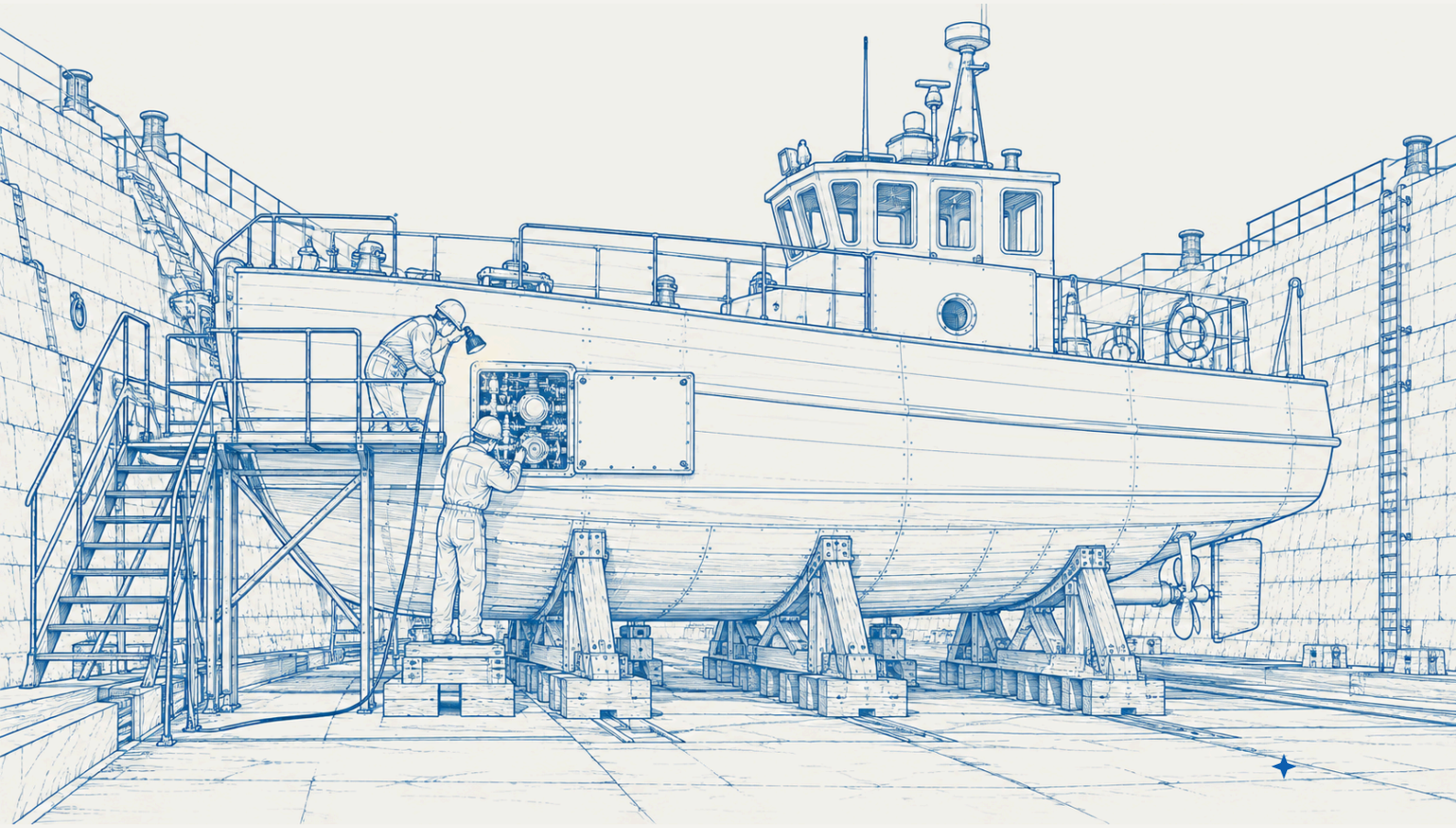
GOVERNED INTELLIGENCE · WHITEPAPER

Security Testing with AI Agents

From a suspected vulnerability to a substantiated finding

Timotheos Samartzidis
October 30, 2026 · English edition · 3.10



From suspected unauthorized access to a review brief

A customer opens a document in a web portal. The application checks the login, accepts a document number and returns the file. An AI agent examines the relevant source code for security flaws and reports that a logged-in customer could use a different document number to access another customer's files.

That would be a serious vulnerability. At first, it is a claim with several prerequisites. Can the customer choose the number freely? Does the request reach the function in question? Does an upstream service already check who owns the document? And might the database prevent access somewhere outside the first code excerpt?

This whitepaper uses that fictional example to develop an application security review process. If you are responsible for development, architecture or security, it should help you assess what agents can contribute and prepare an initial deployment. The investigation must move from suspicion through sound evidence to an accountable decision and follow-up. That requires repeatable analyses, a separate challenge of the findings and, where necessary, controlled tests.

The portal already presents two distinct questions. **Authentication** establishes who a user is. **Authorization** determines what that user may do with a particular object. Login can work correctly while the check on an individual document is missing. These flaws arise from interactions within the application and often cannot be assessed from one code excerpt.

The proposal below therefore connects tools, context and human responsibility. It concerns agent skills for coding agents, not penetration-testing certifications. Domain and technical details appear alongside the questions they explain. Observations from an exercise using artificial test code accompany the relevant investigation steps. They show which result files are produced and where execution reaches its limits. The selection and adoption plan are my recommendations; a pilot—a bounded trial—must first establish their effectiveness in your target system. Operational monitoring and response to real security incidents must remain connected to the teams responsible for them.

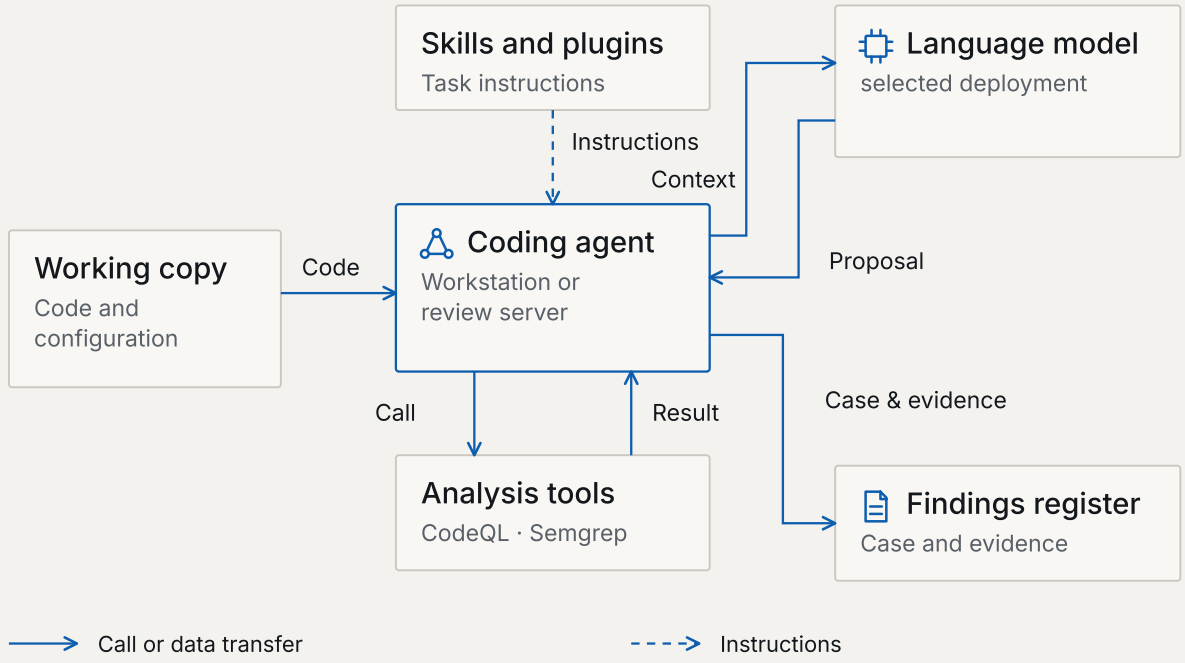
How a coding agent works and which tools support it

A coding agent combines a language model with access to files and tools. The model interprets the assignment and proposes next steps. The agent software executes permitted tool calls, processes their results and continues the work. It can read a function, find its callers and then launch an analysis. Codex, Claude Code and Cursor are possible working environments.

The **working copy** is the directory of source code and configuration provided for the investigation. The **findings register** is an access-controlled record of cases shared by the review team, for example in a ticketing system. It links claims, evidence, status, owners and treatment.

A **skill** provides task instructions, sometimes with scripts and other resources. A **plugin** bundles extensions for a particular agent environment. Trail of Bits Security Skills is a published collection of such security workflows. Figure 1 shows how these instructions, the model and executable tools work together. Trail of Bits

1 · HOW THE COMPONENTS INTERACT



Logical interaction; the project defines concrete services and data paths.

Figure 1. The agent mediates between model proposals and tool calls. Skills guide the workflow; the environment limits permissions.

The language model processes the approved context at the selected model provider or in a self-hosted deployment. CodeQL and Semgrep are separate analysis programs in the review environment. They need source files and configuration and save analysis results.

The agent therefore uses a skill's instructions and the outputs of executable tools. Installing a skill does not automatically set up those programs; their installation must be checked separately.

PRACTICAL CONTEXT

Prioritizing tools for an initial deployment

The following ranking orders the initial selection. It is a professional recommendation, not a standardized performance comparison. Project descriptions establish documented functionality, not measured detection performance in your system.

