skarn

RESEARCH / MEASUREMENT

What leaks into AI coding sessions? We measured it.

130

credential and attack incidents across 62 AI coding sessions, in five assistants.

A teardown of what AI coding assistants persist to disk: 42 secret and attack classes, 11 multi-stage attack chains, and one chain that crosses from one assistant to another.

Executive summary

Every AI coding assistant writes the session to disk: the prompts, the files it read, the commands it ran, and the output it got back. That transcript is not a repository, so the scanner that guards your commits does not open it. This report is a teardown of what ends up in there.

Over a labeled corpus of 62 AI coding sessions across five assistants, Skarn reports 130 credential and attack incidents: 33 critical, 86 high, and 11 medium. They come from 198 rule matches spanning 42 distinct classes of secret and attack, and they include 11 multi-stage attack chains. The aggregate risk score is 100 out of 100. Of the 62 sessions, 23 carry a finding and 11 carry a critical one.

The figure that matters most is not the total. It is that 40 of the 130 incidents exist specifically because a file was read into, or echoed back through, a transcript: the assistant opened a `.env` and its contents landed in the log, or read a credentials file, or repeated a secret back in its own reply. The credential was doing exactly what it was supposed to do, sitting in a gitignored file that no commit scanner looks at, right up until an assistant copied it somewhere that is not gitignored.

The corpus is entirely synthetic. No real developer sessions, accounts, or third-party data were scanned, and every credential in it is a fabricated, non-functional token authored for measurement. This is therefore an anatomy of what these tools persist to disk, not an estimate of how often it happens in the wild. That distinction is load-bearing and is stated again wherever a number appears.

The figures at a glance: 62 sessions scanned across 5 assistants and 36 projects; 130 incidents (33 critical, 86 high, 11 medium) from 198 rule matches; 42 distinct secret and attack classes; 11 multi-stage attack chains across 7 sessions; 1 chain correlated across two different assistants; 40 of 130 incidents transcript-native; 119 incidents carrying an OWASP LLM tag, 98 carrying CWE-798, and 52 carrying a MITRE ATLAS technique; aggregate risk score 100 out of 100.

This report gives the full method, the complete class breakdown, the chain analysis, the standards mapping, and a short checklist for measuring your own exposure with the same tool.

What we measured, and why this is a teardown and not a prevalence study

There is a tempting headline available here, and we are not going to write it. "X% of real developer sessions contain a live credential" would be the strongest possible claim, and this corpus cannot support it. Producing that number honestly requires consent, an ethics review, and an anonymization pipeline over real developers' machines. We have not built those, so we did not scan anyone.

What a labeled, authored corpus can support is the anatomy: what classes of secret and attack accumulate in these files, how a scanner correlates them, and what a commit-oriented tool structurally cannot see. Every number in this report is the output of one command over a corpus you can reason about, run with a named binary version, and check against a committed golden snapshot. It is reproducible and falsifiable. It is not epidemiology, and it does not pretend to be.

The corpus is dense with incidents by construction, because it was built to exercise a detector across its range. A real developer's laptop is not. Read the ratios below as structure, not as base rates.

The surface nobody scans

There are no repositories in this corpus. There are no commits, no branches, and no pull requests. There are 62 session transcripts, the files that Claude Code, Codex CLI, Cursor, Gemini CLI, and GitHub Copilot write to disk as they work, and inside them are 130 incidents that a secret scanner pointed at git history has nothing to look at.

That is the structural point, and the class breakdown makes it concrete. Forty of the 130 incidents exist specifically because a file was read into, or echoed back through, the transcript. The assistant opened a `.env` and its contents landed in the log: 16 incidents. The assistant read a credentials or config file, and the read itself is recorded: 15 more. The model repeated a secret back to the developer in its own reply: 6. Somebody dumped the environment into a shell and the output was captured: 3.

In every one of those cases the credential is doing exactly what it is supposed to do. It is in a gitignored file. It was not committed. No commit scanner, no push protection, and no provider partner-scanner has anything to inspect. Then an assistant reads it, and a verbatim copy lands in a file that nobody gitignored, that no tool scans, and that stays on disk.

The transcript is not a secondary artifact. It is the highest-fidelity record of the work: what was read, what was run, what came back. It is what makes the tool useful. It is also, by construction, the one place your secrets end up in plain text with nothing watching.

How the corpus is built: real formats, synthetic secrets

Two rules govern every session in the corpus.

