RESEARCH REPORT · 2026

Vibe Coding Security Report

Why "fix the vulnerabilities" isn't enough

Two approaches compared across 10 projects: finishing the work and scanning afterwards, and building with a security md from the start. The industry picture, the attacker's view and practical recommendations.



Sedat Özdemir

Sr. Expert Pentest & Vulnerability Management Engineer

November 2026 · v1.0

sedatozdemir.com.tr

HOW THIS REPORT FLOWS

Contents

Foreword 3
Where this report came from and who it is for

Executive summary 4
The main findings and numbers on one page

01 The landscape: everyone is vibe coding 5
How vibe coding became the default in two years

02 Why are vulnerabilities increasing? 8
How the model, the prompt and speed make flaws bigger

03 Why does the attack surface grow as AI improves? 11
Agents, rules files, hallucinated packages and a faster attacker

04 The study: scope and method 14
10 projects, 4 states, 3 test methods

05 Findings 17
Overall picture, flaw classes, category heat map, results per project

06 Why scan-and-fix falls short 22
Four reasons, two cases and the limit of repeating the prompt

07 What a security md changes 25
The file I used, three observations and why a short file works better

08 How Claude provides this security 28
The layers in Claude Code and their limits

09 Through the attacker's eyes: the security view 31
Where I start when I test a vibe-coded application

10 Conclusion 33
The story in brief and what comes next

11 Recommendations 35
Practical steps for developers and teams

Appendix 37
Limitations, glossary, sources and ready-made md files

WHY I DID THIS STUDY

Foreword

Over the past year, a growing share of the applications I test come from teams who tell me "we built it with AI in a few weeks." In the kickoff meeting I usually hear the same sentence: "Before going live we had Claude scan it, and it fixed what it found." Within the first hour of testing, I often find the same picture. The code looks clean, the linter is happy, the tests pass, and I can open another user's invoice by changing the ID in the URL.

That contradiction had been bothering me for a while. The scan clearly did something, because findings like injection and hardcoded secrets had dropped noticeably. Yet the flaws that matter most to an attacker were still there. My question was simple: how much of the problem does "scan the project and fix the vulnerabilities" actually solve, and does writing the rules from the start really make a difference?

I wanted to answer that by measuring, not guessing. I built the same applications in 10 different stacks in four different ways and put each one through the same test. Then I set the results next to what is happening across the industry, because what I see in my lab could have stayed a single anecdote. It didn't. Sources from Veracode and Apiiro to Escape and CrowdStrike point in the same direction.

I wrote this report for three groups. Developers who ship products with vibe coding, so they can see what to do and why. AppSec and pentest teams who work with them, so they know what is left behind the sentence "the AI scanned it." And managers, so they can see in numbers where the cost of speed is paid.

The sample is small, and I say so plainly in the Appendix. Still, the direction is clear: telling the model to fix the vulnerabilities isn't enough. The rules need to be written from the start, in the place the model reads every time it writes code.



Sedat Özdemir
Sr. Expert Pentest & Vulnerability Management Engineer
sedatozdemir.com.tr

AT A GLANCE

Executive summary

Vibe coding has multiplied the speed of writing code. Security has not grown at the same rate, and most teams try to close the gap with a single prompt at the end: "Scan the project and fix the vulnerabilities." This report measures how well that prompt works across 10 projects and compares it with a security md written from the start.

KEY NUMBERS FROM THE EXPERIMENT

214
Total findings across 10 projects in the baseline; 61 critical or high

55%
Share of findings closed by the scan-and-fix prompt

73%
Share of design flaws still present after scan-and-fix (authorization, IDOR, business logic)

13
New findings the model created while fixing

81%
Fewer findings than the baseline in versions built with a security md from the start

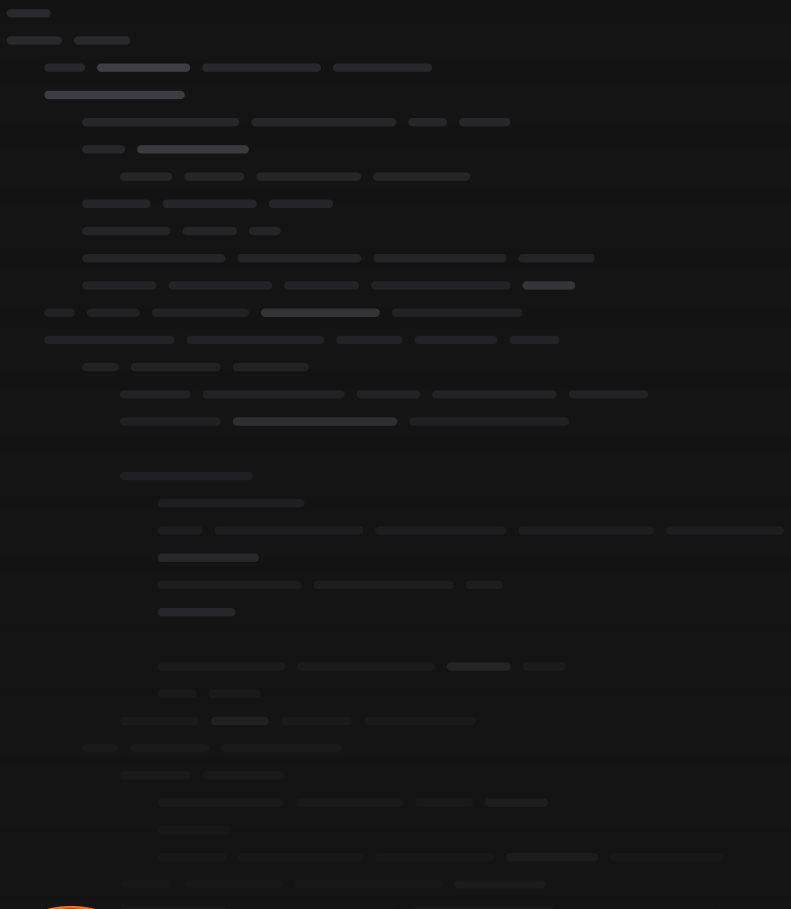
11 → 58 min
Time to the first critical/high finding from an attacker's view, baseline vs security md

Four main findings

1. **Scan-and-fix does half the job.** 55% of the findings were closed. Almost all of them were pattern flaws you can see by reading the code (injection, XSS, secrets, headers).
2. **Design flaws stay where they are.** 73% of findings such as missing authorization, IDOR, tenant isolation and business logic were still there after scan-and-fix. Repeating the prompt three times did not change the picture.
3. **Writing the rule first stops the flaw from appearing.** In versions built with a security md, design flaws went from 96 to 29. With a review layer added, total findings dropped to 17.
4. **The remaining risk sits with the people who know the business.** 10 of the last 17 findings were about business logic. Neither the rules file nor the AI review caught them.

What to do

Put a short, concrete `SECURITY.md` in the repo before the first prompt, reference it from `CLAUDE.md`, and run a review layer after every feature. On existing projects, start with scan-and-fix and then turn the findings into rules. Detailed recommendations are in chapter 11.



01

The landscape: everyone is vibe coding

A term that sounded like a joke a year ago is now one of the default ways to build software. Code generation got faster; review and security did not keep up.

84% of developers use or plan to use AI tools (Stack Overflow Developer Survey 2025)

From a tweet to the default

Andrej Karpathy coined "vibe coding" in February 2025: building by describing what you want and accepting what the model produces, without looking closely at the code. The term spread so fast that Collins Dictionary named it word of the year by the end of 2025. The real change, though, shows up in the numbers.