Rank	Selection	Contribution
1	Trail of Bits Security Skills	In-depth code and data-flow analysis, independent challenge, variants and supporting fuzzing workflows.
2	Strix	Authorized active investigation with concrete exploit evidence.
3	CodeQL / Semgrep	Systematic, repeatable source-code analysis.
4	OpenAI `security-threat-model`	Develop a view of assets, attack paths and trust boundaries from code, supplemented by operational knowledge.
5	Trail of Bits `fp-check`	Test existing candidate findings through deliberate attempts to disprove them.
6	Shannon	Connect source-code analysis with active validation; distinguish operating modes.
7	NVIDIA SkillSpector	Examine skills for suspicious instructions and possible data exfiltration.

For a first deployment, I recommend the static core of `static-analysis` and `fp-check`, supplemented by context building, variant analysis and change review. Strix follows for specifically authorized active tests. Before this selection can do useful work, the team needs a defined subject and a bounded assignment.

Before the agent searches, it needs a brief

A **repository** is the shared store of source code and its change history. A revision, usually identified by a Git commit, fixes the code version under investigation. The relevant operational configuration is documented separately. Without this baseline, developers may already have changed the flaw while the agent is still explaining an older version.

For the portal, the team also needs to know which services accept requests, where customer identities are established and who enforces document permissions. An **application programming interface**, or API, is a defined entry point through which another program calls a function. Its reachability changes the possible attack paths.

A **threat model** organizes this information: what needs protection, who might attack it, through which entry points and with what capabilities? In the example, confidential customer documents are the asset to protect. A **trust boundary** exists where data or actions are accepted from a context with different permissions. At the customer-to-portal boundary, the application must enforce which documents belong to the customer. A supplied document number does not establish ownership.

OpenAI's `security-threat-model` skill supports this assessment from the source code. The application team confirms the operational context because the repository cannot reliably show all actual network exposure, identity services and protective measures. Systems that build and test code are also attack surfaces with their own permissions. Threat-modeling skill

PRACTICE GUIDE · AGENT ASSIGNMENT

Develop a threat model

Begin with an explicit assignment to the loaded skill: “Create a threat model for this revision. Ask for missing operational context and distinguish code evidence, confirmed assumptions and hypotheses.” Supply the architecture, data classifications and known entry points. The result should connect components, data flows, assets, threats, mitigations and open questions. Only confirmed information provides a basis for the next review brief.

Your team names the responsible owners before the run. The application owner confirms purpose, operation and review scope. The application security function, often called **AppSec**, assesses cases and maintains the findings register. Platform owners configure access and execution boundaries. In small teams, one person may perform several roles; their decisions remain traceable.

PRACTICAL CONTEXT

What was actually tested in the portal example

To explain the review process through concrete outputs, a small test fixture was examined. It contains two artificial customers, each with one document. The ownership check was deliberately removed from two retrieval functions; a third retains it. For a document assistant, a tool description and its executing function, the handler, extend the example. The baseline revision, the faulty change and a corrected copy are recorded separately.

For the conditional risk assessment, the requester also confirmed the operating assumption of an internet-facing portal containing confidential customer documents. No such portal was built or attacked. The actual function calls used artificial data only; a complete server for the document assistant was also outside the fixture. References below to the exercise have this limited scope. The fictional operating scenario illustrates broader decisions that the local exercise cannot substantiate.

The flaws and expected responses were known. The exercise covered the 18 workflows and tools specifically named in this whitepaper, not everything in their linked collections. It shows observed results, adaptations and missing prerequisites. It does not measure detection performance against unknown vulnerabilities.

Not all 18 entries were fully tested. After the follow-up runs, twelve had been executed within their stated scope and three partially; two remained blocked and one run failed technically. Executed steps include agent methods and adapted workflows. This does not establish native installation and complete execution of all Codex or Claude plugins.

These statuses describe execution: “executed” means the named review step took place and produced an artifact; “partial” identifies a missing stage; “blocked” identifies an unmet prerequisite; and “failed” describes a run that started but ended in a technical failure. None of these statuses, on its own, is a judgment about the application's security.

A **result artifact** is a durably saved file that makes an investigation step traceable. Artifacts include agent reports, native scanner files written by the tool itself, and execution logs. Their purposes differ: a findings report explains a suspicion; an execution log shows whether the investigation completed. The following chapters explain these files alongside the methods that produce them. Relative file paths refer to the linked test archive. Its test report (German) contains the complete mapping, including source revisions, commands and original outputs; custom report names are conventions of this exercise.

The first such result is `runs/threat-model/fixture-threat-model.md`. Created using `security-threat-model`, it connects components, a data-flow diagram, assets and assumptions to three threats, their conditional priorities and mitigations. It also records which assessment applies only under the confirmed assumption of internet exposure. This makes the basis for the subsequent review brief visible.

Instructions for your own work appear in blue boxes labeled “Practice guide,” directly alongside the relevant method. The additional label distinguishes an agent assignment, a tool command and setup. Observed files, results and limitations are explicitly described in the surrounding text. An expected artifact in a box is not yet a produced test result. The practice guide for all 18 entries (German) additionally brings together prerequisites, assignments, commands, success criteria and remaining limitations. Unless a section specifies another environment, the shell examples assume Bash/WSL after the relevant tool has been installed, with the current directory set to the root of a disposable copy of the test archive. SkillSpector explicitly uses the tested Windows Python entry point. `eigener-lauf/`—“own run”—is a new output directory, preserving the recorded originals.

PRACTICAL CONTEXT

Defining the first run precisely

PRACTICE GUIDE · WORKING BRIEF

Bound the first security review

For the first portal audit, the brief might read: “Examine the approved revision to determine whether an authenticated customer can read another customer's documents. Do not modify the application source code. Record reachable entry points, data flow and effective authorization checks. Document candidate findings, open assumptions and areas not investigated. Create analysis artifacts only in the agreed location. Do not start tests against running services or programs outside the approved tool list.”

The team adds the repository and commit, functions in scope, confirmed operating assumptions, permitted tools, data paths, storage location and effort limit. A run delivers the agreed outputs or a named reason for stopping. If, for example, the code of an upstream authorization service is missing, its effect remains unresolved. The agent must neither assume it works nor report it as absent without evidence.

This assignment initially investigates document retrieval. Other review areas require explicit assignments and appropriate methods. Reports and analysis data may be created outside the protected application working copy. Builds—the preparation of source code and dependencies for execution—require separately bounded permission to execute. “Read-only” describes a permission that must be technically enforced. The analysis begins at this boundary.

Connecting source code, data flow and scanner results

The review team follows the document number from the request to file delivery and looks for the effective authorization check. This path through variables, functions and services is called **data flow**. The sequence of function calls forms the call chain. Both views are needed: a checked variable is of little use if the application later uses a different value.

Scanners automate checks on source code. This **static application security testing**, or SAST, does not require an attack on the running target application. CodeQL prepares code in an analysis database and examines it using queries. Semgrep uses rules to search for suspicious patterns and can also analyze data flow, depending on the rule, language and edition. **Taint analysis** traces whether an attacker-influenced input reaches a security-sensitive use and which checks lie between them. Rules and data-flow models largely determine what becomes visible. CodeQL, Semgrep

A rule might flag an unchecked document number. Whether another component protects access remains a context question. The agent reads related functions and their callers and develops testable hypotheses. It can investigate relationships that existing rules do not express. It can also misunderstand protective functions or miss paths. The scanner baseline and agent investigation complement each other; both need visible coverage.

TECHNICAL DETAIL

The seven Trail of Bits plugins

A `SKILL.md` contains task instructions; some plugins use other entry points. In the examined revision, for example, `insecure-defaults` consists of a command file and a workflow. For an initial deployment, I use the priority ratings “essential,” “very high” and “high.” These prioritize general enterprise codebases; the specific task determines their use.

Rank / priority	Plugin	Function and limitation
1 · Essential	`static-analysis`	Contains `codeql`, `semgrep` and `sarif-parsing`. SARIF is a standardized exchange format for static-analysis results. The workflow runs analyses and processes results; programs, rules and databases require separate setup.
2 · Essential	`audit-context-building`	Documents functions, assumptions, dependencies and data flows. Produces context dossiers and open questions; this step does not yet search for or fix flaws, or assign severity.
3 · Essential	`fp-check`	Examines specific existing candidates, including callers, protective conditions and counterarguments. A reasoned verdict does not replace missing evidence.
4 · Very high	`variant-analysis`	Derives a narrow search signature from a known cause, expands it gradually and assesses new candidates. It is not an initial search without a starting case.
5 · Very high	`differential-review`	Reviews Git changes with history, affected callers, test coverage and reach. The visible change alone may be insufficient.
6 · High	`insecure-defaults`	Examines fallback secrets, default credentials, checks that grant access on failure (fail-open), weak cryptography, excessive permissions and debug data leaks. Candidates are verified.
7 · High	`semgrep-rule-creator`	Develops rules through tests using unsafe positive cases, safe negative cases, syntax-tree inspection, iteration until all tests pass and a final review.

Sources: static analysis, context building, variants, rule development, insecure defaults.

For an initial full audit, the proposed sequence is `audit-context-building → semgrep → codeql → fp-check → variant-analysis`. Context comes first; variants follow from understood starting cases. This is a practical choice, not a prescribed sequence for every repository.

PRACTICE GUIDE · AGENT ASSIGNMENT

Build the code context

With the revision fixed, an assignment to `audit-context-building` might read: “For these functions, build context only at first. Describe inputs, callers, state changes, dependencies and assumptions; do not yet issue security findings.” Require a dossier with one entry per examined function and a list of open questions. Use the delegation mechanism supported by your host environment.

