Security Audit

Look in all three places a real vulnerability hides, prove what you find, and never touch a live system until someone says go.

Produces	Scored private findings, reproducible throwaway PoCs, a coverage-honest report, and — on approval — a client-style fix PR opened but never merged.
Use it when	Someone wants a professional read on whether a repo, PR, tenant, or path is exposed, or wants a specific worry rated, proven, or fixed.
Never	Change anything without approval. Read-only until then, and six human-gates that never cross by themselves.

What it is

A professional security audit is not running a scanner and pasting its output. A scanner reads code and calls that an audit. It misses roughly half of what matters, because half of real findings never appear in git. A database is reachable from the internet because of a firewall rule that lives in Azure, not in a file. A pipeline holds a registry token in cleartext because someone typed it into a variable group. An app registration issues tokens into a URL fragment because of a flow flag set in the tenant. None of that is in the repo. So the audit looks in three places at once: the source, the running cloud, and the identity and CI plane.

What you hand back depends on how far you were asked to go. At minimum, a set of scored findings — each with a CVSS 4.0 vector, a CWE class, an exact `file:line` or resource, and its evidence — reconciled into one report that is honest about what it did not reach. Going further, on the owner's say-so: a reproducible proof-of-concept that stands itself up, demonstrates the exploit and the fix, and tears itself back down. And a fix written as a client-style pull request, opened for review and never merged.

You reach for one of four modes depending on the ask. *Sweep* is the full check-up: find new findings across the whole target. *Review* takes one worry or one candidate and turns it into a scored, written-up private finding. *Poc* proves a finding is real to a skeptic. *Remediate* writes the fix. Sweep is the wide net; the other three act on a single finding, in that order of escalation.

The hard boundary runs through everything. Resolving the target, reviewing, and the entire sweep analysis are read-only. Anything that stands up infrastructure, writes to a client repo, opens a PR, or touches a ticket is gated behind an explicit human approval. Six of those gates are absolute and never crossed by the audit on its own: findings stay private until disclosure is approved; a PoC only ever deploys to a throwaway subscription; every PoC tears down what it stood up; a fix never lands on a client's default branch; a fix PR is never auto-merged; and provenance is verified before anything runs. These are not ceremony. Each one is the scar from a specific way security work goes wrong.

How you'd do it by hand

Strip away the scripts and the craft is the same as what any audit firm does, sequenced. Here it is end to end, with the human action behind each piece of tooling named.

Resolve the target and pin it

Before a single check runs, fix exactly what you're auditing and at what commit. Clone the repo read-only at a pinned SHA and record it, or read the PR diff or the local path in place. Everything normalizes to the same record: kind, root, SHA, platform, scope, provenance. Every finding you write anchors at that SHA, so a `file:line` you cite is true and stays true, and every deep link points at code that can't drift under you.

```
# repo URL  → read-only shallow clone at HEAD, SHA recorded
gh repo clone <owner/repo> -- --depth 1        # or a bearer-authed git clone for
ADO
git -C <clone> rev-parse HEAD                   # record this SHA
# local repo → read in place; PR URL → fetch the diff, scope = changed files
```

The provenance stamp — source, SHA, scope, timestamp, and for a single file a content checksum — is itself the first gate. Auditing the wrong tenant is an incident, so confirming you are pointed at the right, authorized target is not optional. It is the thing you check before you look at anything.

Then, before deep analysis, threat-model the Tier-1 surfaces. Thirty to sixty minutes per surface: draw the data flows, mark the trust boundaries, name the assets, and walk STRIDE against each boundary. This is what turns "here are some shapes" into "here is what an attacker actually does with them." Skip it and you get a bag of pattern hits with no story. The real findings — a token-confusion login flaw, a two-medium chain that composes into a high — come out of the threat model, not the scanner.

The three discovery layers — run all three

This is the center of the method. A finding lives in exactly one of three places, and only one kind of check reaches each. Run one layer and you have missed the other two-thirds by construction.

