so the kernel cannot be talked out of it
kCFG	indirect kernel calls to non-registered targets	the kernel's CFG — same idea as user-mode CFG, different bitmap

SMEP and SMAP together are why modern kernel exploitation looks the way it does. With both on, an attacker with a kernel write primitive cannot execute their own code and cannot even stage data in their own address space for the kernel to read. What is left is reuse of memory the kernel already trusts — which is why so much kernel exploitation writing is about finding a place to put data that the kernel considers its own.

The one that *does* leave a trace

SMAP is CPU state, but **the code that cooperates with it is visible**. A kernel routine that legitimately needs to touch user memory brackets the access with `stac` / `clac` — set and clear the AC flag, temporarily lifting SMAP:

```
stac                    ; SMAP off for this window
mov     rax, [user_ptr] ; a deliberate user-memory access
clac                    ; SMAP back on
```

So in a kernel image, `stac` and `clac` mark **every place the kernel intends to read user memory** — which is a very useful thing to be able to enumerate, and the closest this category gets to an instruction-level signature. On Linux the same windows appear as `copy_from_user` / `copy_to_user` and their inlined forms.

Names, again — this is M2's problem in a new place

Every one of these has more than one name, and the vendor's marketing name and the CPU's name are usually different:

Also written as	Canonical
Supervisor Mode Execution Prevention · PXN (ARM)	**SMEP**
Supervisor Mode Access Prevention · PAN (ARM)	**SMAP**
KVA Shadow (Microsoft) · KAISER · "Meltdown mitigation"	**KPTI**
"Memory integrity" (Windows Security UI) · Hypervisor-protected Code Integrity	**HVCI**
kernel Control Flow Guard	**kCFG**

A report that says "PAN was enabled" and a report that says "SMAP was enabled" are the same report on different hardware. See module M2.

How you actually determine them

Not from the binary. From the running target:

- **Windows** — `systeminfo`, `msinfo32` (Device Guard / VBS lines for HVCI), and the `Win32_DeviceGuard` WMI class. kASLR and KPTI/KVA-shadow status appear in `Get-SpeculationControlSettings`.
- **Linux** — `/proc/cpuinfo` flags for `smep` / `smap`, `/sys/devices/system/cpu/vulnerabilities/*` for the KPTI family, and `dmesg` at boot.
- **Either** — read `CR4` directly from a kernel debugger. SMEP is bit 20, SMAP is bit 21. This is the ground truth; everything above is a report of it.

> **Write this down as configuration, not as a property of the file.** A finding that says "SMEP defeats this" is incomplete unless it also says *on which host*. The same driver on a machine with SMEP disabled — an older CPU, a hypervisor that does not expose it, a deliberately relaxed test rig — is a different finding.

Scope note for this course. These are covered here because you cannot assess a kernel-mode finding without them, and because they extend this module's central claim — *the evidence for a mitigation is not always in the file* — further than any user-mode example can. The course does not ship kernel binaries or a kernel lab; the parked plan for that is TheRange's `todo-m3-kernel-mitigations.md`.

What none of them do

They are all reachability and impact controls. **None of them removes a vulnerability.**

- A stack guard does not stop the overflow; it detects the damage afterwards.
- CFG and CET do not stop a pointer being corrupted; they constrain where the corrupted value may transfer control.
- FORTIFY does not stop a bad length being computed; it refuses the copy at runtime, and only when it can.

So a finding still gets written. What changes is the **impact** section: what an attacker gets, and what they would have to defeat first. Recording that is part of identification; claiming a bug is "not exploitable" because a mitigation is present is not.

What to write down for a binary

1. Which mitigations you checked **in the headers** and what they said.
2. Which you checked **in the code** and what you saw.
3. Which you **could not determine** and why — no indirect calls to reveal CFG, a heap-sized copy that FORTIFY cannot cover.

That third line is what stops the next reader repeating your work.

Module M02-naming

M2 — One mechanism, five names

A short module about vocabulary, because the vocabulary is genuinely confusing and the confusion runs both ways: people treat one mechanism as two things, and treat two different mechanisms as one thing.

Nothing here is a new defence. Module M covered what they do. This is about recognising them when someone else names them.

The worst offender: NX

One bit in a page table entry. Five names in common use:

Name	Whose
NX — No-eXecute	AMD, and the name that stuck generically
XD — eXecute Disable	Intel, same bit
XN / PXN — eXecute Never / Privileged XN	ARM
DEP — Data Execution Prevention	Microsoft, the *OS feature* built on the bit
W^X — write xor execute	OpenBSD, the *policy* of never mapping both

DEP is not a different thing from NX. DEP is Windows' name for enforcing it, including a software fallback on CPUs that lacked the bit. If a report says "DEP bypass" and another says "NX bypass", they are describing the same constraint: you cannot execute the bytes you just wrote into a data page, so the payload has to reuse code that is already executable. That is why ROP exists on both.

`W^X` is a policy statement rather than a hardware name — a page may be writable or executable, never both — which is the same constraint expressed as a rule the allocator obeys.

Stack guard

Windows / MSVC	Linux / GCC
`/GS` (the flag)	`-fstack-protector`, `-fstack-protector-strong`, `-fstack-protector-all`
"security cookie"	"stack canary"
`__security_cookie` (global)	`__stack_chk_guard`, read via `fs:0x28` (TLS)
`__security_check_cookie`	`__stack_chk_fail`
on by default	**opt-in**; distributions enable it by build policy

Also **SSP** — Stack Smashing Protector — which is the original ProPolice name and still appears in GCC documentation and distro hardening notes.

"Cookie" and "canary" mean the same thing. Which word someone uses tells you which platform they learned on.

Address randomisation

Name	What it actually is
ASLR	the OS behaviour: load things at unpredictable addresses
/DYNAMICBASE	the MSVC linker flag that opts a PE in
/HIGHENTROPYVA	asks for 64-bit entropy rather than 32-bit
PIE / `ET_DYN`	the ELF *mechanism* that lets the main executable move
kASLR	the same idea applied to the kernel image

The trap: **PIE and ASLR are not synonyms.** ASLR is the OS randomising what it can. A non-PIE ELF has a fixed load address for its own code no matter how enthusiastic the OS is, so libraries move and the binary does not. "ASLR is on" and "the binary is PIE" are separate facts and you need both.

Forward-edge control-flow integrity

All of these constrain where an indirect call or jump may land:

Name	Whose	How it shows up
CFG	Microsoft, user mode	guarded dispatch through `__guard_dispatch_icall_fptr`
kCFG	Microsoft, kernel mode	the same, in a driver
CET-IBT	Intel hardware	`endbr64` landing pads
BTI	ARM hardware	`bti` landing pads
CFI	clang `-fsanitize=cfi`	compiler-inserted type checks, a different design

"CFI" is ambiguous in conversation — it is both the general category and clang's specific feature. Ask which is meant.

Backward-edge

These protect the *return*, and they are genuinely different mechanisms rather than renames:

Name	Mechanism
Stack cookie / canary	a witness value that corruption must destroy
CET Shadow Stack	a second, protected copy of return addresses, compared on return
PAC (ARM)	the return address is signed; a tampered one fails authentication
SafeSEH / **SEHOP**	x86 Windows only, and about *exception handler* chains — not return addresses

SafeSEH is the one people file in the wrong drawer. It validates that an exception handler is on a list the binary declared; it has nothing to do with return addresses and does not exist on x64, where exception handling is table-driven rather than stack-linked.

Kernel names, for later
Module M3 covers these. The vocabulary is here so it is not new when you meet it:

Name	Also	What it stops
SMEP	PXN (ARM)	the kernel executing user-mapped pages
SMAP	PAN (ARM)	the kernel reading or writing user pages, except between `stac` and `clac`
KPTI	KVA Shadow (Windows), KAISER	the kernel being mapped while user code runs
kASLR	FG-KASLR when function-granular	a predictable kernel base
HVCI	Device Guard, memory integrity	unsigned code being made executable

The pairing to remember: **SMEP is execute, SMAP is access.** SMEP is why kernel exploitation stopped pointing function pointers at userland payloads and moved to reusing kernel code; SMAP is why it also stopped *reading* its data from userland.

Why this matters beyond trivia
Two practical consequences.

Reading other people's work. An advisory saying "requires a DEP bypass", an exploit comment saying "NX is on", and a `checksec` line saying `NX enabled` are three descriptions of one fact. If you index them as three, you will think you have three things to defeat.

Writing your own. A finding that says "mitigations: /GS" is ambiguous to a Linux reader and a finding that says "canary present" is ambiguous to a Windows one. Name the mechanism and then the flag: *"stack guard present (`/GS`, `__security_check_cookie` in the epilogue)"* reads correctly to both.

Module V01-stack-overflow

V1 — Stack buffer overflow (CWE-121)

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

The class in one sentence

A write into a fixed-size buffer in the current stack frame, where the number of bytes written is not constrained by the size of that buffer.

Everything else about it — which function was called, whether a count argument existed, which mitigation was on — is detail. The class is that one property.

The source shape

```
void vuln(const char *input) {
    char buf[16];
    strcpy(buf, input);
}
```

`strcpy` copies until it finds a NUL in the source. Nothing here says 16. The number of bytes written is decided entirely by the attacker's data, and past byte 16 it is writing over whatever the frame put above `buf`.

The bug is not "strcpy was called". The same class appears as `sprintf(buf, "%s", input)`, as `memcpy(buf, input, attacker_len)`, as `strncpy(buf, input, strlen(input))` — bounded, but by the wrong operand — and as `wcsncpy(buf, input, sizeof buf)`, where the operand is right and the *unit* is wrong. A rule about function names fails on the last two, and those are the two that survive code review.

The Windows signature — MSVC `/O2 /GS-`

Six instructions:

```
sub     rsp, 0x38
mov     rdx, rcx                ; src  (2nd arg register)
lea     rcx, [rsp+0x20]         ; dst = buf, at rsp+0x20
call    strcpy
add     rsp, 0x38
ret
```

What to read off it: a `lea` of a frame slot into the first argument register, then a copy call with no count register set up at all. `r8` is never touched, so there is no third argument, so there is no length.