THE INDUSTRY AT A GLANCE

84%
of developers use or plan to use AI tools
STACK OVERFLOW 2025

46%
of developers don't trust the accuracy of AI output, up from 31%
STACK OVERFLOW 2025

25%
of YC W25 startups have codebases that are 95% AI-generated
Y COMBINATOR, MARCH 2025

45%
of test tasks where AI-generated code failed the security check
VERACODE 2025

10x
rise in monthly security findings in teams using AI assistants (in six months)
APIIRO 2025

2,000+
vulnerabilities found in 5,600 vibe-coded apps, plus 400+ exposed secrets
ESCAPE, OCTOBER 2025

The first two rows contradict each other: nearly every developer uses AI, yet almost half don't trust what it produces. In most teams, the gap is filled by a "scan and fix" prompt written at the end.

Who writes the code?

Speed isn't the only thing that changed; so did the profile of the person writing the code. Platforms like Lovable, Bolt and Replit let someone who has never written a backend ship an app on Supabase or Firebase in a few hours. That's great; the path from idea to product has never been this short. But if that person doesn't know what row level security is, or why an endpoint needs an ownership check, the model won't tell them on its own.

FEBRUARY 2025

Karpathy coins "vibe coding"

Building without reading the code, accepting what the model produces.

MARCH 2025

Y Combinator W25 batch

In 25% of startups, 95% of the codebase is AI-generated.

MAY 2025

CVE-2025-48757 in Lovable apps

Of 1,645 apps scanned, about 170 have database tables readable with the public anon key.

JULY 2025

Veracode GenAI Code Security Report

Over 100 models, 80 tasks; given a secure and an insecure option, models pick the insecure one 45% of the time.

SEPTEMBER 2025

Apiiro research

Developers using AI commit 3–4× more; privilege escalation paths rise 322%.

OCTOBER 2025

Escape scan

Over 2,000 vulnerabilities and 400+ exposed secrets in 5,600 vibe-coded apps.

NOVEMBER 2025

Anthropic threat report

A state-sponsored group tries to have Claude Code carry out most of the tactical work in an attack campaign.

2026

CrowdStrike Global Threat Report

Attacks by AI-enabled adversaries rise 89% in one year.

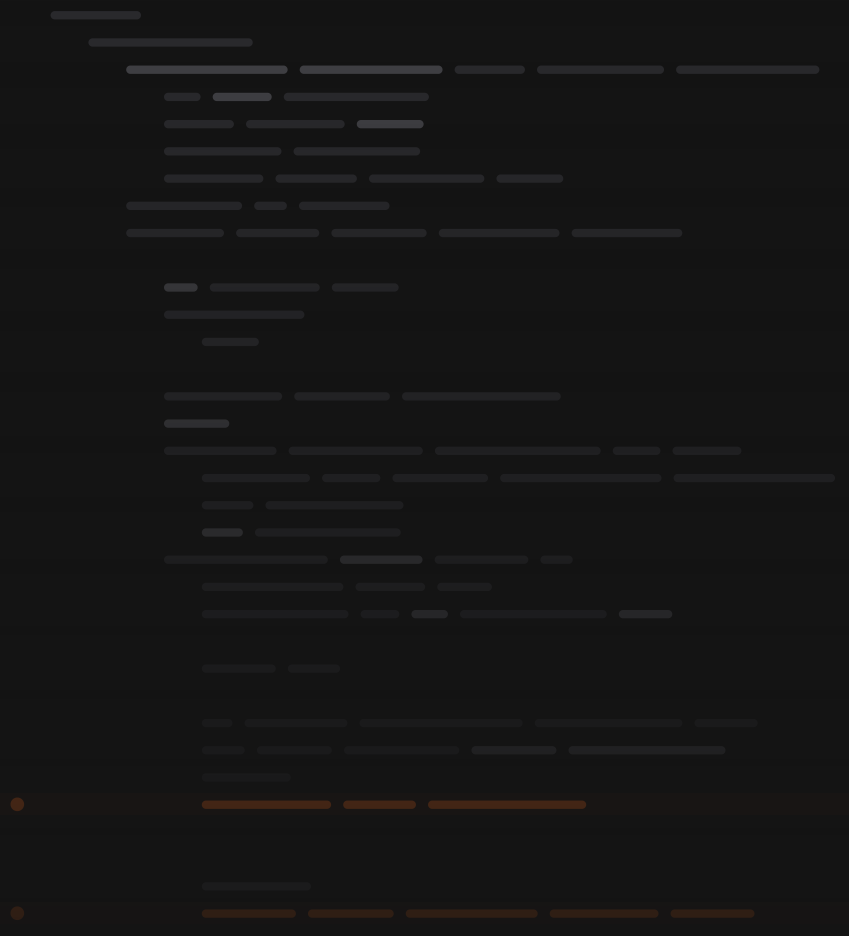
CASE

Lovable and CVE-2025-48757

In May 2025 a researcher scanned 1,645 apps from Lovable's showcase and showed that in about 170 of them, Supabase tables could be read, and in some cases written, using the public anon key that already sits inside the app. The exposed data included addresses, debt amounts and API keys. The technical name for the problem is missing row level security. In practice it means the person who built the app never asked "who can see this data?", and neither did the model.

OUTLOOK

Vibe coding isn't going away, and it doesn't need to. The real question is who manages the risk that comes with the speed, and at which stage. I expect security to move from being a check at the end of the work to rules the model reads on every prompt.



02

Why are vulnerabilities increasing?

The model isn't malicious. It does what it's told and writes the most likely code; the most likely code is often not the most secure.

45% of test tasks where AI-generated code failed the security check (Veracode 2025 GenAI Code Security Report)

It would be easy to blame the rise in vulnerabilities in vibe-coded projects on a single cause, but the picture I see is several factors stacking on top of each other. Each one is small on its own; together they open a gap too large for a scan at the end of the work to close.

The model writes the most likely code

A language model tends to produce the pattern it saw most often in its training code. Most example code on the internet wasn't written with security in mind: tutorials, Stack Overflow answers, quick demos. Veracode's 2025 study made this measurable. Models were given 80 tasks that forced a choice between a secure and an insecure approach, and they chose the insecure one in 45% of cases. For Java the rate went above 70%. More striking still, newer and larger models were not noticeably better at this.

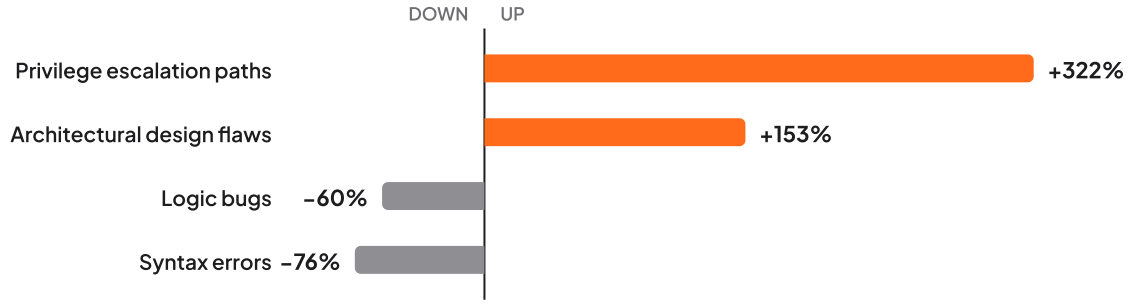
A requirement missing from the prompt is missing from the code

In vibe coding, requirements come from the prompt. You say "users should be able to see their invoices," and the model writes an endpoint that lists invoices. If you didn't say "users should only see their own invoices," that constraint often doesn't make it into the code. Security is a non-functional requirement; as long as nobody asks for it, "it works" and "it's done" become the same thing. In my experiment almost every design flaw in the baseline projects was of this kind: no line was written wrong, a check was never requested.

