Web-App Assessment — pentest tradecraft as Python

Concept cards — generated 2026-08-05

Contents

Track A

Module A1

Four containers, and the assessment job each one does

Why this matters

You will be asked to write recon code in front of someone. The code will be twenty lines long, and roughly four of them will be a container choice. The interviewer is not checking whether you know that a set exists — they are checking whether you picked it *on purpose* and can say what it cost.

There is also a practical edge. A crawl of a real application produces tens of thousands of URLs. The difference between `if url in seen_list` and `if url in seen_set` is the difference between a scan that finishes and one you kill after twenty minutes. That is not a micro-optimisation; it is the same code with one word changed.

Four containers cover almost everything this course asks you to build.

`dict` — the lookup table, and it remembers the order

`dict` maps a key to a value in **O(1)** average time, and since Python 3.7 it **preserves insertion order** as a language guarantee, not an implementation accident.

That second property does more work than people expect. When you need distinct items *in the order you first met them* — endpoints in crawl order, parameters in observation order — the idiom is a dict, not a set:

```
ordered_unique = list(dict.fromkeys(paths))
```

Reach for it when: you need a value per key (parameters per endpoint, sessions per role), or you need dedup that keeps order.

`set` — membership and dedup, and it throws the order away

`set` answers "have I seen this?" in **O(1)** average time. That is the crawler frontier, the visited list, the scope check.

The cost is stated plainly: **a set has no order.** `sorted(set(x))` is not "dedup preserving order", it is "dedup then re-order alphabetically", and those two are different answers. If a downstream diff compares run to run, sorting is often what you want. If a human reads the list expecting to see the site as the crawler walked it, it is wrong. Decide which, and say which.

Reach for it when: membership, dedup where order genuinely does not matter, and set algebra — `observed - documented` is the endpoint list nobody wrote down.

`Counter` — counting and ranking, with one trap

`Counter` is a dict subclass that counts:

```
from collections import Counter
counts = Counter(path for path, _ in requests)
```

The trap is `most_common()`. It sorts by count descending and, for equal counts, **falls back to insertion order** — so two paths seen the same number of times come back in whatever order the log happened to be in. Your test passes on Tuesday's log and fails on Wednesday's, which is the worst kind of bug because it looks like a data problem.

When ties matter, sort explicitly with a compound key:

```
ranked = sorted(counts.items(), key=lambda kv: (-kv[1], kv[0]))
```

Negate the count to sort it descending while the path still sorts ascending — one pass, one stated rule, deterministic on every input.

Reach for it when: frequency, ranking, "which endpoint is hit most".

`deque` — the queue, and the reason it exists

`collections.deque` appends and pops at **either end in O(1)**. A list pops from the front in **O(n)**, because every remaining element shifts down one slot.

A breadth-first crawler pops from the front on every single iteration. With a list that is O(n) per pop over a growing frontier; with a deque it is O(1). The crawler is the canonical case and it is the one you will be asked to write.

```
from collections import deque
frontier = deque([start_url])
while frontier:
    url = frontier.popleft()      # O(1); list.pop(0) is O(n)
```

Reach for it when: BFS, a work queue, a sliding window, a bounded history (`deque(maxlen=100)` of recent responses).

Worked example: the same job, four ways

You have crawl output and you want the endpoints worth testing, in the order the crawler found them.

```
paths = ["/a", "/b", "/a", "/c", "/b"]

list(dict.fromkeys(paths))    # ['/a', '/b', '/c']  distinct, first-seen order
list(set(paths))              # ['/c', '/a', '/b']  distinct, order UNSPECIFIED
sorted(set(paths))            # ['/a', '/b', '/c']  distinct, alphabetical
Counter(paths).most_common()  # [('/a', 2), ('/b', 2), ('/c', 1)]  with the tie trap
```

The first and third look identical here, on five items. They are not the same function, and on the next dataset they will disagree. Knowing *why* they agreed this time is the whole skill.

What to say out loud

Two sentences, and they are the ones the interviewer is waiting for:

"I'm using a set for the visited check so membership is O(1) rather than O(n) per URL."

"Ties here break by path ascending, so I'm sorting on a compound key rather than using `most_common`, which would leave it to insertion order."

Where this goes next

The drill after this card gives you a job and asks for the container. The error-hunt gives you a dedupe pass that is accidentally quadratic and asks you to find the line. Then you write the dedupe yourself, and the capstone ranks endpoints under a ten-minute clock — where the tie-break above is the thing that separates a pass from a near miss.

Idioms that survive a 2-million-line log

Why this matters

Web-assessment code reads text that someone else produced, in volumes you did not choose. An access log from a week of production traffic is two million lines. A crawl of a large application is fifty thousand URLs. The idioms below are the ones that keep working at that size, and the ones whose absence is visible to anyone reading over your shoulder.

They are also, specifically, what a live-coding round is watching for. Not because comprehensions are clever — because reaching for the right one without pausing is the difference between spending your ten minutes on the problem and spending it on the plumbing.

Comprehensions, and the one-character version that scales

A list comprehension builds the whole list in memory:

```
paths = [extract_path(line) for line in log_lines]        # all of it, at once
```

A generator expression — same code, round brackets — yields one at a time:

```
paths = (extract_path(line) for line in log_lines)        # one at a time
```

On eight lines these are interchangeable. On two million they are not: the first materialises two million strings; the second holds one. Chain a generator into `Counter`, `sum`, `max` or a `for` loop and nothing is ever fully in memory:

```
counts = Counter(extract_path(line) for line in log_lines)
```

The rule is small and worth stating out loud: **build a list when you need to index it, iterate it twice, or know its length. Otherwise let it stream.**

Two traps come with generators, and both are worth naming before someone catches you on them. A generator is consumed **once** — iterate it twice and the second pass sees nothing. And it does not have a `len()`, because it does not know one yet.

`enumerate` and `zip`

```
for lineno, line in enumerate(log_lines, start=1):        # start=1 for humans
    ...
```

A log parser that reports "malformed input on line 4,812" needs the number, and `enumerate` is where it comes from. Maintaining `i = 0; i += 1` by hand is the tell that someone is writing C in Python, and an interviewer will notice.

`zip` walks two sequences together — a request list beside its response list, a baseline run beside a mutated one:

```
for baseline, mutated in zip(baseline_responses, mutated_responses):
    if baseline.status != mutated.status:
        ...
```

`zip` stops at the shorter input, silently. When the two are supposed to be the same length, `zip(a, b, strict=True)` raises instead — and in a differential test, a length mismatch is a bug you want raised, not padded over.

Sorting: the key, and the tie you did not think about

`sorted` takes a `key` function and is **stable**, meaning equal elements keep their relative input order.

Stability sounds like a nicety and is actually a trap. If you sort by count alone, the order of the equal-count entries is whatever the input happened to be — so your output changes when the log changes, and your test passes today and fails tomorrow.

State the tie-break instead, with a compound key:

```
ranked = sorted(counts.items(), key=lambda kv: (-kv[1], kv[0]))
```

Read it aloud: *descending by count, then ascending by name.* The negation reverses one component without reversing the other, which `reverse=True` cannot do — it would flip both.

For values you cannot negate, sort twice, least-significant first, and let stability carry the earlier pass:

```
rows.sort(key=lambda r: r.path)               # secondary, ascending
rows.sort(key=lambda r: r.count, reverse=True)  # primary, descending
```

Worked example: the shape of most log tasks

```
import re
from collections import Counter

REQUEST = re.compile(r'"[A-Z]+\s+(?P<target>\S+)\s+HTTP/\d\.\d"')

def rank(log_lines, n):
    counts = Counter()
```

```
    for line in log_lines:                          # streams; never materialised
        m = REQUEST.search(line)
        if not m:                                   # unparsable is data, not an error
            continue
        counts[m.group("target").split("?", 1)[0]] += 1
    return sorted(counts.items(), key=lambda kv: (-kv[1], kv[0]))[:n]
```

Four decisions in nine lines, and each one is a sentence you should be able to say: stream rather than materialise; `search` not `match`, because the quoted request sits mid-line; skip what does not parse rather than raising on it; compound key so the tie-break is stated.

What "malformed input is data" means

Every log has junk lines — a truncated write, a health-check probe in a different format, a line with a quoted space inside the user agent. A parser that raises on the first one is useless on real input, and a parser that silently returns wrong counts is worse.

Skip what you cannot parse, and **count what you skipped**. "3 of 2,000,412 lines did not parse" is a sentence a client can act on. Zero results with no explanation is not.

Where this goes next

The complexity drill after this card asks for the big-O of snippets that appear in exactly this shape of code. Then an error-hunt over a dedupe pass that is accidentally quadratic. The A2 module takes the parsing half further — `urllib.parse`, bytes versus str — and its capstone is this worked example under a ten-minute clock.

Module A2

The URL round trip: split, mutate, rebuild

Why this matters

"Take this URL, change one query parameter, give it back" is the most commonly asked web-flavoured live-coding question there is. It sounds trivial. It is where people lose the round, because the obvious implementation quietly changes three other things about the URL and nobody notices until the target behaves differently.

It is also the inner loop of everything you will build in this course. A parameter fuzzer is this operation in a loop. An IDOR scanner is this operation on a path segment. A reflected-XSS probe is this operation with a marker as the value. Getting it exactly right once means every later module inherits it.

The four functions

```
from urllib.parse import urlsplit, parse_qsl, urlencode, urlunsplit
```

`urlsplit(url)` breaks a URL into five parts and does nothing clever:

```
u = urlsplit("https://shop.example.com:8443/search?q=hat&sort=asc#top")
# u.scheme   'https'
```

```
# u.netloc   'shop.example.com:8443'      host AND port, together
# u.path     '/search'
# u.query    'q=hat&sort=asc'             one flat string, still encoded
# u.fragment 'top'
```

Use `urlsplit`, not `urlparse`. `urlparse` additionally splits out a `params` field for a legacy path-parameter syntax (`/path;key=value`) that essentially nothing uses, and it will eventually surprise you on a URL containing a semicolon.

