BOTTLENECK

NO. 01 · THE UNTRUSTED CLONE · A LIMITED SERIES

GIT CLONE **RUNS SOMEBODY ELSE'S CODE.**

You knew that. You did it anyway, this week, probably twice. A field guide to SecureGit — and to git that can prove what happened.

PLUS: THE MATURITY TABLE — EVERY FEATURE MARKED SHIPPED, DESIGNED, OR PLANNED · CURRENT VERSION 0.12.16

MASTHEAD & HOW TO READ THIS ISSUE

BOTTLENECK *The magazine for engineers becoming AI engineers.*

Issue 01 · The Untrusted Clone

Published by ArmyknifeLabs.

About this issue. Most tooling magazines are written by people who want you to be impressed. This one is written by the people who build the tool, for people who are deciding whether to type one command. Being impressed is not the goal. Typing the command is.

How to read this issue

If you have ten minutes: page 11. That is the quickstart. It ends with you having scanned a real repository. Everything else in this issue is optional after that.

If you have an evening: Part I, pages 10–23. That is Day One in full — install, first acquire, first scan, reading findings, aliases, the pre-commit hook, and a straight answer on speed. You will finish it with SecureGit in your daily workflow.

If you are evaluating this for a team: start at page 56 (the twelve objections), then page 53 (the 30-day rollout), then come back to Part II for the mental model. You need the objections first because your team will raise them before you finish your first sentence.

If you are in a regulated environment: Parts III and IV, pages 38–51. SBOM anchoring, CVE state over time, license posture, and the credential boundary. Then the maturity table on page 63, because some of what you need is `DESIGNED` and not yet `SHIPPED`, and you should know which before you plan around it.

The maturity markers

Every capability in this issue is marked. Here is exactly what the three words mean, and we hold ourselves to them:

`SHIPPED` — in a released build. You can install it today and use it. If you cannot, that is a bug and we want the report.

`DESIGNED` — the architecture is decided and written down in an accepted design record, and implementation is partial or scheduled. It is real engineering intent, not a wish. It is not something you can run.

`PLANNED` — identified, scoped, not yet designed in detail. Directional only. Do not build a procurement plan on a `PLANNED` row.

If a page describes something and you cannot find the marker, treat it as `PLANNED` and write to us, because we made a mistake.

Corrections run at the front of the next issue, at the same size as the original claim.

CONTENTS

EDITOR'S LETTER

You have spent your career securing what you deploy. The thing you never secured is the moment code arrives on your machine — and that moment now happens dozens of times a week, increasingly without you watching.

The Repository Is The Attack Surface

Here is a thing every engineer knows and almost nobody acts on.

`git clone` is not a download. It is a download plus a filesystem write plus, depending on what is in the archive and what you do next, code execution. Hooks. Submodule configuration. Filter drivers. Path handling quirks that have produced real CVEs in real git releases more than once. The repository is not data that you then choose to run. It is a bundle of data and executable configuration that you have invited onto your machine, inside your home directory, with your permissions.

You know this. So does everyone. And then a colleague posts a link, and you clone it, because the alternative is not cloning it, and that is not a real alternative.

The reason this became urgent rather than merely true is that the volume changed.

Two years ago you cloned a handful of repositories a month, and you had usually heard of them. Now an agent working on your behalf pulls down a dependency you have not evaluated, to check whether its API does what a search result claimed. It does that at three in the morning while you are asleep, using your credentials, on your machine. The judgment step that used to sit between "I found this repository" and "it is now on my disk" has been compressed to nothing, and in many workflows removed entirely.

That is the bottleneck this issue is about. Not "can I get the code" — that was solved decades ago. **Can I get the code without the act of getting it being the vulnerability, and can I prove afterward what arrived and what I did with it.**

> git clone is not a download. It is a download that has been granted permission to run.

There is a second half to this, and it is the half that turns a security tool into an engineering tool.

Once you accept that code arrival is an event worth controlling, you notice that code *departure* is too. A push is the first moment your work crosses onto shared infrastructure. It is the moment where "I wrote this" becomes "we shipped this." And in almost every organization, that moment carries no durable proof of anything: not who wrote which lines, not which of those lines came from a human versus an agent, not whether any scanner ever looked at it, not what the dependency posture was at the time.

Six months later somebody asks. They ask because of an audit, or an incident, or a customer questionnaire, or a diligence process. And the honest answer is that you have git history, which tells you what changed and who the commit claims to be from, and nothing else. Git history is a record of assertions, not a record of evidence. It was never designed to be otherwise — Linus was solving a different problem, and solving it very well.

SecureGit is one answer to both halves. It puts a thin gate on arrival, a thin gate on departure, and a cryptographic receipt on both. That is the whole idea,

and the rest of this issue is about whether the gates are thin enough that you will actually keep them.

> Git history is a record of assertions, not a record of evidence. That was the right design for 2005. It is an expensive gap in 2026.

Now the part where I tell you what this is going to cost you, because adoption articles that skip this are the reason engineers distrust adoption articles.

It will cost you a new verb. You will type `acquire` instead of `clone` for untrusted repositories. That is a habit change, and habit changes fail unless you make them cheap. Page 16 is entirely about making it cheap.

It will cost you seconds, and occasionally a minute. Scanning is not free. On a normal working repository it is fast enough that you will stop noticing. On a very large repository it is not, and page 20 gives you real numbers rather than reassurance.

It will cost you some false positives. Every scanner produces them. A scanner that claims otherwise is either lying or not looking hard. Page 60 is a playbook for driving them down, because a tool that cries wolf gets disabled in week two, and a disabled tool has negative value — it provides the feeling of security with none of it.

And it will cost you a conversation with your team, which is the expensive part. Page 56 gives you the twelve objections you will hear, with honest answers, including the three cases where the honest answer is "you are right, do not use this for that."

What you get for it: code that arrives without executing, commits that cannot silently ship a secret, pushes that carry proof, and — the part that surprises people — a genuine answer to the question "what did we know about our supply chain in March," which is a question that currently ends most enterprise sales conversations in an uncomfortable silence.

Start at page 11. It takes ten minutes and it ends with you having scanned something real.

— The Editors

NINETY SECONDS (INFOGRAPHIC SPREAD)

Two timelines, same repository. The top is `git clone`. The bottom is `securegit acquire`. Read them against each other.

Spread headline (spans gutter): WHAT ACTUALLY HAPPENS

Panel A (left page, upper) — TIMELINE ONE: `git clone`

Format: a horizontal timeline, left to right, with events as ticks above the line and *trust state* as a colored band below it. The band starts neutral and turns signal-orange at the first execution point, staying orange to the end.

MOMENT	WHAT HAPPENS	YOUR EXPOSURE
`t+0`	Network fetch begins	None yet
`t+2s`	Objects written into `.git`	Disk write, your permissions
`t+3s`	**Hook scripts land in `.git/hooks`**	**Dormant executables now on disk**
`t+3s`	`.gitmodules`, `.gitattributes`, config land	Filters and submodule directives, unread
`t+4s`	Working tree checkout	Paths written per attacker-influenced names
`t+4s`	Clone reports success	**You believe you have data. You have data and configuration.**
`t+30s`	You `cd` in and run anything git-adjacent	**Hooks may execute. Filters may execute.**
`t+45s`	You open the repo in an editor with extensions	Editor-level execution surface

Annotation (signal, pointing at `t+3s`): *Nothing has run yet. That is the entire window in which a decision is still cheap. `git clone` gives you no place to stand inside it.*

Panel B (left page, lower) — TIMELINE TWO:

`securegit acquire`

MOMENT	WHAT HAPPENS	YOUR EXPOSURE
`t+0`	**Fetched as an archive, not as a live repo**	Inert bytes
`t+2s`	Archive extracted to a staging path	Files on disk, no git configuration active
`t+2s`	**Hooks stripped**	Dormant executables removed, not merely ignored
`t+3s`	Scanners run across the tree	Findings collected
`t+4s`	A sanitization report is written alongside	You have something to read
`t+5s`	Converted into a normal git repository	**Now** it is a repo — after inspection
`t+5s`	`SHIPPED` — a chain receipt records the acquisition	Provenance from moment zero

Annotation: *The order is the product. Fetch, strip, inspect, then confer git-ness.* `git clone` *confers git-ness first and inspects never.*

Panel C (right page, full) — THE THREE THINGS PEOPLE GET WRONG

Format: three tall cards, each with a myth in condensed caps, a correction in serif body, and a one-line takeaway in mono.

MYTH 1 · "I only clone repos I trust." You clone repos your dependencies trust, which is a different set and a much larger one. You also clone repos an agent selected on your behalf from a search result. The trusted-source model assumes a human evaluated the source. Increasingly, none did. `→ The question is not "do I trust this" but "did anyone actually look."`

MYTH 2 · "My scanner catches this." Most scanning runs in CI, which is after the clone, on a machine that is not yours, looking at the tree rather than the repository metadata. The hooks that concern us are in `.git`, which most scanners exclude by default and which CI usually does not receive. `→ Scan the .git directory. Most tools do not, by default. Check yours.`

MYTH 3 · "This is theoretical." Hook execution and path-handling behavior in git have produced real, patched CVEs on multiple occasions. The mitigation is always "upgrade git," which works until the next one, and which does nothing about the window between disclosure and your fleet actually upgrading. `→ Defense in depth exists because point fixes arrive late and unevenly.`

The honest framing SecureGit does not make untrusted code safe. Nothing does. It makes the *acquisition* of untrusted code non-executing, and it gives you a place to stand — a moment where the code is on your disk, inert, and you can look at it before it becomes a live repository.

That moment did not previously exist. That is the entire contribution. Everything else in this issue is built on it.

THE ONE-PAGE MENTAL MODEL

Before any command, this. Five minutes here saves an hour of confusion later.

THE ONE SENTENCE

SecureGit is a git wrapper that puts a gate on the two moments code crosses a trust boundary — arrival and departure — and writes a signed receipt for each crossing.

Everything else is elaboration. If you remember one thing, remember: **arrival, departure, receipt.**

THE FOUR LAYERS

Think of it as four layers stacked, each usable without the ones above it. **You can adopt one layer and stop.** Most people should, at first.

LAYER 1 · ACQUISITION SHIPPED Fetch untrusted code without executing it. `securegit acquire` replaces `git clone` for anything you did not write. Archive-first (ZIP-with-history by default; `zip-only` and `bare-checkout` also available), hooks stripped, twelve scanners run, then converted to a normal git repo. *Adopt this alone and you have already gotten most of the day-one value.*

LAYER 2 · SCANNING SHIPPED Twelve built-in Rust scanners — secrets, patterns, entropy, binary, encoding (Trojan Source), supply chain, CI/CD, container, IaC, deserialization, dangerous files, git internals — plus a plugin system that wraps tools you already use. Run it on a path, on staged changes, on a diff, or in CI. Outputs pretty, JSON, **SARIF 2.1.0**, or **GitLab SAST v15**. `--write-baseline` lets you adopt on a legacy codebase without failing every build. *Adopt this second. It is the layer that changes your commit habits.*

LAYER 3 · CHAIN OF CUSTODY & GOVERNANCE SHIPPED (core, audit log, layered policy) / DESIGNED (offline chain, batch lookup) Every trust-boundary operation emits a cryptographically signed receipt. Push can be gated on whether the commits in range have receipts. Blame can show provenance per line. A separate **hash-chained audit log** captures every security-relevant event (scan, block, hook, baseline write, credential change) and exports as JSONL or CEF for SIEM ingest. **Layered configuration** (`/etc/securegit` → user → repo → env) with `[policy] min_fail_on` , `locked_keys` , and `audit_required` gives security teams a governance surface developers can tighten but not weaken. *Adopt this when you need to prove things to someone who was not there. Two of its three pieces (audit log, policy) work without any signing infrastructure.*

LAYER 4 · SUPPLY-CHAIN PROVENANCE SHIPPED (anchoring, SBOM, compliance report) / DESIGNED (parts) SBOM anchoring, CVE state at a point in time, license posture, lock-file change records — all attached to the same chain. Releases ship a CycloneDX SBOM, `SHA256SUMS` , and a **Sigstore keyless (cosign)** signature. `securegit compliance report` maps findings to OWASP Top 10 (2021) via CWE and summarizes NIST SSDF v1.1 practice coverage — markdown or JSON, straight into a review packet. *Adopt this when procurement, audit, or a customer questionnaire forces the question.*

WHAT IT IS NOT

This section exists because mis-set expectations are the most common cause of abandonment, and every item below has caused a real "wait, I thought it did X" moment.

It is not a replacement for git. It wraps git. Your repositories stay normal git repositories. Your teammates who do not use SecureGit are unaffected.

You can stop using it at any time and lose nothing but the receipts.

It is not a malware sandbox. It prevents *acquisition-time* execution and it scans. It does not analyze behavior, emulate, or detonate anything. Code you subsequently choose to build and run is code you chose to build and run.

It is not a SAST platform. It is not competing with Semgrep or SonarQube on depth of static analysis. It wraps them. If you need deep dataflow analysis, you need those tools, and SecureGit's job is to run them at the right moment and anchor their output.

It is not a secrets manager. It brokers access to secrets that live in a real secrets manager. It is not where your secrets should live. (Page 47.)

It is not a bundled vulnerability scanner — with one deliberate, narrow exception documented on page 40. It anchors the output of the scanners you already run. That boundary was a decision, it was contested internally, and page 40 explains why it moved once and why it will not move again casually.

It does not make your history trustworthy retroactively. Commits made before you adopted it have no receipts. There is a path for retro-attestation, and it is honest about being a different tier of evidence. A receipt that says "we checked afterward" is worth less than one that says "we watched it happen," and the system records which one you have.

THE THREE-WORD TEST

Whenever you are unsure whether SecureGit is involved in something, ask: **is this a boundary crossing?**

Code arriving from elsewhere — yes. Code leaving to shared infrastructure — yes. Anything purely local, in your working tree, between you and your own disk — no.

`status`, `log`, `diff`, `add`, `checkout`, `branch`, `stash` cross no boundary and emit no receipts. That is deliberate. A tool that instruments everything gets turned off.

PART ONE

DAY ONE

Ten minutes to your first scan. An evening to a changed workflow.

Nothing on these pages requires a decision from anyone but you.

TEN MINUTES

Five commands. No configuration. No account. Nothing to uninstall afterward except one binary.

1 INSTALL

```
curl -fsSL https://<release-host>/securegit/install.sh | sh
```

Before you paste that: you are about to pipe a script from the internet into a shell, in an issue about not trusting code from the internet. We are aware of the irony and you should be too. If you would rather not, download the release binary, verify it with the shipped `SHA256SUMS` + **Sigstore keyless signature (cosign)**, and put it on your `PATH`. Both paths are supported and the second one is the one we would pick. Page 12 has the verify commands.