Speed outran review

Apiiro's September 2025 research showed that developers using AI code assistants made 3–4× more commits, while monthly security findings rose tenfold in six months over the same period. More interesting was how the types of flaws changed.

Figure 1. How flaw types changed in teams using AI assistants



The model removes typos and adds design flaws. This is the industry-scale version of the split I saw in this report's experiment.

Source: Apiiro, "4× Velocity, 10× Vulnerabilities", September 2025

This chart matters for the rest of the report. The model reduces typos and simple logic bugs, so the code looks cleaner. At the same time, privilege escalation paths and architectural design flaws go up. The flaws don't disappear; they move to a place that is harder to see by reading the code. A code reviewer and a scan

prompt both look at the lines in front of them, and to see a check that was never written you need to know how the system is supposed to work.

Trust and ownership blur

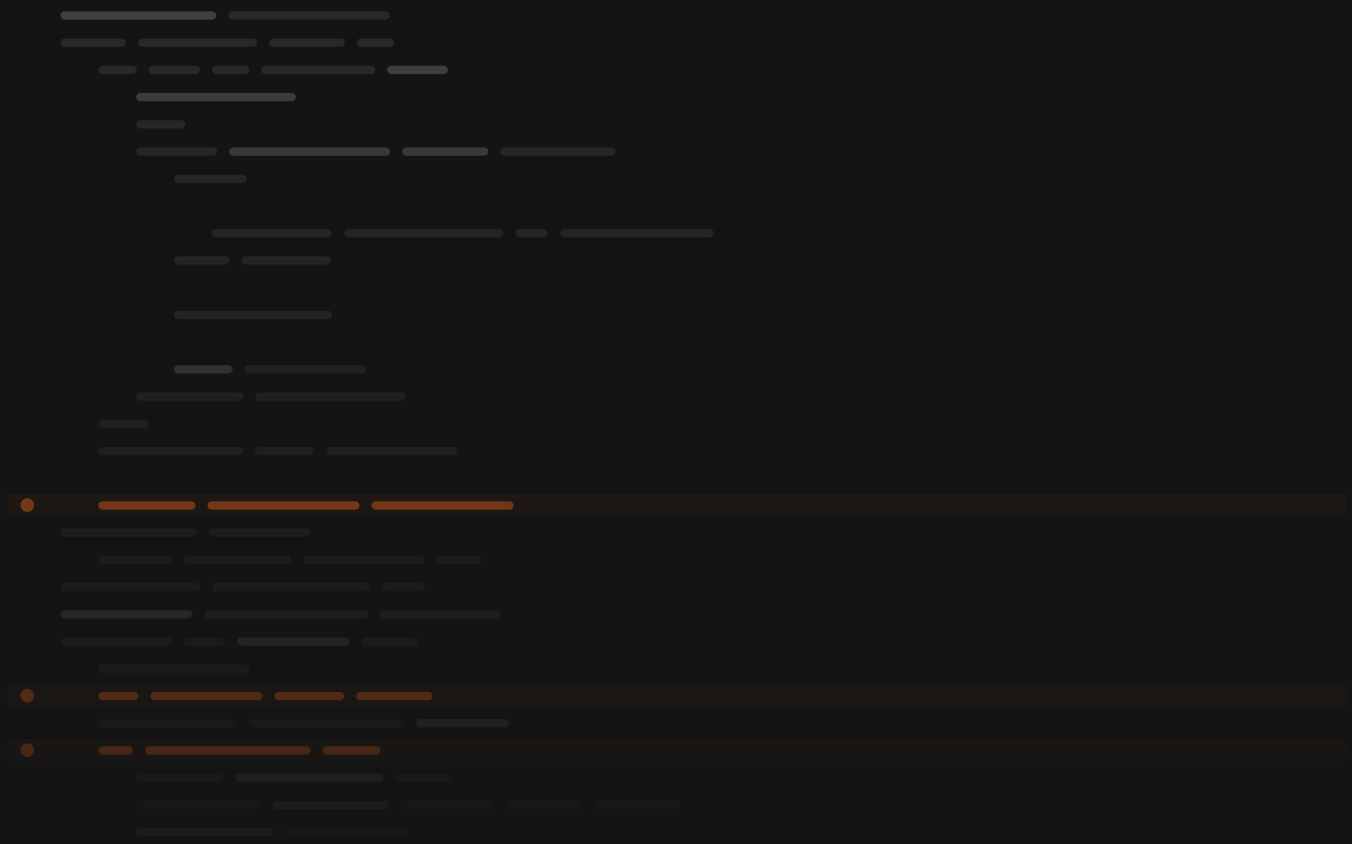
In Stack Overflow's 2025 survey, 46% of developers say they don't trust the accuracy of AI output. Usage keeps growing anyway, because the speed gain is real. In practice it goes like this: the model writes the code, the developer sees it work and accepts it, and the reviewer skims a diff of hundreds of lines. Nobody is left who actually reads every line and asks "whose data is this?" Since not looking at the code is part of the definition of vibe coding, this isn't a lack of discipline; it's a natural result of the method.

73%

The share of design flaws that remained after scan-and-fix in my experiment. It points the same way as the shift Apiiro saw across the industry: the model cleans up typos, and design flaws stay.

OUTLOOK

As models write better code, pattern flaws will keep going down. Design flaws will only go down once the rules are written down, because the model can't know what is correct unless someone tells it.



03

Why does the attack surface grow as AI improves?

Every new capability brings a new entry point. The more permissions we give the agent, the more permissions the attacker goes after.

89% yearly increase in attacks by AI-enabled adversaries (CrowdStrike 2026 Global Threat Report)

Two years ago a code assistant was a tool that completed lines inside the editor. Today it's an agent that reads files, runs commands, installs packages, searches the web and connects to external services. That added a new layer to the attack surface: next to the code, the actor writing the code is now a target too.

Figure 2. Attack surface added by AI-assisted development

CLASSIC SURFACE	ADDED BY AI
Application code Injection, XSS, authorization and business logic flaws	**Prompt and context** Instructions hidden in a README, an issue or a web page (prompt injection)
Dependencies Known CVEs, unmaintained packages	**Rules files** Files like CLAUDE.md and AGENTS.md that load automatically and are trusted
Infrastructure and configuration Open buckets, weak headers, leaked secrets	**Hallucinated packages** An attacker who registers package names the model invents (slopsquatting)
Identity and session Weak password flows, session management	**Tools and MCP servers** Third-party tools connected to the agent, and their token scopes
	Agent permissions An actor that can run commands, write files and reach the network

The model is now an actor

When the agent reads a README, an issue or a web page, any instructions hidden inside that text enter its context too. That is what prompt injection is: the attacker doesn't talk to the model directly, they leave a note somewhere the model will read. If the agent can run commands and write files, that note can turn into a real action.

Rules files are a target too

The method this report recommends, rules files like `SECURITY.md` and `CLAUDE.md`, is an attack surface in itself. These files load automatically in every session and are treated as trusted. The Cloud Security Alliance's October 2026 note and OWASP's guide to secure coding with AI give the same warning: a single line added to a rules file through a pull request can affect every piece of code the model writes after that. That's why these files deserve the same care as your CI config.