`parse_qsl(query, keep_blank_values=True)` turns the query string into a list of `(name, value)` pairs, in order, decoded:

```
parse_qsl("q=hat&sort=asc&debug", keep_blank_values=True)
# [('q', 'hat'), ('sort', 'asc'), ('debug', '')]
```

`urlencode(pairs)` turns pairs back into a query string, encoding as it goes:

```
urlencode([('q', 'red hat'), ('sort', 'asc')])
# 'q=red+hat&sort=asc'
```

`urlunsplit(parts)` reassembles the five components. It takes a 5-tuple in `urlsplit` order, so the idiom is `u._replace(query=new_query)` and hand it back.

The complete round trip

```
def set_param(url: str, name: str, value: str) -> str:
    u = urlsplit(url)
    pairs = parse_qsl(u.query, keep_blank_values=True)
    pairs = [(k, value) if k == name else (k, v) for k, v in pairs]
    if name not in dict(pairs):
        pairs.append((name, value))
    return urlunsplit(u._replace(query=urlencode(pairs)))
```

Six lines, and every one of them is a decision worth being able to defend. Note especially that the replacement happens **in place** — the parameter keeps its position in the query string. That matters more than it sounds like it should: some applications behave differently on parameter order, and a diff of your request against the original should show one change, not a reshuffle.

`parse_qs` versus `parse_qsl`, and the default that deletes findings

These two look interchangeable and are not.

	`parse_qs`	`parse_qsl`
Returns	`dict[str, list[str]]`	`list[tuple[str, str]]`
Order	lost	preserved
Duplicates	grouped into a list	kept as separate pairs

For assessment work, **`parse_qsl` is almost always the right one**, because order and duplication are both signal. `?id=1&id=2` is not noise — how the application resolves that duplicate

is a real finding (HTTP parameter pollution), and `parse_qs` hands you a shape that has already made the decision for you.

Then the default that actually bites:

```
parse_qsl("debug")                          # []          <-- gone
parse_qsl("debug", keep_blank_values=True)  # [('debug', '')]
```

`keep_blank_values` is `False` by default, so a valueless parameter simply **vanishes**. A parameter inventory built without it is missing exactly the feature-flag-shaped parameters (`?debug`, `?admin`, `?preview`) that are the most interesting things in it. There is no error and no warning; the parameter is just not in your output.

Two places the round trip is lossy

Say these out loud before the interviewer says them to you.

Encoding is normalised, not preserved. Decode-then-re-encode gives you a *semantically* equivalent query string, not a byte-identical one. `%2F` may come back as `/`, and a space arrives as `%20` and leaves as `+`. Both are correct per the spec. Both mean your rebuilt URL is not `==` to the original even when you changed nothing, and both can matter to a target that is parsing the raw string itself — which is precisely the kind of target you are testing.

Ordering is only preserved if you preserve it. Round-trip through a dict, or through `parse_qs`, and duplicates collapse and order is whatever the dict happened to do. That is the bug the code above avoids by staying with a list of pairs from start to finish.

When byte-exactness matters — and for raw request manipulation it does — do not round-trip at all. Do a targeted edit on the raw query string and accept that you are now writing a small parser. Knowing *when* to drop down to that is the senior judgment being tested.

What to say out loud

> "I'm using `parse_qsl` with `keep_blank_values` so I keep order, keep duplicates, and don't silently drop valueless parameters."
>
> "Note this round trip normalises the encoding, so the output won't be byte-identical to the input even when nothing changed. If the target is sensitive to the raw form I'd edit the query string directly instead."

Where this goes next

The drill after this card fires the four function names at you until they are reflex. The listing puts a raw transcript in front of you and asks which header decides how the body parses. Then you write `set_param` yourself, and the module capstone parses a week of access logs under a ten-minute clock.

HTTP is bytes, and where you are allowed to forget that

Why this matters

Your client library hands you `response.text` and it works, so for a while the distinction seems academic. Then one of these happens:

- You download a PDF or a JPEG and your parser explodes on invalid UTF-8.
- A page renders as `caf├⌐` in your output and you spend twenty minutes looking for a bug in the target that is actually in your terminal.
- You build an injection payload with a specific byte in it and the library helpfully re-encodes it into something else before it goes on the wire.
- You measure a response length for a differential test and get the character count when the interesting delta was in bytes.

Every one of those is the same misunderstanding, and it is worth clearing once.

The boundary

HTTP bodies and headers are bytes on the wire. Always. A string is what you get *after* somebody decided which encoding those bytes were in.

```
"café".encode("utf-8")          # b'caf\xc3\xa9'   str -> bytes, ENCODE
b"caf\xc3\xa9".decode("utf-8")  # 'café'           bytes -> str, DECODE
```

The direction is the thing people invert under pressure. One sentence fixes it: **encode to bytes, decode to text.** Bytes are the wire format; you encode something to put it on the wire.

`.content` versus `.text`

Every HTTP client gives you both:

```
r = httpx.get(url)
r.content    # bytes — exactly what arrived
r.text       # str   — .content decoded, using a guess
```

`.text` is a **guess**. The library reads the `charset` from the `Content-Type` header if there is one; if not, it falls back to a default or sniffs. When the guess is wrong you get mojibake, and mojibake in your report looks like a finding about the target rather than a bug in your tool.

The rule:

- `.text` when you know it is text and you care about characters — matching a reflected marker, extracting a CSRF token, reading an error message.
- `.content` when it is not text, when you need the exact bytes, or when you are measuring size. `len(r.content)` is the byte length; `len(r.text)` is the character count, and for a differential test those are different numbers that disagree exactly when the response contains anything non-ASCII.

For an assessment tool, prefer `.content` and decode explicitly at the point you need characters. That way the decision is yours and it is visible in the code.

Where each one belongs in a scanner

```
r = client.get(url)

# Size for a differential comparison: bytes. Always bytes.
length = len(r.content)

# Looking for your reflected marker: text is fine, and errors="replace"
# means a decoding problem degrades instead of raising mid-scan.
body = r.content.decode(r.encoding or "utf-8", errors="replace")
reflected = MARKER in body

# Downloading an artefact: never touch .text.
(out_dir / name).write_bytes(r.content)
```

`errors="replace"` deserves a note. In a scanner, a `UnicodeDecodeError` two thousand requests into a run is a terrible failure mode — you lose the run over a single binary response. Replacing the undecodable bytes with `�` lets the scan finish, and the replacement characters are visible in the output if you later care.

Headers are bytes too, and they are ASCII by convention

Header field names and values are bytes on the wire, and by RFC they are restricted to ASCII. Libraries present them as `str` and encode on the way out, which is almost always what you want.

Two consequences that show up in real testing:

- **Non-ASCII in a header value is a compatibility question, not a given.** If you are deliberately putting odd bytes in a header to see what the target does, a helpful client library may normalise or reject them before they leave. That is when you drop to a lower-level request API — the library being helpful is the obstacle.
- **Header names are case-insensitive** (RFC 9110), so `Content-Type` and `content-type` are the same header. Normalise to lowercase the moment you store them, or every downstream check for `authorization` will miss half the time depending on what the server felt like sending.

Worked example: the mojibake that is not a finding

```
raw = b"caf\xc3\xa9"                 # 'café' in UTF-8

raw.decode("utf-8")                  # 'café'      correct
raw.decode("latin-1")                # 'cafÃ©'     the classic mojibake
raw.decode("ascii")                  # UnicodeDecodeError
```

Same bytes, three outcomes. If your output shows the second one, the target sent perfectly good UTF-8 and *you* decoded it as latin-1. Nothing is wrong with the application. Check your decoding before you write the finding.

What to say out loud

> "I'm using `.content` for the length comparison because I want bytes — `.text` would give me a character count, and those differ as soon as the body has anything non-ASCII in it."
>
> "I'm decoding with `errors='replace'` so one binary response can't kill a two-thousand-request scan."

Where this goes next

The drill after this card includes the encode/decode direction and `.text` versus `.content`. The HAR parsing task lowercases header names for exactly the reason above, and the response-diffing work in module B2 depends on comparing byte lengths rather than character counts — a differential that quietly compares the wrong number is a false-positive generator.

Module A3

Red-green-refactor as a narration, not a ritual

Why this matters

The job description weights TDD, and a live-coding round that mentions it is almost never checking whether you can recite the loop. It is checking whether you can *work* that way while someone watches — which means saying what you expect to happen before it happens, and being right.

That is also the honest reason to bother. Writing the test first forces you to decide what the function's contract is while it is still cheap to change. In assessment code, where the function's output becomes a finding in someone's report, an undecided contract is how you end up with a detector whose behaviour nobody can state.

The loop, at the right size

Red. Write one failing test. Run it. Watch it fail, and say why.

Green. Write the least code that makes it pass. Not the good version. The passing one.

Refactor. Improve the code with the test holding it still. Run again.

The step size is the whole discipline. "Write the failing test" means one, not a suite. If your red step takes four minutes, the step is too big and the feedback you get from it is correspondingly vague.

Watching it fail is the part people skip

A test that has never failed has not been tested. It is entirely possible to write an assertion that passes for a reason unrelated to your function — a typo in the name of the thing you imported, a fixture that returns the expected value directly, an assertion on a mock rather than on behaviour.

The red step is what rules that out, and it takes two seconds. Say the sentence out loud:

> "This should fail with a NameError, because `normalise_url` doesn't exist yet."

Then run it, and get the NameError you predicted. Now the test is trustworthy. If it fails a *different* way than you predicted, stop — something is not what you think it is, and finding that out at line four is much cheaper than at line forty.

What to test in web-assessment code

The edge cases are the same handful every time, and naming them fast reads as experience:

- **Empty input.** No lines, no URLs, no parameters. Almost every parser has a wrong answer here and almost no sample data contains it.
- **Malformed input.** A junk log line, a URL that will not parse. Does it skip, raise, or silently corrupt the result?
- **The duplicate.** The same path twice, the same parameter twice. What is the rule, and is it written down?
- **The boundary.** `n=0`, `n` larger than the data, exactly one item.
- **The tie.** Two things with equal counts. If your test data has no tie, your tie-break is untested and it is the first thing that will differ on real data.

Say the ones you are not going to write, and why:

> "I'd also want a case for a response with no Content-Type, and a timeout path — I'm stubbing those for time, but they'd be the next two tests."

Naming a gap deliberately is senior. Leaving it silent and hoping is not, and an interviewer cannot tell the difference between "did not think of it" and "chose not to" unless you say so.