Verify:

```
securegit --version
```

2 ACQUIRE SOMETHING REAL

Pick a public repository you have never inspected. Small is better for a first run.

```
securegit acquire https://github.com/toml-lang/toml /tmp/first-acquire
```

3 CONFIRM THE HOOKS ARE GONE

```
ls -la /tmp/first-acquire/.git/hooks
```

This should be empty. That is the whole product in one directory listing. A normal clone of the same repository will have sample hooks in there and, from a hostile source, could have live ones.

4 READ THE REPORT

```
cat /tmp/first-acquire/.securegit-report.json
```

5 SCAN SOMETHING YOU CARE ABOUT

```
securegit scan . --fail-on high
```

Run this in a repository you actually work in. It will finish in seconds on a normal project.

You are done. That is the tool. Everything after this page makes it faster, quieter, and more useful — but you have already prevented the class of problem this issue is about.

If step 5 produced findings you disagree with, that is expected and it is not a failure. Go to page 60.

GETTING IT ON YOUR MACHINE

Longer than the quickstart, with the parts that go wrong.

PLATFORMS

SHIPPED on Linux and macOS, both Intel and Apple Silicon. Windows support is **experimental** — it works, and it is not yet held to the same bar, and you should expect rough edges around path handling and shell integration. If you are on Windows and this is a blocker, tell us; that feedback moves priority.

Container use is supported and is the recommended way to try it if you do not want a binary on your host.

THE THREE INSTALL PATHS

Path A — release binary. Download, verify, place on `PATH`. Most auditable. Recommended for anyone who reads the rest of this magazine and takes it seriously. Every release ships with a **CycloneDX SBOM**, `SHA256SUMS`, and a **Sigstore keyless signature (cosign)**:

```
sha256sum -c SHA256SUMS --ignore-missing
cosign verify-blob \
  --certificate SHA256SUMS.pem --
signature SHA256SUMS.sig \
  --certificate-identity-regexp
'github.com/armyknifelabs-tools/
securegit' \
  --certificate-oidc-issuer https://
token.actions.githubusercontent.com \
  SHA256SUMS
```

Path B — install script. Fastest. Reasonable for a laptop, not for a fleet.

Path C — build from source. Requires a Rust toolchain. Slowest, most transparent. Use this if your organization requires building security tooling from source, which some do.

For a fleet: do not use Path B. Package it, sign it, and distribute it the way you distribute everything else. A security tool installed by curl-pipe-shell across two hundred machines is a supply-chain finding of its own, and somebody will eventually point that out in a review. Distribute a system-wide `/etc/securegit/config.toml` at the same time — that is where org policy lives (page 45).

WHAT GETS CREATED

```
~/.config/securegit/config.toml
~/.config/securegit/plugins/
~/.local/share/securegit/audit/
# tamper-evident audit log
```

No daemon, no service, no background process, nothing in your login items. Uninstall is deleting the binary and those directories.

CORPORATE NETWORKS

SecureGit reads the OS trust store by default, so a TLS-inspecting corporate proxy with a properly installed root CA "just works." For private CAs held in a file rather than the trust store, point `SECUREGIT_CA_BUNDLE=/etc/ssl/corp-bundle.pem` at a PEM bundle (mirroring `curl --cacert`). Explicit proxying uses standard `HTTP_PROXY` / `HTTPS_PROXY` / `NO_PROXY`, plus `SECUREGIT_PROXY` (or `[network] proxy`) as an override. These cover the two failure modes we hear about most from enterprise pilots.

This matters for adoption more than it sounds. A tool that is trivially reversible gets tried. A tool that installs infrastructure gets a meeting.

YOUR FIRST ACQUIRE, IN DETAIL

```
securegit acquire <url> <destination>
```

You can also acquire into the current directory:

```
securegit acquire <url> .
```

What happens, in order:

1. The remote is resolved and the content is fetched **as an archive**, not as a live repository. This is the load-bearing step. No git configuration is active at any point during transfer.
2. The archive is extracted to the destination.
3. **Hooks are stripped.** Not disabled, not renamed — removed.
4. The configured scanners run across the extracted tree.
5. A report is written to `.securegit-report.json` in the destination.
6. The tree is converted into a normal git repository, with history.
7. `SHIPPED` A chain receipt is emitted recording the remote URL and the resolved `HEAD` at acquisition time. Acquisition never blocks on receipt failure — if the signing daemon is unreachable, you get a warning and your clone.

Point 7 deserves a note. The receipt records what you got, from where, at what commit, at what time. Later, if the upstream force-pushes and history changes underneath you, your receipt still says what you actually received. That is a surprisingly practical thing to have and it costs nothing at acquisition time.

WHAT YOU CAN DO WITH THE RESULT

Everything. It is a normal repository.

```
cd /tmp/first-acquire
git log
git diff
git checkout -b my-branch
```

There is no lock-in, no proprietary format, no wrapper you have to keep using. If you delete SecureGit tomorrow, the repository you acquired is unaffected.

THE THREE THINGS THAT GO WRONG ON FIRST INSTALL

"Command not found" after the install script. The binary landed somewhere not on your `PATH`. Check `~/.local/bin` and `/usr/local/bin`, then restart your shell.

Acquire is slow on a very large repository. Archive-first fetch has different performance characteristics than a shallow clone. For repositories in the hundreds of megabytes, expect seconds, not milliseconds, for acquisition and then a separate scan pass. Page 20 has numbers.

A private repository fails to acquire. You have not registered credentials yet. Page 49. For a first run, use a public repository — do not start your evaluation by debugging auth.

WHAT THE SCANNER SEES

Findings are only useful if you can act on them within about ten seconds of reading them. Here is how to get to that.

THE COMMAND SURFACE

```
securegit scan <path>
securegit scan .
securegit scan . --fail-on high
securegit scan . --min-severity high
securegit scan . --include-git
securegit scan . --format json
securegit scan . --format sarif  --
report-output f.sarif
securegit scan . --format gitlab --
report-output gl.json
securegit scan --staged
securegit scan . --write-baseline
"<reason>"
securegit scan . --baseline .securegit/
baseline.json
```

`--include-git` deserves emphasis. Most scanners in the industry exclude `.git` by default, and the acquisition threat this magazine opens with lives precisely there. When you are evaluating an unfamiliar repository, include it.

`--format sarif` and `--format gitlab` upload directly into the dashboards your reviewers already read — GitHub Advanced Security, Azure DevOps, SonarQube, DefectDojo on one side; the GitLab security dashboard and MR widget on the other. `--report-output` writes the report even when the scan gates, so one CI step can both report *and* enforce. Page 44 is a full worked example.

THE TWELVE BUILT-IN SCANNERS

All `SHIPPED`, all compiled into the binary, all sub-millisecond to low-millisecond per file. Load automatically; no configuration required.

SCANNER	LOOKS FOR	TYPICAL SEVERITY
`secrets`	AWS keys, GitHub/Slack tokens, private keys, DB passwords, OpenAI keys	Critical / High
`patterns`	Dynamic execution, shell injection, reverse shells, persistence indicators	Critical / High
`entropy`	High-entropy strings that look like keys but match no known format	High / Medium
`binary`	Unexpected ELF, PE, Mach-O executables in source trees	Critical / Medium
`encoding`	**Trojan Source (CVE-2021-42574)** — BiDi overrides, homoglyphs, zero-width chars	Critical / High
`supply-chain`	36 known typosquatted packages (npm/PyPI/Ruby), malicious lifecycle hooks, dependency confusion	Critical / High
`ci-cd`	`pull_request_target` abuse, unpinned actions, input injection into `run:`, cache poisoning, `curl|bash` in pipelines, secret exfiltration, Jenkins `@Grab`	Critical / High
`container`	Privileged pods, Docker socket mounts, `cap_add: ALL`, RBAC wildcards, host namespaces	Critical / Medium
`iac`	Open security groups, public S3 buckets, `local-exec` provisioners, Ansible shell pipes	Critical / Medium
`deserialization`	Python `pickle` / `marshal`, `yaml.load()` without SafeLoader, Java `ObjectInputStream`, XXE	Critical / High
`dangerous-files`	`.gitmodules` path traversal (CVE-2018-17456), fsmonitor hooks, filter drivers	Critical / High
`git-internals`	Unexpected hooks after sanitization, dangerous git config keys	Critical / High

`entropy` **is the one that produces the most false positives**, by a wide margin, and it is also the one that catches the credentials that no pattern list knows about yet. That is the trade and it is not resolvable — you tune it, you do not fix it. Page 60.

`encoding` **and** `dangerous-files` **are the ones that will most surprise you the first time they fire.** Trojan Source findings look wrong until you view the file with an editor that shows invisible codepoints; `.gitmodules` findings are usually benign in your own repos and are the whole reason `--include-git` exists for someone else's.

ANATOMY OF A FINDING

```
CRITICAL  secrets          src/config/
settings.py:14
          AWS Access Key
          AKIA••••••••••••••••
          → Rotate this credential, then
remove from history.
```

Severity — `critical`, `high`, `medium`, `low`. Drives `--fail-on` and `--min-severity`.

Scanner — which plugin found it. Useful because it tells you the false-positive profile immediately. A finding from `secrets` is usually real. A finding from `entropy` needs a human look.

Location — file and line. Always. A finding without a location is a bug.

The redacted value — enough to identify it, never enough to use it. **Findings never print full secret values**, including in JSON output. This matters because scan output ends up in CI logs, which are frequently more readable than the repository was.

The action — what to do. A finding that does not tell you what to do is a notification, not a finding.

THE TEN-SECOND TRIAGE

For each finding, in order:

1. **Is it in a test fixture or an example?** Very common, usually benign, and the reason `--skip-paths` exists. But check that the example key is actually fake — real keys get pasted into examples more often than anyone admits.
2. **Is it a real credential?** If yes, stop reading this magazine. Rotate it. Removing it from the file is not sufficient; it is in history, and if the repository was ever public or ever shared, assume compromise. Rotation is the only remediation.
3. **Is it high entropy but not a secret?** A hash, a test vector, a base64 asset, a minified bundle. Suppress it by path, not by disabling the scanner.
4. **Is it a pattern finding you disagree with?** Read the line. `patterns` findings are usually about dynamic execution, and "I know what I'm doing here" is sometimes correct and sometimes the exact sentence that precedes an incident.

THE FIRST SCAN IS ALWAYS THE WORST ONE

Your first scan of a mature repository will produce more findings than every subsequent scan combined. Years of accumulated fixtures, examples, vendored code, and one or two genuine problems.

Do not try to get to zero on day one. The correct first move is:

1. Run it. Look at the `critical` findings only. There are usually few.
2. Fix or rotate anything real.
3. Set `--fail-on high` and configure `skip_paths` for vendor and dependency directories.
4. **From this point forward, gate on new findings, not total findings.**

Teams that try to reach zero before adopting never adopt. Teams that gate the delta and burn down the backlog over a quarter succeed almost every time.

MUSCLE MEMORY

The single biggest predictor of whether you are still using this in a month is whether typing it costs you anything. Spend thirty seconds now.

The command is called `securegit` because the name should be unambiguous when someone reads it in a script, a runbook, or a CI file six months from now. It is not meant to be typed nine characters at a time, forty times a day.

Alias it. Immediately. Before you do anything else in this issue.

BASH / ZSH

Add to `~/.bashrc` or `~/.zshrc`:

```
alias sgit='securegit'
alias sg='securegit'
```

FISH

Add to `~/.config/fish/config.fish`:

```
alias sgit='securegit'
alias sg='securegit'
```

POWERSHELL

Add to your profile:

```
Set-Alias -Name sgit -Value securegit
Set-Alias -Name sg -Value securegit
```

TAB COMPLETION FOLLOWS THE ALIAS

```
# bash
complete -F _securegit sgit
complete -F _securegit sg

# zsh
compdef sgit=securegit
compdef sg=securegit
```

Do this. An alias without completion is worse than no alias, because you lose discoverability and you will forget the flags.

THE WORKFLOW ALIAS SET

This is the part that actually changes behavior. Create `~/.securegit_aliases`:

```
# Acquire instead of clone, with a name
that reminds you why
alias safe-clone='securegit acquire'

# Scan right here
alias scan-here='securegit scan .'

# Scan including repository metadata —
for unfamiliar code
alias scan-deep='securegit scan . --
include-git'

# Scan only what you are about to commit
alias scan-staged='securegit scan --
staged --fail-on high'

# Scan only what changed against your
main branch
alias scan-diff='git diff main --name-
only | xargs securegit scan'

# Skip the usual noise
alias scan-clean='securegit scan . --
skip-paths "**/node_modules/**,**/vendor/
**"'

# Plugin housekeeping
alias plugin-status='securegit plugin
list && securegit plugin check-updates'
```

Source it:

```
echo "source ~/.securegit_aliases" >>
~/.bashrc
```

`scan-diff` **is the one you will use most** and it is the one people discover last. Scanning only what changed against your main branch turns a multi-second operation into a sub-second one and makes the whole habit sustainable on large repositories.

ENVIRONMENT DEFAULTS

Set these once and stop passing flags:

```
export SECUREGIT_FAIL_ON=high
export SECUREGIT_SKIP_PATHS="**/
node_modules/**:**/vendor/**:**/target/
**:**/dist/**"
```

ON THE NAME

We have been asked, repeatedly, whether the binary should be called `sgit` or `safegit` or something shorter.

The current answer is no, and the reasoning is worth stating because it is a general principle. **Optimize the written name for the reader; optimize the typed name for the typist.** A script, a CI file, and a runbook are read far more often than they are written, and by people with less context than the author had. `securegit scan . --fail-on high` is self-explaining to someone who has never heard of the tool. `sg scan . --fail-on high` is not.

Your shell is yours. Alias it to a single character if you like. The canonical name stays long on purpose.

YOUR FIRST GUARDRAIL

One file, six lines. This is the point where SecureGit stops being a thing you remember to run and starts being a thing that protects you when you forget.

Everything up to this page has been manual. Manual controls work exactly as long as your attention does, which is to say not on the Friday afternoon when it matters.

The pre-commit hook is where that changes.

THE HOOK

Create `.git/hooks/pre-commit`:

```
#!/bin/bash
securegit scan --staged --fail-on high ||
{
  echo "SecureGit: staged changes contain
high-severity findings. Commit blocked."
  echo "Review with: securegit scan --
staged"
  echo "Override once with: git commit --
no-verify"
  exit 1
}
```

Make it executable:

```
chmod +x .git/hooks/pre-commit
```

Three deliberate choices in those six lines, and they are all about adoption rather than security.

It scans only staged changes. Not the repository. Staged-only keeps it fast — typically well under a second — which is the difference between a hook you keep and a hook you delete in week two.

It fails at `high`, not `medium`. Start permissive. You can tighten later, and tightening a working gate is easy. Loosening a gate that everyone has learned to bypass is nearly impossible, because you have already taught the team that the gate is noise.

It tells you how to override. `--no-verify` is printed right there. This looks like a weakness and it is the opposite. A gate with no visible escape hatch gets bypassed by a *global* mechanism — someone disables hooks entirely, or removes the file, and now you have no gate at all and no signal. A gate with a visible per-commit override gets bypassed once, deliberately, in a way that is visible in the reflog and in the person's memory.