The same function with `/GS`

```
sub     rsp, 0x48
mov     rax, qword ptr [__security_cookie]     ; a GLOBAL
xor     rax, rsp                               ; mixed with the frame address
mov     qword ptr [rsp+0x30], rax              ; stored above buf
mov     rdx, rcx
lea     rcx, [rsp+0x20]
call    strcpy                                 ; ... identical three lines ...
mov     rcx, qword ptr [rsp+0x30]
xor     rcx, rsp
call    __security_check_cookie
add     rsp, 0x48
ret
```

The copy is unchanged. Same `lea`, same `call`, same absence of a count. The overflow happens in both builds, byte for byte. What `/GS` adds is a value written above the buffer on entry and checked on exit, so that a forward overflow corrupts it before it reaches the return address and the process dies at the check instead of returning to an attacker-chosen address.

> **The cookie changes the impact, not the existence.** A cookied function is not a fixed function. It is the same finding with a different consequence.

Two further things the cookie tells you, both useful:

- **It is evidence a local array exists.** `/GS` only instruments functions that have one. So a cookie in an otherwise opaque function says *there is a buffer in this frame* — and its absence in a function you expected to have one says the array was optimised away.
- **It moved the frame.** `0x38` became `0x48` — sixteen bytes for an eight-byte cookie, the other eight being re-alignment. Mitigations change layout, and layout is exactly what a payload depends on.

The Linux signature — GCC `-O2 -fstack-protector-strong`

```
vuln:
        sub     rsp, 40
        mov     rsi, rdi
        mov     rax, QWORD PTR fs:40      ; canary read from TLS, not a global
        mov     QWORD PTR [rsp+24], rax
        xor     eax, eax
        mov     rdi, rsp                  ; buf is rsp+0
        call    strcpy
        mov     rax, QWORD PTR [rsp+24]
        sub     rax, QWORD PTR fs:40      ; compared inline
        jne     .L5
        add     rsp, 40
        ret
.L5:
        call    __stack_chk_fail          ; cold path, AFTER the ret
```

Same class, same bug, four differences worth knowing cold:

	MSVC `/GS`	GCC `-fstack-protector-strong`
where the secret lives	global `__security_cookie`	thread-local, `fs:40`
derivation	XORed with `rsp`	used raw
the check	`call __security_check_cookie`	inline `sub` + `jne`
failure path	inside the CRT helper	`__stack_chk_fail`, cold, after `ret`

And one difference that is not cosmetic:

	buf → guard
MSVC	**16 bytes** — `buf` at `rsp+0x20`, cookie at `rsp+0x30`
GCC	**24 bytes** — `buf` at `rsp+0`, canary at `rsp+24`

Same `char[16]`, same source file. GCC pads for alignment; MSVC did not need to. **That number is not derivable from the declaration**, and it is the number a payload has to get right.

It is also why a decompiler showing `char buf[24]` for the GCC build is not wrong — it reports the distance to the next object it recovered, which is not the same thing as the array's declared size.

One more, because it is not obvious

GCC emits **no** stack protector unless you ask for it. `gcc -v` shows no `--enable-default-ssp`, and `gcc -O2 -c` alone produces no canary at all. The "Linux has stack protection by default" belief comes from distributions applying `-fstack-protector-strong` through `dpkg-buildflags` when they build *packages* — build the same file yourself in a plain container and you get nothing.

So the accurate statement is: **MSVC `/GS` is on by default; GCC's is opt-in, and distributions turn it on by policy.** A cookie's presence is therefore evidence about how the binary was built, not only about what is in the frame.

Finding this class at scale

Recognising one unbounded copy is the drill. A real target has thousands of functions, and the question becomes *which ones deserve a human.*

The wrong first move is to grep for `strcpy`. Most calls to it in any real binary are correct, and the two variants that survive code review — `strncpy(dst, src, strlen(src))` and `wcsncpy(dst, src, sizeof dst)` — do not contain the string at all.

The rule that actually triages, and the one the module's code task asks you to implement:

```
for each function:
    find the copy call            (not the first call - strlen often precedes it)
    if the routine takes no count            -> UNBOUNDED, look now
    else trace the count register (r8/r8d):
        loaded from strlen of the source     -> SOURCE-BOUNDED, look now
        an immediate                         -> CONSTANT, needs a human
        from a caller-supplied register      -> CALLER-CONTROLLED, look
```

Artifact	Tool	Approach
Binary, headless	**Ghidra** `analyzeHeadless` + a script over `getFunctionManager().getFunctions(true)`	exactly the rule above; this is what the code task implements
Binary, interactive	Ghidra / IDA cross-references from the copy imports	fast for a first pass, poor for the strlen-bounded case
Source	**semgrep** or **CodeQL**	expresses the invariant structurally: a count derived from the source operand
Source, quick	`grep -n 'strn\?cpy|sprintf|memcpy'` then read	fine for a small tree, drowns on a large one

Two accelerators worth knowing:

- **The `/GS` cookie is a free filter.** MSVC only instruments functions that contain a local array, so `__security_check_cookie` in the epilogue narrows a binary to the frames that even *could* have this class. Combine it with a copy call and the candidate set collapses. *(It is a filter, not a finding — most cookied functions are fine.)*
- **`-Wformat-security` / `-Wstack-protector` on a source build** tell you what the compiler already suspects, for free.

What a script must not do is return "safe". A constant count of `32` looks bounded and is wrong by a factor of two in the `wcsncpy` variant. Triage narrows the field; the human decides.

What a finding needs

For this class, three things beyond the class name:

1. **The write and its destination's size** — the instruction, and how big the thing being written into actually is.
2. **What constrains the count** — nothing, something attacker-controlled, or something derived from the wrong operand. If the answer is "nothing", say so; an absent check has no instruction to point at.
3. **Which mitigations are present, and what each one changes** — a cookie changes impact, a non-executable stack changes what the payload can contain, ASLR changes what the payload can name. None of them change whether the bug is there.

Module V02-heap-overflow

V2 — Heap buffer overflow (CWE-122)

Every listing below is real compiler output. The build record is in `notes/build.txt`.

Same class, different neighbourhood

The property is V1′s: **a write into a fixed-size buffer, not constrained by that buffer's size.** What changes is where the buffer lives, and everything follows from that.

```
void vuln(const char *input) {
    char *buf = (char *)malloc(16);
    strcpy(buf, input);
    free(buf);
}
```

Compiled `/O2 /GS`:

```
mov     ecx, 0x10
call    malloc
mov     rdx, rdi
mov     rcx, rax
mov     rbx, rax
call    strcpy              ; unbounded, into a 16-byte chunk
mov     rcx, rbx
jmp     free                ; tail call
```

Start with what is missing

Compare that against V1′s stack version. Same `strcpy(buf, input)`. Same `/GS` flag. And here there is **no cookie** — no `__security_cookie` read, no `xor rax, rsp`, no `__security_check_cookie`.

`/GS` instruments functions containing a **local array**. `char *buf = malloc(16)` is a pointer; the array is not in the frame. There is nothing to protect and nothing is emitted.

This is module M's lesson made concrete in two adjacent listings: **the absence of a mitigation idiom is not evidence the build was unhardened.** Both functions came out of the same compiler with the same flag.

Where the overflow goes

On the stack, past the buffer lies the cookie and then the saved return address, and the epilogue checks the cookie on the way out.

On the heap, past the chunk lies whatever the allocator put there:

- **another live allocation** — you corrupt someone else's object, and nothing anywhere checks it; or
- **allocator metadata** — chunk headers, free-list pointers.

Two consequences worth holding onto:

1. **Nothing is checked on function exit.** The stack cookie fires at `ret`. A heap overflow is detected — if at all — much later, when the allocator next walks the corrupted structure. The crash, when it comes, is somewhere else entirely, in code that did nothing wrong.
2. **The adjacent-object path touches no metadata**, so every allocator hardening measure listed below is irrelevant to it.

Also: `jmp free`

That last instruction is a **tail call**. The epilogue restores registers, adjusts `rsp`, and jumps rather than calls, so `free` returns straight to `vuln`'s caller.

You will see this constantly in small heap wrappers at `/O2`. Read it as "and then calls `free` and returns", not as a jump into unrelated code — and do not scan for a `ret` to find the end of the function, because there is not one.

The shape with no instruction to find

```
size_t n = strlen(input);
char *buf = (char *)malloc(n);     /* should be n + 1 */
strcpy(buf, input);
```

```
call    strlen
mov     rcx, rax          ; n -> malloc's argument, unmodified
call    malloc
mov     rdx, rdi
mov     rcx, rax
mov     rbx, rax
call    strcpy            ; writes n+1 bytes
mov     rcx, rbx
jmp     free
```

There is no off-by-one instruction anywhere. No `inc`, no `lea rcx, [rax+1]`, no `add rcx, 1`. `strlen`'s result goes straight into `malloc`'s argument.

The defect is an instruction that **is not there**, and you can only see it if you know that `strcpy` writes `strlen(src) + 1` bytes — the terminator is not optional. One byte lands past the end of the chunk.

This is the reading V2 actually demands, and it is not V1's:

> **What size was allocated, what size will be written, and are the two derived from the same expression?**

There is no bounds check to look for here, because there was never going to be one. Two calls, two arguments, and a relationship between them that is off by exactly one.

"Only one byte" is not a severity downgrade you can apply without analysis — single-byte heap overflows into metadata have a long history of being fully exploitable.

Heap hardening
Same table as V3, and the same caveat. None of it is compiler-emitted, so **none of it appears in a listing**.

glibc	Windows
chunk size / `prev_size` consistency checks on free	encoded `_HEAP_ENTRY` headers, so a forged header fails its checksum
tcache, and safe unlinking in the large-bin path	LFH randomisation
	the segment heap

These raise the cost of **weaponising metadata corruption**. They do not bound the copy, and — the part people miss — they do nothing at all about an overflow into an **adjacent live allocation**, because that path never touches metadata.

So a heap overflow that lands on another object is invisible to all of it.

Finding this class at scale
V1's triage rule asks one question of one call: *where did the count come from?* On the stack that is enough, because `char buf[16]` is a **declaration** — the destination's size is a fact of the source and cannot be wrong.

On the heap the size is **computed at runtime**, so the same copy is safe or unsafe depending on a `malloc` that may be twenty lines away. The unit of analysis grows from the call to **the pair**.