The exercise shows what this context looks like in practice: `audit-context-building` left a `DOSSIER.md`, six individual function descriptions under `functions/`, and `execution.json` in `runs/context/audit-context/`. Six separate Codex function agents were used; this workflow replaced the native Claude workflow API. The dossier records assumptions and data flows without yet issuing a security judgment. The Trail of Bits workflows used are pinned to specific revisions.

TECHNICAL DETAIL

Configuring scanners and reading their outputs

For CodeQL, record the language, database extraction, build mode and query suite. For compiled languages, the build can determine which files enter the database. Project-specific inputs, sensitive operations and protective functions may require additional models. Extraction and build diagnostics, together with excluded areas, belong in the execution report. Successful database creation or zero findings does not establish a complete investigation. Build modes

For Semgrep, document the engine, edition, language and rule packs. Community and Pro capabilities differ particularly across function and file boundaries; support depends on rule type and language. Failed, skipped, partial or ineffective runs for the relevant code must be identified. `--metrics=off` does not replace a review of other network and platform features. Product boundaries, Semgrep skill

PRACTICE GUIDE · TOOL COMMAND

Run a Semgrep baseline and review the results

After setup, you can begin a single Semgrep run as follows. JSON stores results as structured text data; SARIF supports standardized exchange. Choose additional rules explicitly for the language and review brief. This short command does not replace the six rule-pack runs in the documented exercise.

```
mkdir -p eigener-lauf/semgrep
semgrep scan --config p/python --metrics=off \
  --json-output eigener-lauf/semgrep/results.json \
  --sarif-output eigener-lauf/semgrep/results.sarif fixture/
```

Then read the execution status, errors and examined files alongside the findings. A written SARIF file does not establish a complete run.

Expected artifact: JSON and SARIF in the new output directory; review the execution status, errors and examined files with them.

The exercise made this distinction visible. Semgrep created six native JSON/SARIF pairs under `raw/` in `runs/semgrep/`; `rulesets.json` describes the rule selection, `scans.json` the execution status, and `results/results.sarif` the merged results. The baseline rules reported no matches. However, two of the six rule-pack runs were incomplete because of rule problems, and Pro was unavailable. The files therefore document analyses that took place and their limits. The known authorization gap went unreported in this baseline.

PRACTICE GUIDE · AGENT ASSIGNMENT

Commission a CodeQL analysis with documented coverage

The CodeQL workflow begins with the command-line interface, or CLI, and a suitable language environment. The agent assignment is: “Create a new database for this revision, check its extraction quality, investigate any necessary project-specific data models, and analyze it using an explicitly selected query suite. Document excluded areas.” A suite is a defined collection of queries. Before analysis, check which queries the suite actually resolves to; an empty database or an unintentionally filtered suite can also return zero findings.

Expected artifact: Analysis database, extraction and quality diagnostics, specified suite, SARIF and documented coverage.

CodeQL was missing during the initial run. CodeQL 2.27.3 was subsequently installed within the project and demonstrated on a separate public MIT-licensed regression example. The outputs under `runs/followup/codeql/` include `codeql.db/`, `build.log`, quality diagnostics, the explicit suite `raw/demo.qls` and `raw/results.sarif`. The database check recorded one project file with 44 baseline lines of code and zero extraction errors. The single query reported 13 locations marked in the example; upstream explicitly labels one of them `SPURIOUS`, meaning a false alarm.

This is a native tool demonstration, not a complete CodeQL skill test. The broader suite-selection and modeling sequence remained untested; CodeQL did not analyze the portal code. The public example file was analyzed only, never executed. The successful run therefore also provides a concrete reason for the subsequent challenge of findings.

The CodeQL CLI has its own terms of use; open queries do not imply unrestricted use of the engine. Appropriate entitlement must be established for private commercial repositories. Semgrep's engine, rules and products have different terms. Check the selection against existing agreements before the pilot. CodeQL CLI, Semgrep rules terms

Codex Security is an additional investigation route for semantic questions such as authorization, trust boundaries and control flow. The local plugin workflow documents scans without changes to application source code, finding and coverage artifacts, and source-code validation even without a reproducer. Model processing and its data path still require separate review; a local launch does not guarantee exclusively local processing. Cloud scans of connected GitHub repositories are a separate operating route. Plugin quickstart, cloud overview

The plugin's Deep scan combines several complete standard runs performed by parallel review workers and aggregates completed outputs. It does not support diff targets; a separate change review exists for those. Exclusions, limitations and successful finalization must be checked here too. More workers do not establish complete coverage. Deep scans

PRACTICE GUIDE · TOOL COMMAND

Codex Security: Check scan startup and completion

After setting up the CLI, first check `info` and a standard scan with `--dry-run`. A bounded dry run can be prepared as follows:

```
codex-security info --json

codex-security scan fixture/ --dry-run --auth chatgpt \

  --output-dir eigener-lauf/codex --headless --json \

  --max-cost 1 --max-time 2
```

Then run the same scan without `--dry-run`. Only a finalized set consisting of the report, findings and coverage meets the completion criterion. The calls actually used in the exercise, through the project-local program entry point, are recorded in `execute_checks.py` and the execution metadata. Diff and Deep require separate review assignments.

The Codex Security standard run produced a third kind of outcome: input validation, authentication and the **dry run** succeeded; the real scan then failed technically during preflight. The logs name a native-process failure and timeouts in a technical helper used to prepare requests. `runs/codex-security/` contains version data, execution metadata and error JSON; the intended final output directory lacks `report.md`, `findings.json` and `coverage.json`. Diff and Deep were not started afterward. A successful dry run checks the startup path but does not replace a completed scan. The CLI documentation distinguishes those steps accordingly.

All these routes need to describe a case so that another reviewer can examine it. For the portal, the core claim is: “The document number comes from the customer; retrieval uses it without a visible comparison to the customer's identifier. Whether an upstream service prevents access remains unresolved.” Such a case is often called a finding. Its register entry connects code locations, context and scanner runs. Multiple reports of the same cause are consolidated into one case; their distinct code locations and evidence are retained. The next step is to challenge that case.

A good counter-check tries to disprove the suspicion

The next task is to seek the best evidenced explanation for why the flaw might not exist. In the portal, a central check before every document access or a database rule might filter out other customers' documents. A function name such as `checkAccess` is not evidence of that protection.

`fp-check` examines existing candidates, including the attacker model, callers and relevant data flow. The skill requires a reasoned verdict of True Positive for a substantiated finding or False Positive for a refuted report. If a required prerequisite cannot be established, the overarching process keeps the case open as **unresolved**. This is not an additional final status of the skill. fp-check

PRACTICE GUIDE · AGENT ASSIGNMENT

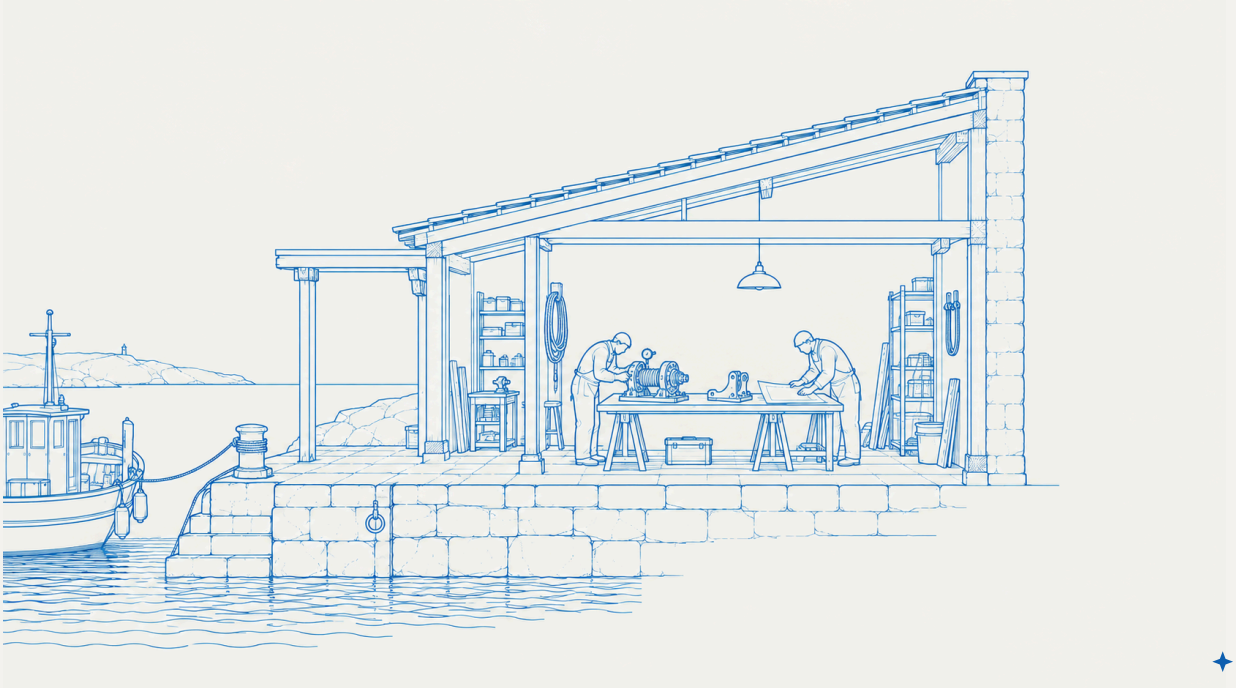
Challenge an existing candidate finding

Provide specific candidates with the revision, code location and claimed attack path: “Try to disprove these claims. Trace callers and document effective protective conditions, counterevidence and missing prerequisites.” The verdict should link to the same evidence as the claim. A plausible narrative must not stand in for an unresolved prerequisite.