> A gate with no visible escape hatch does not get respected. It gets removed. Print the override.

DISTRIBUTING HOOKS TO A TEAM

`.git/hooks` is not version controlled, which is a genuine problem and not one SecureGit invented.

Three options, in increasing order of robustness:

A · A setup script in the repository. `scripts/setup-hooks.sh`, run once by each developer, documented in the README. Simple, and it depends on people running it.

B · `core.hooksPath`. Point git at a version-controlled directory:

```
git config core.hooksPath .githooks
```

Commit `.githooks/pre-commit`. Now the hook travels with the repository. Still requires the one-time config, but it is a single command and it can go in your onboarding script.

C · Managed configuration. Push `core.hooksPath` and SecureGit policy through whatever manages developer workstations. This is the enterprise answer and it is covered on page 55.

Do not rely on option A alone past about five engineers. The failure is silent: someone joins, never runs the script, and is unprotected while believing they are protected. That is worse than no hook, because the team's collective sense of coverage is now wrong.

THE PRE-PUSH HOOK

Same idea, wider net, runs less often:

```
#!/bin/bash
securegit scan --fail-on critical || exit 1
```

Push happens less frequently than commit, so you can afford a broader scan. Note the threshold is `critical` here rather than `high` — a push gate that blocks frequently gets bypassed with `--no-verify` reflexively, and then it protects nothing.

The general rule: the more disruptive the gate, the higher the bar for tripping it.

WHAT TO DO THE FIRST TIME IT BLOCKS YOU

It will block you, and the first time will be at a bad moment. That is when adoption is decided.

Do not reach for `--no-verify` reflexively. Take ninety seconds:

`securegit scan --staged` — read the actual finding.

If it is real: you were about to commit a credential. The hook just did the single most valuable thing it will ever do for you. Rotate and move on.

If it is a false positive: add a `skip_paths` entry or narrow the scanner set. Fix it *now*, in that moment, while it is annoying — because the alternative is that you `--no-verify` past it every day for a month and eventually delete the hook.

The failure mode of security tooling is never a dramatic bypass. It is quiet, incremental erosion by people who were busy.

THE HONEST PERFORMANCE PAGE

Yes, somewhat, in specific places. Here is exactly where, with numbers, so you can decide rather than find out.

Performance is where security tooling adoption actually dies. Not in the evaluation, where everyone is patient. In week three, when a scan adds nine seconds to a loop somebody runs forty times a day.

So: the numbers.

SINGLE FILE

A small source file, all twelve built-in scanners plus one external plugin:

SCANNER	FINDINGS	DURATION
`secrets` / `patterns` / `entropy` (built-in)	6	~5 ms combined
`encoding` / `supply-chain` / `ci-cd` / `container` / `iac` (built-in)	0	~4 ms combined
`deserialization` / `dangerous-files` / `git-internals` / `binary` (built-in)	0	~3 ms combined
External plugin (Python)	3	~23 ms

Total: about 35 ms, of which the external plugin is roughly two-thirds.

That ratio is the whole performance story of this tool, and it repeats at every scale.

THROUGHPUT

PLUGIN TYPE	FILES PER SECOND
Built-in (Rust, in-process)	~1,000–5,000
External (subprocess)	~50–200

Built-in scanners are between ten and a hundred times faster than external ones, because an external plugin pays process-spawn cost — roughly 20–25 ms for an interpreted tool, 5–10 ms for a compiled one — on every invocation.

LARGE REPOSITORY

A substantial open-source repository, several hundred megabytes, five thousand-plus files:

PHASE	TIME
Acquisition	~2.5 s
Extraction	~1.8 s
Scan, all 12 built-in scanners	~35 s
Scan, with ~10 external plugins	~2–3 min

Read those last two rows carefully. Built-in scanners scan a large repository in about half a minute. Adding ten external plugins makes it four to six times slower. That is the trade, stated plainly.

RESOURCE USE

SCALE	CPU	MEMORY
Under 100 files	1 core, under 10%	under 50 MB
5,000+ files	1–4 cores, 40–80%	100–500 MB
Per external plugin	process overhead	+10–50 MB each

Built-in scanners scan a large repository in about half a minute.

Ten external plugins make it four to six times slower. Choose your plugins like you choose dependencies.

THE FIVE RULES THAT KEEP IT FAST

1 · Scan the diff, not the tree. In daily work you almost never need a full scan.

```
git diff main --name-only | xargs
securegit scan
securegit scan --staged
```

This is the single highest-leverage habit in this issue.

2 · Configure `skip_paths` once, properly. `node_modules`, `vendor`, `target`, `dist`, `build`, `.venv`, and whatever your ecosystem's equivalent is. Most first-time "this is slow" reports are a scan of a dependency directory.

3 · Built-in scanners for the hot path. Pre-commit hooks and CI gates should run built-in scanners only. Save the external plugins for scheduled deep scans.

4 · Match the plugin to the stack. A Python security scanner on a Rust repository costs you process-spawn time on every file and finds nothing. Enable only what applies.

5 · Deep scan on a schedule, not on every commit. Nightly or per-PR full scans with the whole plugin set; fast built-in scans inline.

WHAT IS NOT FAST YET, AND IS KNOWN

Stated so you can plan rather than discover:

- **File walking is sequential.** Plugins run concurrently per file, but the walker itself is single-threaded. `PLANNED` — parallel file processing, expected to be a multiple-times speedup on large trees.
- **There is no incremental cache.** Every scan rescans everything in scope. `PLANNED` — hash-based incremental scanning, which would be a large win on repeat scans.
- **No `--jobs` flag yet** to control parallelism explicitly. `PLANNED`.
- **Very large repositories (over a gigabyte) are slow.** Known. Use `--skip-paths` aggressively and scan the diff.
- **First run of an external plugin may fetch a binary.** One-time delay, surprising if unexpected.

None of that is hidden and none of it is fixed today. If your evaluation depends on incremental scanning, it is `PLANNED`, and you should plan for the current behavior.

HOW THIS COMPARES

Honest positioning against tools you may already run:

Gitleaks — very fast, secrets only. SecureGit's built-in `secrets` scanner is in the same performance class and the same scope. If secrets are all you need and you already run gitleaks, you do not need SecureGit for that. You might still want it for acquisition.

Semgrep — moderate speed, much deeper pattern analysis. **Not a competitor.** Wrap it as an external plugin. SecureGit runs it at the right moment and anchors its output.

SonarQube — slow, comprehensive, a different category entirely. Complementary. SonarQube analyzes code quality and security in depth; SecureGit secures the acquisition boundary and the commit gate.

The general rule: SecureGit is an acquisition and boundary layer that happens to include fast scanning. It is not trying to win a static-analysis depth comparison, and a vendor who tells you their tool wins every comparison is telling you something about the vendor.

WHERE YOU SHOULD BE

If you have worked through Part I, this is your state. Tick it off honestly — the gaps are where you will fail next week.

INSTALLED AND VERIFIED

- [ ] `securegit --version` returns a version
- [ ] `~/.config/securegit/` exists
- [ ] I know how to uninstall it (delete binary + that directory)

ACQUISITION

- [ ] I have acquired at least one repository
- [ ] I checked `.git/hooks` and confirmed it was empty
- [ ] I read a `.securegit-report.json`
- [ ] `safe-clone` is aliased and I have used it once instead of `git clone`

SCANNING

- [ ] I have scanned a repository I actually work in
- [ ] I looked at every `critical` finding
- [ ] I rotated anything real (or confirmed there was nothing real)
- [ ] `skip_paths` is configured for my dependency directories
- [ ] I have run `scan-diff` at least once and seen how fast it is

MUSCLE MEMORY

- [ ] `sgit` (or equivalent) is aliased
- [ ] Tab completion works on the alias
- [ ] `~/.securegit_aliases` exists and is sourced
- [ ] `SECUREGIT_FAIL_ON` and `SECUREGIT_SKIP_PATHS` are set

THE GUARDRAIL

- [ ] A pre-commit hook is installed in at least one repository
- [ ] I have seen it pass
- [ ] I know how to override it (`--no-verify`) and why I should not do so reflexively

WHEN IT GOES WRONG

"It found 400 things and I stopped reading." You scanned a dependency directory. Configure `skip_paths`, rescan, and look at `critical` only. Then gate on new findings rather than total findings. Page 15.

"The scan takes too long." You are scanning the tree when you should be scanning the diff. `securegit scan --staged` or `git diff main --name-only | xargs securegit scan`. Page 20.

"The hook blocks me constantly." Your threshold is too low or your `skip_paths` is wrong. Move to `--fail-on high`, fix the paths, and — importantly — fix it the first time it annoys you rather than the tenth. Page 19.

"It won't acquire my private repository." Credentials are not registered. Page 49. Do not debug this during your first hour; use public repositories to evaluate.

"My teammate doesn't have the hook." `.git/hooks` is not version controlled. Use `core.hooksPath` with a committed hook directory. Page 18.

"I forgot to use it." Expected, and the actual reason most adoptions fail. The fix is not discipline. The fix is the pre-commit hook, which runs whether you remember or not, plus a shell alias short enough that `acquire` is not more expensive to type than `clone`.

> **You can stop here.**
>
> Part I is a complete, coherent adoption. Acquisition plus scanning plus a pre-commit hook is a real improvement in your security posture and it costs you almost nothing ongoing.
>
> Parts II through V are for when you want to understand *why* it works the way it does, prove things to other people, or bring a team along. None of it is required. A reader who stops at this page and keeps the habit has gotten more value than a reader who finishes the issue and installs nothing.

PART TWO

THE MENTAL MODEL

Four mechanisms. Why each one is shaped the way it is, and what it costs.

ACQUIRE, NOT CLONE

The entire security argument rests on one ordering decision. Understanding it takes about four minutes and makes everything else in the tool obvious.

Standfirst: *Mechanism 01* · SHIPPED

Why archive-first defeats hooks

THE PROBLEM WITH CLONE, PRECISELY

`git clone` does several things that are individually reasonable and collectively a problem.

It negotiates with a remote and transfers objects. Fine. It writes those objects into a `.git` directory. Fine. It **materializes the repository's configuration and hook directory as live, on-disk state**. And it checks out a working tree using path names that the remote controls.

The trouble is that steps three and four happen before you have looked at anything. By the time `clone` returns success, executable configuration is resident on your filesystem, under your user, and the next git-adjacent command you run — or the next editor you open, or the next build you kick off — may execute it.

There are mitigations. Modern git does not run hooks on clone by default; protocol restrictions exist; `core.hooksPath` can be constrained. All of that is good and none of it is a substitute for not having the executable content on disk in the first place, because mitigations are point fixes against known techniques and your fleet is always some distance behind the current release.

THE REORDERING

SecureGit changes one thing: **the order of operations.**

```
git clone:                  fetch → materialize repo → (inspect never)
securegit acquire:          fetch archive → strip → inspect → materialize repo
```

That is the whole mechanism. Everything else follows from it.

Fetch as an archive. The content arrives as inert bytes. There is no live repository during transfer, so there is nothing for repository configuration to act on.

Strip. Hooks are removed, not disabled. Removal is stronger than disabling because disabling is a configuration state and configuration states can be changed, inherited, or overridden.

Inspect. Scanners run on a tree that is not yet a repository. This is the moment that does not exist in the normal flow — code on your disk, inert, before it has been granted repository status.

Materialize. Now it becomes a real git repository. From this point it is indistinguishable from a cloned one, which is exactly what you want: no lock-in, no wrapper, no special format.

> The order is the product. Fetch, strip, inspect, then confer git-ness. Clone confers git-ness first and inspects never.

WHAT THIS DOES NOT PROTECT YOU FROM

Stated clearly, because overclaiming here would be dishonest and would eventually be found out.

It does not make the code safe to run. If you acquire a repository, read the report, and then run `npm install && npm start`, you have executed the code. That was your decision and it was outside the tool's scope.

It does not detect novel malware. The scanners look for known patterns, credentials, dangerous constructs, and anomalous entropy. A sophisticated, targeted implant designed to look like ordinary code will not be caught by pattern matching, and no tool in this category will catch it.

It does not protect against a compromised upstream you already trust. If a maintainer account is taken over and a malicious commit is published, acquisition works exactly as designed and delivers you the malicious commit — inertly, scanned, with a receipt. The receipt is actually useful afterward. The acquisition did not prevent it.

What it does do is close the window in which merely *obtaining* code can compromise you. That window is narrow, it is real, it is exploited, and it was previously unaddressed by anything in your workflow.

THE REPORT

Every acquisition writes `.securegit-report.json` into the destination.

```
.securegit-report.json
```

It records what was fetched, what was stripped, what the scanners found, and when. **Read it the first ten times.** After that you will have calibrated intuition for what a normal report looks like, and the abnormal one will stand out. That calibration is the actual skill; the file is just how you acquire it.

ACQUISITION AND THE CHAIN

`SHIPPED` Acquisition emits a chain receipt recording the remote URL and the resolved `HEAD` commit at the moment of acquisition.

Acquisition never blocks on the chain. If the signing daemon is unreachable, you get a warning and your repository. This is a deliberate policy choice: read-side operations warn, write-side operations block. A security tool that prevents you from obtaining code because a daemon is down will be uninstalled by lunchtime, and rightly.

The receipt matters later. If upstream force-pushes and rewrites history, your receipt still records what you actually received, at what SHA, at what time. That is the difference between "I think I cloned it in March" and a signed statement about what arrived.

WHEN TO USE PLAIN `GIT CLONE`

There are cases where `acquire` is the wrong tool and you should say so out loud:

Your own repositories, on infrastructure you control. You wrote it. The acquisition threat model does not apply. Use `clone`.

Extremely large repositories where you need a shallow or partial clone. Archive-first fetch has different characteristics. If you need `--depth 1` on a multi-gigabyte monorepo, use git and scan afterward.

Anything inside a build system that expects `git clone` semantics. Do not fight your toolchain. Scan the result instead.

A tool that claims to be correct in every situation is a tool whose recommendations you should discount. `acquire` is for code you did not write, from sources you have not audited. That is a large and growing category, and it is not everything.

SCAN

The architecture is a deliberate two-tier split: a fast core of twelve scanners that always runs, and a slow periphery you opt into. Understanding which tier you are in explains every performance question you will have.

Standfirst: *Mechanism 02* · SHIPPED

Twelve scanners, and a ladder you climb slowly

THE TWO TIERS

Built-in (Rust, in-process). Twelve scanners compiled into the binary — `secrets`, `patterns`, `entropy`, `binary`, `encoding` (Trojan Source), `supply-chain`, `ci-cd`, `container`, `iac`, `deserialization`, `dangerous-files`, `git-internals`. Zero startup cost, shared runtime, direct memory access to file contents, sub-millisecond to low-millisecond per file. These always run.

External (any language, subprocess). Wraps existing tools — the mature security tooling ecosystem that already exists and that you may already run. Language-agnostic, community-maintainable without Rust knowledge, isolated by OS process boundary. Costs 20–50 ms of startup per plugin invocation. Managed by `securegit plugin list|check-updates|update` with a manifest-driven update source; CI can gate on stale plugins with `--fail-on-outdated` and `--fail-on-security`.

