Capstone: Build Foundry-lite

Agentic Cybersecurity — Module 6

Build the thing that does the audits

Where We Are

The Arc, One Step Left

- Day 1: you **built** ShareBox from a markdown spec, with `pi` on a fast model
- Module 4: you **broke** an agent with prompt injection, and measured the defenses
- Module 5: you **audited** the vulnbox with `large` and proved every finding with a flag

Every step so far: you drove the agent by hand, one session at a time.

This block: build the system that drives the agents.

Why a Harness at All

This morning's audit worked. It also doesn't scale:

- One operator, one session, one target
- Recall depends on your prompting stamina and your checklist discipline
- Findings live in a chat scrollback
- Nothing carries over to the next target

Same answer as every automation problem in security: turn the practice into a system, then improve the system instead of repeating the practice.

Foundry

What Foundry Is

- An open specification from Cisco's ASIG: the shape of an agentic security-evaluation system
- Distilled from building and operating one internally, across several iterations
- **No code in the repo, on purpose.** The internal implementation is coupled to Cisco's infrastructure; the design is what transfers
- ~130 functional requirements with inline rationale, 11 inviolable principles, and every organization-specific choice left as an explicit open question for **you** to answer

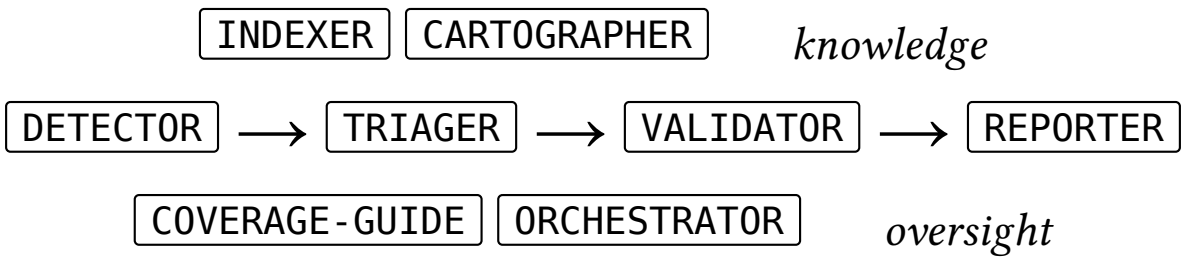
What an Evaluation Harness Needs

Four jobs, whoever builds it:

- **Orchestration**: spawn agents, track liveness, enforce budget
- **Task decomposition**: whole target → claimable units of work
- **Finding verification**: gate every claim on evidence before a human sees it
- **Reporting**: writeups plus a rollup a reviewer can interrogate

You did all four by hand this morning. Foundry pins down what each must guarantee.

The Eight Core Roles



Each role exists to catch the previous one's failure mode:

- Detection without triage: noise
- Triage without validation: plausible fiction
- Validation without coverage: no claim of completeness

Today: **five of these, lite** — a cut §4.2 itself sanctions.

The Substrate

Not agents; the machinery every agent uses:

- **Work queue**: atomic claim, crash-safe release when a holder dies
- **Finding store**: durable, fingerprint-indexed, queryable by every role
- **Sandbox**: egress allowlist and write limits, enforced by infrastructure, not by prompt
- **Budget governor**: spend caps plus yield-based auto-stop
- **Dashboard**: live fleet, findings, coverage, budget

The spec defines these as guarantees, not implementations. SQLite and a loop can satisfy all five.

The Finding Lifecycle

`candidate` ⟶ `verdict` ⟶ `confirmed [exploited?]` ⟶ `published`

- Five verdicts: `true-positive`, `false-positive`, `needs-review`, `not-applicable`, `code-quality`
- Only what survives triage surfaces to a human; the rest is retained with reasoning
- `exploited` is set only by the Validator, after clean-room reproduction. Never inferred, never set by the role that made the claim

The Evidence Gate

The single most important quality control in the spec:

- A `true-positive` verdict must cite code for three legs: **reachability**, **trust boundary**, **impact**
- Every citation is mechanically checked to resolve to real code; a dangling citation demotes the verdict

You met the failure mode this morning: confident, fluent, wrong.