The rule, and the one the module's code task asks you to implement:

```
for each function:
    for each allocator call:                (malloc/calloc/realloc/HeapAlloc)
        size  <- the immediate last moved into ecx/rcx before it
        chunk <- rax, and any register later copied from it
    for each copy call:
        dest  <- whichever allocation rcx is carrying AT THE CALL
        if the routine takes no count           -> UNBOUNDED, look now
        else trace r8/r8d backwards:
            an immediate that fits `size`       -> BOUNDED
            anything else                       -> UNRELATED, look now
```

Do not pair by proximity. "The nearest allocation above the copy" is right for every single-buffer function and wrong the moment a routine allocates two buffers before filling either — it names the wrong chunk *and* clears the one that overflows. Two wrong answers from one shortcut, in a report that gets trusted. Follow the register instead.

Artifact	Tool	Approach
Binary, headless	**Ghidra** `analyzeHeadless` + a script over the function manager	the rule above; this is what the code task implements
Binary, better	Ghidra **P-Code / decompiler API** — `getHighFunction()`, then varnode def-use	the register tracking comes free; worth it once the copies stop being adjacent
Binary, interactive	xrefs from the allocator imports, *not* from the copy imports	the allocation is the scarcer end of the pair and the better starting point
Source	**CodeQL** dataflow: source = an allocation size expression, sink = a copy count	expresses "these two are unrelated" directly, which grep cannot
Source, quick	`grep -n 'malloc|calloc' -A6` and read forward	the `+ 1` cases fall out immediately

Three accelerators worth knowing:

- **`malloc(strlen(...))` is worth a dedicated grep of its own.** The whole `undersized` variant is one missing `+ 1`, it is textual in source, and it is the variant that survives review because the code looks responsible.
- **The `/GS` filter from V1 inverts here.** A heap buffer is not a local array, so a vulnerable function often has **no cookie at all**. Absence of `__security_check_cookie` beside a copy call is weak evidence the destination is *not* on the stack.
- **At `/O2`, look for a constant-offset byte store right after a `movups`.** The vectoriser cannot fold an odd trailing byte into a wide move, so a one-past-the-end write becomes its own instruction with the overrun offset written into it. See `vr-v2-listing-var-loop-vectorised`.

What a script must not do is return "safe". `BOUNDED` here means only *the count is an immediate that fits* — it says nothing about whether the allocation was the right size in the first place, which is exactly the `undersized` variant. Triage narrows the field; the human decides.

What a finding needs

1. **The allocation** — its size, and the expression that produced it.
2. **The write** — its size, and the expression that produced *that*.
3. **The relationship between them.** For the size-mismatch shape this *is* the finding: `malloc(strlen(s))` against a copy that writes `strlen(s) + 1`.
4. **What is adjacent**, if you can tell — another allocation of a known type is a materially different impact from allocator metadata.
5. **Which hardening is in play**, and explicitly whether the overflow path even interacts with it.

Module V03-use-after-free

V3 — Use-after-free (CWE-416)
Every listing below is real compiler output. The build record — sources, compiler banner, exact flags, object SHA-256 — is in `notes/build.txt`.

The class, and why it is different from V1

> A pointer is used after the memory it refers to has been released.

V1's stack overflow is **spatial**: there is a write, it has a destination of a knowable size, and there is a count you can go and inspect. You can point at the instruction.

This one is **temporal**, and that changes how you find it:

- There is **no instruction that marks memory as freed.** `free` returns nothing and changes nothing you can see in the caller.
- The bug is not in any single instruction. It is a *relationship* between one call and a later use.
- Nothing is wrong at the point of the free, and nothing is wrong at the point of the use. Only the pair is wrong.

So there is no idiom to recognise. What you look for instead is **pointer survival**.

The signature: a pointer that outlives the call that killed it

```
mov     ecx, 0x20
call    malloc
mov     rcx, rax
mov     rbx, rax          ; <-- copied into a callee-saved register
call    free              ; <-- ...so it survives this
mov     r8d, 0x1F
mov     rdx, rdi
mov     rcx, rbx          ; <-- and comes back
call    strncpy           ; <-- written into after being freed
mov     byte ptr [rbx+0x1F], 0
```

Read it as a lifetime question rather than as an idiom:

1. What was passed to `free`? The value in `rcx`, which came from `rax`.
2. Is that value still live afterwards? Yes — a copy is in `rbx`.
3. Is it used? Yes, at the `strncpy` and again at the byte store.

The `mov rbx, rax` is the whole tell. A pointer used only *before* the free can live in a volatile register; one used *after* it has to survive a call, so the compiler puts it in a callee-saved register and the prologue saves it.

A very small function that saves `rbx`, `rdi` or `rsi` around a call to `free` is worth a second look. That is not proof — plenty of functions have other reasons to keep a value live — but it is a cheap filter.

The harder shape: the use is on a path you did not walk

```
        ...strncpy into the buffer, held in rdi...
        cmp     byte ptr [rdi], sil
        je      cleanup                 ; empty input skips the strlen
        mov     rcx, rdi
        call    strlen
        mov     rsi, rax                ; rc = length
cleanup:
        mov     rcx, rdi
        call    free                    ; release
        test    esi, esi
        jne     skip                    ; rc != 0 -> nothing more to do
        mov     rdx, rdi                ; <-- freed pointer as printf's %s
        lea     rcx, [rejected_fmt]
        call    printf
skip:
```

Allocate, use, free — correct, on the common path. The use-after-free is reached only when `rc` is zero, which happens only when the input was empty and the function took the reject path.

This is the realistic case. Error paths, cleanup paths and early returns are where this class lives, because they are the paths that get written last, tested least, and read past quickest. A reader tracing the happy path sees a clean allocate-use-free and moves on.

The reading habit that catches it: after you find a `free`, **check every path that reaches the code below it**, not just the one you arrived on.

Where it goes next

The reason this class matters out of proportion to how it looks: the freed chunk gets **reused**. Whatever the allocator hands out next lands on top of the dangling pointer, and if an attacker controls that allocation's contents then they control what the stale pointer now sees.

When the freed object contained a **function pointer** or a **vtable pointer**, the dangling read becomes an indirect call to an attacker-chosen address. That is the bridge from a memory-lifetime bug to control-flow hijack, and it is why this class and V8 (type confusion) shade into each other.

You have already seen the result of that chain — the indirect call in module M's CFG panes is exactly this, one variant further along.

Heap hardening, and what it does not do

Allocators defend themselves. None of it is compiler-emitted, so **none of it appears in a listing** — it belongs in the header/environment bucket from module M.

glibc	Windows
tcache key — a per-chunk marker checked on free, which catches most naive double frees	**LFH randomisation** — small allocations are not handed back in a predictable order
safe-linking — free-list pointers XORed with the chunk address `>> 12`, so forging one needs a heap address	**encoded `_HEAP_ENTRY`** headers
safe unlinking — consistency checks in the large-bin path	**segment heap**, and delayed free for some classes

Read the column headings carefully: every one of these targets the **reuse** step. None of them stops the pointer from dangling, and none of them makes the `strncpy` above safe. They make it harder to *control* what lands in the freed chunk, which raises the cost of exploitation without changing the defect.

So the finding is unchanged. The impact line records which of these is in play.

An asymmetry worth knowing

This class survives optimisation better than V1 does.

Two of BugMuseum's six stack-overflow variants compile to a bare `ret` at `/O2`, because the destination is a local nothing reads and the optimiser removes the write. Every use-after-free variant checked survives intact — `malloc` and `free` have observable side effects the compiler must preserve, so it can neither delete the allocation nor move a write across the release.

Temporal bugs are therefore more reliably present in a release build than dead spatial writes are. That is the opposite of most people's intuition about optimisation hiding bugs.

Finding this class at scale

V1 and V2 are properties of an instruction and of a pair of calls. Neither transfers, because **there is no instruction to match here**. The load at `mov rdx, QWORD PTR buf$[rsp]` is byte-identical to the five legitimate loads above it, and the only thing that makes it a finding is that it sits below the free. You are looking for an **ordering**, not a shape.

The rule, and the one the module's code task asks you to implement:

```
for each function:
    for each release call:                   (free/HeapFree/_aligned_free)
        home <- what last wrote rcx before the call
        if it was not a frame slot -> UNKNOWN-ARG, say so
        scan forward from the release:
            a WRITE to home        -> stop; the pointer was replaced
            a READ  of home        -> USE AFTER FREE, report the line
```

Say `no-finding`, never `clean`. This rule follows one slot, so it is blind to three of the five variants in the drill — `alias`, `owner` and `moved`. Which variants your tool cannot see is a fact about the tool, and it belongs in the output.

Stop at a write to the slot. `buf = NULL` after `free(buf)` is *the* fix for this class. A pass that does not recognise it flags every corrected function, and a pass that flags the fix gets ignored — which costs you the true positives.

Artifact	Tool	Approach
Binary, headless	**Ghidra** `analyzeHeadless` + a script over the function manager	the rule above; text over frame slots, which works at `/Od`
Binary, `/O2`	Ghidra **P-Code**: `getHighFunction()`, then the varnode def-use chain	at `/O2` the pointer is in a callee-saved register and there is no slot to follow — def-use is the only way
Binary, dynamic	**Application Verifier** / **Full Page Heap** (Win), **ASan** or **Valgrind** (Linux)	turns a probabilistic bug into a deterministic fault at the moment of use
Source	**CodeQL** — `free` as a source, a dereference of the same SSA value as a sink	the only tool here that follows an alias across names
Source, quick	`grep -n 'free(' -A8` and read forward for the same identifier	finds the baseline shape only, which is the one that does not ship

Three accelerators worth knowing:

- **A debug allocator is worth more here than in any other class.** Full Page Heap unmaps the freed page, so the use faults *at the use* instead of corrupting silently and crashing somewhere unrelated later. It also forces `realloc` to always relocate, which makes the `moved` variant deterministic.
- **Grep the release for what follows it, not what precedes it.** Every other class is found by reading up from the dangerous call. This one is found by reading down from a harmless one.
- **An indirect call guarded by a compare against a known function is a devirtualisation guess, not a safety check.** `cmp rax, <known fn>` / `jne` / `call rax` means the compiler predicted the pointer's value and provided a fallback — and the fallback is reached exactly when the value has been changed. See `vr-v3-bytes-to-control`.

What a script must not do is decide. It establishes that a slot was read after a release; whether that is reachable, and with what, is the human's job.