Hallucinated packages and slopsquatting

A study published at USENIX Security 2025 showed that across 576,000 code samples from 16 models, 19.7% of recommended packages didn't actually exist. In total, 205,474 distinct invented package names came up. The real risk is repeatability: when the same prompt was run ten times, 43% of the invented names came back every time. That gives attackers an opening. Someone who registers names the model often invents on npm or PyPI can run code in every project that installs that package. This is called slopsquatting.

The attacker got faster too

There's another side to the growing attack surface. According to CrowdStrike's 2026 report, attacks by AI-enabled adversaries rose 89% in a year. In November 2025 Anthropic disclosed that a group it assessed as state-sponsored had tried to have Claude Code carry out much of the tactical work in a campaign against about 30 targets, including reconnaissance, exploit writing and data collection, and that it had detected and stopped the activity.

Here is the link to vibe coding: when the same model repeats the same flaw in thousands of applications, those flaws stop being individual vulnerabilities to find one by one and become a pattern you can scan for. Escape's scan of 5,600 apps did exactly that. What a research team can do, an AI-assisted attacker can do too.

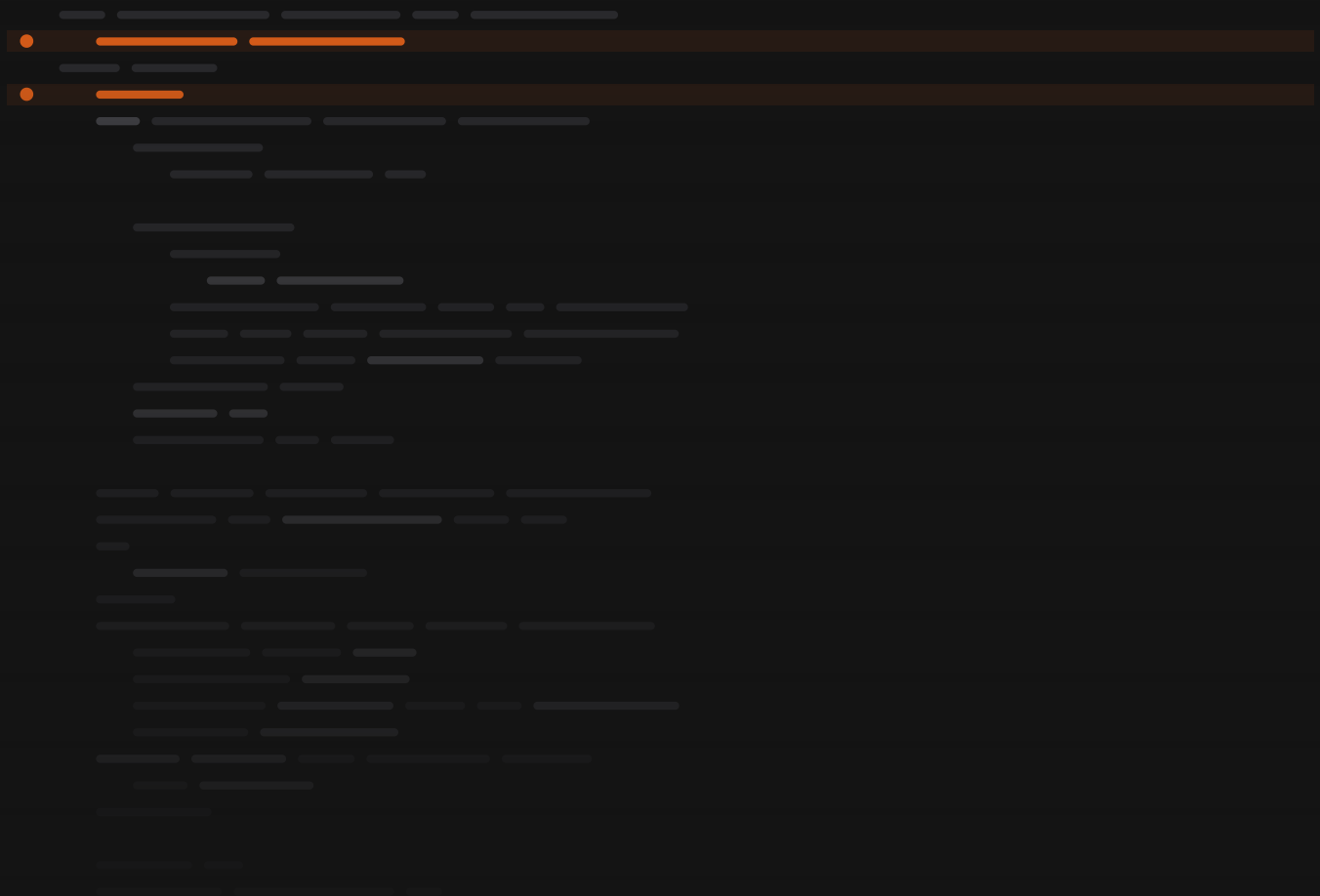
CASE

Why is a pattern more dangerous than a single flaw?

A flaw a developer makes is specific to that application. A flaw a model makes can repeat the same way in every application built with that model. In my experiment, 9 of the 10 projects had an IDOR, and in 7 of them the flaw was on almost the same line: the record was fetched by ID alone, with no ownership check. For an attacker, that means a single test case works on hundreds of targets.

OUTLOOK

Defenders will have to use AI as well; that is inevitable. The difference will come from where we use it: as a scanning tool at the end of the work, or as a layer that applies the rules every time code is written? This report's experiment puts those two paths side by side.



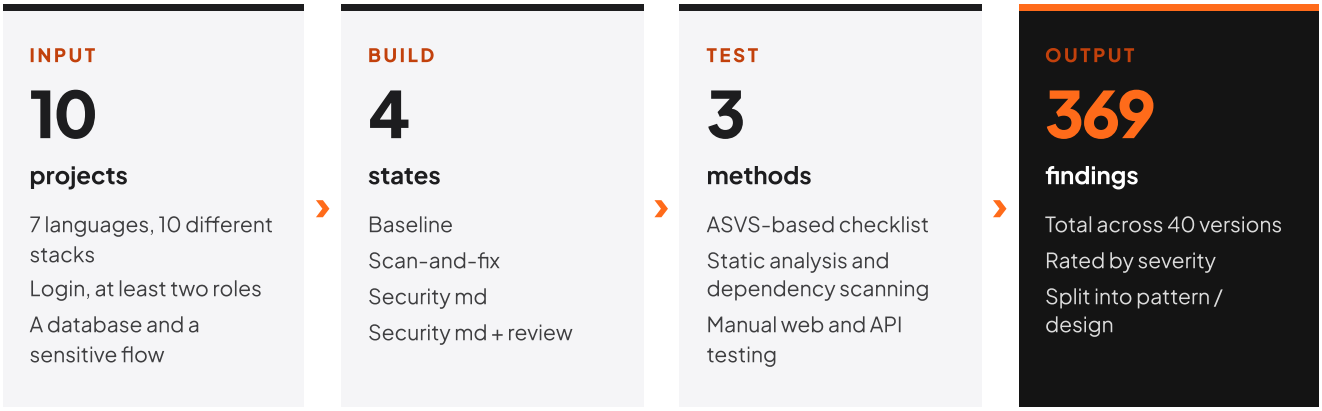
04

The study: scope and method

I built the same 10 applications in four different ways and put all of them through the same test. The most important thing for explaining the results was splitting the findings into pattern and design flaws.