Pin behaviour, not implementation

The failure mode that costs you later is a test that passes while the function is wrong, and the usual cause is asserting on the wrong thing.

```
# Pins your implementation. Passes even if the function returns garbage.
def test_calls_the_client():
    fake = Mock()
    fetch_all(["/a"], client=fake)
    assert fake.get.called

# Pins the behaviour. Fails if the function is wrong.
def test_returns_one_record_per_url():
    records = fetch_all(["/a", "/b"], client=stub_client())
    assert [r.url for r in records] == ["/a", "/b"]
```

The first test breaks when you refactor and survives when you break the function — precisely backwards. A useful heuristic: if you could rewrite the function completely differently and keep the same contract, the test should still pass. If it would not, it is testing the wrong layer.

Worked example: the first ninety seconds

The task: normalise a URL to its endpoint path.

```
# test_normalise.py
from normalise import normalise_url

def test_strips_query_and_fragment():
    assert normalise_url("https://x.test/search?q=a#top") == "/search"
```

Run it. `ModuleNotFoundError` — predicted, so the test harness works.

```
# normalise.py
from urllib.parse import urlsplit

def normalise_url(url: str) -> str:
    return urlsplit(url).path
```

Green. Now the second test, which is where the real contract gets decided:

```
def test_trailing_slash_collapses_but_root_survives():
    assert normalise_url("https://x.test/search/") == "/search"
    assert normalise_url("https://x.test/") == "/"
```

Red again — and notice that writing this test is what forced the root-slash rule to be decided at all. Nobody sits down intending to specify that. It falls out of trying to write the assertion, which is the argument for tests-first in one example.

What to say out loud

> "I'll start with the failing test so I know the harness works and the assertion means something."
>
> "That's green with the simplest thing that works — I'll refactor now that there's a test holding it."
>
> "Edge cases I'd add: empty input, a malformed URL, and a duplicate path. I'll write the first one and stub the other two for time."

Where this goes next

The drill after this card gives you a function shape and asks for the three edge cases an interviewer expects to hear. The error-hunt shows a suite that passes while its function is broken. Then you ship a normaliser **with its own test suite**, which the platform's `coverage` check grades — this is the one module where your tests are the graded artefact, not just the code.

Module A4

Concurrency for people who send a lot of requests

Why this matters

Every tool in this course sends requests in bulk. A parameter fuzzer tries two thousand names. An id enumeration walks a range. A response differ needs a baseline and a mutation for each of them.

Done sequentially at 100 ms per request, two thousand requests is three and a half minutes of your machine doing nothing but waiting. Done concurrently at a sane bound, it is seconds. That is the entire argument, and it is worth being precise about *why* it works, because the reason tells you when it will not.

What async actually buys you

Async does **not** make your code faster. It makes your code stop *waiting*.

A network request spends nearly all its time idle — the bytes are on the wire and the CPU has nothing to do. `await` hands control back to the event loop during that idle time so another coroutine can send its request. One thread, one process, thousands of sockets in flight.

The corollary is the thing to remember: **async helps I/O-bound work and does nothing for CPU-bound work.** Hashing a wordlist, parsing a large document, computing a diff — those keep the CPU busy, and running them "concurrently" on one event loop just interleaves them badly. For that you want processes.

For web assessment, essentially everything is I/O-bound, which is why this model fits so well.

The three ways to run many things

```
results = await asyncio.gather(*(fetch(u) for u in urls))
```

`gather` — run them all, wait for all, get a list back **in the order you passed them in**, not completion order. This is the default choice, and the ordering guarantee is why: `results[i]` still matches `urls[i]` even though the responses arrived scrambled.

```
for coro in asyncio.as_completed(tasks):
    result = await coro          # arrives as each finishes
```

`as_completed` — results as they land. Right for streaming progress to a terminal, wrong when the output has to line up with the input. If you use it and still need the index, carry it through the return value yourself.

```
async with asyncio.TaskGroup() as tg:
    tasks = [tg.create_task(fetch(u)) for u in urls]
```

`TaskGroup` — the modern structured form. Nothing escapes the block; if one task fails the rest are cancelled and the errors surface together. Worth naming out loud as what you would reach for in new code, even if you write `gather` in the interview because it fits on one line.

The mistake that makes an async scan sequential

```
async def check(url):
    r = requests.get(url)        # <-- blocking, synchronous
    return r.status_code
```

`requests` is not async. This call blocks the **entire event loop** — every other coroutine stops, not just this one. Wrap a thousand of these in a `gather` and you have written a sequential scan with extra ceremony, and it will *look* like it is working.

The tell is timing: concurrency that gives you no speedup at all usually means something in the path is blocking. Use an async client (`httpx.AsyncClient`), and for a blocking library you genuinely cannot replace, push it off the loop:

```
result = await asyncio.to_thread(blocking_call, arg)
```

The same trap applies to `time.sleep()` — in async code it is `await asyncio.sleep()`, and the sync version stops everything.

One client, not one per request

```
async with httpx.AsyncClient() as client:          # once, for the whole scan
    results = await asyncio.gather(*(client.get(u) for u in urls))
```

An `AsyncClient` holds a connection pool and, if you are authenticated, a cookie jar. Creating one per request throws away connection reuse — the thing that makes the second request to a host cheap — and loses session state, so an authenticated scan quietly becomes an unauthenticated one.

This is the async echo of module B1′s session lesson, and it is the same bug in a different costume.

Cancellation is not free

When a task is cancelled, a `CancelledError` is raised **inside** it at its next await point. Two consequences:

- A bare `except Exception` will not catch it — `CancelledError` inherits from `BaseException` in modern Python, deliberately, so that "catch everything" handlers cannot accidentally make a task uncancellable.
- Cleanup belongs in `finally`, not in an except block.

You will meet this the first time a timeout fires mid-scan.

Worked example: the shape you will write repeatedly

```
import asyncio, httpx

async def probe_all(urls: list[str], limit: int = 10) -> list[int]:
    sem = asyncio.Semaphore(limit)

    async with httpx.AsyncClient(timeout=10) as client:
        async def one(url: str) -> int:
            async with sem:
                try:
                    r = await client.get(url)
                    return r.status_code
                except Exception:
                    return 0           # a failure is data, not a crashed run
```

```
        return await asyncio.gather(*(one(u) for u in urls))
```

Read the decisions off it: one client for the pool and the cookies; a semaphore so the target is not flooded; a timeout so a hung host cannot stall a slot forever; per-URL failure handling so one bad host does not lose the run; `gather` so `results[i]` still means `urls[i]`.

What to say out loud

> "This is I/O-bound, so async is the right tool — the requests are mostly waiting, and the event loop can use that time."
>
> "One client for the whole scan, so I keep the connection pool and the session cookies."
>
> "`gather` rather than `as_completed` because I need the results to line up with the input list."

Where this goes next

The semaphore card takes the bound seriously — where the `async with` goes, and the two placements that look right and are not. The drill makes the primitives reflexive. The error-hunt is a fuzzer whose semaphore bounds the wrong thing. Then you write the bounded fetch against a grader that measures peak concurrency rather than taking your word for it.

The semaphore, and where the `async with` actually goes

Why this matters

"Fire N requests concurrently, but no more than K at a time" is a named likely prompt for this interview, and it is the shape of every tool you will write in this course: a parameter fuzzer, a wordlist scan, an id enumeration.

It also has a professional edge that the LeetCode version does not. An unbounded `gather` over a ten-thousand-word list opens ten thousand connections at once. Against a client's production application, on an engagement, that is a denial of service you wrote by accident — and "the scanner took the site down" is the fastest way to end a test early. The bound is not politeness. It is the difference between an assessment and an incident.

The primitive

```
sem = asyncio.Semaphore(10)

async with sem:
    ...        # at most 10 coroutines are inside this block at once
```

A semaphore is a counter with a queue. `async with` decrements it, and if it is already zero the coroutine *waits* — without blocking the event loop — until someone leaves. On exit it increments and wakes the next waiter.

It is not thread-safe, and does not need to be: one event loop, one thread. That is why this costs almost nothing.

Where the `async with` goes, and the two ways to get it wrong

This is the part that separates code that works from code that looks like it works.

```
async def fetch_all(urls, limit, fetch):
    sem = asyncio.Semaphore(limit)

    async def one(url):
        async with sem:              # <-- around the AWAIT, and nothing else
            return await fetch(url)

    return await asyncio.gather(*(one(u) for u in urls))
```

Wrong placement #1 — around task creation.

```
for url in urls:
    async with sem:                       # acquires and releases immediately
        tasks.append(asyncio.create_task(one(url)))
```

`create_task` schedules and returns instantly, so the semaphore is released before any request has been made. Every task then runs at once. This bounds *how fast you create tasks*, which is not a thing anyone needed bounded. It looks correct, it runs, and the concurrency limit is decorative — which is exactly why a grader has to *measure* the peak rather than read the code.

Wrong placement #2 — a release that only happens on success.

```
await sem.acquire()
result = await fetch(url)     # if this raises, we never reach the release
sem.release()
```

One failing host and that permit is gone forever. Enough failures and every worker is waiting on a semaphore nobody will ever release — a deadlock that appears only under the error conditions you did not test. Use `async with`, which releases on the way out however you leave.

Ordering: `gather` keeps it, `as_completed` does not

```
results = await asyncio.gather(*(one(u) for u in urls))
```

`gather` returns results in the order of the awaitables you passed in, not the order they finished. That guarantee is free, and it is why `results[i]` corresponds to `urls[i]` even though the responses came back scrambled.

`asyncio.as_completed` yields each result *as it finishes*, which is what you want for streaming progress and is wrong here — the fast endpoints come back first and your output no longer lines up with your input. A differential test that compares `baseline[i]` against `mutated[i]` after an `as_completed` is comparing unrelated pairs, and it will produce findings that are pure artefact.

If you do need completion order plus the original index, carry the index through explicitly:

```
async def one(i, url):
    async with sem:
        return i, await fetch(url)
```

Failures must not lose the run

`gather` defaults to cancelling everything on the first exception. Two thousand requests in, one connection reset, and the whole scan dies.

```
results = await asyncio.gather(*(one(u) for u in urls), return_exceptions=True)
```

`return_exceptions=True` puts the exception object in that slot and lets the others finish. You then decide, explicitly, what a failed slot means — an `"ERROR"` marker, a retry, a logged skip. The important part is that the decision is visible in your code rather than being made for you by a default.