What a finding needs

1. **The release** — which call frees it, and what value was passed.
2. **The use** — which later instruction touches the same memory, and on which path it is reached.

3. **The reachability** — what input drives execution down that path. For the error-path shape this is the whole finding; "reachable only when the input is empty" is a materially different claim from "reachable always".
4. **What lands in between** — whether anything allocates between the free and the use, since that is what decides whether the stale data is attacker controlled or merely stale.

Module V04-double-free

V4 — Double free (CWE-415)

Every listing below is real compiler output. The build record is in `notes/build.txt`.

The class

> The same chunk is released twice.

Not "`free` is called twice on the same variable" — that is one shape of it, and the least interesting. The invariant is about the **chunk**, across however much time and however many aliases that takes.

V3's use-after-free had a dangling pointer and a *use*. Here the second use is `free` itself, and the victim is the **allocator**, not your data.

The baseline shape

```
call    malloc
call    strncpy
movzx   ebx, byte ptr [rdi]    ; a legitimate read, before any free
mov     rcx, rdi
call    free
mov     rcx, rdi               ; same register
call    free                   ; same value, again
```

Two `mov rcx, rdi` / `call free` pairs, adjacent, with nothing between them that could have changed `rdi`. Four instructions and the whole finding is in them.

If every double free looked like this, the class would need a paragraph rather than a module.

The shape that is not in the function

```
call    malloc
call    strncpy
mov     rcx, qword ptr [0x140074C40]   ; load a GLOBAL
call    free                            ; free whatever it held
mov     rcx, rdi
mov     qword ptr [0x140074C40], rdi    ; store THIS buffer into the global
call    free                            ; and free it here too
```

Read the frees: the first releases the *global's previous* value, the second releases *this* invocation's buffer. **Two different pointers.** Neither call is wrong, and this function never frees the same chunk twice.

The defect is what the function **leaves behind**. On return, the global holds a pointer to a chunk that was just freed on the line below the store. The next invocation opens with `free(g_last)` — and releases that chunk a second time.

```
call 1:  free(g_last=NULL)  g_last = A   free(A)      -> g_last dangles at A
call 2:  free(g_last=A)     <-- A freed a second time
```

No single execution of this function contains the bug. It is a property of the function's effect on global state, observed across invocations.

Two things to take from it:

- **The store sits between the two frees**, and it stores the pointer the second `free` is about to release. A reader who sees `g_last = buf` and reads it as "ownership transferred, so it is safe now" has the sequence backwards — the transfer happens and then the transferring function frees it anyway.
- **Global and static state is where this class hides.** A cache of the last buffer, a singleton, a linked list, a handle table: anything that keeps a pointer past the end of the function that allocated it.

The reading habit: when a function stores a pointer somewhere that outlives it, ask **who frees it, and does that owner know about this one.**

What actually breaks

The second `free` hands the allocator a chunk that is already on a free list. Depending on the allocator and the state, that can produce:

- the same chunk appearing **twice on a free list**, so two later `malloc` calls return the *same* address, and two parts of the program now hold what each thinks is a private buffer;
- **corrupted list pointers**, which is the classic route to an arbitrary write.

The first outcome is the one worth understanding, because it turns a lifetime bug into a type-confusion primitive without any memory corruption at all: two correct-looking objects of different types occupying one chunk.

Detection, and how it differs from V2

Unlike heap overflow, this class has **real runtime detection**:

glibc	Windows
the **tcache key** — a per-chunk marker written on free and checked on the next one, which catches the naive adjacent case reliably	the debug CRT detects it
"double free or corruption" checks on the fastbin path	the segment heap fast-fails on some patterns; Page Heap catches it deliberately

And the detection is **usefully placed**: it fires on the second `free`, so the process dies at a point that names the bug fairly precisely. Compare V2, where the crash is displaced into unrelated code minutes later.

But note what that depends on. The tcache key catches `free(p); free(p);` because the chunk is still sitting in the cache with its marker intact. The alias shape's two frees are separated by a whole invocation and arbitrary allocator activity — the chunk may have been handed out and freed again legitimately in between, and the marker is long gone.

So "modern allocators catch double frees" is true of the shape that is easy to find anyway, and least true of the shape that is hard to find.

What a finding needs

1. **Both frees** — where each is, and what each was given.
2. **The proof they are the same chunk.** For the baseline this is register identity. For the alias shape it is an argument about global state across invocations, and it has to be written out.
3. **The window** — what runs between them. Adjacent frees are a crash; frees separated by attacker-influenced allocation are a primitive.
4. **Whether detection would fire**, and honestly. "The allocator would catch this" is a claim about a specific allocator in a specific state, not a general property.

Module V05-integer

V5 — Integer defects (CWE-190 / 191 / 195 / 197)

Every listing below is real compiler output. The build record is in `notes/build.txt`.

Why this class is the productive one

Every defect in this module is **one instruction** away from correct. Not a missing block, not an absent check with a hole where it should be — one mnemonic suffix, or one register width.

And none of them is the bug you will see crash. Each produces a *downstream* heap overflow, so the crash lands at the `memcpy` and the analysis stops there. **The copy is where the crash is; the arithmetic is where the bug is.**

1. The multiply that wraps

```
unsigned int total = count * size;
void *p = malloc(total);
```

```
imul    ecx, edx          ; product straight into malloc's argument
call    malloc
```

`count = 0x10000000`, `size = 0x10` → the product is `0x100000000`, which does not fit in 32 bits, so `total` is **0**. `malloc(0)` returns a valid, minimal allocation, and the caller proceeds to fill it with what it believes is 256 MB of data.

Now the fixed version:

```
test    edx, edx
je      skip
xor     edx, edx
mov     eax, -1
div     r8d               ; 0xFFFFFFFF / size
cmp     ecx, eax
jbe     ok
```

A `div` in front of a `malloc` is an overflow guard.

Division is expensive and compilers avoid it. When one appears immediately before an allocation, dividing a maximum value by one operand and comparing the other against the quotient, it is almost always this check. Learn it as a positive tell — **its absence next to an `imul` feeding `malloc` is the finding.**

One caution: `imul` with no `jo` or `jc` after it is *not* by itself a finding. Unsigned wraparound is defined behaviour and compilers never emit overflow branches for it. The finding is the pair — a wrapping product reaching an allocation size with no range check anywhere ahead of it.

2. The signed compare

```
if (n > 64)                          /* n is int */
    return -1;
memcpy(dst, src, (unsigned int)n);
```

```
cmp     edx, 64
jle     ok              ; SIGNED
```

versus the fix:

```
cmp     edx, 64
jbe     ok              ; UNSIGNED
```

One mnemonic. `cmp` is identical in both — it sets the same flags whatever the operands were meant to represent, and only the branch decides how to read them.

`jle` is jump-if-less-or-equal, signed. Pass `n = -1` and the check passes happily. Then the cast to `unsigned int` turns it into `0xFFFFFFFF`, and the `memcpy` is asked for four gigabytes.

The reading habit, and it is the highest-value one in this module:

> **A `cmp` is never the evidence. The branch is.**

`jl` / `jle` / `jg` / `jge` are signed. `jb` / `jbe` / `ja` / `jae` are unsigned. When a length is compared against a bound, check which family the branch belongs to and whether that matches how the value is used afterwards.

3. The truncation

```
unsigned int len = (unsigned int)n;   /* n is unsigned long long */
unsigned char *p = malloc(len);
memcpy(p, src, n);                    /* the FULL 64-bit n */
```

```
mov     rbx, rdx        ; n kept, full 64 bits
mov     ecx, edx        ; 32-bit view -> malloc's size
call    malloc
mov     r8,  rbx        ; 64-bit view -> memcpy's count
call    memcpy
```

Two instructions, two register widths, one value. `n = 0x1_0000_0040` allocates `0x40` bytes and copies `0x1_0000_0040` of them.

Neither instruction is wrong on its own. `mov ecx, edx` is what a cast to `unsigned int` compiles to; `mov r8, rbx` is what passing a `size_t` compiles to. **The defect is that both describe the same value.**

The tell: **a value used at two different widths across two calls.** On x86-64 this is visible in the register names — `ecx` against `rbx`, `r8d` against `r8`.

There is a subtler variant of the same mistake: check the 64-bit value, then pass a truncated one to the copy. That direction is safe for the copy and merely misreports a length — which is worth knowing, because it means truncation alone is not automatically a memory-safety bug. **Which value is truncated, and where it is used, decides.**

The three tells together

Sub-class	What to look for
Multiply wrap	`imul` feeding `malloc`, and **no `div` guard** ahead of it
Signedness	a **`jl` / `jle` / `jg` / `jge`** branch on a length that is later used unsigned
Truncation	one value reaching two calls at **two register widths**

None is a missing block of code. All three are things you read off the instruction stream in a second once you know what you are looking at — which is exactly why this class rewards drilling and punishes skimming.

What a finding needs

1. **The arithmetic** — the exact instruction, and what the operands are.
2. **The value that defeats it** — `count = 0x10000000, size = 0x10`, or `n = -1`, or `n = 0x1_0000_0040`. A concrete triggering value, not a class of them.
3. **Where it lands** — which allocation is undersized, which copy overruns it.
4. **Where the fix belongs** — before the arithmetic, not at the copy. A bounds check at the `memcpy` treats the symptom.
5. **Not the crash site.** The crash will be the copy. Say so explicitly, so the next reader does not re-triage it as a plain overflow.

Module V06-uninitialised

V6 — Uninitialised memory and information disclosure (CWE-457 / CWE-200)
Every listing below is real compiler output. The build record is in `notes/build.txt`.

The class

> Memory is sent somewhere observable before all of it has been written.

Every other class in this course is about a **write** going wrong. This one is a **read** — of bytes nobody put there — and it discloses whatever the previous occupant of that memory left behind: pointers, keys, other users' data.

There is no corruption. Nothing crashes. Nothing is detected.