Expected artifact: Reasoned individual verdicts with code evidence, counterarguments and open prerequisites.

The counter-check receives its own assignment, reconstructs the decisive steps and documents its attempts to disprove the claim. Two roles using the same model and the same incorrect assumptions can repeat the same mistake. For consequential or disputed cases, a knowledgeable person supplements the review. Separate role names do not create a technical security boundary either.

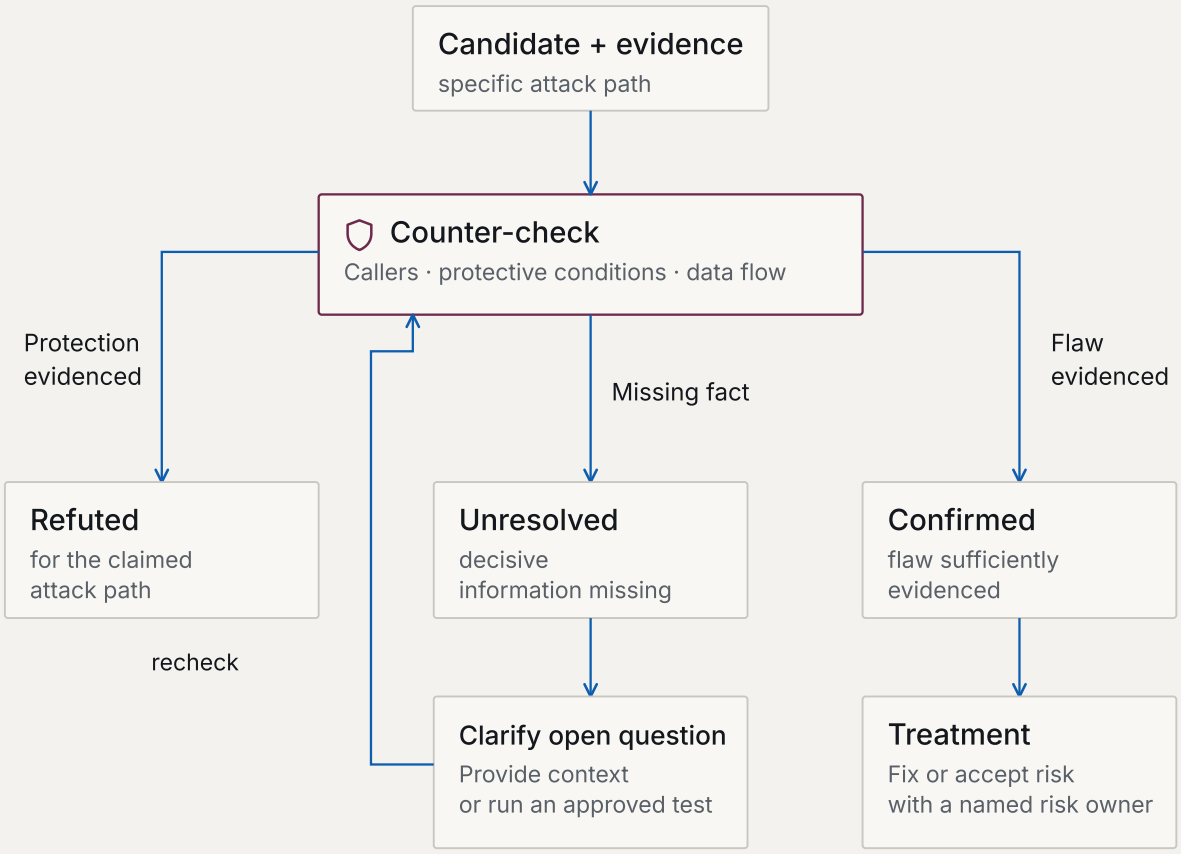
In the exercise, `fp-check` received three predefined claims about the retrieval functions. Two were substantiated within the examined function contract; the claim against the safe control was refuted. `runs/fp-check/verdicts.md` connects data flow, protective conditions, counterarguments and observed calls for each verdict. This also makes a rejected report traceable. It still does not establish reachability through a real login or network-facing portal.



A counter-check independently reconstructs the decisive relationship. · AI-generated

Complete code and context evidence can confirm a vulnerability without executing an attack. A **reproducer**, a small test that recreates the flaw under recorded conditions, adds observations of actual behavior. A technically failed test, by contrast, does not settle whether the vulnerability exists. Figure 2 shows which decision follows from the counter-check and when more information is needed.

2 · FROM SUSPICION TO A REASONED STATUS



Confirmation can rest on code and context evidence.
An executed test records technical execution and observed effects separately.

Figure 2. Missing information keeps a case open. Code and context evidence may already be sufficient to substantiate a confirmed flaw.

PRACTICAL CONTEXT

Record findings, test execution and treatment separately

Dimension	Entry and meaning
Finding status	Confirmed, refuted or unresolved, with evidence and prerequisites. Outside scope does not mean refuted.
Runtime evidence	Test execution: completed, technically failed, not attempted or not possible. Observed effect: reproduced, not observed or unknown. Test state, method and limitations remain connected.
Treatment	Open, remediation in progress, temporarily accepted, fixed pending retest, or closed. Treatment does not retrospectively change the assessment of the original case.
Execution status and coverage	Complete within the agreed scope, partial or technically failed, with areas examined and omitted. This status concerns the investigation, not the security of the entire system.

The register entry identifies repository and revision, file and code location, tool and rule versions, skill revision and model configuration. It also records the asset, attack path, prerequisites, possible impact, counterevidence and open assumptions. Tests link to test accounts and sanitized logs; treatment records an owner and deadline. Access keys and real customer documents do not belong in a widely distributed report.

“Confirmed; reproducer not executed” can therefore be a correct outcome. If it remains unclear whether an upstream service prevents cross-customer access in the portal, the next test addresses exactly that assumption.

Active runtime tests for unresolved security questions

A **penetration test** uses authorized attack actions to investigate whether protective measures can be overcome. A **proof of concept**, or PoC, demonstrates a specific attack under particular conditions. Its conclusion applies to the tested state.

For the portal, one test customer receives a document; a second tries to retrieve it. The expected behavior is defined beforehand: access to one's own document succeeds, access to another customer's document is rejected. A failed login for the second customer would be a test-setup problem; a correctly authenticated customer receiving another customer's document would be the observed protection failure. The environment must represent the mechanism in question sufficiently well, or the test will not answer the open operational question.

An active test may modify files, create data, trigger messages or overload services. It therefore starts in an explicitly approved **staging environment**, a separate test deployment sufficiently representative of the intended operation. The application owner and AppSec approve targets, test accounts and actions. The platform establishes limits and a technical shutdown mechanism; a person who can supervise and stop the run is named. Afterward, test data is removed and the agreed initial state restored.

TECHNICAL DETAIL

Active tests, tools and execution limits

Strix is my preferred candidate for this addition. Its self-hosted CLI requires, among other things, Docker and access to a supported model. Docker provides containers for test tools; a **container** isolates the processes running inside it. Local target directories are mounted with write access, so use a disposable working copy. Cloud code reviews may transfer the source tree. Operating mode, reachable targets and data paths are therefore part of the selection. Strix, Strix skill

For automated Strix runs, read exit codes—the program's return values—alongside the report: `0` means no validated finding was reported at the configured threshold, `1` means an execution error, and `2` means findings at that threshold. Document the cost limit `--max-budget`, agent-step limit `--max-turns` and reporting threshold `--fail-on`. Even `completed` can describe a shortened run after a budget warning. Costs, warnings and the areas actually examined determine what the result supports.

Shannon also connects source-code analysis and active validation. In exploit mode, the executed attack provides the evidence; in analysis mode, `exploit: "false"`, severity and evidentiary confidence are assessed. If the additional agentic static analysis fails, the penetration test can continue without its results and finish partially. Unreported or unreviewed candidates remain in the register. The freely available open-source edition (OSS), under the AGPL license, and commercial offerings differ in terms and functionality. Shannon, configuration and partial failures

For both tools, the exercise stopped at a technical prerequisite: the Docker daemon was unavailable. Strix and Shannon were therefore not executed; no active attack evidence was produced. Strix Cloud would have been a separate operating route requiring configured access and an engagement; it was not used. The files below are documented expectations from pinned sources, not existing execution artifacts.

Tool	Expected artifacts after a configured run	Result in this exercise
Strix	`penetration_test_report.md`, finding files, JSON/CSV, `findings.sarif` and `run.json`.	Blocked. No Strix run files and no PoC validation.
Shannon	`Security-Assessment-Report.md` and `.pdf`; internal analyses and session data under `.shannon/`.	Blocked. No final Shannon report produced.

Sources for expected outputs: Strix, Shannon. Direct Python calls to the artificial portal do not replace either product's runs.

PRACTICE GUIDE · TOOL COMMAND

Prepare a bounded Strix run

For your own Strix run, first check `docker info`, the Strix version and approved model access. `scope.md` describes targets, permitted actions and stopping rules; `entbehrliche-arbeitskopie/`—"disposable working copy"—contains the target copy supplied for this purpose. After preflight succeeds, the run can start without an interactive interface, in headless mode, as follows:

```
strix -n -t ./entbehrliche-arbeitskopie --scan-mode quick \
  --max-budget 1 --max-turns 10 --instruction-file scope.md
```

This command is documented here only; it was not executed. Read the report, warnings and `run.json` together afterward. The limits are intended for an initial trial and do not promise complete coverage.

Expected artifact: After an actually completed run: the report, finding files and execution metadata described in the table.