The design principle: do not rebuild the security ecosystem in Rust. The tools that exist are good, well-maintained, and specialized. Build a fast core for the things that must run on every commit, and wrap everything else.

THE PLUGIN LADDER

Climb this slowly. Each rung is optional and each adds time.

RUNG 0 · Built-in only. SHIPPED Twelve scanners, no configuration. This is where you start and where most teams correctly stop. Fast enough for pre-commit hooks and CI gates. Between them they cover code (`secrets`, `patterns`, `entropy`, `encoding`), dependencies (`supply-chain`, `dangerous-files`), pipelines (`ci-cd`), infrastructure (`container`, `iac`), runtime execution paths (`deserialization`), binaries (`binary`), and repository metadata (`git-internals`).

RUNG 1 · One external plugin that matches your stack. A Python security scanner on a Python codebase; a Go one on Go. One plugin, chosen deliberately. Scheduled scans, not the hot path.

RUNG 2 · A secrets specialist. Wrap a dedicated secrets tool alongside the built-in `secrets` scanner. Different pattern databases catch different things and the overlap is not total.

RUNG 3 · A SAST engine. Semgrep or similar, with rules tuned for your codebase. This is where scan times get long and where the value gets deep. Nightly or per-PR.

RUNG 4 · Specialized and compliance. Container linting, IaC scanning, license detection, malware signatures. Enable per-repository based on what that repository actually contains.

Most teams should live at rung 0 for the hot path and rung 2 or 3 on a schedule. Running rung 4 on every commit is how you end up with a two-minute pre-commit hook and a team that has learned to use `--no-verify`.

WRITING A PLUGIN

The protocol is deliberately trivial: **a plugin is an executable that takes a file path and prints JSON to stdout.**

Python:

```
#!/usr/bin/env python3
import json, sys

file_path = sys.argv[1]
findings = []

# your logic here

print(json.dumps({
    "plugin_name": "my-scanner",
    "findings": findings,
    "scanned_files": 1
}))
```

Bash:

```
#!/bin/bash
FILE="$1"
echo '{"plugin_name":"my-
scanner","findings":
[],"scanned_files":1}'
```

Install:

```
cp my-plugin ~/.config/securegit/plugins/
chmod +x ~/.config/securegit/plugins/my-
plugin
securegit scan /path/to/test/file
```

That is the entire interface. No SDK, no registration, no manifest, no compilation. If you can write a script that prints JSON, you can write a plugin. This was chosen over a richer interface specifically because organizational security policies are idiosyncratic, and the tool that gets custom policies written for it is the one where writing them takes an afternoon.

PLUGIN TYPES, CURRENT AND FUTURE

TYPE	LOCATION	PERFORMANCE	STATUS
Built-in (Rust)	compiled in	0 ms startup	SHIPPED
External (any language)	`~/.config/securegit/plugins/`	20–50 ms startup	SHIPPED
Native dynamic (Rust + FFI)	`plugins/*.so`	near-native	PLANNED
WebAssembly	`plugins/*.wasm`	fast, sandboxed	PLANNED

WASM is the interesting future one and it is PLANNED, not DESIGNED. The appeal is capability-based sandboxing: a plugin with no filesystem or network access unless explicitly granted. Today, external plugins run as subprocesses and inherit your permissions — which is worth stating plainly, because **installing a plugin is installing software with your privileges.**

THE PLUGIN TRUST PROBLEM

We are going to state this directly rather than bury it, because it is the honest weak point of any plugin architecture.

An external plugin runs as a subprocess with your user's permissions. It can read your filesystem. Depending on the tool, it can make network calls. **A malicious plugin in a security tool is an excellent attack.**

Mitigations available today, all of them procedural rather than technical:

1. Read the plugin source. They are small by design — that is a security property, not just a convenience.
2. Prefer plugins that wrap well-known tools, and verify the wrapped binary independently.
3. Test in a container before putting one on your working machine.
4. Do not install a plugin because a search result recommended it.

PLANNED WASM sandboxing addresses this properly. Until it lands, **plugin installation deserves the same scrutiny as adding a dependency**, and we would rather say so than let you discover it.

WHAT THE BUILT-IN SCANNERS ACTUALLY CATCH

From a deliberately seeded test file of about twenty lines, the built-in set found nine issues in roughly four milliseconds:

2 critical — a cloud access key, a hardcoded password **3 high** — a database password, a dynamic execution call, a hardcoded secret **1 medium** — a high-entropy string **3 low** — development comments flagged by an external plugin

That is the shape of a normal result: the built-in scanners handle credentials and dangerous constructs quickly and well, and external plugins add breadth at a cost in time.

Note the low-severity items came from the external plugin. That is typical, and it is why external plugins belong on a schedule rather than in your commit path.

THE GUARDED COMMIT

Any gate can stop a bad commit. The engineering problem is stopping it without teaching people to route around the gate.

Standfirst: *Mechanism 03* · SHIPPED

A gate that survives contact with a deadline

THE COMMANDS

```
securegit status                    #
working tree state
securegit scan --staged             #
scan what is staged
securegit safe-commit -m "message"  #
scan, then commit if clean
securegit commit -m "message"       #
commit with chain receipt
securegit findings                  #
review current findings
securegit review                    #
guided review of changes
securegit diff                      #
inspect changes
```

`safe-commit` is the one to build a habit around. It scans staged changes and commits only if they pass, in a single step, which removes the "I meant to scan first" failure entirely.

WHY THE GATE IS AT COMMIT AND NOT EARLIER

You could gate at `add`. It would be worse.

Staging is exploratory. People stage, unstage, restage, and split changes across several commits. A gate at `add` fires constantly during normal work, most firings are irrelevant because the content changes again before it is committed, and the annoyance-to-value ratio is terrible.

Commit is the first moment the content is declared finished. That is where a gate belongs — at a natural boundary the developer has already decided to stop at, rather than in the middle of their thinking.

WHY THE *ENFORCEMENT* GATE IS AT PUSH

There is a second gate, and it is not at commit. It is at push. This is worth its own explanation because it is counterintuitive and it is deliberate.

Commits are local and revisable. You can rewrite them, amend them, squash them, and — importantly for the chain — attest them retroactively. A commit that is missing a receipt today can have one added tomorrow with no loss of meaning, because the commit has not gone anywhere.

Push is the first crossing onto shared infrastructure. It is the moment your work becomes other people's problem. It is the last point at which stopping is cheap and the first point at which not stopping is expensive.

So the policy is: **commit-time scanning is advisory and fast; push-time chain enforcement is blocking.** Different gates, different jobs, different costs.

Commits are local and revisable. Push is the first crossing onto

shared infrastructure. Gate where stopping is still cheap.

THE FULL WORKING SURFACE

SecureGit wraps the git operations you use daily. All `SHIPPED`:

Everyday — `status`, `add`, `commit`, `safe-commit`, `diff`, `log`, `show`, `blame` **Branching** — `branch_create`, `branch_list`, `branch_delete`, `checkout`, `merge` **Remote** — `push`, `remote_list`, `server_add`, `server_list`, `server_push` **History** — `stash_save`, `stash_pop`, `stash_list`, `undo`, `tag_create`, `tag_list` **Security** — `scan`, `scan_staged`, `findings`, `posture`, `review` **Repository** — `repo_create`, `worktree_add`, `worktree_list`, `worktree_lock` **Backup** — `backup_add`, `backup_list`, `backup_push` **Escape hatch** — `git_raw`

`git_raw` **matters more than its placement in that list suggests.** It passes a command straight through to git. It exists because no wrapper covers everything, and a wrapper that traps you when it hits its limits is a wrapper you will abandon at the first sharp edge. The escape hatch is a feature, and it is documented, and using it is not a failure.

WHICH OPERATIONS EMIT RECEIPTS

Not everything does, and the boundary is principled.

OPERATION	RECEIPT	WHY
`acquire` / `clone`	**Yes**	Code arrives from outside
`commit`	**Yes**	Content declared finished
`push`	**Yes**	Crosses to shared infrastructure
`merge`	**Yes**	Joins two histories
`scan`	**Yes**	Findings become evidence
`fetch` / `pull`	**Yes**	Content arrives from outside
`blame`	**Yes**	Read-only provenance query
`status`, `log`, `diff`, `add`, `checkout`, `branch`, `tag`, `stash`, `config`	**No**	No boundary crossed

That last row is the one that keeps the tool usable. **Instrumenting everything is how you build something people turn off.**

WORKFLOW SCRIPTS

`SHIPPED` Guided scripts for common operations, exposed so you do not need to know where they live:

```
securegit workflow list
securegit workflow info dev-flow
securegit workflow install
securegit workflow run dev-flow --dry-run
securegit workflow run commit-craft
```

Bundled workflows install to `~/.config/securegit/workflows`. Every workflow supports `--dry-run` at the wrapper level, which prints what would happen and executes nothing. Some workflows have their own internal dry-run, passed through after `--`:

```
securegit workflow run pr-prepare -- --dry-run
```

`--dry-run` **before every unfamiliar workflow.** It costs one flag and it is the difference between learning a tool and being surprised by it.

SecureGit also shows contextual tips after some interactive commands — after branch operations it may suggest `dev-flow`; after staged commit work, `commit-craft`. Tips are local, throttled, and suppressed for `--json`, `--quiet`, and `--compact`. Turn them off entirely:

```
SECUREGIT_WORKFLOW_TIPS=0
```

Turning tips off is respected permanently. A tool that keeps helpfully reminding you of something you dismissed is a tool people come to resent.

THE `--NO-VERIFY` POLICY QUESTION

Sooner or later someone will propose blocking `--no-verify` at the organizational level. Usually after an incident.

Push back on this, and here is the argument.

`--no-verify` is a git flag. You cannot remove it; you can only make it costly. Attempts to block it universally produce one of two outcomes: developers stop using the hooks entirely, or they build a shadow workflow you cannot see. Both leave you with less visibility than you started with.

The better design is what the chain layer does: **let the bypass happen, and record it.** A force-push that rewrites history is not blocked outright — it emits a receipt marking the bypass, and policy can require a joint-approval marker. Now the bypass is a visible, attributable event rather than an invisible one.

Visible bypasses are governance. Blocked bypasses are theater with a shadow IT chaser.

THE CHAIN OF CUSTODY

This is the layer that turns git from a record of assertions into a record of evidence. It is also the layer that requires real infrastructure, so read the cost section before you plan around it.

Standfirst: *Mechanism 04* · SHIPPED (core) · DESIGNED (offline + batch lookup)

Receipts, tiers, and the push gate

THE CORE IDEA

Every operation that crosses a trust boundary produces a **receipt**: a signed, timestamped record linking an action to a verified identity and to the exact content involved.

Receipts are cryptographically linked to each other, so the chain is tamper-evident. Modifying a past receipt breaks the links after it.

What a receipt binds together:

- **What** — a content hash. For a commit, the commit SHA. For a clone, the remote URL plus resolved HEAD. For a merge, the merge SHA composed with both parent SHAs.
- **Who** — a verified identity, not a git author field. Git author is self-asserted and trivially forged; the receipt identity is signed.
- **When** — a timestamp from the signing daemon, not from the local machine.
- **Which operation** — clone, push, merge, blame, scan, and so on.

THE SIGNING MODEL

SHIPPED **The signing key never touches SecureGit.**

It lives in a separate signing daemon. In production the key is hardware-rooted; in local development a software key is used. SecureGit is stateless with respect to key material — it composes an envelope and posts it to the daemon's signing endpoint. It cannot leak a key it never holds.

The cost of this design, stated up front: you need the daemon reachable to produce first-class receipts. That is real infrastructure. It is why the chain layer is Layer 3 in the adoption model on page 8 and not Layer 1.

Offline mode DESIGNED — receipts written to a local store and synchronized when the daemon is next reachable. Air-gapped deployment is on this path.

TWO TIERS OF EVIDENCE, AND WHY THE DISTINCTION IS THE POINT

Tier A — forward-attested. The daemon witnessed the operation as it happened. Strongest evidence.

Tier B — retro-attested. The receipt was added after the fact. Real, useful, and weaker — it proves someone asserted something later, not that the daemon observed it at the time.

The system records which tier you have and never conflates them. This is the single most important honesty property in the design. A governance system that lets weak evidence masquerade as strong evidence is worse than no governance system, because it produces confident wrong answers in exactly the situations where being wrong is expensive.

A system that lets retroactive evidence masquerade as witnessed evidence is worse than no system. It produces confident wrong answers precisely when being wrong is expensive.

WHERE THE RECEIPT LIVES — BELT AND BRACES

SHIPPED Commit receipts are stored in **two places at once**, deliberately:

1 · In the commit message as a trailer.

```
X-ContextOS-Receipt: <receipt-id>
```

Travels with the repository. Survives clone. Readable from `git log` with no daemon and no network. Tamper-evident, because changing it changes the commit SHA.

2 · In the daemon's receipt store, indexed by content hash for fast lookup.

Neither is the single source of truth. They cross-validate. A trailer with no matching daemon record is suspicious. A daemon record with no trailer is suspicious. Agreement is evidence.

This is a good pattern generally, and worth stealing even if you never use SecureGit: **when you need durable evidence, write it to two stores with different failure modes and treat disagreement as signal.**

THE PUSH GATE

SHIPPED (core) — This is the enforcement point.

Before a push executes, SecureGit walks every commit in the push range and checks whether each has a receipt. If any commit does not, the push is blocked:

```
error: push rejected — commit <sha> has
no chain receipt.
Run `securegit attest <sha>` to add one,
or set
chain.fail_on_unattested=warn to push
with a marker receipt.
```

Note the shape of that error message. It says what is wrong, gives the exact command to fix it, and names the configuration key to change the policy. That is the standard every error message in a security tool should meet, because a blocking error without a remedy is just an obstacle.

Adopting on an existing repository: your history predates the chain, so nothing has receipts, so your first push is blocked. This is the single most likely moment for someone to give up on the chain layer.

Two ways through:

- `securegit attest <sha>` — add retro-attested (Tier B) receipts. A `--since` flag supports partial attestation from a starting point.
- Set `chain.fail_on_unattested=warn` — push proceeds and un-receipted commits are marked as pre-chain in the record.

Recommendation: for an existing repository, set the policy to `warn`, adopt the chain from today forward, and accept that history before adoption is Tier B or unmarked. Retro-attesting years of history produces a lot of weak evidence and obscures where the strong evidence starts. A clean boundary — "the chain begins here" — is more useful than a uniform blanket of weak claims.

FORCE PUSH

SHIPPED If a push would rewrite published history, a **bypass receipt** is emitted before the push, and policy can require a joint-approval marker in the commit message.

The default is to require it. The pattern mirrors a physical two-person rule for high-stakes operations.

The philosophy again: force-push is not forbidden. Sometimes it is genuinely correct. It is *recorded*, and it requires a second signal. The bypass becomes an attributable event.