Real formats. Each session is written in the genuine on-disk transcript format of the assistant it represents, so the scanner is exercised against the same parsing surface it meets on a real machine, across all five tools. The corpus spans 36 projects, with content timestamps from 2026-02-12 to 2026-06-25.

Synthetic secrets. Not one real credential appears anywhere. Every token is fabricated, format-valid, and non-functional: it matches a detector's shape, and its checksum where one applies, but it unlocks nothing. Skarn masks each of them in its own output regardless, so the values that appear in this report, such as `AKIA****BY`, are the whole of what the tool prints.

The corpus holds three kinds of session. Credential-leak scenarios, where a secret reaches the transcript through a file read, a shell command, or the model's own reply. Attack-chain scenarios, where a poisoned input drives a credential read and then an exfiltration attempt. And clean decoy sessions, built from content hashes, example values, and identifiers, which exist to prove the scanner stays silent where it should: they are the reason a 130-incident total means something.

Extended methodology

What the numbers measure, precisely

Incidents are findings at or above medium severity, which is the CLI's default reporting threshold. That is why the severity split reads 33 critical, 86 high, 11 medium, and 0 low: a run at `--severity low` surfaces more. Every figure in this report is taken at the default threshold, so it is the number a user sees on a first run.

Rule matches and incidents are different counts. The scan records 198 individual rule matches, which deduplicate into 130 reported incidents. The larger number is how often a rule fired; the smaller is how many distinct findings a human is asked to look at.

Chains are counted two ways, and conflating them would overstate the result. The scan reports 11 in-session attack chains, each a multi-phase attack correlated inside one session and reported as a single critical finding. Separately, it reports 1 cross-session chain, where the same secret fingerprint appears in two different sessions. They are different structures and this report keeps them apart.

The aggregate risk score of 100 out of 100 is saturated. It is a property of a corpus built to be dense, and it should not be read as a severity measurement of anything in the field.

Corpus composition

The distribution across assistants tracks how the corpus was authored, not any assistant's relative safety. Most sessions are Claude Code, so most incidents are too. The five-way split matters here as coverage: the same classes fire in every tool, in each tool's own on-disk format.

Assistant	Sessions	Incidents
Claude Code	54	105
Codex CLI	4	14
Cursor	1	4
Gemini CLI	1	4
GitHub Copilot	2	3
Total	62	130

Nothing in this table says that one assistant leaks more than another. It says the corpus contains more Claude Code sessions than anything else, and that the detector reads all five formats.

Redaction

Skarn masks every secret it reports. A finding shows a fixed-form mask, such as `AKIA****BY` or `vT3x****Xn`, which does not disclose the value's length. The redacted report is what a scan produces, what this document quotes, and what is safe to hand to whoever owns the response. The corpus is synthetic, so there is no live value behind any mask in this report in any case.

Honest limitations

Stated plainly so the numbers survive scrutiny rather than invite it.

This is an anatomy, not a prevalence study. The corpus is authored. It shows what these assistants persist to disk and what Skarn surfaces in it. It does not tell you what fraction of real developers have a live key in a transcript right now, and no honest reading of it can. Anyone quoting a percentage of real-world sessions from this document is quoting something we did not measure.

The composition is a design, so the ratios are too. The per-assistant split, the severity mix, the class distribution, and the saturated risk score are all properties of a corpus built to exercise a detector across its range, not samples from a population.

The numbers move with the corpus and the binary. Every figure here is the output of one version of the scanner over one version of the corpus, both named below. Scenarios get added and rules get tuned, so a re-run on a later build can legitimately produce a different count. That is why the command and the version are printed rather than a static claim.

Precision is measured elsewhere. This report counts what the scanner found. It does not measure how often the scanner is wrong. That is a different question, with a different corpus and a build gate behind it, answered in our secret-detection precision and attack-chain detection precision studies.

What is in the transcripts: the full class breakdown

Forty-two distinct classes fired across the corpus. The complete list follows, by incident count.

Class	Incidents	Kind
generic-api-key	23	Credential
dotenv-contents-leaked	16	Transcript-native
attack-chain	11	Attack
dotenv-file-read	11	Transcript-native
aws-secret-access-key	7	Credential
secret-echo-back	6	Transcript-native
connection-string-with-password	5	Credential
exfiltration-channel	5	Attack
credential-file-read	3	Transcript-native
exfiltration-domain-in-command	3	Attack
aws-access-token	2	Credential
env-var-dump	2	Transcript-native
github-pat	2	Credential
npm-access-token	2	Credential
prompt-poisoning-shell-injection	2	Attack
redis-url	2	Credential