Handling it inside the worker is often cleaner, because the marker is right where the failure happened:

```
async def one(url):
    async with sem:
        try:
            return await fetch(url)
        except Exception:
            return "ERROR"
```

Timeouts are not optional

A hung host does not raise. It waits, holding a semaphore permit, for as long as the operating system will let it — and your effective concurrency quietly drops to `limit - 1`, then `limit - 2`. A scan that "got slower near the end" is usually this.

```
async with asyncio.timeout(10):
    return await fetch(url)
```

Worked example: the whole shape

```
import asyncio

async def fetch_all(urls: list[str], limit: int, fetch) -> list[str]:
    if limit <= 0:
        raise ValueError("limit must be positive")
    sem = asyncio.Semaphore(limit)

    async def one(url: str) -> str:
        async with sem:                      # around the await, only
            try:
                async with asyncio.timeout(10):
                    return await fetch(url)
            except Exception:
```

```
                return "ERROR"              # one bad host, not a lost run

    return await asyncio.gather(*(one(u) for u in urls))   # input order preserved
```

Eleven lines, five decisions, and each one is a sentence you should be able to say without looking down.

What to say out loud

> "The semaphore is acquired around the await and nothing else — if I wrapped task creation instead I'd be bounding how fast I schedule, not how many requests are in flight."
>
> "`gather` gives results in input order, so the output still lines up with the URL list. If I'd used `as_completed` I'd have to carry the index myself."
>
> "I'm swallowing per-URL failures into a marker so one bad host can't kill a two-thousand-request run, and there's a timeout so a hung host can't sit on a permit."

Where this goes next

The drill after this card fires the primitives at you — which one, and what it returns. The error-hunt is a fuzzer whose semaphore is in the wrong place, which is the failure this card is really about. Then you write `fetch_all` yourself against a grader that **measures** the peak concurrency rather than trusting your code to be right.

Track B

Module B1

One session, or you are testing as a stranger

Why this matters

Access control is tested by comparing what two different people can reach. That comparison only means something if the two sessions stay *separate* and stay *authenticated* for the whole run. Get the session layer wrong and every finding above it is noise: your "IDOR" was a login page, your "403 means it is fixed" was an expired cookie.

This is the substrate for the whole rest of the course. It is worth an afternoon.

A client is not a convenience wrapper

```
r = httpx.get(url)                      # one-shot: new connection, no cookies kept

with httpx.Client() as client:          # a session
    client.get(url)
```

The `Client` carries three things a one-shot call throws away:

- **A cookie jar.** Log in once, and every later request through that client sends the session cookie. This is the whole reason authenticated scanning works.
- **A connection pool.** TCP and TLS handshakes are re-used, which on a two-thousand-request scan is the difference between minutes and seconds.
- **Default headers and auth.** Set `Authorization` once at construction rather than remembering it at every call site — and forgetting it at one.

The failure mode is quiet: a script that creates a fresh client per request *works*, returns 200s, and is silently unauthenticated for every request after the login. You get a scan of the public site and a report that says the application is fine.

Two roles are two clients

For access-control work you need at least two sessions and they must not leak into each other:

```
with httpx.Client(base_url=base) as alice, httpx.Client(base_url=base) as bob:
    alice.post("/login", data={"user": "alice", "pass": "..."})
    bob.post("/login", data={"user": "bob", "pass": "..."})

    own   = alice.get("/api/users/1")     # control: hers
    other = alice.get("/api/users/2")     # the test: Bob's, with her session
```

Two clients, two jars, no interference. The same shape covers an unauthenticated baseline — a third client that never logs in — which is how you tell "requires any login" from "requires the right login", and those are different findings.

Modelling roles explicitly is what lets you build the access matrix from module C1 in code rather than by hand.

Auth carriers, and where each one breaks

Carrier	Set it	Breaks when
Cookie	login response `Set-Cookie`; the jar handles it	you make a new client, or the cookie is `Secure` and you are on http
Bearer token	`headers={"Authorization": f"Bearer {t}"}`	it expires mid-scan, or a redirect crosses hosts
Custom header	`X-API-Key`	a redirect drops it, or a proxy strips it
Basic	`auth=("user", "pass")`	almost never used any more; recognise it, do not expect it

Token expiry mid-scan is the one that wastes an afternoon. Requests start returning 401 halfway through, your detector reads that as "access denied, correctly", and you conclude the endpoint is secure. Re-assert the control request periodically: if the request you *know* should succeed starts failing, your session died and the run is invalid from that point.

The redirect that eats your header

```
client.get(url, follow_redirects=True)
```

Convenient and worth understanding before you switch it on.

- A **cross-origin** redirect drops the `Authorization` header, by design — you do not want your token sent to whatever host a 302 names. So a test that "returned 200 without auth" may have been redirected somewhere that needs no auth at all.
- A **301/302** may turn your POST into a GET. Your carefully-built body is gone, and the 200 you are looking at answers a different question than the one you asked. `307` and `308` preserve the method, which is precisely why they were added.
- The status you see is the **final** one. A 302⟶200 chain reports 200, and the interesting event — that there was a redirect at all — is invisible unless you look at `r.history`.

For assessment work the default should be `follow_redirects=False`. You want to see the 302 itself: where it points, what it set, and what it dropped. Follow it deliberately when you have decided to.

Headers you set on purpose

- `User-Agent` — identify your scan. On a real engagement the client's SOC should be able to tell your traffic from an attacker's, and a header saying so is professional courtesy that also saves you a phone call.
- `Accept` — many APIs return a different body shape for `application/json` versus `text/html`, and you can miss a reflection entirely by asking for the wrong one.
- `X-Forwarded-For`, `X-Original-URL` — client-supplied, sometimes trusted by the application, and therefore test material rather than configuration.

Worked example: the shape B1 is building toward

```
import httpx

def login(base: str, user: str, pw: str) -> httpx.Client:
    client = httpx.Client(base_url=base, timeout=10, follow_redirects=False)
    r = client.post("/account/login", data={"user": user, "pass": pw})
    if r.status_code not in (200, 302):
        client.close()
        raise RuntimeError(f"login failed for {user}: {r.status_code}")
    return client                       # cookies now live in this client
```

Note the failure check. A login helper that returns a client regardless of whether the login worked is how an entire unauthenticated scan gets mistaken for an authenticated one — and the error surfaces four modules later as a finding that will not reproduce.

What to say out loud

> "One client per role, so the cookie jars stay separate and I keep the connection pool."

> "`follow_redirects=False` — I want to see the 302, because following it can drop the Authorization header and turn my POST into a GET."
>
> "I re-check the control request periodically; if the one I know should work starts failing, my session expired and everything after that point is invalid."

Where this goes next

The proxy card puts your traffic and your manual testing on one screen. The header drill makes the common ones reflexive. The listing walks a real login exchange and asks which line established the session. Then you wrap a sender so every call emits a typed `RequestRecord` — the substrate that modules B2 through D3 all log against.

One view of the traffic: routing Python through your proxy

Why this matters

You will test an application two ways at once: by hand in Burp, and with the Python modules you are building here. If those two live in separate worlds you spend the engagement translating between them — "the script says 403, let me rebuild that request by hand and see".

Route the script through the same proxy and the translation disappears. Every request your code sends appears in the history, next to the ones you sent by hand. You can select one, edit it, replay it, and then go fix the script. That loop is the single biggest workflow improvement available in this course, and it costs one keyword argument.

It is also the same habit as reading your own logs in reverse engineering: **look at what your tool actually did**, not at what you meant it to do.

The one argument

```
with httpx.Client(proxy="http://127.0.0.1:8080") as client:
    r = client.get("https://target.example.com/api/users/1")
```

Every request through that client now goes via Burp or mitmproxy on 8080. The proxy sees the real bytes — including the headers your library added that you never wrote.

That last part is worth dwelling on. Your library sets `Accept-Encoding`, `Connection`, a `User-Agent`, and possibly `Accept` on your behalf. When a target behaves differently for your script than for your browser, the difference is usually in headers you did not know you were sending. The proxy is where you find out.

TLS, and the honest version of `verify=False`

The moment you proxy HTTPS you get a certificate error, because the proxy is deliberately performing a man-in-the-middle and presenting its own certificate.

Everyone's first fix:

```
client = httpx.Client(proxy="http://127.0.0.1:8080", verify=False)
```

This works and it turns off certificate verification **entirely**, for every host that client talks to, for the life of the process. In a lab against your own seeded target, that is a reasonable trade and you should still know you made it.

The better fix, and the one to use on anything resembling real work, is to trust the proxy's CA:

```
client = httpx.Client(proxy="http://127.0.0.1:8080", verify="/path/to/mitmproxy-ca-cert.pem")
```

Now verification is on, the proxy works, and a genuinely broken certificate on the target still shows up as an error — which is itself a finding you would otherwise have silenced.

Note what you lose with `verify=False`: any TLS misconfiguration on the target becomes invisible to you, because you told your client not to look.

The `trust_env` trap

`httpx` reads `HTTP_PROXY`, `HTTPS_PROXY` and `NO_PROXY` from the environment by default. Two failure modes, in opposite directions, and both are confusing:

- You set `HTTPS_PROXY` in your shell weeks ago, forgot, and now every scan is routed somewhere you are not looking.
- You explicitly pass `proxy=` and something *else* — a corporate `NO_PROXY` covering the target's domain — quietly exempts your traffic from it, so the proxy history stays empty and you conclude the proxy is broken.

```
client = httpx.Client(proxy="http://127.0.0.1:8080", trust_env=False)
```

`trust_env=False` makes the client use exactly what you passed and nothing else. For a tool whose results you intend to put in a report, being explicit about where traffic goes is worth the extra argument.

The check that settles it: send one request and look at the proxy history. If it is not there, your traffic is not going where you think, and everything you conclude from that run is about a path you have not seen.

Make it a switch, not an edit

Hard-coding the proxy means commenting it out to run without one, which means someone eventually commits the commented-out version.

```
import os

def build_client(base_url: str) -> httpx.Client:
    proxy = os.environ.get("SCAN_PROXY")            # unset -> direct
    return httpx.Client(
        base_url=base_url,
        proxy=proxy,
        verify=False if proxy else True,            # lab trade, made explicit
        trust_env=False,
        timeout=10,
```

```
        follow_redirects=False,
    )
```

`SCAN_PROXY=http://127.0.0.1:8080 python scan.py` routes it; unset runs direct. The `verify` line states the trade in one place instead of scattering it.