BLAME WITH PROVENANCE

SHIPPED — exposed in the agent tool surface as well as the CLI.

```
securegit blame <file> [--commit <sha>]
[--format=human|json]
```

Standard blame output, augmented per line with the receipt tier and operation type:

```
<sha>  <identity>  A  human-edit   fn
handle_request() {
<sha>  <identity>  A  agent-exec
let body = req.body();
<no-receipt>       ?  (unattested) }
```

That third column is the one that matters, and it is why this feature exists. In a codebase where humans and agents both commit, "which lines did an agent write" stops being an archaeology exercise and becomes a query.

Blame degrades gracefully by default. Unattested lines show `?` rather than erroring. `--strict` errors instead.

The principle: **visibility tools never block; only gate operations block.** Blame is read-only. If it refused to run on partially-attested history, nobody would ever run it on a real repository.

MERGE

SHIPPED A merge joins two receipt chains, so its receipt encodes both parent SHAs in its content hash — preserving the history graph inside the evidence chain.

Multi-author merges — detected from author/committer mismatch or co-author trailers — can require a joint-approval marker. Without it: a warning, and a receipt marked unverified rather than clean.

POLICY AND CONFIGURATION

SHIPPED Policy resolves in a fixed order: **git config → environment variables → built-in defaults.**

Git config first is a deliberate operational choice: it means managed-workstation tooling can push policy through standard git configuration, which every organization already has a mechanism for. Environment variables override, which is what CI needs.

KEY	DEFAULT
`chain.daemon_url`	local endpoint
`chain.fail_on_daemon_error`	`warn`
`chain.fail_on_unattested`	`block`
`chain.require_joint_approval_on_force_push`	`true`
`chain.offline_store`	local path

Hard defaults are always enforced: block on unattested, require joint approval on force-push and multi-author merge. You can loosen them explicitly. You cannot accidentally end up without them.

THE HONEST COST OF THE CHAIN LAYER

Before you plan around this, know what it asks of you.

You need a signing daemon. That is infrastructure to deploy, operate, and monitor. For a solo developer this is probably not worth it. For a team with a compliance requirement it usually is. Decide honestly which you are.

First push on an existing repository will be blocked. Mitigated by `attest` and the `warn` policy, but it will surprise someone. Warn your team before you enable it, not after.

Large pushes add latency. The gate looks up every commit in the push range. On a wide range this is noticeable. A batch lookup endpoint is DESIGNED and not yet shipped; today, expect a per-commit cost.

The daemon is a dependency. Read operations warn, write operations block, and that is configurable — but a daemon outage will interrupt pushes if you have configured it to. Decide your failure policy deliberately, in advance, and write it down.

PART THREE

SUPPLY CHAIN

Ninety percent of your application is code nobody on your team wrote.

This part is about proving what you knew about it, and when.

SBOM, OSV, AND THE NUMBER THAT MATTERS MORE THAN ZERO

An architectural decision worth understanding, because it tells you exactly what this tool will and will not do for your dependencies — and because the boundary moved once, deliberately, and the reasoning is instructive.

Standfirst: *Supply chain* · SHIPPED (anchoring) · DESIGNED (parts)

Anchoring, not scanning

THE GAP

The chain of custody in Part II proves what humans and agents contributed to your repository. It says nothing about the overwhelming majority of a modern application, which is open-source dependencies you did not write and mostly have not read.

Federal procurement, defense customers, and enterprise diligence all ask the same four questions:

1. What is in your software? (SBOM)
2. What is its license posture?
3. What was its known-vulnerability state at a given moment?
4. Can you prove how it was built?

Git history answers none of these.

THE DECISION: ANCHOR, DON'T BUNDLE

SHIPPED **SecureGit records the canonical hash of each supply-chain tool's output at the moment it ran, and anchors that hash in the chain.**

You keep your existing tooling — whatever vulnerability scanner, SBOM generator, and signing tool you already run. SecureGit is the anchor, not the scanner.

The reasoning, which was contested internally and decided in a strategic review:

Bundling scanners expands the maintenance surface without bound. Every ecosystem needs its own. They change constantly. You are now maintaining a scanner fleet instead of a chain.

Fork drift. A bundled copy of somebody else's scanner falls behind, and the gap between your version and theirs is invisible to your customer.

License coverage narrows. Bundling constrains what you can ship and to whom.

And customers already run these tools. Telling a security team to replace a scanner they have tuned for three years is how you lose the deal.

We are the chain anchor, not the scanner. That single sentence decides

what this product is and — more usefully — what it will never become.

THE ONE EXCEPTION, AND WHY IT MOVED

SHIPPED There is exactly one place the anchor-don't-bundle rule was amended, and documenting the amendment precisely is how you keep it from becoming a pattern.

The problem with the pure position: most enterprise customers assume "SBOM provenance" means the system takes an SBOM, looks up vulnerabilities, and anchors the result. They do not want to install another tool. They want the join to happen inside the system.

The amendment: SecureGit bundles a **vulnerability-database lookup client**, not a scanner.

What it does: given a package and a version from an existing SBOM, query a public vulnerability database and return known advisories. Stateless. No local database. No manifest parsing. No dependency-tree resolution. One API, one schema.

What it explicitly does not do: manifest discovery, transitive dependency resolution, license analysis, SBOM generation, or wrapping any of the ecosystem-specific audit tools. Those all go through the normal ingest path.

The distinction that justifies the amendment: a database lookup against a pre-existing SBOM is qualitatively different from running a scanner. The customer already did the hard part — dependency resolution — when they generated the SBOM. This adds a search step, not an analysis step.

Naming the boundary this precisely is the point. A vague boundary erodes. A boundary stated as "lookup against an existing SBOM, never resolution or discovery" is one where future scope creep is detectable by anyone reading the design record.

FOUR SAFETY PROPERTIES WORTH STEALING

Whether or not you use SecureGit, these four decisions are good engineering and transfer to any system that anchors third-party output.

1 · Opt-in, never default. SHIPPED The lookup is an explicit flag. It is never triggered automatically. **Regulated environments must know when an outbound network call happens** — a security tool making a surprise outbound request is a finding, not a feature.

2 · The anchor never waits on the scan. SHIPPED The SBOM receipt is emitted unconditionally. If the vulnerability lookup fails — network error, API down, partial batch failure — the SBOM anchor still succeeds and the scan is recorded as degraded. The auditor learns "SBOM anchored, scan was partially complete," which is true and useful, instead of nothing at all.

3 · `scan_completeness` is a first-class field. SHIPPED Every vulnerability-state receipt carries a completeness value from 0 to 1.

This is the number that matters more than the CVE count.

> **"0 CVEs" without "completeness = 1.0" is not a clean bill of health.** It might mean the scan covered 12% of your components and found nothing in those. Those are wildly different facts and most tooling renders them identically.

4 · Absence is not evidence of absence. SHIPPED If no vulnerability receipt exists for a commit, the correct reading is **"no scan was run,"** not "no vulnerabilities." Any report generated from the chain must surface that distinction explicitly.

That fourth one is the most commonly violated principle in the entire security-tooling industry. Dashboards render "no findings" and "not scanned" the same way constantly, and executives read both as green.

AIR-GAPPED ENVIRONMENTS

SHIPPED Customers who cannot make outbound calls point the tool at a local mirror of the vulnerability database. The mirror's snapshot date is recorded in the receipt, so an auditor can see how current the vulnerability data was at scan time.

We document the download; we do not ship the data. That keeps licensing clean and keeps the tool from carrying a stale copy of somebody else's dataset.

RESCAN, BECAUSE ADVISORIES KEEP ARRIVING

SHIPPED The same dependency set scanned in April may have more known vulnerabilities in May. Nothing about your code changed; the world's knowledge did.

```
securegit sbom rescan <receipt-id>
```

Runs a fresh lookup, bypassing cache, and emits a new vulnerability-state receipt linked to the same SBOM.

The chain accumulates a timeline, and the timeline is the actual product here. "What did we know, and when did we know it" is answerable as a sequence of signed records rather than a reconstruction from memory.

This is what makes monthly compliance refresh cheap. You do not re-anchor the SBOM. You add a new point to the vulnerability timeline against the same anchor.

THE SINGLE-COMMAND FLOW

The whole supply-chain story, once configured, is one pipe:

```
<your-sbom-generator> | securegit
sbom emit --with-osv-scan
```

Two receipts land in the chain — the SBOM anchor and the vulnerability state — cryptographically linked to each other.

The link key is a content hash, not an identifier. This is a small design decision with a large consequence: a content hash is reproducible across serializations, while a generated identifier changes if a record is ever re-signed. Choosing the reproducible key means the join still works years later, after re-signings, migrations, and format changes.

If you build anything that joins records across systems and time, choose the content hash.

LOCK FILES AND LICENSES

Two quieter capabilities that answer two questions auditors ask early: when did this dependency arrive, and are we permitted to ship it.

Standfirst: *Supply chain* · SHIPPED (core) · DESIGNED (coverage)

What changed, and what you are allowed to ship

LOCK-FILE CHANGE RECEIPTS

SHIPPED When a dependency lock file changes, a receipt records the change: the file's content hash before and after, and where format support exists, a count of dependencies added, removed, and version-bumped.

Format coverage is explicitly tiered, and the tiering is honest:

Full parsing SHIPPED — the major lock formats for the ecosystems most represented in the codebase get real dependency-count diffs.

Change-only SHIPPED — every other lock format records **that** the file changed and hashes it, but reports zero counts.

This distinction is deliberate and it is surfaced, not hidden. A record saying "changed, counts unavailable for this format" is materially different from "changed, zero dependencies affected." Reporting the second when you mean the first is how audit records become misleading.

When more formats get full parsing: DESIGNED. Check your build.

LICENSE DETECTION

SHIPPED License detection runs at acquisition time using file heuristics and a standard license identifier database.

Three properties, each of which is a deliberate choice:

`UNKNOWN` **is a valid recorded value.** Not an error, not a blank. Recording "we looked and could not determine this" is more useful than recording nothing, because it distinguishes "unlicensed" from "unexamined."

It is best-effort and says so. License detection from files is genuinely hard. Dual licensing, per-directory exceptions, vendored code with different terms, and modified license text all defeat heuristics. The tool does not pretend otherwise.

It never blocks the clone. SHIPPED License detection at acquisition is **visibility, not enforcement**.

That last one deserves defending, because it looks like a gap.

If license detection blocked acquisition, you could not obtain a repository in order to *examine* its license — which is the most common reason to obtain an unfamiliar repository in the first place. Blocking would make the tool actively counterproductive for the exact workflow it was built to support. Enforcement belongs at a later gate, where you have information and intent.

UNKNOWN is a valid value. "We looked

and could not determine this" is a materially different fact from "we did not look," and the record should be able to tell them apart.

WHAT THIS GIVES YOU IN PRACTICE

Three answers you can produce from the chain that you probably cannot produce today:

"When did this dependency enter our tree?" — the lock-file receipt timeline, with a signed timestamp and an attributed actor.

"What was our license posture at release 4.2?" — the license records anchored around that release point.

"Did anyone review this dependency addition?" — the lock-file change receipt is linked to a commit, which is linked to an identity, which may be linked to an approval.

None of those are exotic questions. All of them currently require an engineer to spend an afternoon in `git log` and produce an answer hedged with "as far as I can tell."

THE TRAP IN DEPENDENCY COUNTS

A tempting metric: "dependencies added this quarter." Easy to compute from lock-file receipts, easy to chart, and nearly meaningless.

A lock file update that bumps four hundred transitive versions after a single direct dependency change looks enormous and is routine. A single added direct dependency that pulls in an unmaintained package with network access looks trivial and is the actual risk.

Use the receipts for provenance, not for scoring. The value is answering "when and by whom," not producing a number that goes on a slide. Numbers that go on slides get optimized, and a team optimizing for fewer lock-file changes is a team batching risky changes into larger, less reviewable updates.

REPORTS THAT LAND WHERE REVIEWERS ALREADY LIVE

Everything in Parts II and III exists to make one specific conversation go differently. The way it lands in that conversation is: as a report your reviewer's tools already read.

Standfirst: *Supply chain & governance* · SHIPPED (SARIF, GitLab SAST, baselines, audit log, compliance report) / DESIGNED (integrated chain audit)

SARIF, GitLab SAST, baselines, audit

SARIF 2.1.0, AND GITLAB SAST V15

SHIPPED One flag on the scan command; the report uploads directly into whatever dashboard your reviewers already open.

```
securegit scan . --format sarif  --
report-output securegit.sarif
securegit scan . --format gitlab --
report-output gl-sast-report.json
```

SARIF 2.1.0 is the OASIS standard for static-analysis output. It carries CWE mappings, stable fingerprints (line-drift resistant), severity, and location, and is understood by **GitHub Advanced Security**, **Azure DevOps**, **SonarQube**, and **DefectDojo**. GitLab SAST v15 is GitLab's native format and lights up the security dashboard and MR widget without additional plumbing.

`--report-output` writes the file even when `--fail-on` gates the exit code, so a single CI step can both *enforce* a bar and *publish* findings to the reviewer's dashboard. That was the single most common failure mode we saw in early pilots: teams that gated hard but had nowhere for developers to see findings ended up disabling the gate.

BASELINES: ADOPTING ON LEGACY CODE WITHOUT FAILING EVERY BUILD

SHIPPED The pattern that unblocks adoption on a mature codebase.

```
securegit scan . --write-baseline "Legacy
findings accepted 2026-08-03; expires
2026-11-03" \
  --expires 2026-11-03
git add .securegit/baseline.json && git
commit -m "chore(security): baseline v1"

# thereafter, every scan and every CI run
securegit scan . --fail-on high --
baseline .securegit/baseline.json
```

Baselines suppress known findings by a **stable fingerprint** that includes rule, file, snippet, and CWE — not by line number. That means shuffling code up or down a file does not invalidate a suppression, and does not re-fail a build. Every suppression carries `reason`, `created_at`, `created_by`, and optionally `expires` (RFC3339). Expired entries automatically re-fire; the

log records who added them and why. `--no-baseline` re-runs without suppressions for review.

Adopt in this order: run once; commit a baseline with an expiry three months out; gate PRs on `--fail-on high`; use the three months to burn down the baseline; renew what you cannot fix with a shorter fresh expiry. The teams that succeed do exactly this. The teams that try to reach zero before turning on gating do not adopt.

THE TAMPER-EVIDENT AUDIT LOG

SHIPPED A separate, always-on log — independent of the chain — that captures every security-relevant event: scan runs, scan blocks, hook execution, baseline writes, credential store operations, policy denials, tool updates.

Every entry is JSONL with a `prev_hash` and `hash` field. Break the chain — edit any entry, delete any entry, reorder entries — and `securegit audit verify` fails with the exact index of the break.

```
securegit audit show --last 100
securegit audit verify
securegit audit export --format cef > /
var/log/securegit.cef   # ArcSight/
Sentinel/QRadar
securegit audit export --format jsonl --
output audit.jsonl      # Splunk/Datadog/
Loki
```

The log lives under `~/.local/share/securegit/audit/` and rotates automatically. Two of its three consumers — SIEM ingest and internal review — work without any signing infrastructure at all, which is why we count the audit log as an on-ramp to Layer 3 that requires no daemon.