Layer 1 — source and IaC. Two passes, because pattern-matching and semantic reading catch different things. First the scanners, by hand: `gitleaks` for committed secrets, `osv-scanner` for known-vulnerable dependencies against the lockfile, `checkov` and `trivy` for IaC and container posture, `semgrep` for injection and dangerous-sink patterns. These are fast and exhaustive over their pattern set, and blind to whether a shape actually bites. So the second pass is a semantic deep-read: read the real code and trace attacker-controlled input from an entry point to a sink. That is what catches the defects the patterns can't. A `PATCH /users/:id` that lets a non-admin write their own `role`. A rate-limiter mounted only when `env === 'production'` while the config can only ever read `'prod'` — a dead guard that never fires. A sanitiser that HTML-encodes the JSON envelope and corrupts the stored data. The deep-read fires per surface unconditionally — not only where a scanner lit up — because semantic composition is exactly what the patterns can't see.

```
gitleaks detect  --source <repo> --no-banner
osv-scanner      --lockfile <repo>/<lockfile>
checkov -d <repo> --quiet ;  trivy config <repo>
semgrep --config auto <repo>
```

Layer 2 — live Azure running-state. Query the deployed estate read-only, `list` and `show` only, and flag the state that no code read can see. The recurring finding classes:

- a Postgres server with `publicNetworkAccess = Enabled` or a `0.0.0.0` "allow all Azure" firewall rule
- a single home IP allow-listed into prod
- an engine past end-of-life
- `md5` password encryption where SCRAM is available
- a Key Vault on public network with purge protection off
- a container registry with `adminUserEnabled` (a shared, non-attributable push credential)
- a storage account defaulting to TLS 1.0 or still allowing shared-key access
- Defender left on the free tier so the databases get no anomaly detection
- resources shipping zero diagnostic settings, so failed-auth events are ephemeral and go nowhere

```
az postgres flexible-server list  -o json   # publicNetworkAccess, firewall,
version, params
az keyvault list -o json                    # network ACL, purge protection, access
```

```
model
az acr list --query "[].{n:name,admin:adminUserEnabled}" -o json
az storage account list -o json              # minimumTlsVersion,
allowSharedKeyAccess, network
```

Layer 3 — live Entra and ADO. The identity and pipeline plane. On the tenant, read the app registrations (`az ad app list/show`) for:

- reply URLs still pointing at `localhost`, `http://`, postman, or ngrok on a real app-reg
- implicit or hybrid grant issuing tokens into the URL fragment
- client secrets or certs valid for more than five years
- reply URLs on non-Azure hosts you have to confirm are still owned (an orphaned one is a takeover)
- one app-reg spanning prod and non-prod, where a non-prod compromise mints prod tokens On the build system, read variable groups *and* build-definition variables for values not marked secret that look like tokens by name or shape — the build-definition path is the one a variable-group-only check misses, and it is exactly where a Sonar or registry token tends to sit in cleartext.

The read-only guarantee here is not a promise in prose. Each probe refuses any `az` verb that isn't `list`, `show`, or `account show`; the ADO reads are GET-only, never `-X POST/PUT/PATCH/DELETE`. You can point all three layers at production because they structurally cannot write.

Run the mode, as a manual sequence

The three layers feed whichever mode you're in.

Sweep is the full audit. Walk eight categories in order — secrets, dependencies, IaC posture, auth, transport and headers, storage and data-at-rest, input handling, CI/CD gates — running the scanners and the deep-read across each, then the live probes. Every scripted hit gets triaged: does the shape bite, given the deployment context and any composition? Adversarial verification runs on *every* candidate, not only the interesting ones — light for a low (recompute the checksum, re-read the pinned lines, hold up one alternative reading), heavier for a high. After the flat list is triaged, one dedicated pass looks across all confirmed findings for chains. Two mediums can compose into a high that neither part carries alone. A plaintext storage key in an app setting plus a storage account left internet-reachable over weak TLS yields unauthenticated data access from the internet. Record the chain and its effective severity, not just the pieces.