What the proxy gives you that a log file does not

You could log every request yourself — and in module B1′s code task you will, into a typed `RequestRecord`. The two are complementary, not redundant:

- **The proxy gives you replay and edit.** Select the request your script sent, change one byte, send it again. That loop is where most manual testing actually happens.
- **Your records give you structure.** Timings, status codes and headers you can diff, count and assert on across a run. A proxy history is for a human; a `RequestRecord` list is for a regression harness.

Use both. They answer different questions.

What to say out loud

> "I route the client through Burp so my scripted traffic and my manual testing share one history — I can replay anything the script sent."
>
> "`verify=False` here because the proxy is intercepting TLS; on a real engagement I'd trust the proxy CA instead, so a genuinely broken certificate on the target still surfaces."
>
> "`trust_env=False` so an environment variable can't silently reroute or exempt my traffic."

Where this goes next

The header drill covers the headers a proxy will show you being sent. The error-hunt is a client that follows a redirect and loses the header it was testing — the failure this card's `follow_redirects` note is really about. Then the code task turns each call into a typed `RequestRecord`, which is the structured half of the pair described above.

> **Note on grading.** Proxy routing is taught and drilled here but not deterministically graded, because grading it needs a second service that records what passed through. That is escalated as `todo-runtime-webapp.md` §2(b); until it exists, the graded code task asserts on the records you emit rather than on the path they travelled.

Module B2

Crawling: the frontier, the scope rule, and three ways to loop forever

Why this matters

Every vuln-class module in Track C starts with "given an endpoint". Something has to produce that list, and what it misses, you never test. A crawler is not glamorous work — it is the thing that decides whether your whole assessment covered the application or covered the front page.

It is also where the two most expensive mistakes in scanning live: going somewhere you were not authorised to go, and never finishing.

The whole algorithm

```
frontier = deque([start_url])          # to visit — popleft() is O(1)
visited  = set()                       # already queued or done — O(1) lookup

while frontier:
    url = frontier.popleft()
    if url in visited:
        continue
    visited.add(url)

    for link in extract_links(fetch(url), page_url=url, scope_host=host):
        if link not in visited:
            frontier.append(link)
```

That is it. Three containers and a loop, and it is exactly the A1 material: a `deque` because you pop from the front on every iteration, a `set` because you test membership on every link.

Everything hard about crawling is in the two functions this loop calls — `extract_links` and the scope rule inside it.

Resolving a link is not string concatenation

A page at `https://shop.example/catalog/shoes/` can contain any of these:

In the HTML	Resolves to
`https://shop.example/cart`	absolute — use as-is
`/cart`	root-relative — `https://shop.example/cart`
`boots`	path-relative — `https://shop.example/catalog/shoes/boots`
`../hats`	parent-relative — `https://shop.example/catalog/hats`
`//cdn.example/x.js`	**protocol-relative** — `https://cdn.example/x.js`

The last one catches people. `//cdn.example/x.js` is not a path; it inherits your scheme and names a **different host**. A crawler that treats it as a path requests `https://shop.example//cdn.example/x.js`; one that resolves it correctly but does not re-check scope walks straight off the application.

```
from urllib.parse import urljoin
absolute = urljoin(page_url, href)      # handles all five, correctly
```

Use it. The hand-rolled version is where crawlers escape.

The scope rule, and the substring test that breaks it

Scope is a professional boundary before it is a technical one. On an engagement you are authorised for named hosts, and a crawler that wanders is a problem no amount of good findings offsets.

The tempting check:

```
if scope_host in urlsplit(link).netloc:      # WRONG
```

`"shop.example" in "evil-shop.example"` is `True`. So is `"shop.example" in "shop.example.attacker.com"`. A substring test admits any host an attacker can name — and an attacker who can get a link onto the page decides where your authenticated crawler goes next.

```
if urlsplit(link).hostname == scope_host:    # right
```

Exact equality. If subdomains are in scope, say so explicitly with a rule you wrote deliberately — `host == scope or host.endswith("." + scope)` — rather than inheriting it from an accident of string containment.

Note `hostname` rather than `netloc`: `netloc` includes the port and any `user:pass@` prefix, and `https://shop.example@evil.example/` has a `netloc` that starts with your scope host and a *hostname* of `evil.example`. That URL form exists specifically to fool people reading left to right.

Three ways to loop forever

Infinite parameters. `/calendar?month=1` links to `month=2`, which links to `month=3`, for ever. Every URL is genuinely distinct, so the visited set never saves you. Bound it: a maximum page count, a depth limit, or a rule that caps how many distinct values of one parameter you will follow.

Session noise in the URL. `/x?sid=abc` and `/x?sid=def` are the same page wearing two names. Dedupe on a *normalised* form — drop known session parameters before the visited check — or your crawler re-walks the site once per request.

Path recursion. A broken relative link produces `/a/b/a/b/a/b/...`, each level a new URL. A depth limit catches it; nothing else will.

The general defence: **always have a hard bound.** A crawler with no page cap is a program with no defined end, and it will find its way to that behaviour on someone else's application.

What a crawler cannot find

Say this out loud in an interview, because it is the difference between someone who has run a crawler and someone who has relied on one:

- Endpoints reached only by JavaScript — `fetch()` calls that never appear as an `href`.
- API routes with no link anywhere. This is most of an API's surface.
- Anything behind a form you did not submit, or a state you did not reach.
- Anything gated behind a role your session does not hold.

That is why crawling is *paired* with wordlist-driven discovery, and why the capstone for this module is parameter discovery rather than more crawling. A crawler maps what the application admits to; a wordlist asks about what it does not.

Be polite, and record it

Rate-limit, identify yourself in the `User-Agent`, and respect the bound. The `asyncio.Semaphore` from A4 is the mechanism. An unbounded crawler against a production application is an availability incident you authored — the same point as A4′s, arriving from the other direction.

What to say out loud

> "Frontier as a deque, visited as a set, scope checked on hostname equality — not a substring, because `shop.example` is a substring of `evil-shop.example`."
>
> "`urljoin` for resolution so protocol-relative links are handled; those name a different host and are the classic way off-scope."
>
> "Hard page cap, because a calendar or a session id in the URL generates infinite distinct URLs and the visited set won't stop it."

Where this goes next

The diffing card covers what makes a response *different*, which is the other half of recon. The scope error-hunt is a crawler that walks off a protocol-relative link. The inventory table has you reconstruct an endpoint/parameter map from a transcript set. Then you write `extract_links` yourself — with the scope rule as the graded property — and the module capstone does wordlist-driven parameter discovery against a live target.

What makes a response different, and what only looks different

Why this matters

Almost every detection signal in this course is a **comparison**. Blind SQLi is a differential. Parameter discovery is a differential. Reflected XSS is "did my marker come back, and in what context". Access control is one session's response against another's.

So the quality of your differ decides the quality of every detector built on it. Get it slightly wrong in the permissive direction and you produce a tool that flags all five thousand words in your list — which is not a scanner, it is a random number generator with a progress bar.

The signals, ranked

Status code. The strongest and the cheapest. A 200 where you expected 404, a 500 where you expected 200, a 302 that appeared. **A status change is always worth looking at**, even when the bodies are identical — a 403 and a 401 with the same error page are different facts about the application.

Body length in bytes. `len(response.content)`, not `len(response.text)`: bytes, no decode, no charset guess. Strong when the page is stable, noisy when it is not — which is why the normalisation below exists.

Body content after normalisation. The real signal. Exact comparison of normalised bodies is usually better than any similarity threshold, because a threshold has a number in it that you cannot justify.

Timing. The weakest, and the only one available for blind injection. It needs a baseline and repetition, because network jitter is larger than most real signals. Module C2 treats this properly.

Headers. Easy to forget and often the tell — a `Set-Cookie` that only appears on one branch, a `Content-Type` that changes from HTML to JSON, a `WWW-Authenticate` on one path.

The noise that changes on every request

Fetch the same page twice, unchanged, and the bodies differ. Every time. Because real pages contain:

- a **CSRF token** — 32 random hex characters, new each request
- a **timestamp** — rendered into the footer
- a **request id** — for their tracing, useless to you
- **your own marker**, reflected back, when you are fuzzing with one

A differ that does not account for these reports every request as interesting. The fix is to normalise both sides before comparing:

```
import re

HEX32 = re.compile(r"\b[0-9a-fA-F]{32}\b")
TS    = re.compile(r"\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:\d{2}")

def normalise(body: str, marker: str) -> str:
    if marker:                                  # empty marker would blank it all
        body = body.replace(marker, "")
    body = HEX32.sub("<TOKEN>", body)
    body = TS.sub("<TIME>", body)
    return " ".join(body.split())               # collapse whitespace
```

Two details in there are load-bearing.

The marker is removed first. If your marker happens to be 32 hex characters, removing it before the token rule keeps it handled as a marker rather than as noise you might have wanted to see.

The empty-marker guard. `body.replace("", "")` is harmless, but any normalisation built around an empty marker tends toward blanking everything — after which every response is identical and your detector never fires again. Silent, total, and it looks like a clean run.

Your own reflection is not a difference

This is the rule that makes reflection-based fuzzing work at all.

You send `?q=CANARY123`. The page comes back containing `CANARY123`, because it echoes the search term. Of course it differs from the baseline — you changed the input. That difference tells you nothing about whether the parameter is *processed* in any interesting way.

Strip your marker, then compare. What is left is the application's behaviour rather than your own input coming home.

The exception, and it is the point of module C3: you *do* care where the marker landed and whether it was escaped. That is a separate question — context analysis — from "is this response different", and conflating the two is how a reflected-XSS detector reports every search box on the internet.

A differ needs a control

Same discipline as the authorised request in C1, arriving in a different costume.