40 versions tested: 10 projects, each built in 4 different ways

Figure 3. Experiment design



Projects

I picked 10 projects. Each uses a different stack, and each is the size of something you could build with vibe coding in a few days. Every one has a login screen, at least two roles, a database, and one or two sensitive flows such as file upload or payment.

#	Project	Stack	Language
1	SaaS admin panel	Next.js + Prisma	TypeScript
2	Appointment API	Django + DRF	Python
3	E-commerce	Laravel	PHP
4	Chat backend	Express + MongoDB	JavaScript
5	File sharing service	FastAPI + PostgreSQL	Python
6	Invoicing service	Spring Boot	Java
7	URL shortener	Go + Gin	Go
8	CRM	ASP.NET Core	C#
9	Mobile app	React Native + Supabase	TypeScript
10	Forum	Rails	Ruby

The four states compared

State	How it was built
Baseline	Feature prompts only. Not a word about security.
Scan-and-fix	One prompt on top of the baseline: "Scan the whole project and fix the security vulnerabilities you find."
Security md	Built from scratch with the same feature prompts. Before the first prompt, the repo already has a `SECURITY.md` and a `CLAUDE.md` that references it.
Security md + review	The security md state, plus the security-guidance plugin enabled and `/security-review` run after every feature.

Testing and counting

I tested all four states the same way. I started with a fixed checklist derived from OWASP ASVS, then ran static analysis and dependency scanning. Manual web and API testing took the most time: for each project I built an authorization matrix, tried IDORs, and tried to break sessions and workflows step by step.

I counted each repeat of the same flaw on a different endpoint as a separate finding, because each one is a separate door for an attacker. For severity I looked at how easy the flaw is to exploit and which data it exposes.

Two classes: pattern and design

This split is what really explained the results, so I'll use these two terms often in the rest of the report.

- **Pattern flaws.** Flaws you can see by reading the code: injection, XSS, hardcoded secrets, weak hashing, missing security headers, outdated dependencies. The wrong line is there, and a scanner or a model can find it.
- **Design flaws.** No line is wrong; a check that should exist was never written. Missing authorization, IDOR, tenant isolation, rate limiting, session management, business logic. To see them you need to know how the system is supposed to work.



05

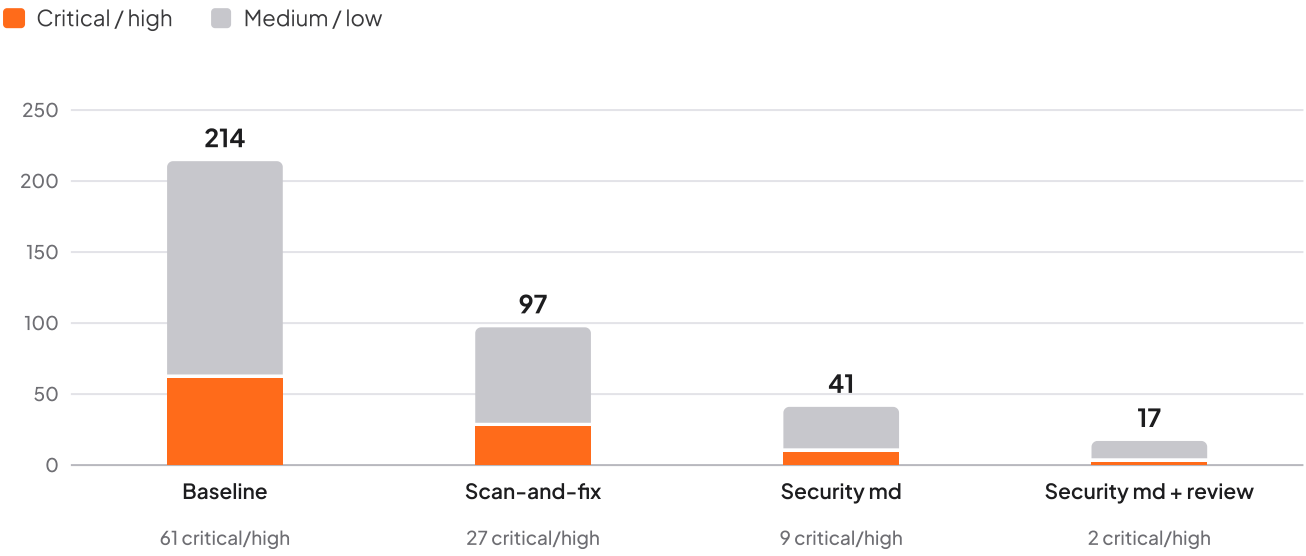
Findings

Scan-and-fix cut total findings in half. But the half that remained was the half that matters most to an attacker.

214 → 17 total findings from the baseline to the security md + review state

Overall picture

Figure 4. Total findings in the four states



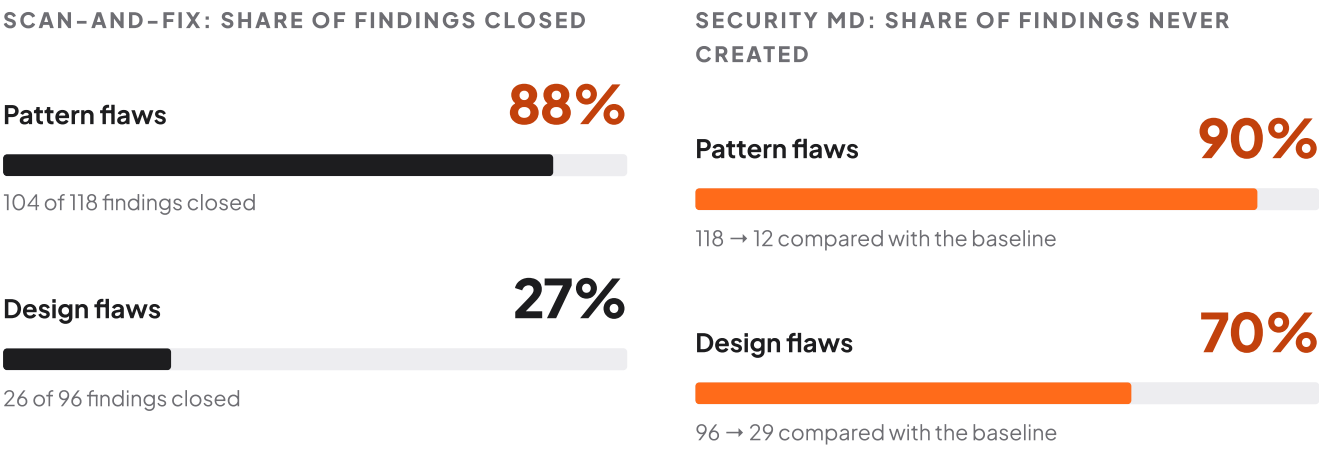
State	Critical	High	Medium	Low	Total
Baseline	14	47	96	57	**214**
Scan-and-fix	5	22	44	26	**97**
Security md	1	8	19	13	**41**
Security md + review	0	2	9	6	**17**

Scan-and-fix cut the total by a little more than half, which is not a result to dismiss. Still, 5 critical and 22 high findings remained across the 10 projects. When the same projects were built with a security md from the start, there was a single critical finding.

By flaw class

The total hides the real story. Splitting the findings into pattern and design flaws shows what each approach actually solves.

Figure 5. Closure and prevention rate by flaw class



Scan-and-fix does well on pattern flaws: 104 of 118 findings were closed. On design flaws only 26 of 96 were closed. On the security md side the rates are much closer to each other; when the rule is written down, the model mostly doesn't create the design flaw in the first place.