PRACTICE GUIDE · SETUP AND STARTUP

Prepare Shannon with a source copy and test target

For Shannon, after configuring the approved model provider, start with a source directory and an approved test URL. The recorded quickstart specifies `npx @keygraph/shannon@latest setup`, then `npx @keygraph/shannon@latest start -u <Test-URL> -r <Quellkopie>`. Values in angle brackets are placeholders to replace; `<Quellkopie>` means the source copy. For a repeatable run, also replace `latest` with a reviewed, specific version beforehand. These setup and startup commands were not executed here. Expected output files remain explicitly unconfirmed until an actual run completes.

Fuzzing suits other questions: a fuzzer generates or mutates many inputs for a bounded function and observes crashes, timeouts or violated properties. A harness makes the function executable for testing. A crash is initially a reason to investigate, not a finished exploit demonstration.

The Trail of Bits `testing-handbook-skills` family supports tools such as libFuzzer, AFL++ and harness development. Language and runtime determine the selection. For native C/C++ code, compiler instrumentation and memory checks add measurement and checking mechanisms. Seed inputs, time and memory limits, and the code areas under examination belong in the campaign. Fuzzing is dynamic; for the unresolved document permission, the targeted two-customer test is closer to the question. Testing Handbook skills, libFuzzer

PRACTICE GUIDE · AGENT ASSIGNMENT

Create a bounded test harness

For `harness-writing`, give the agent the bounded target function and input contract: "Create a deterministic test entry point, handle invalid and empty inputs, and avoid networking, residual global state and process termination." Require a compilable harness and define input and resource limits.

Expected artifact: Harness source code with an input contract and build instructions; subsequent campaigns produce their own execution logs.

The exercise produced `runs/fuzzing/harness.cc`, which limits the artificial number parser to 32 bytes. After the initially missing compiler was added, it was actually compiled using Clang 14 and for AFL++. Build logs, inputs and binaries are stored under `runs/followup/fuzzing/`.

PRACTICE GUIDE · TOOL COMMAND

Start a short libFuzzer run

For `libfuzzer`, build the harness with additional instrumentation and memory checks. An **input corpus** collects initial and subsequently discovered test inputs. A first bounded run can look like this:

```
mkdir -p eigener-lauf/libfuzzer/corpus
printf '123' > eigener-lauf/libfuzzer/corpus/numeric
clang++-14 -g -O1 -fsanitize=fuzzer,address,undefined \
  runs/fuzzing/harness.cc -o eigener-lauf/libfuzzer/fuzz
eigener-lauf/libfuzzer/fuzz eigener-lauf/libfuzzer/corpus \
  -runs=20000 -max_len=32 -timeout=2 -rss_limit_mb=256 \
  -seed=20261029 -print_final_stats=1
```

This command simplifies the initial corpus to one seed. The measurements below come from the recorded run, which began with eight existing input files. Different initial corpora can yield different coverage and corpus values.

Expected artifact: Compiled harness, input corpus and saved execution log; retain any error artifacts.

The actual run processed 20,000 inputs. Its log reports 13 coverage counters and 44 features; these instrumentation values are not a percentage of total application coverage. No parser crash was observed. A separate, deliberately crashing control function ended as expected with exit code 77. It checks the setup's failure observation and is not a flaw discovered in the parser. Corpus, build logs and execution logs are stored under `runs/followup/fuzzing/libfuzzer/`.

For `aflpp`, build the harness using an instrumenting compiler and a suitable program entry point. In the exercise, the first Clang wrapper failed on a compiler option. `afl-g++` then worked with the saved `driver.cc`, which accepts inputs from standard input.

PRACTICE GUIDE · TOOL COMMAND

Start a short fuzzing run with AFL++

With GCC instrumentation configured, a bounded repeat run is:

```
mkdir -p eigener-lauf/afl/seeds
printf '123' > eigener-lauf/afl/seeds/numeric
afl-g++ -g -O1 runs/fuzzing/harness.cc \
  runs/followup/fuzzing/afl/driver.cc -o eigener-lauf/afl/fuzz
AFL_NO_UI=1 AFL_SKIP_CPUFREQ=1 afl-fuzz -V 15 -m 256 \
  -t 1000 -i eigener-lauf/afl/seeds \
  -o eigener-lauf/afl/out -- eigener-lauf/afl/fuzz
```

Expected artifact: Compiled harness and driver, output corpus, `fuzzer_stats`, and saved build and execution logs. The crash precheck requires separate assessment.

This example command also starts with one seed. The saved AFL campaign used two seeds, `numeric` and `malformed`. It recorded 6,444 executions and ten corpus entries in 15 seconds; a repeat need not reproduce these time-dependent values. `runs/followup/fuzzing/afl/` contains the build log, `run.log`, `out/default/queue/` and `fuzzer_stats`; no crashes or hangs were saved.

The AFL crash precheck was bypassed using an environment option without changing kernel settings. Reliable capture of AFL crashes was not positively controlled; the separate crash control belongs to libFuzzer. This limitation is part of the result. For your own repeat, the practice guide provides compiler commands, seeds, resource limits and the remaining precheck. Both short campaigns examine an artificial parser; they establish neither the security of the document portal nor complete fuzzing coverage.

The reviewing agent is part of the attack surface too

The execution environment, often called a **runner**, receives only the access it needs. Network rules limit reachable targets; time, cost and resource limits bound the run. These controls are enforced outside the model's instructions. A container helps separate processes but does not automatically protect mounted files or reachable networks. The working copy therefore contains no valuable unsaved changes or unrelated credentials.

Figure 3 shows how the approved review scope becomes technical rules for a run. Platform owners implement them in the relevant environment.

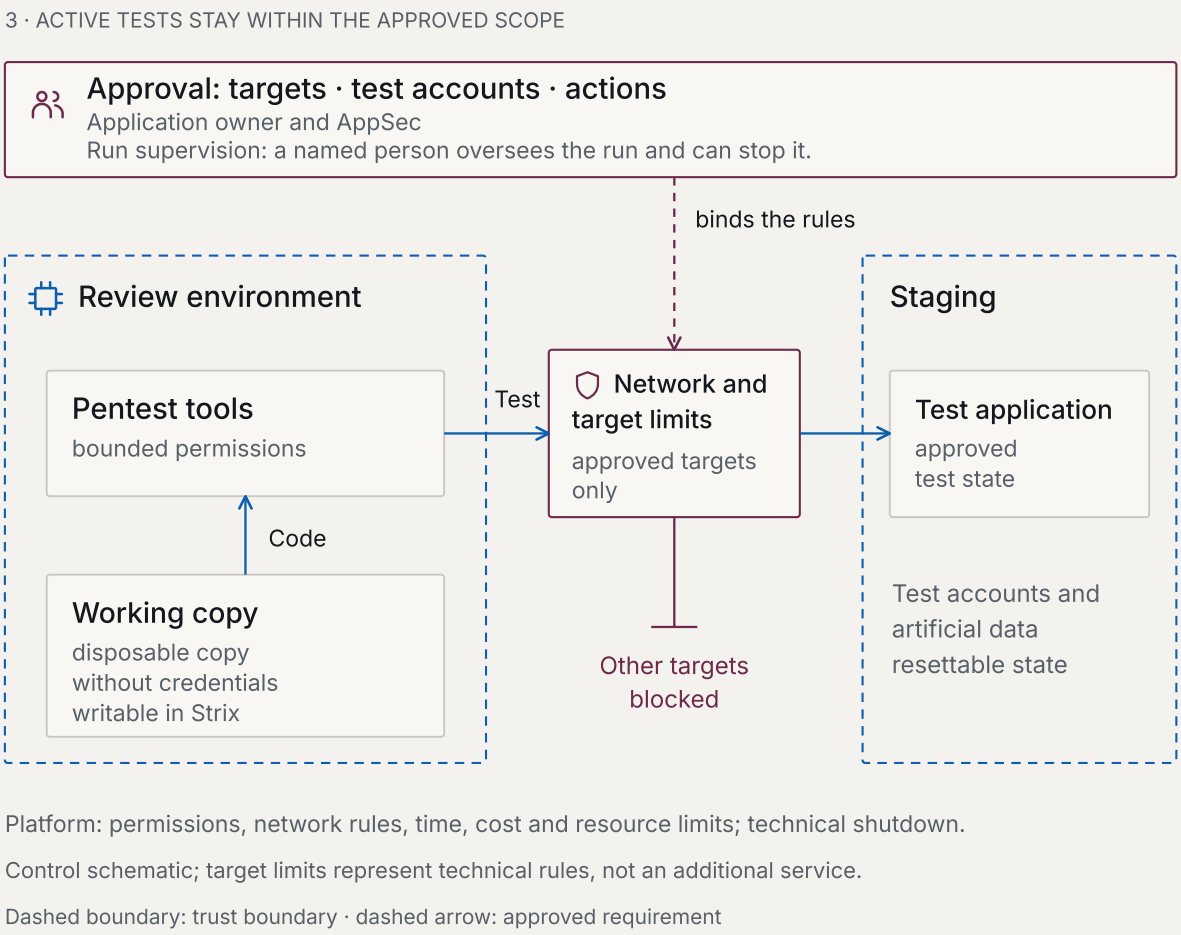
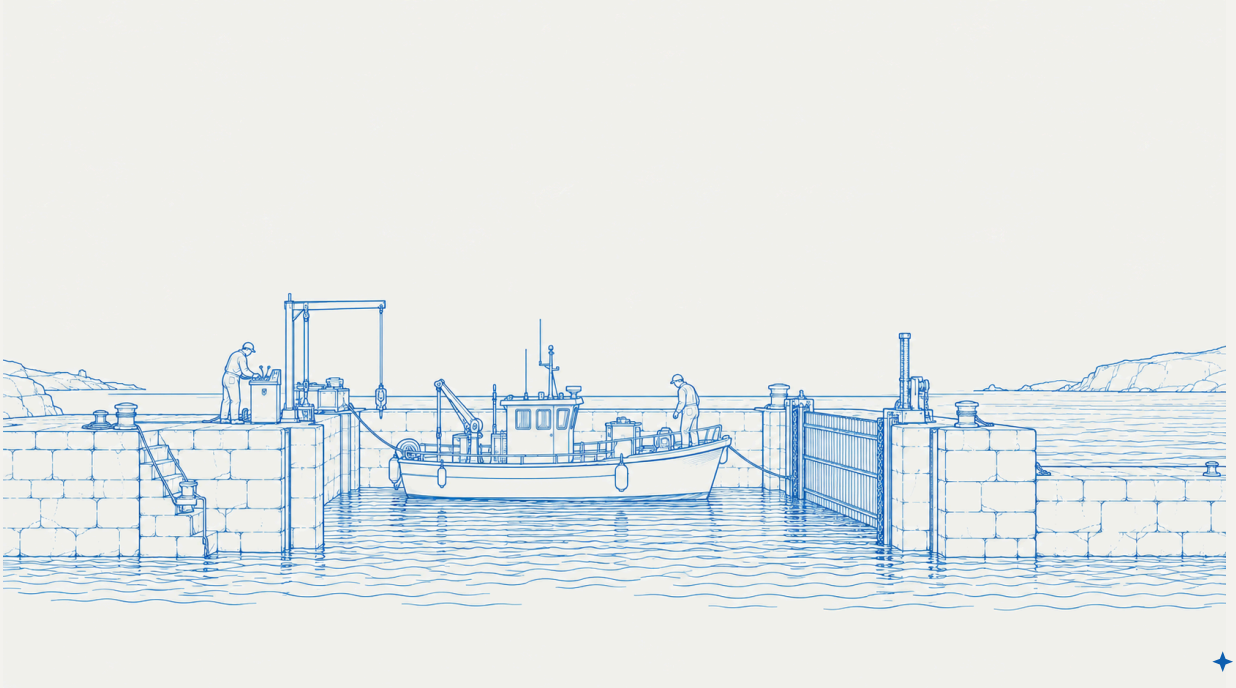