Class	Incidents	Kind
ssh-private-key	2	Credential
anthropic-api-key	1	Credential
base64-exfiltration-pipeline	1	Attack
bearer-token-in-curl	1	Credential
digitalocean-pat	1	Credential
gcp-api-key	1	Credential
gcp-service-account-key	1	Credential
gitlab-pat	1	Credential
google-maps-api-key	1	Credential
huggingface-access-token	1	Credential
inline-credential-assignment	1	Credential
mcp-chain-exploitation	1	Attack
mcp-config-file-read	1	Transcript-native
mongodb-connection-uri	1	Credential
npm-fine-grained-access-token	1	Credential
npmrc-auth-token	1	Credential
printenv-secret	1	Transcript-native
private-key	2	Credential
prompt-poisoning-file-combined	1	Attack
prompt-poisoning-web-injection	1	Attack
pypi-upload-token	1	Credential
sendgrid-api-token	1	Credential
slack-bot-token	1	Credential
stripe-access-token	1	Credential
twilio-account-sid	1	Credential
twilio-api-key	1	Credential

They fall into three groups, and the third is the one that separates a session scanner from a secret scanner.

Credentials, by service. Cloud keys lead the count: AWS secret access keys and access tokens, Google Cloud API keys and a service-account JSON, a DigitalOcean token. Then the source and package hosts: GitHub and GitLab personal access tokens, npm access tokens both classic and fine-grained, an `.npmrc` auth token, a PyPI upload token. Then the SaaS keys a service actually runs on: Stripe, Twilio, SendGrid,

Slack. Then AI provider keys, Anthropic and Hugging Face. Then the datastores, where the password is in the URI itself: connection strings with embedded passwords, Redis URLs, a MongoDB URI. Then private keys: 4 incidents across SSH keys and PEM blocks. The single most common class is the generic API key, at 23 incidents: a credential-shaped value with no vendor prefix to identify it.

The transcript-native classes. These have no equivalent in a git scanner, because they are not about a file's contents. They are about an event that only a transcript records. `dotenv-contents-leaked` fires when a `.env` file's contents appear in the log. `dotenv-file-read`, `credential-file-read`, and `mcp-config-file-read` fire on the read itself. `secret-echo-back` fires when the model repeats a secret back in its reply. `env-var-dump` and `printenv-secret` fire on a shell dumping the environment. Together they are 40 of the 130 incidents.

The behavioral classes. Prompt poisoning through a web fetch, a shell tool result, or a file. Exfiltration channels and exfiltration domains named in a command. A base64 exfiltration pipeline. An MCP tool-chain exploitation. These are not secrets at all. They are the attack, caught in the act, and they are what the chains are assembled from.

The attacks, not just the keys

A leaked key is an incident. A poisoned input that drives a credential read and then an exfiltration is an attack. The scan correlates 11 of them, across 7 sessions, in two different assistants: 8 chains form in Claude Code sessions and 3 in Codex CLI sessions.

A chain forms when the phases co-occur. Something poisons the context: a web page fetched into the session, a tool result, a file. Something reads a credential. Something moves it out: a request to a domain that is not yours, a base64 pipeline, a DNS lookup with the secret substituted into the hostname. Skarn correlates the phases and reports the chain as one critical finding rather than three unrelated ones, which is the difference between an alert a human can act on and three they have to assemble themselves.

The chain worth walking end to end is the one that leaves the tool entirely. Skarn fingerprints each secret it finds and looks for the same fingerprint elsewhere in the corpus. It reports one cross-assistant chain:

```
recon   2026-05-28T09:14:40Z  claude  dotenv-file-read    demo-virtucon-recon-001
exfil   2026-05-29T16:04:00Z  codex   secret-echo-back    demo-virtucon-xfil-001
secret  vT3x*****Xn           crossCli: true
```

On the first day, a Claude Code session reads a project's `.env` file, and the value lands in that transcript. Thirty-one hours later a Codex CLI session, a different assistant from a different vendor, writing a different file on disk in a different format, echoes the same secret back.

Neither session on its own tells you very much. Correlated on the secret's fingerprint, they describe a credential that has spread across two tools and two days, and the developer has no reason to know it. A scanner that reads one assistant's format, or one session at a time, does not see this. It is the clearest single illustration of why the session is the right unit of analysis: the secret is not in a repository, it is not in one tool, and it is not in one moment.

Mapped to the standards

Every incident carries its taxonomy tags, so findings land in the frameworks a security team already reports against rather than in a vocabulary we invented. Of the 130 incidents, 119 carry an OWASP Top 10 for LLM Applications tag, 98 carry a CWE, and 52 carry a MITRE ATLAS technique.