THE COMPLIANCE REPORT

SHIPPED A single command that produces a review packet against two frameworks security teams actually cite.

```
securegit compliance report --format
markdown --output compliance.md
securegit compliance report --format
json     --output compliance.json
```

Findings are grouped by **OWASP Top 10 (2021)** via CWE mapping, and by **NIST SSDF v1.1** practice (PS.1–PS.3, PW.1–PW.9, PO.1–PO.5, RV.1–RV.3). The markdown output is intended for direct inclusion in review packets; the JSON output is a starting point for a GRC integration.

It is not a substitute for an audit. It is a substitute for two engineer-days of copy-and-paste when the vendor questionnaire arrives, and it is the artifact that most reliably ends the “but can you prove any of it?” exchange described on the next page.

WHAT THE CHAIN ADDS ON TOP

DESIGNED A report generated directly from chain data — no engineer reconstruction — covering human contribution (who committed what, at which evidence tier), supply-chain state as anchored at a point in time, build provenance, **and the gaps**. Explicitly. Commits without receipts. Scans that did not complete. Periods with no vulnerability data. **The report names its own holes**, which is the property that makes it credible rather than merely impressive.

WHY GAP-REPORTING IS THE CREDIBILITY FEATURE

A report with no gaps is not trustworthy. Every real system has gaps — a daemon outage, a repository adopted late, a scan that timed out, a format without full parsing support.

A report that shows none is either describing a system nobody uses or hiding something. An auditor who has read more than a few of these knows that immediately, and a clean report makes them look harder rather than less hard.

A report that says "these forty commits are attested, these three are not, and here is why" is stronger than one that claims forty-three. It is stronger because it demonstrates that the system is

capable of noticing a problem — which is the only real evidence that its clean findings mean anything.

A report with no gaps is not trustworthy. Every real system has gaps. Showing yours is what makes the rest of it believable.

VERIFICATION WITHOUT ACCESS

SHIPPED A design property worth understanding, because it comes up in every enterprise conversation.

Some anchored payloads live in customer-controlled storage. A verifier without access to that storage **can still verify the hash chain** — the anchoring, the timestamps, the identities, and the integrity of the sequence — without reading the payload contents.

This means an auditor can confirm that a record is authentic, unmodified, and correctly sequenced without you handing over your source code, your SBOM contents, or your scan output.

That separation — verify the chain, gate the contents — is what makes third-party audit possible without disclosure. It is a genuinely useful property and it is not obvious until you need it.

A CAVEAT THAT MUST TRAVEL WITH THE NUMBERS

SHIPPED Vulnerability counts drift. The same dependency set, unchanged, will show more known vulnerabilities next quarter, because advisories are published continuously.

This is correct behavior and it confuses people constantly. A customer sees "3 CVEs in March, 11 in June, nothing changed in our code" and reasonably asks what went wrong.

Nothing went wrong. The world learned more.

Any report generated from this data must explain that, in the report, near the numbers — not in a footnote, and not left to a support conversation. **The rescan timeline is the explanation**: it shows the same anchor with a sequence of point-in-time observations, which makes the drift legible as knowledge accumulation rather than as decay.

PART FOUR

SECRETS & AGENTS

Your agent needs credentials. Your agent must never hold them.

Both of those are true at once, and the resolution is a handle.

HANDLES, NOT VALUES

A narrow idea with a wide consequence: the thing doing the work gets the ability to act without ever receiving the value it acts with.

Standfirst: *Mechanism 05* · SHIPPED (core) · DESIGNED (most of the surface — markers throughout)

The secret broker

THE PROBLEM, SHARPENED

An agent needs to push to a repository. Pushing requires a token. Therefore the agent needs the token.

That reasoning is wrong, and finding where it is wrong is the whole design.

The agent needs the *capability* to push. It does not need the *value* of the token. Those are separable, and separating them is the entire product.

Why it matters more with agents than with humans: a human who sees a token generally does not write it down. An agent operates in a context window that may be logged, summarized into memory, included in a trace, sent to a model provider, or echoed into terminal output that ends up in a transcript. **A secret that enters an agent's context has entered every downstream system that touches that context.** Not maybe — by construction.

THE MECHANISM

SHIPPED **Late binding at the execution boundary.**

1. A user or agent requests an action, referring to a secret by a **handle** — a stable name, never a value.
2. SecureGit validates the requested action against the handle's profile. Is this handle allowed to be used by this command, against this host?
3. Only then does SecureGit resolve the actual value from the backing store.
4. The value is injected directly into the child process, HTTP client, or credential callback.
5. The value is **redacted from all observed output** — stdout, stderr, logs, error messages, and structured responses.
6. A metadata-only audit event is recorded: which handle, which provider, which host, which command family, when, and what the result was. **Never the value.**

The value exists for the duration of one operation, inside one process boundary, and is never rendered anywhere a human or a model can read it.

```
securegit run --with-secret
OPENAI_API_KEY=openai-dev -- npm test
```

The agent typed `openai-dev`. The agent never saw a key.

The agent needs the capability to push. It does not need the value of the token. Everything here follows from noticing

that those are different.

THE SAFETY DEFAULTS

SHIPPED Six rules, enforced rather than recommended:

1. **No command prints a secret value by default.**
2. **Structured and agent-facing responses return handles, provider metadata, and status only** — never values.
3. **Known values are redacted** from stdout, stderr, logs, responses, and error messages.
4. **Use is audited as metadata**: provider, handle, target host, command family, timestamp, result.
5. **Handles can be constrained** by provider, host, repository, command family, expiration, and permitted environment variable names.
6. **Raw reveal, export, or copy requires an explicit human-only path** and is disabled for agent interfaces by default.

Rule 6 is the one that makes the rest coherent. Without it, an agent asks for the value and gets it, and every other rule is decoration.

WRITES NEVER PASS THROUGH SHELL HISTORY

SHIPPED Storing a secret reads the value from an environment variable rather than an argument:

```
OPENAI_API_KEY_VALUE=... securegit secret
set openai-dev \
  --value-env OPENAI_API_KEY_VALUE --yes
```

Never `--value <the-actual-secret>`**.** Command-line arguments are visible in process listings and durable in shell history — two places a secret is very hard to remove from once it lands. This is a small ergonomic cost and it removes an entire class of leak.

Production writes and deletes require both explicit confirmation and an explicit production flag. **Production targets are visually distinct in prompts and logs.** Making the dangerous environment *look* different is a cheap and effective control.

PROVIDER PROFILES, NOT KEY-VALUE SPRAWL

DESIGNED The intended model is that secrets are **provider access profiles**, not arbitrary key-value pairs.

A profile records the handle, the provider, the credential kind, the host, the capabilities it grants, the commands it is permitted for, and where the value actually lives. **The value stays in the backing store.**

```
handle:            gitlab-main
provider:          gitlab
kind:              pat
host:              gitlab.example.com
capabilities:      vcs.read_repo,
vcs.write_repo, vcs.create_repo
allowed_commands:  acquire, clone, fetch,
push, server
backend:           <secrets-manager>
backend_ref:       <path-in-that-manager>
```

Why profiles rather than key-value: a profile carries enough metadata for the tool to apply safe defaults and to refuse obviously wrong uses. A key-value pair carries none, so every safety decision has to be made by the caller, which means it is made inconsistently or not at all.

BACKENDS

SHIPPED Metadata discovery, execution-time value resolution, guarded writes and deletes, and a local store ("credential store v2") adequate for single-user work.

Credential store v2, in detail. Encryption is **ChaCha20-Poly1305** AEAD, keyed to the machine (macOS ioreg UUID, Linux `/etc/machine-id` / DMI product UUID, Windows machine GUID). Every write is atomic — temp file, fsync, rename — with an integrity check on read; a tampered file fails closed rather than silently returning junk. An optional environment variable `SECUREGIT_CREDSTORE_PASSPHRASE` layers a passphrase-derived key on top of the machine key. The store lives at `~/.securegit/credentials.encrypted`

and is not portable between machines by design; migration is deliberate, one credential at a time.

DESIGNED The intended backend order:

1 · Your existing secrets manager. For teams that already have one, this is the source of truth. SecureGit stores **bindings** — handle to location — not copies. Nobody should duplicate hundreds of secrets into a new tool's config directory to adopt it.

2 · OS keychain. For local development. Uses the platform's native credential store.

3 · The built-in encrypted store (v2, above). Compatibility and single-user work. Fine for a laptop; **explicitly not the high-assurance layer** and documented as such.

Backend endpoints are always user-configurable and never hardcoded. Cloud, local, LAN, or an internal hostname — the tool must not assume.

DISCOVERY WITHOUT DISCLOSURE

SHIPPED The most useful single property for agent workflows:

```
securegit secret discover --target dev --
keys-only
```

Returns names, paths, comments, tags, versions, and metadata. **Returns no values.**

This gives an agent enough context to bind the right profile — to know that a key called `PAYMENTS_API_KEY` exists in the production path — without ever exposing what it is. The agent can reason about the shape of your secret inventory while being structurally incapable of reading it.

THE OPEN QUESTIONS, PUBLISHED

These are genuinely undecided, and printing them is how you find out what people actually need before you build the wrong thing.

Should local development default to the OS keychain, even before a full secrets-manager integration ships?

Should agents be allowed to create secrets, or only bind handles a human created? The safe answer is bind-only. The safe answer is also annoying, and annoying controls get bypassed.

What is the minimum useful audit format before secret usage gets full chain receipts?

Which providers ship in the first catalog? Version control, cloud, and model APIs are the obvious three. Order within them is not obvious.

If you have an opinion on any of these, it will change what gets built. That is not a courtesy — it is the fastest available way to avoid building the wrong thing.

FINDING REPOSITORIES WITHOUT HANDLING TOKENS

The same handle discipline, applied to the everyday problem of "which repository was that." · SHIPPED

REGISTERING SERVERS

```
securegit server add github-main --platform github --api-url https://api.github.com
securegit server add gitlab-main --platform gitlab --api-url https://gitlab.example.com/
api/v4
```

Credentials are stored keyed to the server name and resolved through the normal credential chain. **The token is never typed into a search command.**

SEARCHING

```
securegit server search <query>                       # all registered servers
securegit server search <query> --server gitlab-main  # one server
securegit server search <query> --json                # machine-readable
securegit server search <query> --limit 10            # cap per provider
securegit server search <query> --include-groups      # orgs and groups too
```

One command across GitHub and GitLab. Results normalize across providers: server name, provider, repository path, description, web URL, clone URLs, visibility, default branch, and last activity where the provider supplies it.

`--include-groups` returns organizations from GitHub and groups from GitLab, normalized into the same record shape.

THE CREDENTIAL BOUNDARY

SHIPPED Resolution order:

1. A per-server environment variable
2. The encrypted stored credential, keyed to the server name
3. Host-based fallback from existing stored auth

The rule that matters: an agent should call `securegit server search` — or the equivalent agent-facing tool — rather than reading `.env` files or handling personal access tokens directly.

This is a small thing that eliminates a large and extremely common leak path. Agents reading `.env` files to find credentials is not a hypothetical; it is a default behavior of many agent setups, and the resulting token ends up in a context window and then in a log.

> COMPATIBILITY SURFACE
>
> Testing confirmed that GitLab instances expose GitHub-compatible API endpoints for discovery, authentication, and repository listing.
>
> Practical consequence: the same code path works against GitHub, GitHub Enterprise, GitLab, and the smaller self-hosted git servers that implement the same surface.
>
> If you run self-hosted git, this is worth ten minutes of testing before you assume you need something custom.

THE MCP SURFACE

*The same tool, exposed to an agent, with the safety properties preserved rather than bolted on.** · SHIPPED

WHY THIS BELONGS IN THIS MAGAZINE

Every discipline in the AI-engineering transition converges here. Your agents commit code. They pull dependencies. They need credentials. They push.

Every one of those is a trust boundary crossing performed by something that is not a person, at a volume no person could match, with an audit trail that does not currently exist.

An agent doing git operations through raw shell commands has your full permissions, your tokens in its context, and no record beyond whatever the shell happened to log. An agent doing git operations through a brokered tool surface has scoped capabilities, handles instead of values, and a receipt for every crossing.

That is the entire argument for this page.

THE EXPOSED SURFACE

SHIPPED Agents get a structured tool surface covering the same operations humans use — **32 tools** across ten families — delivered as `securegit-mcp`, a standard MCP server:

Repository state — status, log, show, diff, blame **Change** — add, commit, safe-commit, undo, stash, snapshot **Branching** — branch create / list / delete, checkout, merge, worktrees, stack **Remote** — push, remote list, server add / list / push, repository create **Security** — scan, scan staged, findings, posture, review, baseline, audit **Discovery** — search repositories across registered servers (`securegit_search_repos`) **Release** — tag create / list, release list, CI status, pull request list **Backup & recovery** — backup add / list / push, snapshot list / restore **Meta** — update-check (throttled daily, on startup), version, health **Escape hatch** — raw git passthrough (guarded)

THE FOUR PROPERTIES THAT MAKE THIS SAFE

1 · Handles, not values. Agent-facing tools accept secret handles. They do not accept, return, or display credential values. An agent asks to push using `gitlab-main`; it never learns what `gitlab-main` is.

2 · Guarded destructive operations. Operations that destroy or overwrite require explicit confirmation. Some are unavailable to agents entirely, by policy, regardless of what the agent believes it has been authorized to do.

3 · Everything is scanned and anchored. An agent commit passes the same gates a human commit does. An agent push emits the same receipt. **The chain does not distinguish between human and agent for the purpose of requiring evidence — it distinguishes for the purpose of recording which one it was.**

4 · Provenance survives. SHIPPED Blame with chain overlay means "which lines did an agent write" is a query, not an investigation. In a codebase where agents contribute meaningfully, this is the difference between reviewable and unreviewable.

An agent with a raw shell has your permissions and your tokens. An agent with a brokered tool surface has scoped

capabilities and a receipt. The difference is not a policy. It is an architecture.

GRAPH INTEGRATION

DESIGNED Repository operations can update a knowledge graph after acquire, fetch, pull, and push — building a queryable model of code, contributors, and change over time.

```
GRAPHRAG_ENABLED=1
GRAPHRAG_API_URL=<your-endpoint>
securegit push
```

Off by default, and it should stay off until you have a reason.

The connection to the wider argument: an agent that can traverse a repository graph can answer questions that no amount of file reading answers — which commits touched the module that broke, who reviewed them, what the dependency posture was at the time. That is retrieval over structure rather than over text, and it is a different capability class.

THE HONEST WARNING

Do not give an agent write access to a repository you cannot afford to have rewritten, on your first day.

Start read-only. Let it scan, search, review, and report. Watch what it does for a week. Then grant commit access on a branch. Then, much later, push.

The receipts are what make that progression safe rather than a leap of faith. You are not trusting the agent. You are building a record that lets you check, and expanding scope as the record earns it.