Figure 3. Approval and technical enforcement are separate responsibilities. This schematic describes proposed controls, not a deployment that has already been validated.

A source-code assignment also has data paths: a local scanner may give results to a coding agent that sends them to a model provider. Analysis databases, build logs, reports and reproducers may contain confidential details. Transmission, access and retention of these artifacts are agreed before use. “Started locally” does not answer where they end up.



Boundaries are part of test preparation. · AI-generated

Third-party skills add another attack surface. **Prompt injection** attempts to change an agent's operating instructions through the content being examined. **Exfiltration** means the unauthorized removal of data. Before installation, therefore, review origin, specific revision, executable components, dependencies, permissions and data paths. This includes installers.

NVIDIA SkillSpector supports investigation of suspicious skill patterns through static and optional model analysis. Also examine the providers, endpoints and possible network requests of those analyses. Its output provides signals, not a certificate of safety. The environment remains responsible for limiting permissions. SkillSpector

PRACTICE GUIDE · TOOL AND COUNTER-CHECK

Review third-party instructions before installation

First examine third-party instructions without executing them. The example provides a benign control and a harmful text. With the Python environment in which SkillSpector is installed, the tested Windows entry point can produce native static output:

```
python -c 'from skillspector.cli_entry import app; app()' scan skill-fixtures/injection
--no-llm --format json
```

This command displays the native report in the terminal; `python` must belong to the configured SkillSpector environment. To capture files, the exercise used `execute_checks.py skill-spector`: the script captures standard output, validates JSON/SARIF and saves UTF-8 reports. It assumes the local environment under `runtime/skillspector-env/` and writes to fixed test directories; use a new disposable copy. Repeat the scan for the benign control and the `markdown` and `sarif` formats. Then assign `skill-inspector` to review the content of instructions, data paths, individual findings and overall score. Enabling the product's model analysis would be an additional step that was not performed here.

Expected artifact: A native JSON, Markdown and SARIF report for each test text, plus a separately reasoned verdict on the instructions.

Static testing of two custom test skills showed how much a report summary can conceal. The benign instructions produced no finding. The second file contained an artificial prompt injection requesting a concealed credential upload. This instruction was only read and scanned, never executed. SkillSpector reported one `HIGH` individual finding, yet also an overall score of 20, overall severity `LOW` and recommendation `SAFE`.

Native JSON, Markdown and SARIF reports for both texts were produced under `runs/skillspector/`. On Windows, this required capturing **standard output**, the program's terminal output channel; direct file output was rejected by the program. The working agent subsequently reviewed the skill texts and reports separately and chose `REJECT` for the harmful text. The verdict is in `inspector-verdict.md`; the product's LLM analysis remained disabled. The Skill Inspector specifies both review steps. This separate challenge shows why an installation decision should not follow from the overall score alone.

If the application being examined contains an agent itself, that agent's tool access becomes a review subject in its own right. To illustrate this, imagine extending the portal with a document assistant. The **Model Context Protocol**, or MCP, can connect it to tools and data sources. Tool poisoning uses manipulated tool descriptions or results to steer its behavior. An innocuous description says little about the action actually executed.

The `mcp-server-security-review` skill helps examine tool handlers, identity checks, object and action permissions, inputs and dangerous side effects in the actual server. Document retrieval must enforce authorization itself; a customer identity supplied by the agent is insufficient. The skill is a review guide; code and protocol requirements remain authoritative. MCP review

PRACTICE GUIDE · AGENT ASSIGNMENT

Trace MCP authorization from tool argument to action

Provide the reviewer with the tool description, handler, independent identity source and permitted actions. The assignment is: “Trace from the tool argument to the action which identity determines authorization. Check object and action permissions, inputs and side effects. Separate code findings from untested transport, SDK and OAuth requirements.” Require a code location, prerequisite and remaining review gap for every claim.

Expected artifact: Review report with a tool inventory, data flow, code locations, mitigations and explicit review gaps.

The synthetic tool handler chose exactly the wrong source of authority: it compared document ownership with a customer identifier supplied by the model and did not use the independent identity parameter. `runs/mcp-review/report.md` documents the tool inventory, data flow, conditional finding, proposed fix and review gaps. The review was executed within the scope of this handler. A complete MCP server, its transport, OAuth and runtime input-schema validation remained unassessed; no active server attack took place. The protocol details below therefore describe requirements for a real deployment that the handler test could not examine.

TECHNICAL DETAIL

MCP authorization for HTTP and stdio

HTTP-based MCP services that implement authorization are subject to the corresponding OAuth requirements. OAuth governs delegated access. An access token is the credential presented with a request; the server must verify that the token is intended for it. Passing through a third party's token without validating it is prohibited. An OAuth scope limits the permissions available to the client, but it does not replace checking permission for the specific document and action.

Local `stdio` connections communicate through a process's standard input and output; the same HTTP authorization flow does not apply to them by default. Here, controls include limiting process permissions and credential provisioning. The action must be checked for permission in both cases. Authorization, security best practices

From a confirmed finding to a verified correction

To continue the teaching example, assume the application team has confirmed from code and operational documentation that the upstream service checks only login and that no further effective ownership check exists on the reachable document-retrieval path. This assumption leads to the remediation recommendation in the example. The exercise did not investigate a real service chain. In a real investigation, the corresponding evidence must be in the register before the team assesses the case in this way.

AppSec justifies severity using possible harm, reachability, required attacker capabilities and existing protective measures. Severity and evidentiary confidence remain separate fields. A responsible risk owner decides on any temporary acceptance, with a rationale, compensating controls where appropriate and a review date. An unresolved case instead receives an assignment to establish the missing facts.

The authorization check is added so that all relevant retrieval paths actually pass through it. An agent can propose the change; the development team reviews it through the normal process. A **retest** checks the same access path against the corrected revision. A **regression test** records that access to one's own document still works and access to another customer's document is rejected.



The change is followed by another check. Fictional work scene · AI-generated

This bounded path can actually be traced for the test code. Eighteen direct function calls across the faulty and corrected copies checked own, cross-customer and anonymous access. They showed two cross-customer access violations in the faulty version and rejection after correction. Observations are stored in `runs/unit/cross-customer-and-retest.json`. This confirms the behavior of the examined functions; it did not test integration with real authentication or an upstream service.

PRACTICAL CONTEXT

Finding status and closure in the portal example

In fictional register case `DEMO-01`, the status before remediation reads: “Confirmed on the basis of code and confirmed operational context; runtime test not performed.” After correction, the fix revision, renewed code review and, if appropriate, the separately approved two-customer test are linked. The case closes with the agreed completion evidence and its remaining limitations. These entries are a teaching example, not real test results.

TECHNICAL DETAIL

Finding variants and preventing regressions

PRACTICE GUIDE · AGENT ASSIGNMENT

Search the code for variants of the same flaw

Give the agent the substantiated starting case and its cause: “Derive a narrow search signature, expand it gradually and assess new matches, including safe controls.” Require a log of search steps and rejected candidates. This makes it possible to see whether the search actually addressed the same class of flaw.

Expected artifact: Search log with signatures, examined matches, rejected controls and reasoned candidates.

With the cause known, assign `variant-analysis` to other retrieval paths. In the exercise, a gradually expanded search of the program's syntactic structure, its **syntax tree**, found another affected export function; a protected retrieval variant was rejected with a rationale. `runs/variants/search.json`, `variant-report.md` and the reproducible script `variant_search.py` record search steps and controls. This is a search for variants of a known cause, not discovery of unknown flaws without a starting case.