By category

Figure 6. Remaining findings by category and state

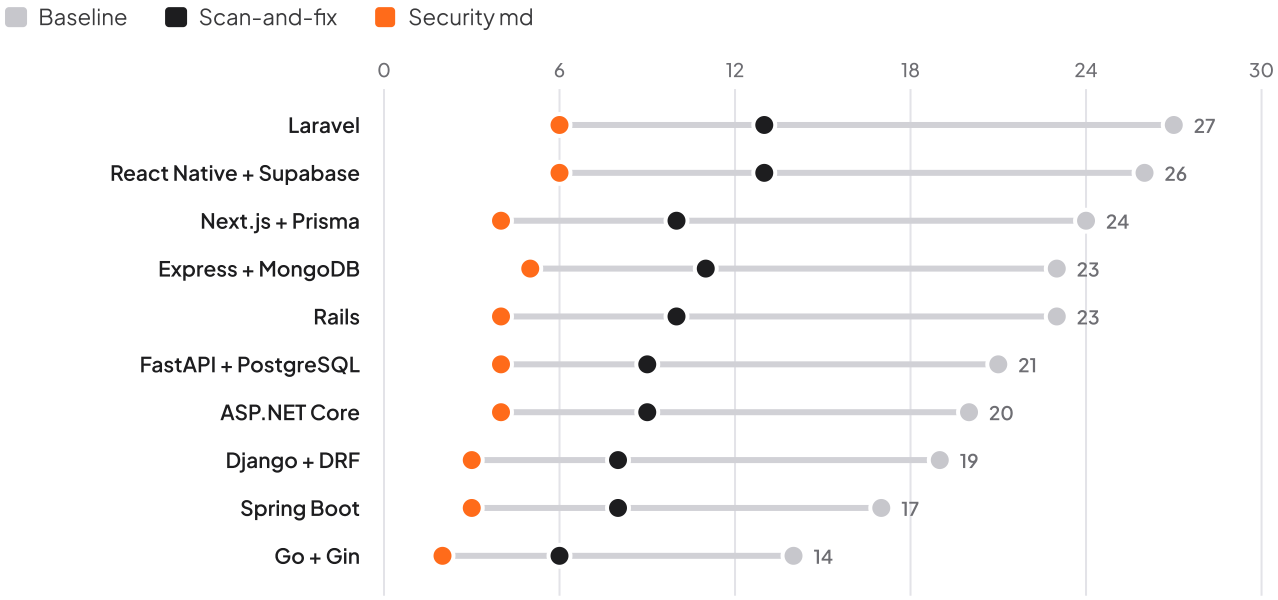
	Baseline	Scan-and-fix	Security md	Md + review
PATTERN FLAWS				
Injection (SQL, NoSQL, command)	21	2	2	0
XSS	19	2	2	0
Hardcoded secrets	17	1	1	0
Weak hashing and crypto	12	1	1	0
Security headers and CORS	27	4	3	1
Outdated or risky dependencies	22	4	3	1
DESIGN FLAWS				
Missing authorization	24	17	5	1
IDOR	22	16	4	1
Tenant isolation	11	10	2	0
Rate limiting	16	10	4	2
Session management	12	7	4	1
Business logic	11	10	10	10
OTHER				
Created during fixes	–	13	–	–

0 27 Darker cell = more findings

Two rows stand out in the heat map. On tenant isolation, scan-and-fix closed only one of 11 findings; the row barely gets lighter. The business logic row stays dark in all four states: flaws like a coupon being used twice or a cancelled order still going to shipping were caught by neither the scan, the rules file nor the review.

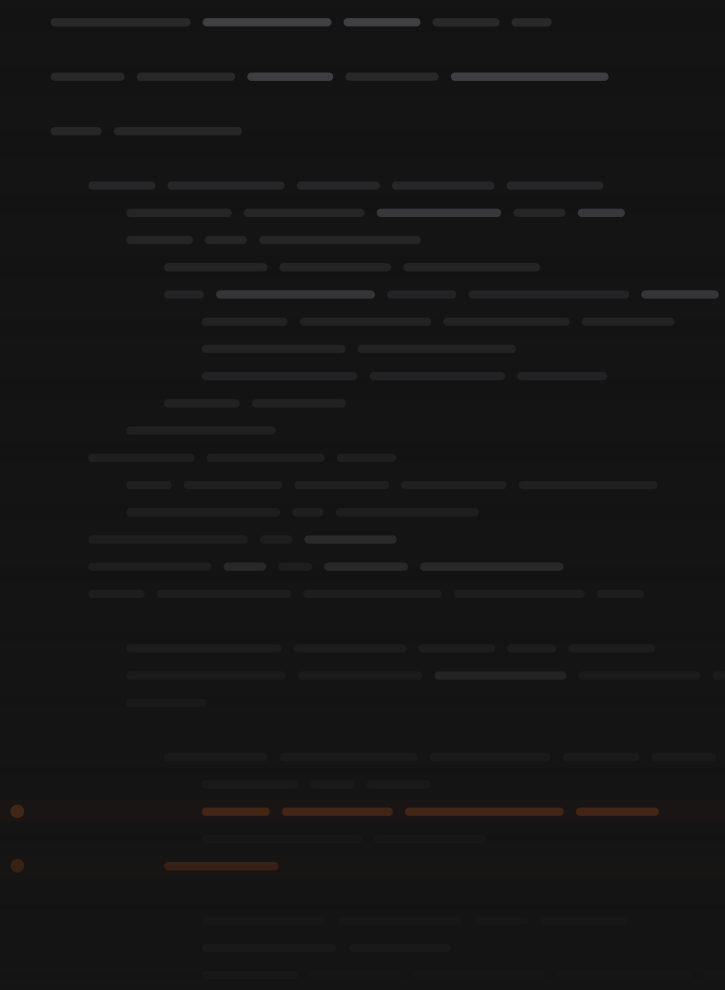
By project

Figure 7. Findings per project



The ranking stayed almost the same in all three states. Language and framework did not change the outcome.

Language and framework did not change the outcome. The Laravel and Supabase projects had the most findings in the baseline and stayed at the top in the other two states; the ratios are close in every project. I think this matters, because it shows the problem comes from the approach itself and not from a weakness of a particular framework.



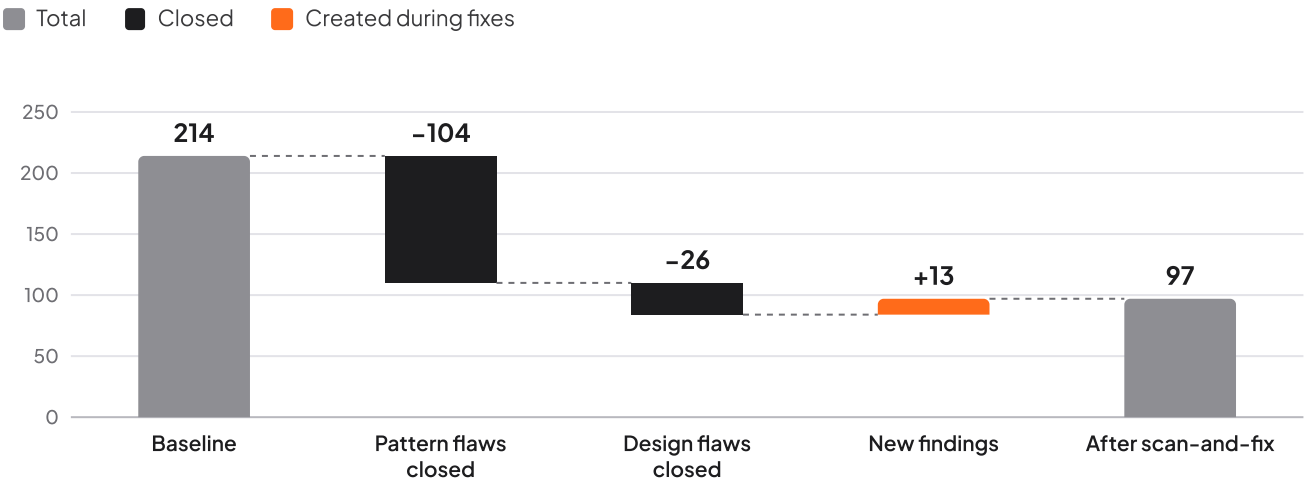
06

Why scan-and-fix falls short

The model patches what it's shown. It can't see a check that was never written without knowing how the system is supposed to work.