Shape 1: the padding you did not write, because you cannot see it

```
struct Record {
    uint32_t id;      /* +0  */
    uint8_t  kind;    /* +4, then 3 bytes of padding */
    uint32_t value;   /* +8  */
};

r.id = id;  r.kind = kind;  r.value = value;
emit(&r, sizeof r);
```

Every field is assigned. The struct is fully initialised as far as the source is concerned. And `sizeof r` is **12**, not 9 — the compiler inserted three bytes after `kind` so that `value` lands on a 4-byte boundary.

```
mov     dword ptr r$[rsp],   ecx     ; +0, 4 bytes
mov     byte  ptr r$[rsp+4], dl      ; +4, 1 byte
mov     dword ptr r$[rsp+8], r8d     ; +8, 4 bytes
```

```
mov     edx, 12
call    emit                            ; 12 bytes emitted
```

Count the stores: 4 + 1 + 4 = 9 bytes written, 12 emitted. Offsets +5, +6 and +7 are never touched. They hold whatever the last function to use that stack region left there.

This is the shape worth the module. The source names no padding, so no amount of reading the C reveals it — you find it by counting bytes in the listing, or by knowing the struct's layout rules cold.

Shape 2: the tail you did not fill

```
char buf[64];
if (n > 64) n = 64;
memcpy(buf, name, n);
emit(buf, sizeof buf);     /* all 64 */
```

```
call    memcpy
mov     edx, 64             ; CONSTANT: the buffer's size
call    emit
```

versus the fix:

```
mov     edx, ebx            ; the clamped length that was actually copied
call    emit
```

One instruction. **Is the length passed to the output a constant, or a value derived from what was written?** A constant equal to the buffer's declared size, following a partial fill, is the finding.

This one *is* visible in the source — `sizeof buf` against `n` — so it is the easier sibling. Worth drilling because the listing version is so mechanical: find the output call, look at what is in the length register.

The fix, and how not to look for it

```
xor     eax, eax
mov     dword ptr r$[rsp+8], r8d
mov     qword ptr r$[rsp],   rax    ; 8-byte ZERO over +0..+7
mov     dword ptr r$[rsp],   ecx
mov     byte  ptr r$[rsp+4], dl
```

That is `memset(&r, 0, sizeof r)` — and **there is no call to `memset`.** At `/O2` a small one is folded into one or two wide stores.

> Do not look for `call memset`. Look for a **wide zero store covering the whole object, before the fields are written.**

A reader searching for the call will conclude the fixed version is also vulnerable.

The mitigation: `-ftrivial-auto-var-init`

GCC, same function, with and without `-ftrivial-auto-var-init=zero`:

```
        sub     rsp, 24
        mov     qword ptr [rsp+4], 0     ; <-- the only added instruction
        mov     dword ptr [rsp+4], edi
```

One instruction. The compiler zeroes the whole local before anything else runs, so the padding is defined even though the program never assigns it. MSVC's equivalent is `/d1initall`.

Note what this does and does not do:

- It **removes this class wholesale** for stack locals, which is unusual — most mitigations in this course change impact rather than existence.
- It does **nothing for heap allocations.** `malloc` returns uninitialised memory and no compiler flag changes that.
- It is **not on by default** in either toolchain, so its absence tells you nothing about the build's hardening in general.

Two things about the `/GS` cookie here

Both MSVC specimens carry one, because both declare a local aggregate. It is **not related to the defect**: the cookie protects the return address, and this bug reads memory already inside the frame and hands it out legitimately.

Do not read a cookie as relevant just because it is present. Module M's rule again — an idiom is evidence only about the thing it is for.

Why the severity is not what it looks like

A userland padding leak discloses three bytes of the same process's stack. Easy to rate low.

The same defect in a kernel driver's ioctl output buffer discloses three bytes of **kernel** stack to an unprivileged caller — and **a single kernel-address leak defeats kASLR for the entire image.** An information disclosure is often the enabling half of a chain whose other half is a memory-corruption bug that needed an address.

So this class is rated by **what the disclosed bytes are next to**, not by how many of them there are.

What a finding needs

1. **How many bytes are unwritten, and exactly which offsets.** "+5 through +7" is actionable; "some padding" is not.
2. **What reaches the output** — the call, and the length it was given.
3. **What previously occupied that memory**, if you can establish it. Three bytes of stack that reliably hold part of a pointer is a different finding from three bytes of noise.
4. **Whether it is a padding leak or a length mistake** — they have different fixes and different review stories.
5. **The trust boundary the bytes cross.** Same process, process to process, or kernel to user. That is what sets the severity, and it is the part a reader cannot reconstruct from the listing.

Module V07-format-string

V7 — Format string (CWE-134)

Every listing below is real compiler output. The build record is in `notes/build.txt`.

The class

> A format string an attacker controls is an **interpreter** they control.

Every other class in this course needs a length to be wrong, or a lifetime, or an arithmetic result. This one needs nothing to be wrong at all — the code calls a standard function correctly, with the arguments the programmer intended, and the *data* decides what happens.

The whole vulnerable function

```
void log_vuln(const char *msg) { printf(msg); }
```

```
jmp     printf
```

That is the entire function. One instruction.

The caller put `msg` in `rcx`, `printf`'s first parameter *is* `rcx`, so there was nothing to do but jump.

Now the correct version:

```
void log_fixed(const char *msg) { printf("%s", msg); }
```

```
mov     rdx, rcx                                ; msg -> 2nd argument
lea     rcx, OFFSET FLAT:??_C@_02DKCKIIND@?$CFs@        ; "%s" -> 1st argument
jmp     printf
```

Two extra instructions. The decorated symbol is MSVC's name for a string constant, and the object's `CONST` segment confirms what it holds:

```
??_C@_02DKCKIIND@?$CFs@ DB '%s', 00H
```

> **The vulnerable version is shorter.** That inverts the usual intuition — you are not looking for something extra and wrong, you are looking for something missing and ordinary. A one-instruction function reads as a thin wrapper, which is exactly why this shape gets skimmed past.

The tell

The format argument's register receives a `lea` of a read-only string constant in the correct version, and a caller-supplied pointer in the vulnerable one.

As a procedure:

1. **Identify which parameter is the format.** First for `printf`, second for `fprintf`, third for `snprintf` and `sprintf_s`, fourth for the `vsnprintf` family after the buffer and count.
2. **Find what puts a value in that register.**
3. A `lea` of an `.rdata` symbol is correct. Anything tracing back to a caller-supplied pointer is the finding.

Note step 1 carefully — the register is *not* fixed. Here is the same mistake in a wrapper, where the format is the third parameter:

```
; vulnerable
mov     rdi, rcx
mov     edx, edx
call    snprintf                ; r8 still holds the caller's `user` pointer

; correct
mov     r9, r8                                          ; user -> 4th argument
lea     r8, OFFSET FLAT:??_C@_02DKCKIIND@?$CFs@         ; "%s" -> 3rd
call    snprintf
```

Same tell, different register. Teaching "check `rcx`" would be wrong for half the family — and under System V it is `rdi` and `rdx` instead.

And do **not** look for a missing `"%s"`. There is no instruction for an absent format specifier.

(`mov edx, edx` in both wrapper builds is a zero-extension of the 32-bit `cap` into a 64-bit `size_t` parameter. Not part of the defect.)

What the attacker gets

The format string is a small language, and controlling it means executing it:

Specifier	Effect
`%x`, `%p`	walk the argument registers, then the stack — **disclosure** of whatever is there
`%s`	dereference one of those values as a pointer — **disclosure of arbitrary memory**, or a crash
`%n`	**write** the number of bytes emitted so far to an address from the argument list

`%n` is the one that turns a disclosure into an **arbitrary write**, and with width specifiers the value written is controllable too.

The disclosure half is the part to take seriously even where `%n` is unavailable. Stack contents from a printf-family call regularly include return addresses and frame pointers — and a leaked code address defeats ASLR, which is often the missing half of some other chain.

The mitigations, stated precisely

- **MSVC's CRT disables `%n` by default.** Re-enabling it takes an explicit `_set_printf_count_output` call.
- **glibc with `_FORTIFY_SOURCE` rejects `%n`** when the format string lives in writable memory.
- **`-Wformat-security` / `-Wformat-nonliteral`** catch this at compile time — which is why the class is much rarer in code that is built with warnings enabled and treated as errors.

None of that makes the class harmless. Both runtime mitigations target `%n` only, both are per-runtime rather than universal, and neither touches the disclosure primitive.

What a finding needs

1. **The call, and which argument is the format** — name the function and the position, since the position varies.
2. **The provenance of that pointer** — how attacker data reaches it.
3. **Which primitives are available**: disclosure always; arbitrary write only if `%n` is enabled in the target's runtime. Check rather than assume.
4. **What the format-call's stack looks like**, if you can say — it decides what `%x` and `%s` actually reach.

Module V08-type-confusion

V8 — Type confusion (CWE-843)

Every listing below is real compiler output. The build record is in `notes/build.txt`.

The class

> Memory is interpreted as one type when it holds another.

Nothing is out of bounds. Nothing is freed. Every read is inside a live, valid object — the object is simply **not the type the code thinks it is**, so a field read at a given offset returns something that was put there for another purpose.

The mechanism is almost always a **discriminant that was not consulted**: a tag, a length, a version field, a vtable pointer. Something in the data says what the data is, and the code did not ask.

Shape 1: the tagged union

```
struct Tagged {
    uint32_t kind;                /* +0 */
    union {
        uint64_t number;          /* +8 */
        void (*handler)(void);    /* +8 */
    } u;
};

void dispatch_vuln(struct Tagged *t) { t->u.handler(); }
```

```
rex_jmp QWORD PTR [rcx+8]
```

One instruction. A tail call through whatever is at offset +8. The tag at +0 is never loaded, never compared, never branched on.

If `kind` is `KIND_NUMBER`, that quadword is an integer some other part of the program stored — and the program calls it.

The correct version:

```
cmp     dword ptr [rcx], 1        ; kind == KIND_HANDLER?
jne     skip
rex_jmp qword ptr [rcx+8]
skip:
ret
```

(`rex_jmp` is MSVC's listing spelling for a REX-prefixed indirect jump — a tail call through a function pointer. Bytes `48 ff 61 08`.)

Shape 2: the downcast

```
struct Header { uint32_t type; uint32_t len; };                    /* 8 bytes */
struct Record { uint32_t type; uint32_t len; uint8_t body[64]; };  /* 72 */

const struct Record *r = (const struct Record *)h;
emit(r->body, 64);
```

```
add     rcx, 8
mov     edx, 64
jmp     emit
```

Sixty-four bytes read from offset +8 of an object whose declared type is eight bytes long. Whatever follows the header in memory is emitted.