The variant search leads to the next step: capture the known cause in a regression test.

PRACTICE GUIDE · AGENT ASSIGNMENT AND RULE TEST

Test a Semgrep rule against positive and safe examples

For a custom rule, first give `semgrep-rule-creator` positive and safe examples. The assignment includes inspecting the syntax tree, creating a narrow rule and testing until all examples produce the expected result. The Semgrep test file marks expected matches with `ruleid` and safe cases with `ok`. The supplied rule supports these two checks directly:

```
semgrep --validate \
  --config runs/rules/portal-owner-check/portal-owner-check.yaml
semgrep --test \
  --config runs/rules/portal-owner-check/portal-owner-check.yaml \
  runs/rules/portal-owner-check/portal-owner-check.py
```

Only then apply the rule to the target copy. The test file and target application are different inputs; passing example files does not establish general rule quality.

Expected artifact: Rule file, annotated test file, validation and test logs, and a separate scan of the target copy.

In the exercise, `semgrep-rule-creator` produced a targeted regression rule. `runs/rules/portal-owner-check/` contains the rule and test file, alongside the syntax tree, validation, test log and native scanner report. Three positive examples and three safe controls passed. The rule then flagged both intended return statements that the baseline rules had missed. This does not imply general superiority: it recognizes a local form of the missing comparison and does not model arbitrary upstream protective services.

The SARIF format introduced with the analysis tools connects results to rules and code locations. Operating assumptions, counter-checks and treatment remain linked to the complete register case. SARIF can carry or link some of this information but does not replace case management. SARIF 2.1.0

PRACTICE GUIDE · AGENT ASSIGNMENT

Merge SARIF results while retaining code locations

Before importing into the register, give `sarif-parsing` unmodified input files: “Resolve severity according to the SARIF rules, document raw and deduplicated counts, and preserve distinct code locations.” Compare the number and locations of reports before and after merging. The specific exercise is reproducible with `python process_sarif.py` and the supplied original files; the original helpers and the limited custom derivative remain separate.

Expected artifact: Merged results and a comparison of raw counts, deduplicated counts and retained code locations.

The meaning of these outputs must also survive further processing. In the exercise, `sarif-parsing` merged two reports with different code locations into one. Both carried the placeholder `requires login` as their **fingerprint**, the identifier used to recognize a report. The unmodified SARIF helper treated that placeholder as a unique identity. The original outputs were retained; a documented custom post-processing step ignores only that placeholder and uses the existing fallback identifier based on rule, path, line and message. Both locations were then retained.

`runs/sarif/` contains `combined.sarif`, `summary.json` and the derived `findings.json` and `findings.csv`. These make the loss and limited correction traceable. When merging reports, therefore, also check whether code locations and evidence survive processing. A shared case may contain several affected locations.

One retrieval path has now been examined and treated. A second overview shows which questions remain open for the rest of the application.

Reviewing other functions, dependencies and configurations

For each accessible function, the team records which protective condition should apply and which method examines it. In the portal, this includes uploads, modifications and deletion of documents, exports and administrative access, alongside reading. Variants of a known cause remain distinguishable from functions that have not yet been considered.

Further reviews cover included libraries, exposed credentials and deployment. Secret scanning looks for items such as keys and passwords in the source tree and its history. Dependency review examines dependencies and their known vulnerabilities. A code audit of document retrieval does not automatically cover these tasks. Credentials, dependencies

The owner revokes or rotates a discovered credential; deleting it from a file is insufficient. A library finding is assessed against the deployed version and usage context. Container and infrastructure configurations require suitable review methods, data and assessments of their own.

TECHNICAL DETAIL

Trace defaults through to their security impact

`insecure-defaults` requires an evidenced path from a suspicious default to its security impact. The test code contained a fallback key and a debug value enabled by default, neither used by any examined function. These did not become confirmed product flaws. A suspicious value alone does not establish whether an attacker can exploit it.

PRACTICE GUIDE · WORKFLOW ASSIGNMENT

Trace default settings to their effects

Provide configuration together with the code that uses it. Require review of all six categories in the plugin overview and, for each candidate, a data or control path to security impact. Categories not searched and candidates not verified belong in the coverage record. Offline tests of the workflow are separate checks of its sequence and error handling.

Expected artifact: A report organized by category, identifying configuration consumers, verified paths to impact and open areas.

Technical testing of this plugin remained partial: its offline tests using predefined agent responses passed 102 checks; all 36 deliberately mutated workflow versions were detected as faulty. A separate search-pattern comparison against 18 documented examples found one example without a matching hit on the Windows host. `runs/insecure-defaults/` contains test logs and `report.md` with adaptations and limitations. The native agent workflow did not run. These files substantiate the examined workflow sequencing and error handling, not the detection performance of a complete analysis agent.

The coverage overview connects method, owner, result and limitations for each area. It marks areas as included, partially investigated, not investigated or outside scope. “Not investigated” does not mean “no issues.” Cases enter the same register. New entry points or a different actual permission distribution also update the threat model and subsequent review assignments. This baseline leads to ongoing change review.

Integrating agent reviews into development

A **diff** describes the change between code versions; a **pull request** presents it for shared review. `differential-review` examines its security implications. The proposed sequence is `differential-review` with Semgrep/CodeQL, followed by `fp-check` for new candidates. It accompanies the **software development lifecycle**, or SDLC: the development and maintenance of the application. Context from the initial audit is checked for currency. Change review

PRACTICE GUIDE · AGENT ASSIGNMENT

Review changes with history, callers and tests

Explicitly select the baseline and target revisions. The assignment to `differential-review` is: "Examine the security implications of this change using history, affected callers and test coverage. Do not modify the target working copy, and identify callers not examined." The review should connect the diff, code locations, tests and open questions. A saved Git diff alone does not fulfill this assignment.

Expected artifact: Review report with diff, history and test evidence, affected callers and open questions.

In the exercise, `differential-review` connected the four removed protective lines in two functions to real Git history and unchanged existing tests. The safe baseline revision was exported without switching the working copy under investigation. `runs/diff/review.md` refers to diff, history and blame outputs—the mapping of code lines to changes. The three original unit tests passed despite both ownership checks having been removed. This exposes a concrete gap: the existing green tests were insufficient to preserve the protective condition across the change.

Existing **continuous integration**, or CI, automatically runs defined checks when code changes. During the pilot, AppSec and the application team agree whether every change or a clearly defined selection triggers agent review. For a selection, they name the functions or components and ownership of the selection rule. Reports and register cases reach the responsible team through pull requests and linked tasks. Active tests against shared staging are scheduled separately, so multiple changes do not alter the same test state without coordination. Code scanning

The effect on release approval also needs agreement. In this proposed pilot, new agent reports do not initially block a change on the strength of their text alone; existing mandatory checks continue to apply. For confirmed findings, the team defines which severity requires remediation or an accountable risk decision before approval. Open cases and technically failed mandatory checks receive explicit treatment. A missing report is not a pass.

To make these rules workable, a team must first trial the process within a bounded scope. The pilot examines both the additional insights and the route by which they reach the people responsible.

Setting up a pilot and assessing its value

The application team names a pilot owner to bring together scope, capacity and decision criteria. Domain decisions remain with the owners described above. Start with a selected repository, confirmed operational context and the bounded assignment from the brief chapter. First set up the five core plugins and scanners and check their actual permissions.

The exercise provides concrete setup questions. The fuzzers installed later enabled bounded native campaigns; CodeQL was demonstrated on a public example. The complete CodeQL skill, two Semgrep rule-pack runs and the native `insecure-defaults` workflow remain partially or incompletely examined. Strix and Shannon still require a working Docker service; Codex Security did not reach a final report. Decisions about your pilot rest on these individual pieces of evidence and gaps, not a general success rate.

TECHNICAL DETAIL

Installing plugins and checking host support

PRACTICE GUIDE · INSTALLATION

Set up Trail of Bits plugins in Codex

The following Codex commands match the Trail of Bits documentation checked on October 28, 2026. Run them in the terminal of the approved agent environment; they require an installed Codex CLI. This plugin installation was not performed in the exercise: pinned instructions were applied directly and the required scanners installed separately. Codex installation

```
codex --version
codex plugin marketplace add trailofbits/skills
codex plugin add static-analysis@trailofbits
codex plugin add audit-context-building@trailofbits
codex plugin add fp-check@trailofbits
codex plugin add variant-analysis@trailofbits
codex plugin add differential-review@trailofbits
codex plugin list --json
```

For additional review tasks, install `insecure-defaults@trailofbits` and `semgrep-rule-creator@trailofbits` using `codex plugin add`. Load the Handbook skills and other linked instructions through the skill mechanism supported by the host. Scanner programs, language tools, build dependencies and model access follow separately. A trial run checks which programs start, where they create files and which permissions they use. These installation instructions do not claim that all native host integrations were tested in this exercise.

PRACTICE GUIDE · INSTALLATION

Install the Strix skill separately

For the individual Strix skill, the kickoff also specifies:

```
npx skills add usestrix/strix --skill penetration-testing-with-strix
```

This installs skill contents. Strix, Docker and model configuration follow separately. `npx` can also execute third-party code during installation and is subject to the supply-chain review described earlier. Strix skills

Codex and Claude Code have their own plugin mechanisms. Cursor supports agent skills in locations such as `.cursor/skills/<name>/SKILL.md`. A portable skill file does not guarantee identical execution of all additional actions, subagents and command files on every host. On Windows, Bash-based components need a suitable environment; this does not imply that every scanner requires Windows Subsystem for Linux, or WSL. Cursor skills, Claude Code plugins

The installation record captures host and version, origin and specific package revision, plugin versions, available programs, licenses, data paths and the trial run. Linked `main` pages can change. The test archive therefore documents the source and tool versions actually used. It does not replace the setup record for your own agent environment or a review of the plugins loaded there.