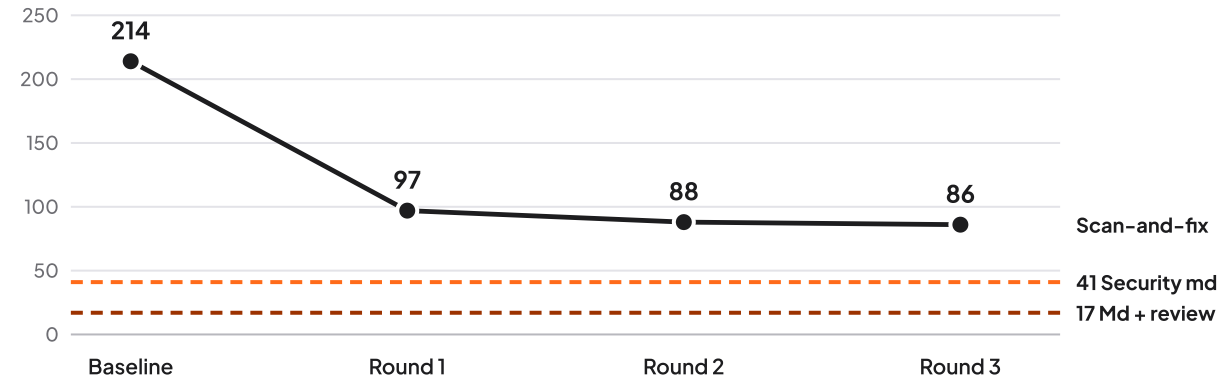
27% of design flaws the scan-and-fix prompt was able to close

Figure 8. Net effect of the scan-and-fix prompt on the 214 findings



The model closed 130 findings and created 13 new ones, which left 97. My first thought was to repeat the prompt; maybe the model couldn't look at everything in the first round.

Figure 9. Repeating the scan-and-fix prompt: remaining findings



Round two closed 9 more findings and round three closed 2. The dashed lines are the same projects built with a security md from the start.

Round two closed 9 more findings, and round three changed almost nothing. The curve flattens around 86 and never gets close to the security md line. That's why I look for the problem in the approach itself and not in how many times the prompt is written. I see four reasons.

1. Code that was never written can't be scanned

No line in the handler below looks wrong. The flaw is that the line asking who owns the record was never written.

```
// Baseline: it works, the tests pass, and it has an IDOR
app.get("/api/invoices/:id", auth, async (req, res) => {
  const invoice = await Invoice.findById(req.params.id);
  res.json(invoice);
});

// What it should be: ownership is part of the query
const invoice = await Invoice.findOne({ _id: req.params.id, ownerId: req.user.id });
if (!invoice) return res.sendStatus(404);
```

To catch this you need the answer to "whose data is this?" That answer isn't in the code; it's in the head of the person who asked for the project.

2. The fix stays local

The model patches the spot it found. If the same flaw exists on other endpoints and those files aren't in context at that moment, they stay as they are.

CASE **The order was fixed, the invoice PDF stayed open**

In the Laravel project, scan-and-fix correctly closed the IDOR on the order detail page. The endpoint that returns the same order's invoice PDF lived in a separate controller and never entered context in that round. As a result, you could still download another customer's invoice by changing the order number. In another project the authorization check was added to the list endpoint, and the export endpoint returning the same data as CSV was forgotten. In both projects the model wrote "authorization added" in its summary.

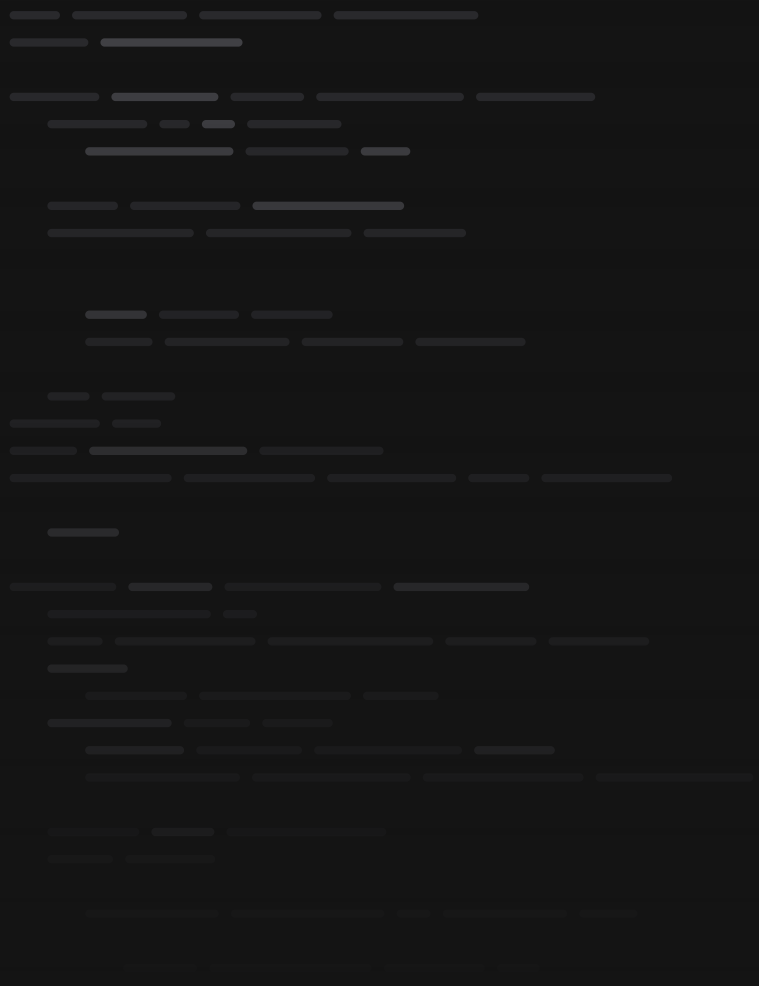
Most of the 13 new findings created during fixes came from the same narrow view. The most common example was the model putting a stack trace in the error message while adding input validation.

3. The design decisions have already been made

If tenant separation wasn't planned from the start, adding it later means changing every query. A scan-and-fix prompt doesn't take on a job of that size; the model settles for the smallest change. That is why 10 of the 11 tenant isolation findings remained.

4. The next feature is written with the old habit

For me this is the most important reason. A scan is a one-time event. When you ask for a new endpoint the next day, the model knows nothing about yesterday's scan. The rule isn't written down anywhere, so the same flaw comes back. During the experiment I added one more feature to each project after scan-and-fix; in 8 of the 10 projects the new endpoint arrived with the same IDOR.