Before you trust a delta, send the baseline **twice** and diff it against itself. If two identical requests already produce a "difference", your normalisation is incomplete and every finding built on it is noise. This costs one request and it is the single highest-value habit in this module.

```
b1, b2 = fetch(url), fetch(url)
assert not is_interesting(b1, b2, marker)     # if this fails, fix the differ first
```

Worked example: what the fuzzer actually does

```
baseline = fetch(f"{url}?nonexistent_param_zzz=1")   # a param nothing handles

for word in wordlist:
    marker = "CANARY" + str(hash(word) % 100000)
    candidate = fetch(f"{url}?{word}={marker}")
    if is_interesting(baseline, candidate, marker):
        report(word)
```

The baseline is a request with a parameter the application definitely does not process — that is the control, and it captures exactly the per-request noise you want normalised away. Anything that then differs from it did so because of the *name* you supplied, which is the finding you came for.

What to say out loud

> "I normalise the CSRF token, the timestamp and my own reflected marker on both sides before comparing — otherwise every request differs and the tool flags everything."
>
> "Status change is always interesting, even with an identical body."
>
> "I diff the baseline against itself first. If that shows a difference, my normalisation is wrong and nothing downstream is trustworthy."

Where this goes next

The drill turns observed deltas into what they imply. The error-hunt is a crawler that walks off-scope. The inventory table has you build the endpoint/parameter map a fuzzer consumes. Then you write `is_interesting` yourself, against a grader whose datasets contain every noise source above plus one real change — and the known-bad it must beat is a differ that forgot to strip its own marker.

Track C

Module C1

Authentication answered; authorization was never asked

Why this matters

Broken access control is the most common serious finding in web application testing, and it is the one automated scanners are worst at. A scanner does not know that invoice 4192 belongs to Bob and not to you. It cannot tell a deliberately public endpoint from a missing check. That judgment is the tester's, which is precisely why the work is worth automating carefully rather than buying off the shelf.

It is also the cleanest possible introduction to the discipline this whole track is built on: **the finding is a comparison, not an observation.**

Two questions, and only one of them gets asked

Every request that touches a protected object asks two separate questions:

	Question	Fails as
Authentication	Who are you?	401 — no session, bad token, expired
Authorization	May you have *this*?	403 — valid session, wrong object

Broken access control is almost always the second question going **unasked**. The application knows exactly who you are. It correctly identified your session, looked up your user id, and then served the object anyway, because nobody wrote the line that compares the two.

That is why "authentication bypass" is the wrong label and worth not reaching for. Nothing was bypassed. The gate did its job; there was no second gate.

Horizontal and vertical

Horizontal — same privilege level, wrong object. Alice reads Bob's invoice. Both are ordinary customers. In API terms this is **BOLA**, Broken Object Level Authorization, and it is API1 in the OWASP API Top 10.

Vertical — crossing a privilege boundary. A customer reaches an admin function.

The distinction is not vocabulary for its own sake, because **the fixes differ**:

- Vertical is fixed by a **role** check: *is this caller an admin?*
- Horizontal is fixed by an **ownership** check: *does this caller own this object?*

A role check does nothing for horizontal, because both parties have the same role — and a team that has just added role-based access control will often believe they are done. They have fixed one of the two.

The user-controlled key

Look at the shape of the endpoint:

```
GET /api/users/{id}
GET /api/invoices/{id}
GET /api/orders/{id}/items
```

Every one of those takes an **object identifier from the client**. CWE-639 calls it a "user-controlled key", and that phrase is the whole diagnosis: the client names the object, and something has to check the client is entitled to name it.

This is why *enumeration* is the test. You change the key and see what comes back. Sequential integers make that trivial; UUIDs make it harder and change nothing about whether the check exists.

The authorised request is half the test

Here is the part people skip, and the reason this module has a no-false-positive ladder.

Suppose you request ids 1 through 10 with your token and get data back every time. What have you learned?

Nothing yet. Two very different worlds produce that result:

1. The endpoint has no ownership check — a real finding.
2. The endpoint is deliberately public — a design decision, and reporting it is a false positive.

The way to tell them apart is the **control**: make the request you ARE entitled to make, and compare. If the authorised request and the unauthorised request are indistinguishable, you have evidence the application is not distinguishing them either. Without that comparison you have an observation, not a finding.

This is the same discipline as the baseline request in a blind injection oracle, and it is the same discipline as the patched-variant check the grader runs against your code. A measurement without a control is an anecdote.

And your own id is not a finding. Reading your own record is the application working correctly. A scanner that reports it has not found a vulnerability; it has found the feature. This sounds too obvious to state, and it is the single most common defect in a first IDOR module — which is why this pack ships a deliberately-wrong solution that does exactly that, and requires the grader to reject it.

Worked example

Alice holds `tok-alice` and owns user id 1. The endpoint is `GET /api/users/{id}`.

```
GET /api/users/1   Authorization: Bearer tok-alice   ->  200   her own record
GET /api/users/2   Authorization: Bearer tok-alice   ->  200   Bob's record
GET /api/users/9   Authorization: Bearer tok-alice   ->  404   no such user
```

Read the three lines together:

- Line 1 is the **control**. It proves the token works and the endpoint serves owners. It is not a finding.
- Line 2 is the **finding**. Same token, an object she does not own, and the application did not object.
- Line 3 is neither. A 404 for an id that does not exist is the object not being there, not a decision about who may have it — although notice that a 404 for an id that *does* exist and that you do not own would be an application deliberately hiding existence, which is a legitimate design.

The finding you write is about the missing comparison, and the evidence is lines 1 and 2 side by side.

Severity, honestly

Severity is a claim you have to defend, and access control is where it gets argued.

What raises it: the sensitivity of the exposed field, whether writes are possible as well as reads, and whether the ids are **enumerable** — sequential integers mean an attacker gets the whole table with a loop.

What lowers it without fixing it: unguessable identifiers. Swap sequential ids for UUIDs and leave the check absent, and the flaw is *identical* — but an attacker now has to obtain an id rather than count to one. Real exposure drops. Report the missing check; be honest that the identifier reduces practical risk.

That honesty is not a concession. It is what makes the severities you *do* assign believable.

Where this goes next

The drill after this card gives you object-reference shapes and asks how you enumerate each. The listing puts this module's actual target in front of you — the vulnerable handler beside the patched one — and asks which line is the check. The error-hunt shows a detector that reports the authorised id too. Then the capstone: detect it against the running service, prove it in `evidence`, and return **nothing at all** against the patched build.

The Finding contract: evidence is the proof, severity is a claim

Why this matters

A detector that returns `True` has told you almost nothing. It cannot be composed into an attack path, it cannot be diffed against last week's run, it cannot be handed to an agent, and it cannot be pasted into a report. The boolean is where the information goes to die.

Everything downstream in this course — the chaining in D2, the regression harness in D3, the MCP tool wrapper in D1 — consumes **structured findings**. So the contract is not paperwork around the interesting part. It *is* the interface that makes the interesting part reusable.

The contract

Enforced-contract tasks import these from the shared `contracts` package, so every module in your toolkit speaks the same shape:

```
Severity = Literal["info", "low", "medium", "high", "critical"]

class Finding(BaseModel):
    vuln_class: str                                  # "IDOR", "SQLi", "XSS", ...
    severity: Severity
    endpoint: str                                    # "/api/users/2"
    method: str = "GET"
    description: str
    evidence: dict = Field(default_factory=dict)   # the proof
```

One source of truth, one import. When a later module chains an IDOR into a privilege escalation, it is reading these fields — not parsing your prose.

`evidence` is what separates a report from a claim

`evidence` defaults to an empty dict. Nothing in the type system will stop you shipping a finding without it, which is exactly why this has to be a discipline rather than a validation error — and why this module's grader asserts on it directly.

The test to apply: **could a stranger reproduce this from the evidence alone, without asking you anything?**

```
Finding(
    vuln_class="IDOR",
    severity="high",
    endpoint="/api/users/2",
    method="GET",
    description=(
        "GET /api/users/2 returns user 2's record to a session authenticated "
        "as user 1. The authorised request to /api/users/1 succeeds identically, "
        "so the endpoint applies no ownership check."
    ),
    evidence={
        "request": "GET /api/users/2  Authorization: Bearer <token for user 1>",
        "status_code": 200,
        "response_excerpt": '{"id": 2, "username": "bob", ...}',
        "control_request": "GET /api/users/1 -> 200 (own record, expected)",
    },
)
```

Four things are doing work there:

- **The request**, so it can be replayed.
- **The status code**, so "it returned something" is a fact rather than a summary.
- **A response excerpt that identifies the other user** — this is the part that proves it was *Bob's* record and not an empty 200 or an error page. A status code alone does not establish that.
- **The control request.** The authorised call, right there beside the unauthorised one. This is what makes the finding an argument rather than an observation, and it is the first thing a developer will ask for.

Redact the credential, keep the shape. Write `Bearer <token for user 1>`, not the token. A report is a document that gets emailed, attached to tickets, and stored for years.

What goes in `description`

State the finding, not the walk to it. A developer reading it should know where to look and what is missing:

> "GET /api/users/{id} applies no ownership check: any valid bearer token can read any user record."

Compare with what a scanner emits:

"Information disclosure detected on /api/users/2."

The second one names an impact and no cause. Nobody can act on it without redoing your work.

Severity is an argument you have to win

Pick the level and be ready to defend it on the axes a client will push on:

- **What is exposed?** An email address and a username is not a password hash.
- **Read or write?** An IDOR that lets you *modify* another user's record is a different finding from one that lets you read it.
- **How enumerable?** Sequential integers mean the whole table with a loop. Unguessable identifiers mean an attacker needs to obtain one first.
- **Who can reach it?** Unauthenticated, any registered user, or one specific role?

For this module's target — email and username, read-only, sequential ids, any valid token — **high** is defensible and **critical** is not, because there is no credential material and no write. Being able to say *why* it is not critical is what makes it believable when you do say critical.

The failure mode is inflation. Mark everything high, and the severity field stops carrying information; the client starts triaging by reading your descriptions, which is the work you were supposed to do for them.

Why a false positive costs more than a miss

This is the discipline the whole track is built on, and it is worth stating in the terms it actually plays out in.

A false positive spends a real engineer's afternoon reproducing nothing. They close it. Do that twice and your genuine critical is the third item in a queue nobody trusts any more. A miss costs you one bug; a false positive costs the weight of **everything else you reported**.

That is why the grader for this module runs your detector against a **patched** build of the same service — byte-identical but for one authorization check — and requires it to return an empty list. Silence against a secure target is a graded outcome, not an edge case.