The correct version needs **two** guards, and this is the part worth remembering:

```
cmp     dword ptr [rcx], 7        ; the type tag
jne     skip
cmp     dword ptr [rcx+4], 72     ; the length: 0x48 == sizeof(Record)
jb      skip
add     rcx, 8
mov     edx, 64
jmp     emit
```

Either guard alone is insufficient. Checking the type without the length still trusts an object that claims to be a Record and was truncated. Checking the length without the type still reinterprets the wrong structure at the right size.

The tell

A field read at an offset belonging to one type, with no compare of the discriminant before it.

The positive form is easier to search for, and it is the one to learn:

> a `cmp` of a field at a small offset against a small constant, followed by a conditional branch that guards the rest of the function.

That is a tag check. Its absence in front of a union-member access or a downcast is the finding.

Two things worth noticing

The vulnerable function is shorter again. One instruction against four. That also happened in V7 — and it is worth naming *why*, not just noticing it: when the defect is a **missing check**, the buggy code is the correct code with instructions removed.

Do not generalise it further than that. V1′s overflow pair are the same length, and V5′s fixed multiply is longer only because a `div` was added.

This is where V8 and module M meet. `rex_jmp qword ptr [rcx+8]` is exactly the indirect call that CFG guards. Enabling CFG changes that instruction to a guarded dispatch — and **does nothing to restore the tag check**. A `kind` value that happens to be a valid indirect-call target still passes. Forward-edge CFI narrows the target set; it does not tell you the object was the wrong type.

And a class boundary worth being precise about

BugMuseum's `variant-07-type-confusion-reuse` — the specimen behind module M's CFG panes — reaches an indirect call through a **reused freed chunk**. That is a **V3** bug (use-after-free) whose *consequence* is type confusion.

The difference matters for the finding:

- **V3 → confusion:** the object's lifetime ended. The fix is the lifetime.
- **V8 proper:** the object is valid and correctly allocated. The fix is the missing check.

Reporting one as the other sends the fix to the wrong place.

*(Incidentally, `struct Tagged` puts the union at +8 rather than +4, because a `uint64_t` member forces 8-byte alignment. The four bytes at +4 are padding — so this declaration is also a **V6** specimen the moment anything emits it whole.)*

What a finding needs

1. **The discriminant** — which field says what the object is, and where it is.
2. **The access that assumed** — which offset was read, as which type.
3. **What the two interpretations disagree about** — an integer read as a function pointer is a control-flow finding; a short header read as a long record is a disclosure or an out-of-bounds read.

4. **Who controls the discriminant.** If an attacker sets `kind`, this is directly reachable. If it is set internally and confused by some other bug, the finding is that other bug.
5. **Whether it is this class or a lifetime bug ending here** — see above.

Module V09-double-fetch

V9 — Double fetch / TOCTOU (CWE-367)

Every listing below is real compiler output. The build record is in `notes/build.txt`.

The class

> A value is read once to be checked, and read again to be used. Between the two reads, someone else changes it.

The check passes on the first value. The operation runs on the second. Both reads are of the same address; neither is wrong on its own.

```
if (*user > 0x40)                              /* first read */
    return -1;
memcpy(dst, (const void *)(user + 1), *user);  /* second read */
```

```
mov     eax, dword ptr [rcx]      ; read 1 - for the check
cmp     eax, 64
jbe     ok
...
ok:
mov     r8d, dword ptr [rcx]      ; read 2 - for the copy
lea     rdx, qword ptr [rcx+4]
call    memcpy
```

The fix captures once:

```
mov     eax, dword ptr [rcx]      ; the only read
cmp     eax, 64
jbe     ok
...
ok:
mov     r8, rax                   ; reuses the value that was checked
call    memcpy
```

The tell is countable

Two loads from the same address — one before the guard, one after. One load means the checked value is the used value.

That is the whole technique, and it is the same shape as V6's: don't look for a wrong instruction, count something and compare.

The finding that makes this module worth having

Here is `fetch_novol` — the vulnerable function with the `volatile` qualifier removed and **nothing else changed**. The source still reads the location twice:

```
mov     eax, dword ptr [rcx]      ; the only read
cmp     eax, 64
jbe     ok
...
ok:
mov     r8, rax
call    memcpy
```

One load. It is byte-identical to the fixed version.

The compiler is entitled to this. Without `volatile` there is nothing to suggest the memory can change between the two reads, so caching the value in `rax` is an obvious optimisation.

Three consequences, and they are the reason this class is not simply "reading twice is bad":

1. **The class is only visible when something forces both reads** — `volatile`, an intervening call the compiler cannot see through, or an explicit barrier.
2. **A source-level double fetch the optimiser removed is not exploitable in that build.** One read happens; there is no window. It is a latent defect that returns if the code is recompiled differently or a `volatile` is added later.
3. **Two functions with different source correctness can be byte-identical.** `fetch_fixed` is correct by construction; `fetch_novol` is correct by accident. **A listing cannot distinguish them** — and that limit belongs in any finding drawn from disassembly.

`volatile` is not a mitigation

It is easy to read backwards, so state it plainly: `volatile` **preserves** the double fetch. It does not defend against anything.

Its presence tells you the programmer expected this memory to change asynchronously — which makes a double fetch **worse**, not better, because it is documentary evidence that the race is real.

Why this is mostly a kernel class

The window between the reads is only useful to someone who can write the memory during it, so the memory has to be shared. In practice: a user buffer a driver reads directly, shared memory, or a memory-mapped file.

The classic case is a Windows driver handling a `METHOD_NEITHER` IOCTL, which hands the driver raw user pointers. Validate a length, then re-read it, and you have exactly the vulnerable function above. `ProbeForRead` followed by a re-read rather than a capture is the same mistake with a validation call in front of it.

Two ordinary userland functions reading the same heap object twice is not this class — nothing else can write between the reads.

What a finding needs

1. **Both loads** — their addresses in the listing, and that they read the same location.
2. **The guard between them**, and what it established about the first value.
3. **Who can write that memory during the window.** Without an answer this is not a finding; with one it is often a serious one.
4. **The capture fix** — read once into a local, check the local, use the local. Not a lock, not a re-validation.
5. **The volatile caveat, explicitly.** If you are working from disassembly, say that the listing shows two loads. If you are working from source, say whether the build actually preserves them — because a listing showing one load is consistent with both a correct function and a defective one.

Module V10-unchecked-return

V10 — Unchecked return value (CWE-252 / CWE-476)

Every listing below is real compiler output. The build record is in `notes/build.txt`.

The class

> A function reports that it failed, and the caller carries on regardless.

The report is always there — a null pointer, a status code, a negative count. The caller simply does not look at it, and then uses a result that was never produced.

Flavour 1: the pointer

```
unsigned char *p = malloc(n);
memcpy(p, src, n);
```

```
call    malloc
mov     r8d, ebp
mov     rdx, rdi
mov     rcx, rax          ; the return value, straight into memcpy's destination
mov     rsi, rax
call    memcpy
```

`rax` flows directly into an argument register. **That is easy to misread as handling the value** — it is being used, after all. Using is not checking.

The correct version:

```
call    malloc
mov     rbx, rax
test    rax, rax          ; the check
je      skip
...
```

Flavour 2: the status

```
fill(buf, n);
emit(buf, 64);
```

```
call    fill
mov     edx, 64          ; eax is never referenced again
lea     rcx, buf$[rsp]
call    emit
```

Here `eax` is **not mentioned at all** after the call. Nothing on the page connects to the callee's result — which is a different reading problem from the pointer case, where the result was at least visible.

```
call    fill
test    eax, eax         ; the check
jne     skip
```

The tell
A call whose return register is never tested before the next use.

The positive form is a compact two-instruction idiom right after the call:

```
test    rax, rax  /  je      ; a pointer
test    eax, eax  /  jne     ; a status
```

The register width is a useful secondary signal: `test rax, rax` says the callee returns a 64-bit pointer, `test eax, eax` says a 32-bit status. That helps when you do not have the callee's prototype.

The honest limitation, and it is unique to this class
You cannot judge this from the call site alone.

`printf`'s return value is discarded correctly millions of times a day. `memcpy`'s return is almost never used. An unchecked return is a defect only if **this callee can fail** and **failure changes what the caller may do next** — and neither of those is in the listing.

Every other class in this course is decidable from the instructions: count the bytes, count the loads, read the branch mnemonic, find the missing `lea`. This one needs the callee's contract. **The listing is necessary and not sufficient.**

Practically, that means the finding requires one of: the callee's documentation, its own disassembly, or a name recognisable enough to know its contract.

Do not over-rate the malloc case
An unchecked `malloc` is not automatically a serious finding.

On Linux with default overcommit, `malloc` rarely returns NULL for ordinary sizes, and a null dereference at a small offset is a clean crash rather than a primitive.

It becomes serious when:

- the **size is attacker-influenced**, so NULL is reachable on demand rather than by luck; or
- the subsequent write is at a **large attacker-controlled offset** from the null pointer, which on some platforms reaches mapped memory.

Say which of those applies. "malloc unchecked" on its own invites a reader to assume the worst or dismiss it, and neither is analysis.

Where this class meets another

Look again at the status flavour. When `fill` fails, `buf` is left **partly unwritten** — and all 64 bytes are emitted anyway.

That is simultaneously a **V6** disclosure. The unchecked return is the root cause; the disclosure is the consequence. Report the cause, note the consequence — a fix at the `emit` call would leave the caller still trusting a result that was never produced.

What a finding needs

1. **The call, and its contract** — what the callee returns and when it fails. Cite where you established that.
2. **The absence** — that no `test` / `cmp` of the return register precedes the next use.
3. **What the caller does with the unproduced result** — dereferences it, emits it, treats it as a length.
4. **How failure is reached.** Attacker-influenced allocation size, a short/hostile input to the callee, resource exhaustion. Without this the finding is theoretical.
5. **The consequence class**, if it becomes another one — null dereference, disclosure, or use of stale data.

Module W01-attack-surface

W1 — Attack surface: where does untrusted input enter?

Modules V1–V10 hand you a function and ask what is wrong with it. **This track starts one step earlier: nobody has handed you anything.**

You have a binary, or a source tree, or a device. The first question is not "where is the bug" — it is **"where could a bug matter?"** Everything downstream is scoped by the answer, and getting it wrong wastes the entire engagement.

The definition that does the work

> An **entry point** is a place where data crosses a trust boundary into code you are auditing.