The gate doesn't ask the model to be more careful. It requires the claim to be checkable, and checks it.

The Constitution

Eleven principles, each one a production failure the authors shipped, diagnosed, and wrote down:

- Evidence over assertion
- Surface only what survives
- Exploited means demonstrated
- Sandbox by infrastructure, not by prompt
- The operator outranks every agent

Where the constitution and the spec conflict, the constitution wins. Your plan and tasks get checked against it.

Spec-Driven Development

What spec-kit Is

- GitHub's toolkit for spec-driven development with coding agents
- One workflow: `/speckit.clarify` → `/speckit.specify` → `/speckit.plan` → `/speckit.tasks` → `/speckit.implement`
- The spec is the source of truth; the plan and task list derive from it; `/speckit.analyze` cross-checks all three against the code for drift

The agent still writes the code. The spec decides what the code has to be.

Why a Spec Beats Vibes Here

Day 1 you built ShareBox from a one-page spec and a test suite. That worked because the whole target fits in one session's head.

Foundry doesn't: eight roles, a lifecycle, a substrate, governance.

- Vibes drift between sessions; a spec is checkable at any point
- Decisions get made once and written down, not re-made per prompt
- Tasks derive from numbered requirements, so nothing silently drops

Same idea as Day 1, industrialized. The spec is the contract; the test is whether the system runs and finds.

Working From a Seed

Foundry ships **deliberately underspecified**: ~3 dozen `[NEEDS CLARIFICATION]` markers where the answer depends on your environment.

1. Adopt the constitution; seed the spec into your project
2. `/speckit.clarify`: answer the markers
3. `/speckit.specify`: harden into your spec
4. Iterate until no markers remain, **then** plan, task, implement

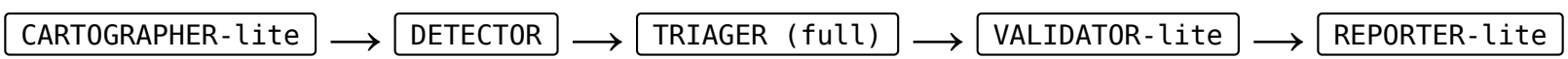
A marker that survives into planning becomes a guess baked into your design.

The Lab

The Cut: Foundry-lite

Full Foundry: eight roles + a substrate. You: three hours.

§4.2 invites the cut: which roles will **you** merge or omit? Answering it in `/speckit.clarify` is the lesson.



Skipped, with the spec's blessing: Orchestrator (a ~20-line runner), Indexer (target fits in context), Coverage-Guide (timebox = done), the §8 substrate, all extensions

Omitting roles: sanctioned. Dropping a gate: never.

FR Walk: Map & Detect

`CARTOGRAPHER-lite` — one doc: endpoints, auth model, trust boundaries (§7.3: the boundary map makes the gate tractable)

`DETECTOR` — two modes:

- **FR-037** — rule sweep: every rule × every function
- **FR-041** — the corpus is a versioned artifact: `rules/rules.md` v1
- **FR-040** — exploratory: the flaws no rule describes
- **FR-043** — candidate = location + class + why + technique
- **FR-044** — candidates never reach humans

FR Walk: Triage — the Crown Jewel

- **FR-050** — one verdict, five values
- **FR-051** — investigate before judging: trace entry to sink
- **FR-052 / FR-087** — three legs, each cited: **reachability**, **boundary**, **impact**
- **FR-088** — citations mechanically resolved; dangling ⟶ `needs-review`
- **FR-053** — "likely real, unproven" = `needs-review`, never TP
- **FR-054** — reasoning recorded, or the store rejects the verdict

Lab 4: a hallucinated token. Lab 5: triaged fiction. Now: build the gate.

FR Walk: Validate & Report

`VALIDATOR-lite` — you have a live target; use it:

- Reproduce the headline impact: a runnable `curl` PoC vs. your vulnbox
- **FR-089** — only the Validator sets `exploited`, never the claimant

`REPORTER-lite`:

- **FR-057** — only `true-positive` surfaces; the rest stays in the store
- Render through your own Lab 2.4 `vuln-report` skill

The Store Is a Directory

Finding store: JSON files in a directory. No queue, no claims, no heartbeats.