Where this goes next

The error-hunt after this card shows a detector that reports the authorised id alongside the real ones — competent code that would put a false positive in a client's report. The role-matrix table asks you to write down, per role and per endpoint, what the expected verdict is *before* you test, which is how you avoid deciding what counts as a finding after seeing the response. Then the capstone returns real `Finding` objects with real evidence, and returns nothing at all against the patched build.

Track D

Module D1

Four properties that make a module agent-drivable

Why this matters

The job description asks for something specific: the same web-assessment tasks you solve by hand, **solved autonomously later**. That is not a different skill bolted on afterwards — it is a constraint on how you write the modules in the first place.

A detector that prints a nicely-formatted report to a terminal is finished work for a human and useless to a driver. One that returns a sorted list of typed records is both. The difference is four properties, and none of them costs more than a few lines.

This is also the part of the course where your existing agent-orchestration experience transfers directly. The discipline is the same one you already apply to a tool layer; only the domain is new.

The four properties

Headless. No prompts, no interactive confirmation, no reading from stdin. A tool that waits for input hangs the loop that called it. Everything it needs arrives in the arguments.

Structured output. JSON in, JSON out. Not prose, not a table, not a log line a caller has to parse with a regex. This is what the `Finding` contract from C1 was for; the tool boundary is where those models become plain dicts.

Deterministic. Same input, same output — including **ordering**. This is the one people skip, and §Determinism below is why it matters more than it looks.

Idempotent. Calling it twice changes nothing and returns the same answer. A driver that retries after a timeout must not double-report, and a harness that re-runs your corpus must get the same numbers.

Determinism is mostly about ordering

Two runs of the same detector against the same target find the same three findings. If they come back in a different order each time, three things break:

- **Diffing two runs.** "What changed since last week" becomes noise. Every finding looks moved.
- **The regression harness** in D3. Its whole job is comparing a run against a known-good baseline, and an unordered baseline is not a baseline.
- **An agent's reasoning.** A driver that decided to act on "the first finding" now acts on a different one per run, and its trajectory stops being replayable.

Where does the nondeterminism come from? Almost always from a `set`, a `dict` iteration in an old codebase, concurrent completion order from module A4's `as_completed`, or the order the target happened to answer in.

The fix is one line, and it belongs at the boundary rather than being hoped for upstream:

```
findings.sort(key=lambda f: (f.endpoint, f.vuln_class))
```

Sort on a key you chose, not on whatever the data arrived as. If two findings can tie on your key, add a third component until they cannot.

Failure is data, not an exception

```
# Kills the caller.
def run_tool(request: dict) -> dict:
    return {"ok": True, "findings": detector(request["base_url"])}

# Reports, and lets the caller decide.
def run_tool(request: dict) -> dict:
    try:
        findings = detector(request["base_url"], request["candidate_ids"])
    except Exception as exc:
        return {"ok": False, "error": str(exc), "findings": [], "count": 0}
    ...
```

An exception crossing the tool boundary ends the agent loop. A `{"ok": false, "error": ...}` lets the driver retry, try a different tool, or record the failure and continue — which is the behaviour you want from a scan that hits one broken host out of two hundred.

Note the failure case still returns the **same keys** as the success case. A caller that has to check which shape it got before it can read the result is doing your validation for you, and will eventually get it wrong.

Validate before you act

```
if not isinstance(request.get("base_url"), str) or not request["base_url"]:
    return {"ok": False, "error": "base_url must be a non-empty string",
            "findings": [], "count": 0}
```

Two reasons this matters more for a tool than for a function you call yourself:

- **The caller may be a language model**, which will occasionally send a string where a list belongs, or omit a field entirely. That is not a bug to be offended by; it is an input you have to handle.
- **A validation error should not reach the detector.** Rejecting a malformed request before any network traffic happens is the difference between a clear error and a confusing partial scan.

And the error message is part of the interface. `"base_url must be a non-empty string"` tells a caller what to fix; `"KeyError: 'base_url'"` makes it guess.

Redundant fields must not be able to disagree

```
return {"ok": True, "findings": findings, "count": len(findings)}
```

`count` is derivable from `findings`, so why include it? Because callers want it without deserialising the list — and the moment you include it, it becomes a thing that can be wrong. Compute it from the data at the point of return; never maintain it alongside.

The general rule: a redundant field that can drift is worse than no field. If you publish it, derive it.

Worked example: the whole shape

```
def run_tool(detector, request: dict) -> dict:
    base_url = request.get("base_url")
    ids = request.get("candidate_ids")

    if not isinstance(base_url, str) or not base_url:
        return {"ok": False, "error": "base_url must be a non-empty string",
                "findings": [], "count": 0}
    if not isinstance(ids, list) or not all(isinstance(i, int) for i in ids):
        return {"ok": False, "error": "candidate_ids must be a list of integers",
                "findings": [], "count": 0}

    try:
        findings = detector(base_url, ids)
    except Exception as exc:
        return {"ok": False, "error": str(exc), "findings": [], "count": 0}

    ordered = sorted(findings, key=lambda f: (f.endpoint, f.vuln_class))
    dumped = [f.model_dump() for f in ordered]          # models -> plain dicts
    return {"ok": True, "findings": dumped, "count": len(dumped)}
```

Twenty lines, and every one of the four properties is visible in it. Note `model_dump()`: pydantic objects are not JSON-serialisable, and a tool that returns them works perfectly in your tests and fails the moment a real caller tries to send the result anywhere.

Track D is typed strictly, on purpose

Every task in this track is graded with `mypy --strict`. That is not gratuitous: a tool's signature *is* its contract, and an agent-facing interface with implicit `Any` in it has no contract at all. If the types are vague, the schema you generate from them will be too.

What to say out loud

> "Headless, structured, deterministic, idempotent — and the deterministic part means ordering, because otherwise you can't diff two runs."
>
> "Failures come back as `ok: false` with a message rather than raising, so one bad host doesn't end the driver's loop."
>
> "`model_dump()` at the boundary — pydantic objects aren't JSON-serialisable, and that's exactly where a tool has to hand off."

Where this goes next

The MCP card covers the *schema* half — how a tool describes itself so a caller knows what to send. The drill trains the interface smells. The error-hunt is a tool whose output ordering changes between runs, which is this card's third property failing quietly. Then you wrap a C-track detector as a real JSON tool, and the capstone publishes two of them as a manifest plus a dispatcher.

The schema is the contract: describing a tool to its caller

Why this matters

The previous card made a module *callable*. This one makes it **discoverable**: a caller — a driver, a model, another engineer — has to know the tool exists, what it does, and what to send, without reading your source.

That description is not documentation. It is the interface. If it is vague, a model calls the tool with the wrong arguments, and every failure that follows looks like your tool being broken.

The three fields

Every tool-definition format worth knowing — MCP, the Claude API's tool use, OpenAI's function calling — is the same triple wearing different key names:

```
{
    "name": "scan_idor",
    "description": "Detect horizontal IDOR on an object endpoint by requesting "
                   "each candidate id with one user's session and comparing "
                   "against that user's own record.",
    "input_schema": {
        "type": "object",
        "properties": {
            "base_url":      {"type": "string"},
            "candidate_ids": {"type": "array"},
        },
        "required": ["base_url", "candidate_ids"],
    },
}
```

`name` — stable and machine-readable. Renaming it breaks every caller and every recorded trajectory, so choose it once.

`description` — what the tool does *and when to reach for it.* This is the field people underwrite. A model choosing between six tools reads only the descriptions; "scans for IDOR" does not distinguish itself from "scans for access control issues". Say what it needs, what it returns, and what it is not for.

`input_schema` — JSON Schema. In practice you need five types: `object`, `array`, `string`, `integer`, `boolean`. Anything more elaborate than that is usually a sign the tool should be two tools.

The schema does not validate anything

This is the part that surprises people, and it is worth being blunt about.

A schema is a **description**, not an enforcement mechanism. Nothing between the caller and your function is obliged to check it. A model will occasionally send a string where an array belongs, omit a required field, or invent one you never declared — not maliciously, just as a consequence of generating structured text.

So the schema tells the caller what to send, **and you still validate on arrival**. Both, always. The schema reduces the error rate; your validation handles the residue.

If that feels redundant, compare it to a function signature and an `isinstance` check at a trust boundary. You write both there too, for the same reason.

Sort the manifest, and sort `required`

```
manifest = sorted(entries, key=lambda e: e["name"])
required = sorted(properties)
```

Same argument as the finding order from the previous card, one level up. A manifest built by iterating a dict comes out in whatever order the dict was built, so two runs of your toolset produce two different manifests that describe identical capabilities.

That matters as soon as you want to diff "what tools did this agent have on Tuesday versus Friday", or commit the manifest, or assert on it in a test. An unordered manifest is a file that changes every time you look at it.

Derive it, do not write it twice

The manifest must be built **from the tools that exist**, not typed out beside them:

```
def build_manifest(detectors: dict[str, Callable[..., list[Finding]]]) -> list[dict]:
    return sorted(
        (describe(name, fn) for name, fn in detectors.items()),
        key=lambda e: e["name"],
    )
```

A hardcoded manifest is correct on the day you write it and wrong the first time someone adds a detector — and the failure is quiet: the tool exists, the agent cannot see it, and nobody notices the coverage gap. This is exactly the defect the module capstone's known-bad implements, because it passes every test written against the two tools you started with.

Errors are part of the interface too

A caller cannot recover from what it cannot distinguish. Three failures, three different messages:

```
{"ok": False, "error": "unknown tool: scan_xss",           ...}
{"ok": False, "error": "missing required property: base_url", ...}
{"ok": False, "error": "connection refused",               ...}
```

The first means the driver asked for something that does not exist — it should re-read the manifest. The second means it sent bad arguments — it should fix them and retry. The third is the target's problem — it should retry later or move on. Collapse all three into `"error"` and the driver's only strategy is to give up.

What a tool should not be

Two smells, both common in code that grew from a script:

Stateful across calls. A tool that remembers the last `base_url` and lets you omit it is convenient for a human and unusable for a driver, because call order now changes call meaning. Every call carries everything it needs.

Doing three things. `scan_and_report_and_email` cannot be composed, cannot be retried safely, and cannot be described in one schema. Split it. Composition is the driver's job — which is module D2's subject.