OWASP Top 10 for LLM Applications	Incidents
LLM02:2025 Sensitive Information Disclosure	114
LLM01:2025 Prompt Injection	4
LLM06:2025 Excessive Agency	1

MITRE ATLAS technique	Incidents
AML.T0037 Data from Local System	32
AML.T0025 Exfiltration via Cyber Means	9
AML.T0057 LLM Data Leakage	6
AML.T0051.001 Indirect Prompt Injection	4
AML.T0053 AI Agent Tool Invocation	1
AML.T0068 LLM Prompt Obfuscation	1

CWE	Incidents
CWE-798 Use of Hard-coded Credentials	98

The distribution is the story in miniature. The overwhelming majority of what accumulates in a session log is sensitive information disclosure: the assistant did its job, and the job left a credential behind in the transcript. The prompt-injection and exfiltration tags are the smaller, sharper set. Not an accident, but an attack, and the reason the chains are rated critical.

Why a session leak goes unseen

A key pushed to a repository has a fire brigade. Push protection blocks it, the provider's partner scanner sees it and revokes it, and the pre-commit hook that should have caught it fails loudly. The whole machinery is pointed at one place: the commit.

A key sitting in an assistant's session file is in none of those paths. It was not committed, so push protection has nothing to inspect. It is on a laptop, not in a repository, so no partner scanner receives it. The file is not in the project tree, so a repository scan walks straight past it. Nothing in the standard toolchain reads it.

In this corpus the credential from the cross-assistant chain sits in one transcript for a full day before a second assistant repeats it, and both files remain on disk afterwards, in plain text, readable by any process running as that user.

The vendor answer to this is that they do not train on your data, which is true and beside the point. The exposure is not the model. It is the file the client wrote on your own machine, in your own home directory, that nothing on the machine is looking at.

Reproducibility

The figures are the output of one command over a named corpus with a named binary, not a slide.

The binary is skarn 0.17.0, the signed release build, run at its default severity threshold of medium and above.

The scan is a single unwindowed pass over the whole corpus, emitted as JSON. Incidents, severities, rule classes, match counts, chain count, and risk score are all fields of that report. The session count is the recall enumeration, whose `session_count` is 62, the same figure the scan itself prints.

```
bin/skarn-demo check  --hours 0      --format json
bin/skarn-demo recent --hours 876000 --format json
```

The corpus carries a committed golden snapshot of the scan, and a verification script diffs a live scan against it. It prints `verify: OK (live scan matches golden)`, which is the guard that the numbers in this report still describe the corpus. The corpus is private, so the scan is reproducible by us and auditable by method, not downloadable.

The figures published here are page-local literals with a documented reproduce command behind them, so they are re-derived and re-checked against the golden before each publication rather than carried forward.

How to assess your own exposure

The corpus shows the shape of the problem. Measuring yours takes a few minutes and does not leave your machine.

1. Install Skarn. It is a single static binary for macOS, Windows, and Linux, on both Intel and ARM.
2. Run `skarn check`. It scans your AI coding-assistant sessions locally, across Claude Code, Codex CLI, Cursor, Gemini CLI, and GitHub Copilot, with no network calls by default. Nothing is uploaded.
3. Read the redacted report. Secrets are masked to a fixed form that does not disclose the value's length, so the report is safe to keep and to forward.
4. Own the response. Skarn surfaces what is in the transcripts, with the rule that fired, the session it fired in, and the chain it belongs to. What to do about a finding is your team's call, not the tool's. It is a detection and awareness tool, and it does not act on your credentials for you.

Sources and further reading

The taxonomy identifiers used throughout are the published frameworks, cited as the scan emits them.

- OWASP Top 10 for Large Language Model Applications (2025): https://owasp.org/www-project-top-10-for-large-language-model-applications/
- MITRE ATLAS, the adversarial threat landscape for AI systems: https://atlas.mitre.org/
- CWE-798, Use of Hard-coded Credentials: https://cwe.mitre.org/data/definitions/798.html
- Skarn secret-detection precision, the companion measurement of how often the scanner is wrong: https://getskarn.com/research/secret-detection-precision/
- Skarn attack-chain detection precision, the same method applied to the chain surface: https://getskarn.com/research/attack-chain-detection-precision/
- Skarn, local-first AI coding session security: https://getskarn.com/

Measured 2026-07-12 with skarn 0.17.0. The corpus is synthetic; no real developer sessions, accounts, or third-party data were scanned.