`candidate` ⟶ `triaged` ⟶ `validated` ⟶ `published`

- **FR-090** — fingerprint = hash(path, symbol, class); no line numbers, no snippets
- The demo: run it twice on the unchanged target. Zero duplicates, or the fingerprint has a bug

Success = SC-001, Verbatim

> On a target with at least one known seeded vulnerability, an end-to-end run produces a published `true-positive` finding for it with evidence satisfying §7.3, with no operator intervention between up and publication.

Your seeded target: the Lab 5 vulnbox. You captured its flags this morning — you **know** the vulns are there.

Your Three Hours

- **~20 min** — `/speckit.clarify`: answer the open questions with the lite cut
- **~90 min** — build with `pi`: `/speckit.plan` → `tasks` → `implement`
- **~40 min** — run on the vulnbox, twice (the dedup proof)
- **~30 min** — room walk: caught / missed / hallucinated

Behind? Cut breadth (fewer rules, thinner map), never a gate.

Stretch: `exploited` via a live PoC; then your Day-1 ShareBox, or a neighbor's (source only).

Ground Rules

- **Build lab.** Attack nothing. One exception: your Validator may reproduce findings against your own hosted vulnbox instance
- Throttle raised for this block, but **no parallel agent fleets**: run your stages sequentially, back off when the proxy pushes back
- Findings never leave your instance. The vulnbox's bugs are teaching material for the next class

You Hold Ground Truth

This morning you exploited the vulnbox. This afternoon your system audits the same source. Score it honestly:

- **Caught**: surfaced findings that match vulns you proved
- **Missed**: vulns you exploited that it never surfaced
- **Hallucinated**: findings that die on the gate or the live instance

Almost nobody gets same-day ground truth on a security tool they built. You do. Use it.

Build.

Debrief when time is called.

Debrief

The Table

Per student (filled in live): caught / missed / hallucinated on the vulnbox; did the second run dedup cleanly?

Class-wide:

- Which vulns did every implementation catch? Which did none?
- How many hallucinations did the gate stop, and pass?
- Rule-gaps (FR-042): exploratory finds no rule describes?

Compare with your manual Lab 5 numbers: that delta is what you built today.

Beyond This Class

Do You Ever Need to Train a Model?

Almost certainly not.

- Prompting + context engineering covers the overwhelming majority of needs; you spent two days proving how far it goes
- **Fine-tuning / LoRA**: teaches style and format, not knowledge; earns its cost only at volume, with a narrow task, and an eval to prove the delta
- If you can't measure the improvement, you don't need the training

Decision order: prompt → context → tools/skills → bigger model → **then** maybe tuning.

From Demo to Production

What you built this week were demos. Production adds:

- **Cost control**: caps and per-key attribution (you lived under one all week)
- **Logging**: every prompt, tool call, and output; you'll need it for incidents
- **Evals**: your first API script, grown up into a benchmark; regression-test your prompts and models like code
- **The trifecta check**: on every deployment, every tool addition, every MCP server (five minutes, forever)

What to Build First

Back at your desk, in order:

1. Extensions for your actual workflow (the Lab 2.4–2.5 habit)
2. A private eval for one task you care about; pick models with evidence
3. An agentic review pass, with caught/missed/hallucinated tracking
4. Grow today's Foundry-lite build toward the full spec, against a codebase you own

And audit every AI system you meet with the trifecta check. It's the cheapest security control in this entire field.

What You Can Now Answer

- What an LLM is, and why it lies with confidence
- How to make one reliable enough to automate against
- What agents are, how to drive, extend, and **distrust** them appropriately
- How AI-written code fails, and how to find it before someone else does
- What prompt injection breaks, which defenses hold, and the one that actually works
- What an AI security review is worth, because you measured it
- What it takes to turn that review into a system, because you built one

Thanks.

Questions, war stories, and evidence-gate disputes welcome.