Both halves are load-bearing.

Data crossing. A function that takes a `char *` is not an entry point. A function that takes a `char *` *the attacker chose* is. Every entry point has a source outside your trust domain.

A trust boundary. If the caller is already at least as privileged as the callee, nothing crossed. A root-only configuration file parsed by a root daemon is not attack surface — it is configuration. This is the single most common way to waste a week: auditing code that only a more-privileged party can reach.

So the first artifact of an engagement is not a bug list. It is a **surface inventory**: entry points, ranked, each with who can reach it.

The standard surfaces

Learn them as a checklist, because the point is not to be clever — it is to avoid missing an entire category.

Surface	Where you find it
Command line & environment	`argv`, `getenv`, and anything read before privileges drop
Files & formats	any parser: images, archives, fonts, configs, saved state
Network	listeners, clients parsing responses, and the state machine on either side
IPC	pipes, sockets, shared memory, COM/RPC, D-Bus, mach ports
Device interfaces	IOCTLs, sysfs/procfs writes, driver read/write handlers
Inherited state	file descriptors, handles, current directory, umask, `PATH`

The last row is the one people skip, and it is where privilege-escalation bugs live: a setuid binary does not have to *read* anything to be attacked if it inherits a hostile environment.

Finding them mechanically

Do not read the program from `main`. Work from the outside in.

In a binary:

1. **Imports.** The import table tells you what the program can *do* before you read a single instruction. `recv`, `CreateFileW`, `DeviceIoControl`, `RegQueryValueEx` each imply a surface.
2. **Exports and dispatch tables.** A driver's `DriverEntry` sets up a dispatch table; a DLL's exports are its API. Both are entry-point lists somebody already wrote down for you.
3. **Strings.** Format strings, paths, protocol verbs, registry keys. Strings are a map of what the program expects to receive.
4. **Cross-references from the above.** Every call site of `recv` is a place attacker bytes arrive. That is your worklist.

In source: the same order — build system and entry points first, then grep for the read primitives, then follow the data.

Ranking, and why you must

An inventory is not useful until it is ordered. Rank each entry point by:

1. **Who can reach it.** Unauthenticated remote > local unprivileged > local admin > same-privilege. This dominates everything else.
2. **How much of the input the attacker controls.** A parser reading a fully-attacker-supplied file beats one reading a single integer.
3. **How complex the handling is.** Complexity is where bugs live. A hand-rolled parser outranks a call into a well-tested library.
4. **How far it is from a known-dangerous operation.** Surface that reaches a copy, an allocation size, or an indirect call is worth more than surface that reaches a comparison.

The output is a ranked list with a sentence each. **You have found no bugs at this point, and that is correct** — you have decided where looking is worthwhile.

The trust-boundary questions to ask out loud

For each candidate, answer these before spending time:

- **Who is the attacker?** Name them: unauthenticated network peer, local user, another container, a malicious file the victim opens.
- **What do they already have?** If the answer is "code execution as the same user", most of what you are about to audit is not a boundary.
- **What would crossing this boundary get them?** If a bug here yields the privileges they started with, it is not a finding.

A "bug" that requires root to trigger and yields root is a bug report nobody will act on. Deciding that in W1 costs a minute; discovering it after a week of reversing costs a week.

What this module does not do

It does not find bugs. It produces a **ranked surface inventory** — and the discipline is to stop there and go on to reachability (W2) rather than diving into the first interesting-looking function.

The failure mode this prevents is the most common one in amateur VR: opening a binary, finding a `strcpy`, and spending three days on code no attacker can reach.

The deliverable

For each entry point that survives ranking:

1. **Where it is** — the function, the IOCTL code, the port, the file format.
2. **Who reaches it** — the attacker, named.
3. **What they control** — all of the input, a length field, one flag.
4. **What it reaches** — the dangerous operations downstream, if you know yet.
5. **Why it is ranked where it is** — one sentence.

Module W02-reachability

W2 — Reachability: can that input actually get here?

W1 produced a ranked list of places attacker data enters. This module answers the question that decides whether any of it matters:

> **Does attacker-controlled data reach this dangerous operation, and is it still attacker-controlled when it arrives?**

Both clauses fail independently, and both failures are common.

Why this is the expensive part

BugMuseum's README puts the cost honestly: recognising `strcpy(buf, attacker)` is instant and roughly fixed per bug class. **Proving the argument is reachable and unbounded is ~90% of the effort in a real audit**, and it does not live in the buggy function.

That is why V1–V10 cannot teach it. They contain one function. Reachability is a property of the paths between functions.

The two directions

Sink-first. Start at a dangerous operation and walk *upwards* through callers until you arrive at an entry point or run out of callers.

- Cheap to start: the sink list comes from imports and cross-references.
- Converges fast when the call graph is shallow.
- Fails on indirect calls — the upward walk stops dead at a function pointer.

Source-first. Start at an entry point and walk *downwards*, following the data.

- Matches how the attacker actually reaches the code.
- Naturally answers "is it still attacker-controlled", because you are carrying that fact with you.
- Expensive: the fan-out from one entry point is enormous.

In practice you do both and meet in the middle. Walk up from the sink until the graph gets wide, walk down from the ranked entry point until it gets deep, and look for the overlap.

Taint: what "still controlled" means

Data does not arrive at the sink the way it entered. Between them it may be:

Transformation	Effect on control
Copied / moved	none — still fully controlled
Validated	control reduced to whatever the check permits
Parsed into a field	control reduced to that field's range
Replaced	control lost entirely — this path is dead
Combined with other data	partial; say exactly which bytes

The useful discipline is to carry a **one-line statement of what the attacker controls** at every hop, and to narrow it rather than to forget it. By the time you reach the sink you should be able to say *"the attacker controls the low 16 bits of this length, and nothing else."*

That sentence is the finding. "It's attacker controlled" without the qualifier is a claim you have not made.

Guards that look correct

The hard cases are not missing checks. They are checks that exist and do not do what they appear to:

- **The off-by-one** — the check is present and admits one more than it should.
- **The wrong operand** — bounded by the source rather than the destination (V1's `strncpy(dst, src, strlen(src))`).
- **The wrong unit** — bytes against elements (V1's `wcsncpy` with `sizeof`).
- **Upstream overflow** — the check is correct and the value it checked already wrapped (V5).
- **The re-read** — the value checked is not the value used (V9).
- **The unreached check** — a validating branch that this path skips entirely.

The last one is what makes reachability and validation the same problem. **A check on another path is not a check.** You have to establish which path runs.

Where the walk breaks

Be explicit about these rather than walking past them:

1. **Indirect calls.** A call through a function pointer, a vtable, a dispatch table or a callback registration. The static call graph has no edge here, and your tool will silently show none.
2. **Dispatch tables.** An IOCTL handler table or a message-type switch is an edge the disassembler may not have recovered as a call.
3. **Cross-module boundaries.** An export called by another binary you are not looking at.
4. **Data-driven control flow.** A state machine where the reachable next state depends on parsed input.

At each of these, either resolve the target set by finding where the pointer is assigned, or **record the break as an assumption** in the finding. An unstated assumption at hop four invalidates everything after it.

The output: a path, not a verdict

Reachability produces a **path record**, and it is the `path` element of the finding format used throughout this pack:

```
source       the entry point, and the attacker who reaches it
path         the call chain, hop by hop, with the break points named
             and any assumption stated
control      what the attacker still controls on arrival, precisely
guards       every check on this path, and what each establishes
sink         the dangerous operation, with its address
precondition what has to be true for this path to run
```

Note what is not in there: whether it is exploitable. That is W3.

When to stop

Three honest stopping conditions:

- **You reached an entry point.** Write the path.
- **You ran out of callers and none is an entry point.** The code is unreachable from outside; record that and move on — a negative result that is written down is worth having.
- **You hit an unresolvable indirect call.** Record the break, state what would resolve it, and rank whether resolving it is worth the time.

The failure mode is the fourth option nobody admits to: walking until it feels like enough and writing "attacker controlled" without a path. That claim cannot be checked by a reviewer, and it is the one most often wrong.

Module W03-primitives

W3 — Primitives: what does this bug actually give you?

W2 produced a path: attacker data reaches a dangerous operation, and you can say precisely what is controlled on arrival. This module answers the next question, and it is the one that decides severity:

> **What capability does that give the attacker?**

A capability is a **primitive**. Naming it is the difference between "there is a memory-safety bug here" and a finding someone can act on.

The taxonomy

Learn these as a fixed set. Almost every memory-safety bug reduces to one or more of them.

Primitive	What the attacker can do
Relative write	write past a buffer, at a *fixed or bounded* distance from it
Write-what-where	write a chosen value to a chosen address
Relative read	read past a buffer — usually a disclosure
Arbitrary read	read from a chosen address
Uninitialised disclosure	read memory nobody wrote — contents not chosen, but leaked
Allocation control	choose a size, or the lifetime, of an allocation
Reuse control	choose the *contents* of memory something else still points at
Control-flow transfer	cause a branch to an address influenced by the attacker

The ordering matters: each row is roughly more powerful than the one above, and the job of an exploit is to climb.

Every class maps onto this

The class modules named the bug. Here is what each *gives* you:

Class	Typical primitive
V1 stack overflow	relative write — bounded by how far past the buffer you can reach
V2 heap overflow	relative write, into a neighbour or into allocator metadata
V3 use-after-free	**reuse control** if you can groom the reallocation; otherwise a stale read
V4 double free	allocator-state corruption → often two allocations at one address
V5 integer defects	**allocation control** — you chose the size, and it was wrong
V6 uninitialised	uninitialised disclosure
V7 format string	arbitrary read via `%s`, and write-what-where via `%n`
V8 type confusion	whatever the confused field is — often control-flow transfer
V9 double fetch	allocation or length control, by changing the value after its check
V10 unchecked return	usually a null dereference; occasionally reuse of a stale result

Two things to notice.

One bug can yield several. A heap overflow into an adjacent object with a length field gives you a relative write *and*, through that length, a wider relative read. Say all of them.

The primitive is not the class. Two stack overflows can have completely different value: one reaches four bytes past a buffer into an unused field, another reaches the return address. Same CWE, different findings.