PART FIVE

ADOPTION

The engineering was the easy part.

Thirty days, twelve objections, and the failure modes nobody writes down.

SOLO, THEN TEAM, THEN ORG

A sequence that has survived contact with real teams. Each phase is independently valuable, so stopping early is a success rather than a failure.

DAYS 1–3 · YOURSELF

Goal: you are using it daily without thinking about it.

- Install. Alias it. Set up completion.
- Acquire three real repositories with `acquire` instead of `clone`.
- Scan two repositories you actually work in. Fix or rotate anything critical.
- Configure `skip_paths` properly.
- Install a pre-commit hook in one repository.

Exit criteria: you have used it five days running without consulting documentation, and the pre-commit hook has passed at least twenty times and blocked you at most once.

If you do not hit that: stop. Do not bring a team into a tool you are not yet fluent in. Every question they ask, you will be answering by reading docs in front of them, which is the fastest way to lose a rollout.

DAYS 4–10 · ONE REPOSITORY, ONE TEAM

Goal: a shared gate in one place, with the team's consent.

- Pick **one** repository. Not the most critical. Not a toy. Something real that a few people touch.
- Move hooks into version control: commit a hook directory and set `core.hooksPath`.
- Add scanning to CI as **advisory** — reporting, not blocking. (Page 58.)
- Run for a week. Collect false positives. Tune `skip_paths` and scanner selection.
- **Then** flip CI to blocking at `--fail-on high`.

Exit criteria: one week of CI runs with a false-positive rate low enough that nobody has asked to disable it.

The critical sequencing rule: advisory first, blocking second. A gate that blocks before it is tuned generates an incident, a meeting, and a permanent reputation as the thing that broke the build. That reputation outlives every subsequent improvement.

DAYS 11–20 · THE SECOND AND THIRD REPOSITORY

Goal: prove it generalizes, and find out where it does not.

- Add two more repositories, ideally in different languages or ecosystems.
- Notice what breaks. Different stacks produce different false-positive profiles, and the tuning you did on a Python service will not transfer cleanly to a frontend monorepo.
- Write down your organization's `skip_paths` baseline and share it.
- Add one external plugin matched to your stack, on a **schedule**, not in the hot path.

Exit criteria: three repositories, three teams, no open complaints about noise.

DAYS 21–30 · POLICY AND SCALE

Goal: it is infrastructure, not a personal preference.

- Distribute configuration through whatever manages developer workstations. Git config is the intended channel because every organization already has a mechanism for it.
- Decide your chain-layer position. **Be honest about whether you need it** — it requires a signing daemon, and if you have no compliance driver, Layers 1 and 2 may be your correct final state.
- If you do need it: stand up the daemon, set `chain.fail_on_unattested=warn` initially, and communicate before enabling — not after.
- Add supply-chain anchoring if procurement or audit is driving.

Exit criteria: a new engineer joining gets SecureGit configured by your standard onboarding, without anyone explaining what it is.

Advisory first. Blocking second. A gate that blocks before it is tuned earns a reputation that outlives every improvement you make to it afterward.

THE FOUR WAYS ROLLOUTS DIE

Each of these has killed a real adoption. Each has a specific counter.

1 · The noisy first scan. Someone runs it on a mature repository, gets four hundred findings, concludes the tool is useless, and tells everyone. **Counter:** never let the first team-visible scan be untuned. Configure `skip_paths` and `--fail-on high` before anyone else sees output. Gate on new findings, not total findings.

2 · The blocking gate nobody agreed to. Enabled on a Friday. Blocks a release. The release goes out with the gate disabled and it never comes back. **Counter:** advisory for a full week, minimum. Announce the switch to blocking with a date. Let people object beforehand — the objections are usually right and always cheaper before than after.

3 · The champion leaves. One enthusiastic engineer set it all up. They change teams. Configuration rots, nobody knows how it works, it gets removed in a cleanup. **Counter:** version-controlled hooks, documented `skip_paths` baseline, CI configuration in the repository. If it only lives in one person's shell, it dies with their tenure.

4 · The chain layer adopted too early. A team with no compliance requirement stands up a signing daemon, hits the first-push block, spends a week on it, and concludes the whole product is heavy. **Counter:** Layers 1 and 2 are a complete adoption. **Do not adopt Layer 3 without a specific question you need to answer for someone outside your team.** If you cannot name the person who will ask, you are not ready and you do not need it.

WHAT TO SAY IN THE FIRST TEAM MEETING

Keep it to four sentences. Longer pitches invite longer arguments.

"We're adding a scanner to CI. For the first week it reports and doesn't block, so we can see what it finds and tune out the noise. After that it'll fail the build on high-severity findings — mostly credentials. If it's noisy, tell me and I'll fix the configuration rather than lowering the bar."

Note what that does not contain: zero-trust, supply chain, chain of custody, provenance, attestation. **Those are real and they are not why an engineer will accept a new gate in their build.** They accept it because it catches credentials and because you promised to fix the noise.

Save the architecture for the people who ask.

ANSWERED HONESTLY

Including three where the honest answer is that you are right.

1 WE ALREADY HAVE A SECRETS SCANNER IN CI.

Then you have covered one of four layers, at the latest possible moment. CI catches secrets after they are committed and pushed — the credential is already in history and already on the remote. A pre-commit gate catches it before it exists anywhere durable. And CI covers none of the acquisition problem. Keep your CI scanner; add the gate that runs earlier.

2 THIS WILL SLOW DOWN OUR BUILDS.

Built-in scanners run in the tens of milliseconds per file and scan a large repository in about half a minute. External plugins are ten to a hundred times slower. Use built-in scanners in the hot path and external plugins on a schedule, and the build cost is negligible. If you enable ten external plugins on every commit, your build will get slower and that will be a configuration decision, not a tool property. Page 20 has the numbers.

3 DEVELOPERS WILL JUST USE `--NO-VERIFY`.

Some will, sometimes, and that is by design — the override is printed in the error message. A gate with no visible escape is removed entirely rather than bypassed occasionally, and then you have no gate and no signal. The goal is not zero bypasses. It is that bypasses are deliberate, rare, and visible.

4 WE DON'T CLONE UNTRUSTED REPOSITORIES.

You may be right.

If your engineers only ever work in repositories your organization owns, on infrastructure you control, the acquisition layer is not for you. Adopt scanning and skip acquisition. But check the actual behavior first — including what your CI pulls, what your build tooling fetches, and what your agents clone. The answer is often different from the policy.

5 THIS IS ANOTHER TOOL TO MAINTAIN.

It is one binary and one config directory. No daemon, no service, no background process, for Layers 1 and 2. Uninstall is deleting two things. The chain layer does require infrastructure, and that is precisely why it is Layer 3 and optional.

6 OUR SECURITY TEAM WILL NEED TO APPROVE IT.

They should. Hand them page 8 (what it is and is not), page 30 (the plugin trust problem, stated against our own interest), page 44 (SARIF/GitLab SAST reports, the audit log, the compliance report against OWASP Top 10 and NIST SSDF v1.1), and page 63 (the maturity table). The single artifact that most reliably ends the “can you prove any of it?” exchange is `securegit compliance report`, and it takes one command to produce. A tool that publishes its own weak points evaluates faster than one that does not, because the reviewer's first job is finding what you did not tell them.

7 THE FALSE POSITIVES WILL BURY US.

Partly right.

The first scan of a mature repository will be noisy. Every scanner is. The entropy scanner in particular trades false positives for catching credential formats no pattern list knows yet. The fix is tuning, not tolerance — page 60 is a full playbook. But if you adopt this without tuning it, you will be buried, and that outcome is on the adoption, not on you.

8 WE'RE A SMALL TEAM, THIS IS ENTERPRISE STUFF.

Layers 1 and 2 are a solo-developer adoption and cost you ten minutes. Layers 3 and 4 are enterprise and you should skip them until someone external asks a question you cannot answer. Nothing about the first two layers requires a team, a process, or a budget.

9 WHAT HAPPENS WHEN THIS PROJECT IS ABANDONED?

Legitimate, and worth answering directly.

Your repositories remain normal git repositories. Acquisition produces standard git. Receipts live in commit trailers, which are plain text in your history and readable with `git log` forever, daemon or no daemon. Scan output is JSON. **There is no proprietary format and no lock-in anywhere in the design** — which is a deliberate property precisely because this objection is correct to raise about any tool.

10 WE NEED THIS TO WORK AIR-GAPPED.

Scanning, acquisition, credential store v2, the audit log, and the compliance report all work offline today. Vulnerability lookups support a local mirror, with the mirror's snapshot date recorded in the receipt. `securegit update-check` is throttled and offline-tolerant — it warns, it does not phone home to gate. Chain offline mode (fully offline receipt anchoring) is `DESIGNED` and not yet shipped — if air-gap chain-of-custody is a hard requirement, that is the one real gap today and you should plan around it rather than around a roadmap.

11 CAN IT HANDLE OUR MONOREPO?

Partly. Scanning is fast per file and the walker parallelizes across cores, but there is no incremental cache — `PLANNED`. On a very large monorepo, scan the diff rather than the tree, gate on `--baseline` for existing findings, and use `skip_paths` aggressively. If you need full-tree scans of a multi-gigabyte repository on every commit, this is not there yet and pretending otherwise would waste your time.

12 WHY NOT JUST USE GITLEAKS AND COSIGN?

Do, if that is what you need. Gitleaks is excellent at secrets. Signing tools are excellent at signing — in fact SecureGit uses cosign itself for its own release signatures. SecureGit's contribution is the acquisition boundary (which neither addresses), the eleven scanners that are not `secrets`, the SARIF/GitLab SAST/audit/compliance pipeline in one binary, and the chain that links commit, scan, dependency, and push into one queryable evidence trail rather than four disconnected outputs. **If you do not need the chain and you have secrets covered, you may only want the acquisition layer.** That is a legitimate outcome and we would rather you adopt one layer than reject four.

GATES PEOPLE DO NOT DISABLE

CI is where security tooling goes to be turned off. The difference is entirely in how you introduce it.

THE BASIC INTEGRATION

```
name: Security Scan
on: [push, pull_request]

jobs:
  scan:
    runs-on: ubuntu-latest
    permissions:
      security-events: write   # for
SARIF upload
      contents: read
    steps:
      - uses: actions/checkout@v4
      - name: Install SecureGit
        run: curl -fsSL https://<release-
host>/securegit/install.sh | sh
      - name: Scan (report + gate in one
step)
        run: |
          securegit scan . \
            --format sarif --report-
output securegit.sarif \
            --baseline .securegit/
baseline.json \
            --fail-on high
      - name: Upload to GitHub Advanced
Security
        if: always()
        uses: github/codeql-action/
upload-sarif@v3
        with:
          sarif_file: securegit.sarif
```

In production, do not curl-pipe-shell in CI. Pin a version, verify a checksum, or use a container image. A security scan step that installs itself from an unpinned URL is a supply-chain finding inside your supply-chain control, and a reviewer will find it.

GITLAB EQUIVALENT

```
security-scan:
  image: registry.example.com/
securegit:0.12
  script:
    - securegit scan . --format gitlab --
report-output gl-sast-report.json
        --baseline .securegit/
baseline.json --fail-on high
  artifacts:
    reports:
      sast: gl-sast-report.json
```

The `sast` artifact lights up GitLab's security dashboard and MR widget with zero further plumbing.

THE FOUR-STAGE INTRODUCTION

Stage 1 · Advisory, informational. Scan runs, prints findings, always exits zero. One week minimum. You are collecting the false-positive profile of your actual codebase, which you cannot predict.

Stage 2 · Advisory, annotated. Findings appear as PR annotations. Still non-blocking. People start fixing things voluntarily because the finding is in front of them at the moment they are looking at the code.

Stage 3 · Blocking on new findings only. Scan the diff, not the tree. New high-severity findings fail; existing ones do not. **This is the sustainable steady state for most teams**, and many should stop here permanently.

Stage 4 · Blocking on total. Only after the backlog is burned down. Many teams never reach this and should not feel bad about it.

SCAN THE DIFF, NOT THE TREE

The single most important CI configuration decision:

```
git diff origin/main --name-only | xargs
securegit scan --fail-on high
```

Three reasons this is right, and they compound:

Speed. Seconds instead of minutes. **Relevance.** The author can act on it. A finding in code they did not touch is somebody else's problem, delivered at the worst possible moment. **Fairness.** Nobody should be blocked by a finding that predates their branch. This is the one that determines whether the gate is perceived as reasonable, and perception determines survival.

WHERE EACH GATE BELONGS

GATE	SCOPE	THRESHOLD	SPEED
Pre-commit	staged only	`high`	under a second
Pre-push	push range	`critical`	seconds
PR / CI	diff vs. main	`high`	seconds
Nightly	full tree, all plugins	report only	minutes
Release	full tree + SBOM + CVE	`critical`	minutes

Note the nightly row reports rather than blocks. Nightly is where you run everything expensive and let it be noisy, because nobody is waiting on it. That is the correct home for the deep plugin set.

THE METRIC TO WATCH

Do not track findings. Track **the disable rate**.

Count how often people use `--no-verify`, skip the CI step, or add a suppression. That number is the real health of your rollout, and it moves before anything else does.

A rising disable rate means the gate is producing more friction than perceived value, and it means it *now* — weeks before anyone raises it in a meeting, and months before someone removes the gate in a cleanup PR.

Findings count measures your codebase. Disable rate measures your adoption. Only one of them tells you whether the tool will still be here next quarter.

MAKING IT QUIET

A tool that cries wolf gets disabled. Driving noise down is not maintenance — it is the work that determines whether any of this survives.

THE FALSE-POSITIVE PLAYBOOK

STEP 1 · CLASSIFY BEFORE YOU SUPPRESS

Every finding you are about to dismiss is one of four things. **Naming which one changes what you do about it.**

A · Wrong path. Vendor code, dependencies, build output, fixtures. Not your code, not your problem. → `skip_paths`.

B · Wrong scanner for this stack. A Python security scanner on a Go repository. → Disable the plugin for this repository.

C · Genuinely a pattern, deliberately used. Dynamic execution you meant to write. → Suppress narrowly, at that location, with a comment explaining why. **Not repository-wide.**

D · High entropy, not a secret. Hashes, test vectors, base64 assets, minified bundles. → `skip_paths` for asset directories; narrow suppression for individual cases.

Never suppress category (C) globally. "We use dynamic execution in one place" becomes "we no longer detect dynamic execution anywhere," and the next occurrence is invisible.

STEP 2 · FIX THE PATHS FIRST

Most first-week noise is path noise. This one setting resolves the majority of it:

```
export SECUREGIT_SKIP_PATHS="**/
node_modules/**:**/vendor/**:**/target/
**:**/dist/**:**/build/**:**/.venv/**:**/
testdata/**"
```

Or per-invocation:

```
securegit scan . --skip-paths "**/
node_modules/**,**/vendor/**"
```

Write your organization's baseline down and share it. Every team rediscovering this independently is wasted effort and inconsistent results.