What to say out loud

> "Name, description, input schema — and the description is functional, because it's the only thing telling the caller when to pick this tool over another one."
>
> "The schema describes; it doesn't enforce. I still validate on arrival, because a model will send the wrong shape occasionally."
>
> "The manifest is derived from the registered tools and sorted, so it doesn't change between runs and doesn't go stale when someone adds a detector."

Where this goes next

The drill trains the interface smells this card names. The error-hunt is a tool whose output order changes between runs. The code task wraps a single detector properly. Then the capstone publishes two detectors as a manifest plus a dispatcher — and its known-bad is the hardcoded manifest described above, which is right until a third tool appears.

Module D3

A coverage case becomes a permanent test, or it rots

Why this matters

This is a named responsibility in the job description — *build and maintain a library of regression and benchmark test cases* — and it is the module that turns a pile of detectors into something you can trust next quarter.

The reason is unglamorous. Detectors decay. You refactor the HTTP client and a header stops being sent. You tighten a false-positive filter and it now filters out a real signal. You normalise one more thing in the differ and a detection disappears. **Every one of those failures is silent**: the tool still runs, still reports findings, still looks healthy, and quietly covers less than it did.

A regression harness is the only thing that notices.

What the harness compares

Two runs, and a stored baseline is what makes it possible:

```
baseline  = {target: [endpoints reported]}     # committed, known-good
current   = {target: [endpoints reported]}     # today
```

Four differences fall out, and they are not the same kind of thing:

Difference	Meaning
in baseline, not in current	**regression** — coverage you had and lost
in current, not in baseline	**improvement**, provisionally — see below
target only in current	**new target** — the corpus grew
target only in baseline	**dropped target** — the corpus changed

The whole value is in the first row. The other three exist so the first row stays readable.

A new finding is not automatically an improvement

The harness cannot tell a new detection from a new false positive. Both look identical: an endpoint appears that was not there before.

So a new finding is reported as an **improvement** and it never sets the verdict. `verdict` is `"regressed"` when coverage was lost and `"clean"` otherwise — because a run that gained ten findings and lost none may have gained ten bugs in your filter, and a harness that celebrated that would be lying to you.

Improvements are for a human to look at. Regressions are for the build to fail on. Keeping those two responses separate is the design.

A dropped target is not lost coverage

This is the rule that decides whether anyone keeps using your harness.

Remove one target from the corpus — it was retired, the lab moved, the case was wrong — and a naive harness reports **every finding it used to have** as a regression. Twelve red rows for one routine edit. Do that twice and people start passing `--force`, and the harness stops being consulted at all.

So a dropped target is reported **once**, as a dropped target, and its findings are not counted as regressions. The signal is "the corpus changed", not "the detector broke", and conflating them is how a working alarm gets disabled.

Same instinct as the false-positive discipline in the next card: an alert nobody believes has negative value.

The baseline is a file you commit

```
json.dump(results, f, indent=2, sort_keys=True)
```

Two properties matter. **Sorted**, so writing it twice produces the same bytes and a real change shows up as a small diff instead of a reshuffle. And **committed**, so the baseline moves under

review — updating it is a deliberate act with a reason attached, rather than something a script does silently when it does not like today's numbers.

A harness that rewrites its own baseline on failure is not a harness. It is a program that agrees with itself.

The corpus needs negative cases

A corpus of nothing but vulnerable targets cannot measure a false positive. `return [finding for every endpoint]` scores perfect recall on it.

So the corpus carries targets that are **known clean**, and the patched variant from Track C is exactly where they come from — the same service with the authorization check present. Those cases are what make precision meaningful, and without them your benchmark measures only one half of the thing that matters.

Flaky is the same as broken

A harness that fails intermittently gets ignored, and then a real regression arrives inside the noise and nobody looks. So the inputs have to be deterministic in exactly the way module D1 insisted on:

- ordered output, so comparison is possible at all
- fixed seed data, so the ground truth does not move
- no live third-party dependency in the corpus

If a case cannot be made deterministic, take it out of the harness and test it some other way. One flaky case costs you the credibility of every stable one.

Worked example

```
def check_regressions(baseline, current):
    shared = baseline.keys() & current.keys()

    lost = [{"target": t, "endpoint": e}
            for t in shared
            for e in sorted(set(baseline[t]) - set(current[t]))]
    gained = [{"target": t, "endpoint": e}
              for t in shared
              for e in sorted(set(current[t]) - set(baseline[t]))]

    return {
        "regressions":     sorted(lost,   key=lambda r: (r["target"],
r["endpoint"])),
        "improvements":    sorted(gained, key=lambda r: (r["target"],
r["endpoint"])),
        "new_targets":     sorted(current.keys() - baseline.keys()),
        "dropped_targets": sorted(baseline.keys() - current.keys()),
        "verdict":         "regressed" if lost else "clean",
    }
```

Note `shared` on the first line. Comparing only the targets present in **both** runs is what implements the dropped-target rule — and it is one line, in the right place, rather than a special case bolted on afterwards.

What to say out loud

> "The verdict fails on lost coverage only. A new finding might be a real detection or a new false positive, and the harness can't tell — so it's reported for a human and doesn't fail the build."
>
> "Dropped targets are reported once as a corpus change, not as a wave of regressions. Otherwise one routine edit produces a red run and people stop trusting it."
>
> "The baseline is committed and sorted, so updating it is a reviewed decision rather than something the script does when it dislikes today's numbers."

Where this goes next

The false-positive card explains why precision is the metric a client feels. The metrics drill makes the confusion matrix reflexive. The error-hunt is a benchmark that scores a detector against its own output — the most self-satisfied bug in this course. Then you score a real corpus, and the capstone is the harness itself, with a regression the corpus was built to catch.

Why a false positive costs more than a miss

Why this matters

Say it as a sentence first, because the arithmetic that follows is a way of measuring it rather than a substitute for it:

> A miss costs you one bug. A false positive costs the weight of everything else you reported.

A false positive spends a real engineer's afternoon reproducing nothing. They close it. Do that twice and your genuine critical is the third item in a queue nobody trusts. The findings you were right about stop being acted on — which is the same outcome as not having found them, except you also spent someone's time.

This is why every Track C grader runs your detector against a **patched** build and requires silence. It is not an edge case in the marking; it is the property being marked.

The four cells

	Expected vulnerable	Expected clean
Reported	true positive	**false positive**
Not reported	false negative	true negative

Two ratios come out of it:

```
precision = tp / (tp + fp)      # of what I reported, how much was real
recall    = tp / (tp + fn)      # of what was there, how much did I find
```

Precision is what the client feels. It is the fraction of your report that survives contact with a developer. **Recall is what the engagement is judged on afterwards** — what you missed, discovered by someone else.

Both matter; they trade against each other; and the honest move is to report both rather than the one that flatters the run.

Accuracy is the wrong headline

A corpus of 100 endpoints with 3 genuinely vulnerable ones. A detector that reports nothing at all:

```
accuracy = 97/100 = 97%
recall   = 0/3    = 0%
```

Ninety-seven percent, and it found nothing. Accuracy is dominated by the majority class, and in scanning the majority class is always "not vulnerable". Never lead with it.

Undefined is not zero

This is the one that shows up as a bug in a real harness, and it is a small dishonesty rather than a maths error.

If a detector reports nothing, `tp + fp == 0`, and precision is `0/0` — **undefined**. Return `None`, not `0.0`:

```
precision = round(tp / (tp + fp), 4) if (tp + fp) else None
```

Because `0.0` and `None` say different things to whoever reads the report:

- `0.0` reads as *"it tried and everything it said was wrong"*
- `None` reads as *"it never said anything"*

The second is the truth, and it is a much more urgent problem. A silent detector scoring `precision: 0.0` looks like a tuning issue in a dashboard; scoring `precision: null` looks like the broken thing it is.

Same for recall when the corpus has no positive cases at all: `None`, because recall over nothing is not a number.

A corpus without clean cases cannot measure precision

If every case in your corpus is vulnerable, `fp` is always zero and precision is always 1.0. The trivially useless detector —

```
def scan(target):
    return [Finding(...) for endpoint in every_endpoint]
```

— scores **perfect precision and perfect recall** on that corpus. The benchmark is measuring nothing.

So the corpus must carry **known-clean** cases, and Track C already produced them: the patched variant of every seeded target, byte-identical but for the check. A case whose `expected` list is empty is not filler; it is the only thing in the corpus that can catch crying wolf.

Rule of thumb: if removing all the negative cases would not change your headline number, the headline number is not measuring precision.

Deduplicate before you count
A detector that reports `/api/users/2` three times has not found three things. Count distinct `(target, endpoint)` pairs, or your true-positive count inflates with the noisiness of the implementation rather than its correctness.

The same applies to the report you hand a client: three copies of one finding reads as carelessness and invites them to discount the rest.

A failed case is not a clean case
If the detector raises on a target — connection refused, malformed response, a bug in your own code — that case has to be recorded as **failed**, not scored as "found nothing".

Scoring it as clean quietly improves your precision (no false positive was possible) and quietly damages your recall, and nothing in the output says a case did not run. Keep a `failed_cases` list and report it beside the metrics. "Scored 48 of 50 cases; 2 failed" is a sentence someone can act on.

This is the same distinction as `ok: false` versus an empty finding list from module D1: *I could not look* and *I looked and found nothing* are different facts.

Worked example
Corpus: 5 cases, 4 genuinely vulnerable endpoints across them, 1 known-clean target.

```
detector reports:  /a (real), /b (real), /c (NOT expected)
expected:          /a, /b, /d, /e

tp = 2      fp = 1      fn = 2

precision = 2/3 = 0.6667      recall = 2/4 = 0.5
```

The honest report is both numbers plus the corpus size, and the useful next question is which of `/d` and `/e` was missed and why — a metric is a prompt for an investigation, not a conclusion.

What to say out loud

> "Precision is what the client feels — the fraction of my report that survives a developer trying to reproduce it. Recall is what I get judged on later."
>
> "Undefined precision comes back as null, not zero, because zero reads as 'tried and was wrong' when the truth is 'never reported anything'."
>
> "The corpus has known-clean cases, otherwise precision is always 1.0 and a detector that flags everything scores perfectly."

Where this goes next

The metrics drill makes the four cells reflexive. The error-hunt is a benchmark that scores a detector against its own output, so it cannot fail. Then you score a real corpus — with `None` for undefined metrics as a graded property — and the capstone is the regression harness that decides whether coverage moved.