Review takes one candidate and scores it. Trace the whole path at the pinned SHA — exports, callers, the verifier and the controller that trusts it — end to end before rating anything. State the worst plausible impact assuming the surface is reachable. Then list the exact deployment questions that would move the rating: an edge WAF, the network posture, whether a dangerous env value can reach a live host. Then try to disprove it, re-checking against the pinned library version actually deployed rather than the latest docs. A finding that doesn't hold is re-rated in place with the verification shown, never quietly deleted. Score it with a CVSS 4.0 vector and a CWE, keep it private, route it to a named owner with an SLA tied to severity, and write it up as a GHSA-shaped advisory.

Poc proves it. Pull the evidence by provenance — from the canonical repo at an exact commit, verified against a recorded `sha256`, stored as a pointer and fetched on demand, never hand-copied. One command runs the full cycle and always tears down: verify, up, attack, fix-demo, down. Demonstrate the consequence *and* the fix blocking the same actor — a finding without a shown fix is half a record. When a local model stands in for the real system (Docker networks modelling Azure reachability, a mock IdP), say so plainly and point at the thing that proves the rest. Any credential in the repo is obviously synthetic and labelled. Then a second pass tries to refute the whole thing: did the exploit really run, was the checksum really checked, did teardown really happen.

Remediate writes the fix. On a `security/<sec-nn>` branch, never the default. Read the client's last several merged PRs and match their voice, section shape, labels, and commit convention — this workspace's style does not go on their repo. The PR body describes the fix, not the exploit, and references the advisory by ID rather than pasting any exploit string. Include a regression test that fails without the fix. Run the client's

own test suite, and re-run every verification layer that failed on the vulnerable source — each must now stand down, or the fix is incomplete and the PR does not open. Cite the CVSS vector, include a rollback plan, link the ticket both ways. Then open it. A human merges.

Reconcile, score honestly, and report

Merge every layer's findings into one report: an executive brief, a severity distribution, a per-finding card with evidence and `file:line` or resource, CVSS and CWE. The report footer names which layers actually loaded and which were absent, so it can never imply coverage it didn't run.

If there's a prior audit to compare against, account for every one of its findings. The probes emit neutral class names. A client-supplied map attributes a neutral finding to the client's own ID by matching against the evidence — all the client-specific naming lives in that map file, never in your method. Each prior finding comes back *found* (and which layer found it), *remediated* (verified absent in the live estate), or *gap* (nothing matched — a real hole). Sixty-two known findings in, sixty-two accounted for, or the grid tells you which ones aren't.

Every sweep ends with a coverage claim in a fixed shape: N surfaces swept, M rules applied, K raw hits, J triaged, C confirmed, S stood down, U un-triaged with a reason. On any real sweep the un-triaged count is non-zero — state it. A missing scanner downgrades its check to *not covered* and says so; it never silently becomes "clean."

Why it's built this way

Every guardrail here maps to a specific, expensive failure.

Private-first disclosure. A finding is a loaded weapon until it's fixed. Put its detail in a public issue title, a branch name, or a screenshot and you've handed an attacker a map to an unpatched hole, plus a clock. So findings stay private until the owner approves disclosure, drafted privately and migrated into a private advisory before they circulate. This is why the audit never opens a public artifact carrying finding detail on its own.

Throwaway-only PoCs and mandatory teardown. The point of a PoC is to prove exploitability, and the fastest way to cause the incident you're demonstrating is to run the exploit against the client's real environment. So a PoC only ever deploys to a throwaway subscription that shares no client id, tenant, or database — and every `up` has its `down`, verified gone. A PoC that leaves infra running is a live cost and a new exposed surface. The teardown isn't tidiness; it's the difference between proving a risk and creating one.

Never push to a client default, never auto-merge. The fix belongs to the client's engineers and their process. Landing it on their default branch, or merging it because the ruleset happens to permit admin-merge, takes a decision that is theirs. It also skips the review that catches the fix that breaks their API or reintroduces a bug. The audit opens the PR; a human on their side merges it. Keeping strictly to that line is what keeps the audit a collaborator rather than an unrequested committer.

Provenance verified before any run. A hand-copied snapshot drifts, and an audit of the wrong target is an incident, not a mistake. Pinning the SHA and checking the checksum before anything executes is how you know the `file:line` in the finding points at the code you actually read, and how you know you're looking at the estate you were authorized to look at.