STEP 3 · RIGHT-SIZE THE SCANNER SET

```
securegit scan . --plugins
secrets,patterns
```

For a commit gate, `secrets` and `patterns` are usually the right pair. `entropy` **belongs in scheduled scans**, not in the path where someone is trying to commit a fix at 6pm.

STEP 4 · TUNE THE THRESHOLD, NOT THE COVERAGE

Prefer `--fail-on high` over disabling scanners. Keep the coverage and raise the bar. You still see the medium findings; they just do not block. Disabling a scanner loses information permanently; raising a threshold loses only the interruption.

STEP 5 · RE-MEASURE

```
securegit scan . --format json | jq
'[.findings[] | .severity] | group_by(.)
| map({severity: .[0], count: length})'
```

Target: under five findings per hundred files at `high` or above on a tuned repository. Above that, keep tuning. Below that, you are in the range where people read findings instead of dismissing them.

Under five findings per hundred files is where people read them. Above that, they dismiss them. There is no third behavior.

COMMON PROBLEMS

Scan is slow. Scanning the tree instead of the diff. Or scanning a dependency directory. Or too many external plugins in the hot path. In that order of likelihood.

Acquire fails on a private repository. Credentials not registered. `securegit server add`, then retry. Page 49.

Pre-commit hook does not run. Not executable (`chmod +x`), or `core.hooksPath` points elsewhere, or you are committing from a GUI that bypasses hooks — which many do, silently.

Push blocked with "no chain receipt." Expected on a repository adopted after its history began. Either `securegit attest` the range, or set `chain.fail_on_unattested=warn`. Page 36.

Plugin does not run. Not executable, not in `~/.config/securegit/plugins/`, or its output is not valid JSON. Test it standalone first: run it against one file and check that it prints parseable JSON.

Findings differ between local and CI. Different scanner sets, different `skip_paths`, or different versions. **Pin the version in CI and share the configuration.** Divergence between local and CI results destroys trust in both.

Scan finds nothing on a repository you know has issues. Check `--include-git`. Check that your `skip_paths` is not excluding the code. Check the version. In that order.

WHEN TO GIVE UP ON A REPOSITORY

Some repositories are not worth gating, and admitting that is better than a permanently ignored gate.

A twelve-year-old codebase with vendored dependencies, generated files checked in, and test fixtures full of realistic-looking fake credentials may produce noise you cannot tune below the usable threshold in reasonable time.

For those: scan on a schedule, report, do not block. Put the gate on new repositories and on the parts of the old one that are actively developed.

A gate that is always red is not a gate. It is a broken light that everyone has learned to walk past, and it makes every other gate you install less credible.

QUICK REFERENCE

```
ACQUISITION
  securegit acquire <url> <path>          acquire safely (zip+history)
  securegit acquire <url> .               acquire here

SCANNING  ·  12 built-in scanners
  securegit scan <path>                   scan a path
  securegit scan --staged                 scan staged changes
  securegit scan . --fail-on high         non-zero exit at high+
  securegit scan . --min-severity high    display high+ only
  securegit scan . --include-git          include .git directory
  securegit scan . --format json          machine-readable
  securegit scan . --format sarif  --report-output f.sarif   GHAS/Azure/Sonar
  securegit scan . --format gitlab --report-output gl.json   GitLab SAST
  securegit scan . --skip-paths "<glob>"  exclude paths
  securegit scan . --plugins a,b          named scanners only
  securegit scan . --write-baseline "<reason>" --expires <RFC3339>
  securegit scan . --baseline .securegit/baseline.json

EVERYDAY
  securegit status / add / diff / log / blame
  securegit safe-commit -m "<msg>"        scan, then commit
  securegit commit -m "<msg>"             commit with receipt
  securegit commit --ai                   AI-suggested commit message
  securegit findings / review / posture
  securegit undo                          universal undo (18 mutating cmds)
  securegit snapshot                      commit-independent restore points
  securegit snapshot list / restore <id>

BRANCHING & STACKS
  securegit branch-create / branch-list / checkout / merge
  securegit worktree-add <path>
  securegit conflicts / resolve            conflict inspection & resolution
  securegit stack                          stacked-diff workflow
  securegit absorb                         git-absorb port (auto-fixup)

REMOTE
  securegit push
  securegit server add <name> --platform <p> --api-url <url>
  securegit server list / search <query>
  securegit repo-create <name>

SETTINGS  ·  layered: /etc → user → repo → env
  securegit settings show / path / init
  /etc/securegit/config.toml               org policy (min_fail_on, locked_keys, audit_required)
  ~/.config/securegit/config.toml          user
  <repo>/.securegit/config.toml            repo

AUDIT  ·  hash-chained, tamper-evident
  securegit audit show --last <N>
  securegit audit verify
  securegit audit export --format jsonl --output audit.jsonl
  securegit audit export --format cef  > securegit.cef

COMPLIANCE  ·  OWASP Top 10 (2021) via CWE  ·  NIST SSDF v1.1
  securegit compliance report --format markdown --output compliance.md
  securegit compliance report --format json     --output compliance.json

PLUGINS & UPDATES
  securegit plugin list / install <name> / info <name>
  securegit plugin check-updates
  securegit plugin update --all
  securegit update-check                   throttled daily on startup
```

```
WORKFLOWS  ·  20 shipped scripts, 4-level config override
  securegit workflow list / info <name> / install
  securegit workflow run <name> --dry-run

SECRETS  ·  handles, never values  ·  ChaCha20-Poly1305 store
  securegit secret add <handle> --provider <p>
  securegit secret list / info / test <handle>
  securegit secret discover --target <t> --keys-only
  securegit secret rotate <handle>
  securegit run --with-secret NAME=<handle> -- <cmd>

CHAIN & SUPPLY
  securegit attest <sha>  |  --since <sha>
  securegit sbom emit [--with-osv-scan]
  securegit sbom rescan <receipt-id>

ANALYTICS & AI CONTEXT
  securegit gain                          workflow adoption analytics
  securegit <cmd> --compact               60–90% token reduction for LLM contexts

ESCAPE HATCH & HELP
  securegit git-raw -- <any git command>
  securegit --help / <command> --help / --version

CONFIG PATHS
  /etc/securegit/config.toml              org policy (fleet)
  ~/.config/securegit/config.toml         user
  ~/.config/securegit/plugins/            external plugins
  ~/.config/securegit/workflows/          workflow overrides
  ~/.securegit/credentials.encrypted      credential store v2
  ~/.local/share/securegit/audit/         audit log

ENVIRONMENT
  SECUREGIT_FAIL_ON=high
  SECUREGIT_SKIP_PATHS="<glob>:<glob>"
  SECUREGIT_CA_BUNDLE=/etc/ssl/corp-bundle.pem
  SECUREGIT_PROXY=http://proxy.corp.example.com:3128
  SECUREGIT_CREDSTORE_PASSPHRASE=<passphrase>
  HTTP_PROXY / HTTPS_PROXY / NO_PROXY
  SECUREGIT_VERBOSE=1  ·  SECUREGIT_WORKFLOW_TIPS=0
```

WHAT IS REAL TODAY

Every capability in this issue, in one table. This is the page to hand your security reviewer.

AT A GLANCE: As of v0.12.16, every capability that was DESIGNED in the first two categories has shipped except two (offline chain, batch receipt lookup). The maturity table is denser at the top than at the bottom, which is the direction any honest table should move.

CAPABILITY	STATE
Archive-first acquisition (zip+history / zip-only / bare), hooks stripped	SHIPPED
Sanitization report on acquire	SHIPPED
Twelve built-in scanners (`secrets` , `patterns` , `entropy` , `binary` , `encoding` / Trojan Source, `supply-chain` , `ci-cd` , `container` , `iac` , `deserialization` , `dangerous-files` , `git-internals`)	SHIPPED
External plugin system (any language) + managed update manifest	SHIPPED
Staged / diff / tree scanning, severity thresholds	SHIPPED
Output formats: pretty, JSON, **SARIF 2.1.0**, **GitLab SAST v15**	SHIPPED
`--report-output` (report even when gate fails)	SHIPPED
Baselines with stable fingerprints, RFC3339 `expires` , `--no-baseline`	SHIPPED
Pre-commit and pre-push hook patterns	SHIPPED
Workflow scripts with dry-run — **20 shipped**, 4-level config override, language auto-detection	SHIPPED
Server registration and cross-provider search	SHIPPED
Credential resolution chain	SHIPPED
Agent tool surface (**32 MCP tools**, `securegit-mcp`)	SHIPPED
Secret handles, execution-boundary resolution, redaction	SHIPPED
Metadata-only secret audit events	SHIPPED
Guarded production writes and deletes	SHIPPED
Credential store v2 — ChaCha20-Poly1305, machine-bound, atomic, fails closed	SHIPPED
Optional passphrase overlay (`SECUREGIT_CREDSTORE_PASSPHRASE`)	SHIPPED
Chain receipts: acquire, commit, push, merge, scan	SHIPPED
Receipt storage in commit trailers + daemon store	SHIPPED
Tier A / Tier B evidence distinction	SHIPPED
Push gate on unattested commits	SHIPPED
Force-push bypass receipts + joint approval	SHIPPED
Layered configuration (`/etc` → user → repo → env)	SHIPPED
Org policy (`min_fail_on` , `locked_keys` , `audit_required`)	SHIPPED
`securegit settings` **command** (`show` / `path` / `init`)	SHIPPED

CAPABILITY	STATE
Hash-chained audit log with `audit verify`	SHIPPED
Audit export — JSONL (SIEM), **CEF** (ArcSight/Sentinel/QRadar)	SHIPPED
Compliance report — OWASP Top 10 (2021) via CWE + NIST SSDF v1.1	SHIPPED
SBOM anchoring	SHIPPED
Vulnerability-database lookup (opt-in)	SHIPPED
`scan_completeness` field	SHIPPED
Offline vulnerability mirror	SHIPPED
SBOM rescan timeline	SHIPPED
Lock-file change receipts (major formats)	SHIPPED
License detection (best-effort, non-blocking)	SHIPPED
Blame with per-line chain overlay	SHIPPED
Diamond merge receipts + multi-author approval	SHIPPED
Signed releases — CycloneDX SBOM + `SHA256SUMS` + Sigstore keyless (cosign)	SHIPPED
Corporate networks — OS trust store, `SECUREGIT_CA_BUNDLE`, standard proxy env	SHIPPED
`securegit undo` — universal, 18 mutating commands journaled	SHIPPED
`securegit snapshot` — continuous, commit-independent restore points	SHIPPED
`securegit commit --ai` — AI-suggested commit messages	SHIPPED
`securegit conflicts` / `resolve` — first-class conflict UX	SHIPPED
`securegit stack` — stacked-diff workflow	SHIPPED
`securegit absorb` — auto-fixup, port of git-absorb	SHIPPED
`--compact` **mode** — 60–90% token reduction for LLM contexts	SHIPPED
`securegit gain` — workflow adoption analytics	SHIPPED
`securegit update-check` — throttled daily startup check for binary, MCP, plugins	SHIPPED
Secrets-manager backend: metadata discovery + value resolution	SHIPPED
Chain offline store with sync-on-reconnect	DESIGNED
Batch receipt lookup for large pushes	DESIGNED
Integrated chain audit report generator	DESIGNED
Provider profile catalog	DESIGNED
OS keychain backend (Keychain / DPAPI / Secret Service)	DESIGNED
Full lock-file parsing for remaining formats	DESIGNED
Repository graph integration	DESIGNED
Incremental scan cache	PLANNED

CAPABILITY	STATE
`--jobs` parallelism control	PLANNED
Native dynamic (FFI) plugins	PLANNED
WebAssembly sandboxed plugins	PLANNED
Plugin marketplace	PLANNED

GLOSSARY

ACQUIRE — Fetch a repository as an archive, strip hooks, scan, then convert to a normal git repository. The ordering is the security property.

ANCHOR — Record the content hash of an external tool's output in the chain, without bundling that tool.

ATTEST — Add a receipt to a commit after the fact. Produces Tier B evidence.

AUDIT LOG — A tamper-evident JSONL log, independent of the chain, capturing every security-relevant event with `prev_hash` linkage. Verified by `securegit audit verify`; exports to JSONL or CEF.

BASELINE — A file (`.securegit/baseline.json`) suppressing a known set of findings by stable fingerprint. Every entry carries reason, creator, timestamp, and optional RFC3339 expiry.

BUILT-IN SCANNER — One of the **twelve** scanners compiled into the binary. Zero startup cost.

CEF — Common Event Format. The audit-log export format understood by ArcSight, Microsoft Sentinel, and QRadar.

CHACHA20-POLY1305 — The AEAD used by credential store v2. Machine-keyed by default; optionally passphrase-augmented.

CHAIN OF CUSTODY — A tamper-evident sequence of signed receipts linking actions to identities and content.

COMPLIANCE REPORT — `securegit compliance report`. Findings grouped by OWASP Top 10 (2021) via CWE and by NIST SSDF v1.1 practice.

COSIGN / SIGSTORE KEYLESS — The signing model used for SecureGit releases. Signatures anchor to short-lived OIDC identities rather than long-lived keys.

DIAMOND RECEIPT — A merge receipt encoding both parent commits, preserving the history graph inside the evidence chain.

EXECUTION BOUNDARY — The last moment before an operation runs; where a secret value is resolved and injected.

EXTERNAL PLUGIN — An executable that takes a file path and prints JSON findings. Any language. Subprocess cost.

FORWARD-ATTESTED (TIER A) — The signing daemon witnessed the operation as it happened.

HANDLE — A stable name referring to a secret. Used by humans and agents; never a value.

JOINT APPROVAL — A second signal required for high-stakes operations such as force-push or multi-author merge.

PUSH GATE — Enforcement at push time, checking that every commit in range carries a receipt.

RECEIPT — A signed, timestamped record binding an operation to an identity and a content hash.

RETRO-ATTESTED (TIER B) — A receipt added after the fact. Real evidence, weaker than Tier A, always distinguished from it.

SANITIZATION REPORT — The record written at acquisition describing what was fetched, stripped, and found.

SCAN COMPLETENESS — What fraction of a scan actually finished. Zero findings without full completeness is not a clean result.

SARIF 2.1.0 — OASIS static-analysis report format understood by GitHub Advanced Security, Azure DevOps, SonarQube, DefectDojo.

SKIP PATHS — Glob patterns excluded from scanning. The primary false-positive control.

SNAPSHOT — A commit-independent restore point captured by `securegit snapshot`.

TRAILER — A key-value line in a commit message. Where receipt identifiers live so they survive without a daemon.

TROJAN SOURCE — CVE-2021-42574. The class of source-code attack that hides intent using BiDi override, homoglyph, or zero-width Unicode. Detected by the `encoding` scanner.

UNDO — `securegit undo`. Universal reversal for 18 mutating commands, backed by an operation journal separate from git reflog.

YOU ALREADY RAN THE CODE. YOU JUST CALLED IT A CLONE.

BOTTLENECK · NO. 01 · THE UNTRUSTED CLONE

START AT PAGE 11. IT TAKES TEN MINUTES.