Qualify it, or you have not named it

An unqualified primitive is nearly useless. Every one needs its limits:

- **How far?** A relative write of four bytes is not the same as one of four kilobytes.
- **What values?** Can you write arbitrary bytes, or only ASCII, or only non-NUL? A `strcpy` primitive cannot write a zero byte.
- **How many times?** Once, or repeatedly? Repetition turns a weak primitive into a strong one.
- **What lands there?** For heap bugs — what is the neighbour, and do you get to choose it?
- **Under what preconditions?** Carried forward from the W2 path record.

> "Relative write, up to 31 bytes past a 16-byte stack buffer, no NUL bytes, once per request, reaching the saved return address"
>
> is a finding. **"Stack overflow" is a class name.**

What the bug does *not* give you
State this explicitly. It is the part that gets skipped and the part a reviewer most needs.

- A stack overflow behind a cookie gives a relative write **and not** a reliable return-address hijack (V1/M).
- A format string on a runtime with `%n` disabled gives arbitrary read **and not** a write (V7).
- A use-after-free where you cannot influence the reallocation gives a stale read **and not** reuse control (V3).
- A type confusion behind CFG gives a transfer to a *valid indirect-call target* **and not** arbitrary code (V8/M).

Each of those is a mitigation constraining a primitive rather than removing a bug — module M's rule, applied.

Chaining, at the level this course goes to
Real exploitation composes primitives: leak an address to defeat ASLR, then use a write to redirect control. You should be able to **recognise which link a bug provides**:

- An **information disclosure** is often the enabling half — it defeats ASLR or reveals a heap address, and by itself does nothing.
- A **write** without a leak may have nowhere useful to aim.
- **Allocation control** is what makes heap grooming possible, and grooming is what makes reuse control reliable.

This is why a V6 info leak beside a V2 heap overflow is worth more than either alone, and why a finding should say which link it is.

And this course stops here. Naming the primitive, its limits, and which link it provides is identification. Building the chain is not — that is the ceiling stated in `reverse-engineering-course.md` and inherited here.

The deliverable
Added to the W2 path record:

```
primitive     the row from the taxonomy, or several
qualifiers    distance, permitted values, repeatability, what lands adjacent
denied        what this bug does NOT give, and which mitigation denies it
link          enabling (leak) or acting (write/transfer), and what it would
              need to pair with
```

Module W04-variant-analysis

W4 — Variant analysis: you found one, where are the others?
The single largest productivity multiplier in vulnerability research, and the one most often skipped because the first bug feels like the finish line.

> **A bug is evidence about the codebase, not just about itself.**

Somebody misunderstood an API, or copy-pasted a handler, or applied a convention inconsistently. Whatever produced this bug very likely produced others, and you now hold the best possible query for finding them: a confirmed instance.

Why bugs cluster
Four mechanisms, and they suggest different searches:

Mechanism	Where the siblings are
Copy-paste	the same code shape elsewhere — often the adjacent handler
A misunderstood API	every other call to that API, across the whole codebase
A convention applied unevenly	the places that did not get the memo
One author, one habit	other code by the same author, in the same period

The second is the highest-yield in practice. If a developer believed `strncpy` NUL-terminates, or that `snprintf`'s return is the number of bytes written, that belief is in every call they made.

Turning the bug into a query
This is the actual technique, and it has three steps.

1. Strip the incidentals. Your finding contains facts that are specific to this instance — a function name, a buffer size, an offset. Remove them. What remains is the *shape*.

2. Name the invariant that was violated. Not "this function has a bug" but "a length derived from the source is used to bound a copy into the destination" or "a discriminant is not consulted before a union member is used". The invariant is the query.

3. Choose the widest search that expresses it. In descending order of precision:

Where	How
Source	`grep` for the API, then read; `semgrep`/CodeQL for a structural query
Binary, symbols	search symbols and cross-references for the same call, then diff the guard sequences
Binary, stripped	search for the *idiom* — the instruction pattern the class compiles to, which is exactly what V1–V10 trained
Across versions	diff releases and look for the same shape appearing or being fixed
Across products	shared libraries and vendored code carry their bugs with them

The stripped-binary row is why the class modules matter here. "Find every call site where a count register is loaded from `strlen` of the source" is a searchable idiom because you learned what it looks like.

The patch as a variant oracle

When a vendor fixes a bug, the patch tells you two things:

1. **What the invariant was** — the fix states it more precisely than any advisory.
2. **Which instances they knew about.**

The second is the opportunity. Vendors routinely fix the reported instance and miss its siblings. Diff the patched function to learn the shape, then search the rest of the binary for it. **A patch is a free, authoritative query, and the set of places it was not applied is a candidate list.**

This composes directly with patch diffing: the diff gives the shape, the sweep gives the variants.

Where to look first

Ranked, because the sweep must be bounded:

1. **The same file.** Adjacent functions, especially ones with parallel names — `handle_read` / `handle_write`.
2. **The same dispatch table.** If one IOCTL handler is wrong, its siblings were written the same afternoon.
3. **Every other call to the same API.** The misunderstood-API case.
4. **The same parser for other formats or versions.** Version 1 was audited; version 2 was added later by someone else.
5. **Other consumers of the same shared code.** Vendored copies do not get the fix.

Discipline that keeps it honest

Write the query down. A sweep you cannot state is a sweep nobody can repeat or check — including you, next month.

Each instance gets its own reachability. This is where variant analysis goes wrong. Finding the same *shape* is not finding the same *bug*: the sibling may be unreachable, may be guarded elsewhere, or may be called only with constant arguments. Every candidate goes back through W2 before it is a finding.

Record the negatives. "I checked all 14 other call sites of this API and 13 pass a destination-derived bound" is a strong statement — it bounds the problem and it stops the next person repeating your sweep.

Say when you stopped and why. A sweep is never exhaustive. "Searched the same binary and the two other drivers in the package; did not search the rest of the product" is honest and useful. Silence implies completeness you did not achieve.

The deliverable

```
invariant   the rule that was violated, stated generally
query       exactly what was searched, and where — reproducible
candidates  every match, with its location
triaged     for each: reachable (W2) and what primitive (W3), or why it is not
             a finding
scope       what was searched and what was not
```

The last line is the one that makes the report trustworthy. Everything else can be re-derived from it.

Module W05-crash-triage

W5 — Crash triage: working backwards from the fault
Every module before this one runs **forwards** — from a defect, to a path, to a capability. Real work usually starts at the other end:

> **You have a crash. Is it a bug, is it *this* bug, and where did it actually start?**

A crash is evidence. It is not a finding, and the distance between the two is where most triage goes wrong.

First: what actually faulted
Before any theory, extract the facts:

1. **The faulting instruction** — what it was trying to do: read, write, execute.
2. **The faulting address** — and its relationship to something meaningful.
3. **The faulting module** — whose code was executing.
4. **The call stack** — and whether it is trustworthy (a corrupted stack produces a fictional one).
5. **Which register held the bad value**, and what it should have held.

Those five, written down, before any sentence beginning "this looks like".

Classify the fault

Symptom	Usual meaning
Read/write at or near **0**	null dereference — an unchecked return (V10)
Write at a **large controlled** address	genuine write primitive
Read of an unmapped address, value looks like ASCII	a length or pointer taken from attacker data
Fault **inside the allocator**, not your code	earlier heap corruption (V2/V4) surfacing late
Process killed by a **guard check** rather than faulting	a mitigation fired — `__stack_chk_fail`, `__security_check_cookie`, a fast-fail
Fault at a **plausible-looking but wrong** address	often a type confusion (V8) or a stale pointer (V3)

Two of these deserve emphasis.

"Near null" is not automatically boring. A null dereference at offset 8 is a crash. A null dereference at attacker-controlled offset `0x40000` may reach mapped memory. Record the offset, not just "null deref".

A fault inside the allocator means the damage happened earlier. The crash address tells you almost nothing about the defect. That is the signature of heap corruption, and it is the case where working backwards matters most.

The discipline that separates triage from guessing

> **A crash in the module you are studying is not evidence you triggered the bug you were looking for.**

This is the single most valuable habit in this module. Software crashes for many reasons: your harness is wrong, the environment is unusual, you hit an unrelated defect, or the fault is a legitimate error path you drove into.

When you are reproducing a *known* vulnerability, there is a mechanical check:

Does the fault land in code the patch changed?

Take the patch, list the functions it modified, and compare against the faulting function and the frames above it. If the crash is nowhere near the patched code, you have found *something* — possibly something new and worth its own record — but you have not reproduced the CVE.

Without that check, an incidental crash gets written up as a successful reproduction, and the error is invisible because the observable outcome — the target crashed — is exactly what was expected.

Working backwards

From the faulting instruction to the defect:

1. **Identify the bad value.** Which operand was wrong? An address, a length, a pointer, an index.
2. **Find where it was produced.** Walk up the data flow — not the call stack. The frame that faulted often merely *used* a value produced elsewhere.
3. **Find where it stopped being valid.** This is the defect. It may be far above the fault: a wrapped multiply (V5), an unchecked return (V10), a stale pointer (V3).
4. **Confirm the causal chain** by predicting a second observation and checking it. If the theory says the length came from field X, change field X and the crash address should move predictably.

Step 4 is what makes it triage rather than a story. A theory that explains the crash but predicts nothing has not been tested.

Deduplicating

With more than one crash, group by **root cause**, not by symptom:

- The **stack** is a poor key — one defect produces many stacks, and heap corruption produces stacks unrelated to the defect.
- The **faulting address** is worse — it varies with allocation layout.
- The **defect** is the only stable key, which means dedup happens *after* working backwards rather than before.

Practical compromise: bucket by faulting function plus fault type to triage cheaply, and re-bucket by defect once you understand each.

What a triage record has to contain

```
observed     faulting instruction, address, module, and the bad register value
classified   which row of the table, with the offset if near-null
attribution  does the fault land in the code you were targeting? evidence either
             way
backwards    the bad value, where produced, where it stopped being valid
confirmed    the second prediction you made, and whether it held
dedup        which defect bucket, and why this crash belongs to it
```

The `attribution` line is the one most often missing, and it is the difference between "the target crashed" and "I reproduced the vulnerability".

The two honest outcomes people skip

- **"This crash is not the bug I was looking for."** It may still be a finding, and it goes in its own record.
- **"This crash is not a security bug at all."** An assertion, a legitimate error path, a harness defect. Write it down and move on — an unexplained crash left in the pile will be re-triaged by someone else next month.