Evidence tiers, stated on every finding. "I traced X and confirmed Y at this SHA" is a different claim from "this reads as though it depends on running state." Confirmed-in-source outranks deployment-dependent, and collapsing the two is how a report loses its credibility — one wrong confident bug and the reader starts discounting all of it. Terraform in the repo is not proof of running infra. When you're not sure, a visible stand-down ("I looked, here's why it isn't exploitable on the deployed version") beats a confident wrong bug every time, because the stand-down builds trust and the false positive spends it.

Honest coverage — never a silent all-clear. "Covered everything" is the most dangerous sentence in an audit, because it hides the scanner that didn't run and the surface nobody reached. The fixed-shape coverage

claim exists so that gap is always visible: what was swept, what was hit, what's still un-triaged and why. A quiet "clean" that skipped a category is worse than a loud "I didn't get to CI/CD, here's why."

A check that mocks the surface a bug lives on proves nothing. This is the through-line. An end-to-end test that stubs the very interface the vulnerability sits behind has proven something about the stub, not the bug. If the finding is about running Azure state, a source read alone can't confirm it — it can only say "deployment-dependent." If the finding is about a login flow, you have to exercise the flow. Proof means touching the real surface, at the right tier of evidence, and saying honestly which tier you reached.

The whole method reduces to this: a security audit is trustworthy exactly to the degree that it looked in all three places, its specific claims are provable at a pinned commit, and it is honest about what it didn't reach. Everything above serves that.

At a glance

```
# Resolve + pin — every finding is true at this SHA
gh repo clone <owner/repo> -- --depth 1 ;  git -C <clone> rev-parse HEAD

# Layer 1 — source & IaC (scanners, then a semantic deep-read per surface)
gitleaks detect --source <repo> --no-banner
osv-scanner --lockfile <repo>/<lockfile>
checkov -d <repo> --quiet ;  trivy config <repo> ;  semgrep --config auto <repo>

# Layer 2 — live Azure running-state (read-only: list / show only)
az postgres flexible-server list -o json     # public access, firewall, version, md5
az keyvault list -o json                      # network ACL, purge protection,
access model
az acr list -o json ; az storage account list -o json   # admin cred, TLS floor,
shared key

# Layer 3 — live Entra + ADO (read-only Graph / pipeline)
az ad app list -o json                        # reply URLs, implicit/hybrid, long-
lived creds
#   ADO variable groups + build-definition vars: GET only, hunt cleartext token
shapes
```

The three layers — run all three, because each reaches findings the others structurally cannot:

- **Source / IaC** (git + scanners + deep-read) — auth logic, mass-assignment, CORS, dead guards, IaC posture, dependency debt.
- **Live Azure running-state** (ARM) — public DBs, weak TLS floors, open vaults, shared admin creds, no diagnostics.
- **Live Entra + ADO** (Graph / pipeline) — implicit-flow app-regs, long-lived secrets, cleartext pipeline tokens.

The four modes — sweep is the wide net; the rest escalate on one finding:

- **sweep** — find new findings across eight categories; triage each; chain the confirmed ones; claim coverage honestly.
- **review** — trace one candidate at the pinned SHA, refute it, score it CVSS 4.0 + CWE, write it up private.
- **poc** — prove it in a throwaway that stands up, shows exploit and fix, and tears itself down.
- **remediate** — client-style fix PR on a `security/` branch, regression test, rollback plan — opened, never merged.

The six human-gates — never crossed by the audit on its own:

- Private-first — no finding detail in any public artifact until disclosure is approved.

- Throwaway-only PoCs — never a client's real or shared environment.
- Mandatory teardown — every `up` has a verified `down`.
- Never push to a client default branch — fixes land on `security/<sec-nn>`.
- Never auto-merge — a human on the client side merges.
- Provenance verified before any run — right target, authorized, checksum checked.

Standing methodology to judge every audit against, as of its own date:

- PTES for the overall shape
- OWASP WSTG for web surfaces
- NIST SP 800-115 for authorization and documentation discipline
- CVSS 4.0 for scoring
- coordinated disclosure for how a finding travels from private to fixed