For this exercise, instructions and tool versions were recorded separately on October 29, 2026. Semgrep 1.182.1 ran in a separate Ubuntu/WSL environment, SkillSpector 2.12.0 in its own Python environment; Codex Security CLI 0.8.1 included plugin 0.1.95. The setup also used Clang 14, AFL++ 4.00c and CodeQL 2.27.3. Compilers and CodeQL were added exclusively to the project-local test environment; origins and hashes are recorded. These versions and the adaptations described are part of the results. A different installation may have different prerequisites and outputs.

TECHNICAL DETAIL

Organizing the brief, runs and findings

PRACTICE GUIDE · FILING CONVENTION

Organize the brief, runs and findings

The following structure can be used outside the protected application working copy. The names are a suggested convention, not automatically generated standard files from a product. German path names are retained so that they match the original example.

```
security-audit/<lauf-id>/
  auftrag.md            Revision, scope, permissions, owners
  kontext.md            Entry points, data flows, open assumptions
  laeufe/               Scanner configuration, status and logs
  befunde/              One case per identifier, with evidence links
  tests/                Approved demonstrations and results
  abdeckung.md          Examined and omitted areas
  bericht.md            Assessment and next decisions
```

The authoritative register can remain in the ticketing system and link to these artifacts. The platform configures access; confidential evidence retains its restrictions when linked. AppSec keeps review status current, and the application team takes responsibility for remediation tasks.

PRACTICAL CONTEXT

Using pilot data to decide on expansion

The comparison starts with a scanner baseline on the same revision. Agent investigation is then added. The decision uses the report, findings register, coverage overview and effort record. Track additional confirmed cases, refuted reports, open candidates and unexamined areas, as well as human review and remediation time, scanner licenses, model resources and execution resources. Agree limits and available capacity before starting; only setup and baseline measurement provide a basis for a credible schedule.

The proportion of confirmed cases among reports with a final substantive assessment describes usefulness in the pilot. It does not measure all missed flaws. Known vulnerabilities or explicitly introduced test cases help make misses visible; success on them cannot be extrapolated to unknown production flaws. Without a known, complete set of flaws, an overall detection rate cannot be determined credibly.

Where possible, at least one sufficiently substantiated case should also pass through remediation and retest. If no suitable finding exists, this stage remains unassessed or is trialed using an explicitly identified test case. Expansion makes sense when additional insights are confirmed, the counter-check remains manageable and treatment works. Otherwise, close process gaps first or narrow the purpose. Unanswered operational questions, repeatedly aborted runs and missing responsibilities receive tasks and owners.

Conclusion

At the document portal, value turns on a specific question: can the team substantiate whether a customer can retrieve someone else's documents and verify any necessary correction? An agent can reconstruct the path through code, use analysis tools and prepare a counter-check. A sound assessment rests on the brief, context and evidence; a separate retest establishes whether a correction works. Named people remain responsible for access rights, approvals and residual risks.

The exercises show why these connections matter. Existing unit tests stayed green despite ownership checks being removed. When a harmful skill instruction was examined, a high-severity individual finding appeared alongside an overall recommendation of SAFE. Anyone taking only the green test result or aggregate score loses the decisive context. Result files must therefore be read together with execution status, individual evidence and open questions.

These observations come from bounded exercises with known flaws and artificial data. They show concrete results and technical gaps; they do not establish detection performance for unknown production flaws. Not all 18 workflows were fully tested. In particular, active Strix and Shannon runs and a completed Codex Security scan remain outstanding. The documented instructions and expected artifacts help with preparation but do not replace a successfully completed run of your own.

If you adopt this approach, start with one repository, a confirmed threat model and a clearly bounded security question. My recommendation remains the static core of Trail of Bits, suitable CodeQL or Semgrep analyses and a separate counter-check using `fp-check`. Follow at least one suitable case through remediation and retest. If an unresolved question requires active evidence, you can trial Strix in a separately approved pilot. The test environment and its technical boundaries must be in place first.

Then decide on expansion according to whether additional confirmed insights emerge and whether the team can address them at a reasonable cost. The measure is a review process in which a plausible claim becomes a traceable decision and a verified improvement. The next extension of the security stack should be judged by that standard too.

Source dates and limits of the claims

The underlying documentation was reviewed as of October 28, 2026; test sources and tool versions were recorded separately on October 29. The portal case is synthetic. Tools actually executed include Semgrep and SkillSpector; the Codex Security run started but failed technically. Execution limitations and result files are described alongside each method. No external applications were attacked or production vulnerabilities confirmed. A pilot must test effectiveness in your own target system.

Style reference: “Enterprise Architecture für agentische KI.” Image provenance, prompts and diagram sources: `CREDITS.md`, `image-prompts.json`, `diagrams/`. The generated scenes provide no evidence of security effectiveness.

Primary sources

Sources are linked beside the relevant claims in the text. This list repeats the same references for consultation. Underlying documentation reviewed: October 28, 2026; additional test sources and execution: October 29, 2026. The exercise source manifest records its exact revisions.

1. Trail of Bits
   https://github.com/trailofbits/skills
2. Threat-modeling skill
   https://github.com/openai/skills/tree/main/skills/.curated/security-threat-model
3. CodeQL
   https://docs.github.com/en/code-security/concepts/code-scanning/codeql/codeql-cli
4. Semgrep
   https://docs.semgrep.dev/semgrep-pro-vs-oss
5. static analysis
   https://github.com/trailofbits/skills/blob/main/plugins/static-analysis/README.md
6. context building
   https://github.com/trailofbits/skills/blob/main/plugins/audit-context-building/skills/audit-context-building/SKILL.md
7. variants
   https://github.com/trailofbits/skills/blob/main/plugins/variant-analysis/skills/variant-analysis/SKILL.md
8. rule development
   https://github.com/trailofbits/skills/blob/main/plugins/semgrep-rule-creator/skills/semgrep-rule-creator/SKILL.md
9. insecure defaults
   https://github.com/trailofbits/skills/tree/main/plugins/insecure-defaults
10. Trail of Bits workflows
    https://github.com/trailofbits/skills/tree/2be6433b39e009cd2b6fd3be1895f6b1faa5af9a/plugins
11. Build modes
    https://docs.github.com/en/code-security/concepts/code-scanning/codeql/codeql-for-compiled-languages
12. Semgrep skill
    https://github.com/trailofbits/skills/blob/main/plugins/static-analysis/skills/semgrep/SKILL.md
13. CodeQL 2.27.3
    https://github.com/github/codeql-action/releases/tag/codeql-bundle-v2.27.3
14. public MIT-licensed regression example
    https://github.com/github/codeql/tree/18b48aac513b7915adf742aa3fe226963a6ebcb6/python/ql/test/query-tests/Security/CWE-078-CommandInjection
15. Semgrep rules terms
    https://semgrep.dev/legal/rules-license/
16. Plugin quickstart
    https://learn.chatgpt.com/docs/security/plugin
17. cloud overview
    https://help.openai.com/en/articles/20001107-codex-security
18. Deep scans
    https://learn.chatgpt.com/docs/security/plugin/deep-scans
19. CLI documentation
    https://learn.chatgpt.com/docs/security/cli

20. fp-check
https://github.com/trailofbits/skills/blob/main/plugins/fp-check/skills/fp-check/SKILL.md
21. Strix
https://github.com/usestrix/strix
22. Strix skill
https://github.com/usestrix/strix/blob/main/skills/penetration-testing-with-strix/SKILL.md
23. Shannon
https://github.com/KeygraphHQ/shannon
24. configuration and partial failures
https://github.com/KeygraphHQ/shannon/blob/main/docs/configuration.md
25. Strix
https://github.com/usestrix/strix/blob/6b7ae2ebe4d47037a927aade23265ff4e4f27e84/skills/penetration-testing-with-strix/SKILL.md
26. Shannon
https://github.com/KeygraphHQ/shannon/blob/b9744f6cecc5429333873b86a6762f1d70c54fc2/docs/development.md
27. Testing Handbook skills
https://github.com/trailofbits/skills/tree/main/plugins/testing-handbook-skills
28. libFuzzer
https://llvm.org/docs/LibFuzzer.html
29. SkillSpector
https://github.com/NVIDIA/skillspector
30. Skill Inspector
https://github.com/NVIDIA/SkillSpector/blob/b5987a332c9a118ec4714a5c79eb76641b546251/skills/skill-inspector/SKILL.md
31. MCP review
https://github.com/howardhsieh/agent-security-skills
32. Authorization
https://modelcontextprotocol.io/specification/2026-07-28/basic/authorization
33. security best practices
https://modelcontextprotocol.io/docs/2026-07-28/tutorials/security/security_best_practices
34. SARIF 2.1.0
https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html
35. Credentials
https://docs.github.com/en/code-security/concepts/secret-security/secret-scanning
36. dependencies
https://docs.github.com/en/code-security/concepts/supply-chain-security/dependency-review
37. Change review
https://github.com/trailofbits/skills/blob/main/plugins/differential-review/skills/differential-review/SKILL.md
38. Code scanning
https://docs.github.com/en/code-security/concepts/code-scanning/about-code-scanning
39. Strix skills
https://github.com/usestrix/strix/tree/main/skills
40. Cursor skills
https://cursor.com/docs/skills
41. Claude Code plugins
https://docs.anthropic.com/en/docs/claude-code/plugins