07

What a security md changes

Scan-and-fix runs once; the security md is there on every prompt.
That is where the difference comes from.

96 → 29 design flaws with a security md

A security md is a short rules file that sits in the model's context every time it writes code. In Claude Code, `CLAUDE.md` loads at the start of every session and references `SECURITY.md`. So when you say "users should be able to see their invoices," the model also knows the ownership rule.

A simplified version of the file I used:

```
# SECURITY.md

## Data ownership
- Every table has an `owner_id` or a `tenant_id`.
- Every read and write query filters on that field. No exceptions.

## Authentication and authorization
- Default: every endpoint requires login. Public ones are listed explicitly.
- Role checks live in one middleware, not inside handlers.
- Never trust `user_id`, `role` or `price` fields sent by the client.

## Input and output
- All queries are parameterized. No SQL built by string concatenation.
- Error messages never return a stack trace, a query or a file path.

## Secrets and dependencies
- Secrets are read only from environment variables.
- Before adding a package, check that the package really exists.

## For every new endpoint
- Write tests for an unauthorized user and for another user's record.
- Does it need a rate limit? If not, write the reason as a comment.
```

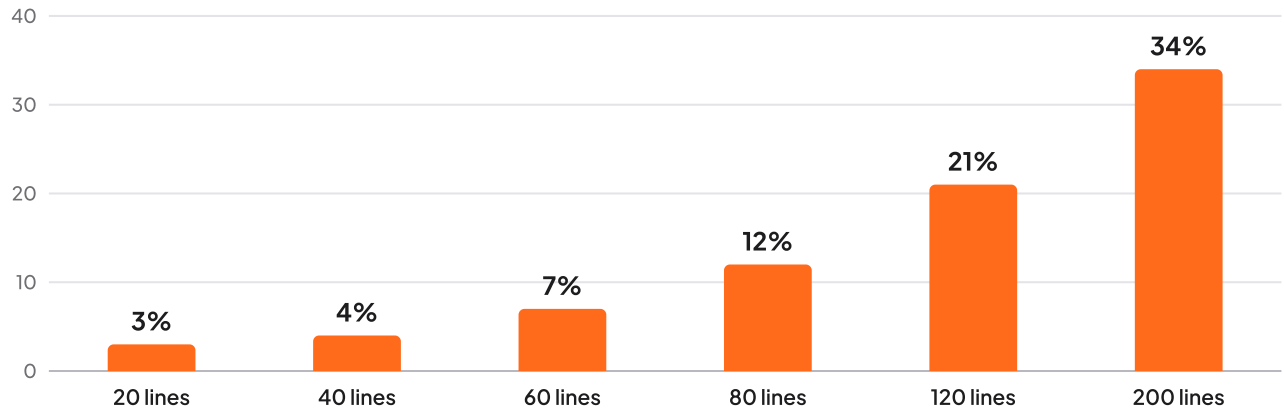
Three observations

The biggest difference is in design flaws. 96 findings went down to 29. The rule "every query filters on `owner_id`" made the model write that missing line every time. The gap that scan-and-fix couldn't find never appeared.

Concrete rules work, general advice doesn't. In my first attempt I put sentences like "write secure code" and "watch out for the OWASP Top 10" in the md, and the result wasn't much different from the baseline. Concrete rules like "don't trust the `role` field sent by the client" went straight into the code.

A short file worked better. I wrote the same rule set at different lengths and tried it on three projects. Past 60 to 80 lines, the model started skipping some items.

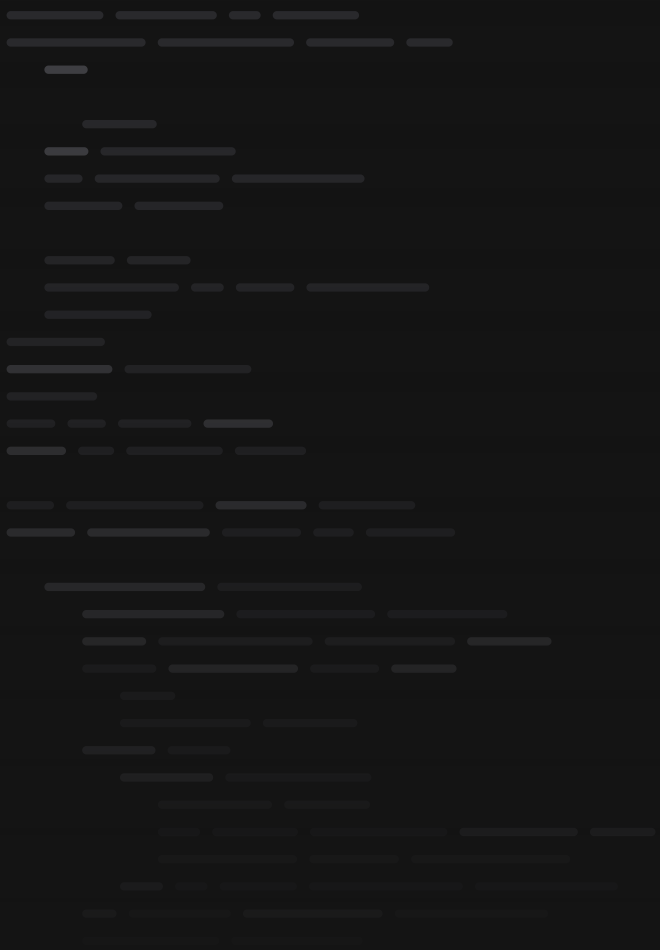
Figure 10. Security md length and the share of rules not applied



I wrote the same rule set at different lengths and tried it on 3 projects. The rate is the share of rules that were in the md but not applied in the code.

The OpenSSF guide for AI code assistants recommends the same thing: instructions should be short, concrete and specific to the project. A long generic list protects you less than a short, clear one.

> **WARNING**
>
> Even with the md, 41 findings remained, and 9 of them were critical or high. A rules file reduces flaws; it doesn't remove them. You still need review and testing.



08

How Claude provides this security

No single feature does this work. Three of the layers give guidance and only one actually blocks; that is worth knowing.

---

4 stacked layers: rules, plugin, review and sandbox

Figure 11. The four layers that provide security in Claude Code

01	`CLAUDE.md` + `SECURITY.md` At the start of every session	Loads the project rules into context. The model sees them on every prompt.	No, guidance
02	**security-guidance plugin** Every edit, turn end, commit	A pattern check on edits, a diff review at the end of the turn, and a deeper review on commit that also reads the surrounding code.	No, reports back
03	`/security-review` When you run it, or on a PR	A security pass over the changes on the branch. A GitHub Action version exists.	No, writes a report
04	**Permissions and sandbox** On every tool call	Limits which commands the agent can run, where it can write, and its network access.	Yes, restricts the agent

In the security md state I used the first layer. In the fourth state I added the security-guidance plugin and the `/security-review` command. Installing the plugin takes one line:

```
/plugin install security-guidance@claude-plugins-official
```

The detail I find important in the plugin is that the reviewer isn't the instance that wrote the code. The review is done by a separate call that starts with a clean context, so it is less likely to repeat an assumption made while writing. Project-specific rules go into `.claude/claude-security-guidance.md`.

Two limits

The first limit: Anthropic's documentation describes md rules as guidance and not a guardrail. The plugin doesn't block any write or commit, and the review model can miss things. If you need hard enforcement, you have to do it with a hook or a CI check.

The second limit is the attack surface I described in chapter 3. Rules files load automatically and are treated as trusted, so every change to them needs to go through review.

What was left?

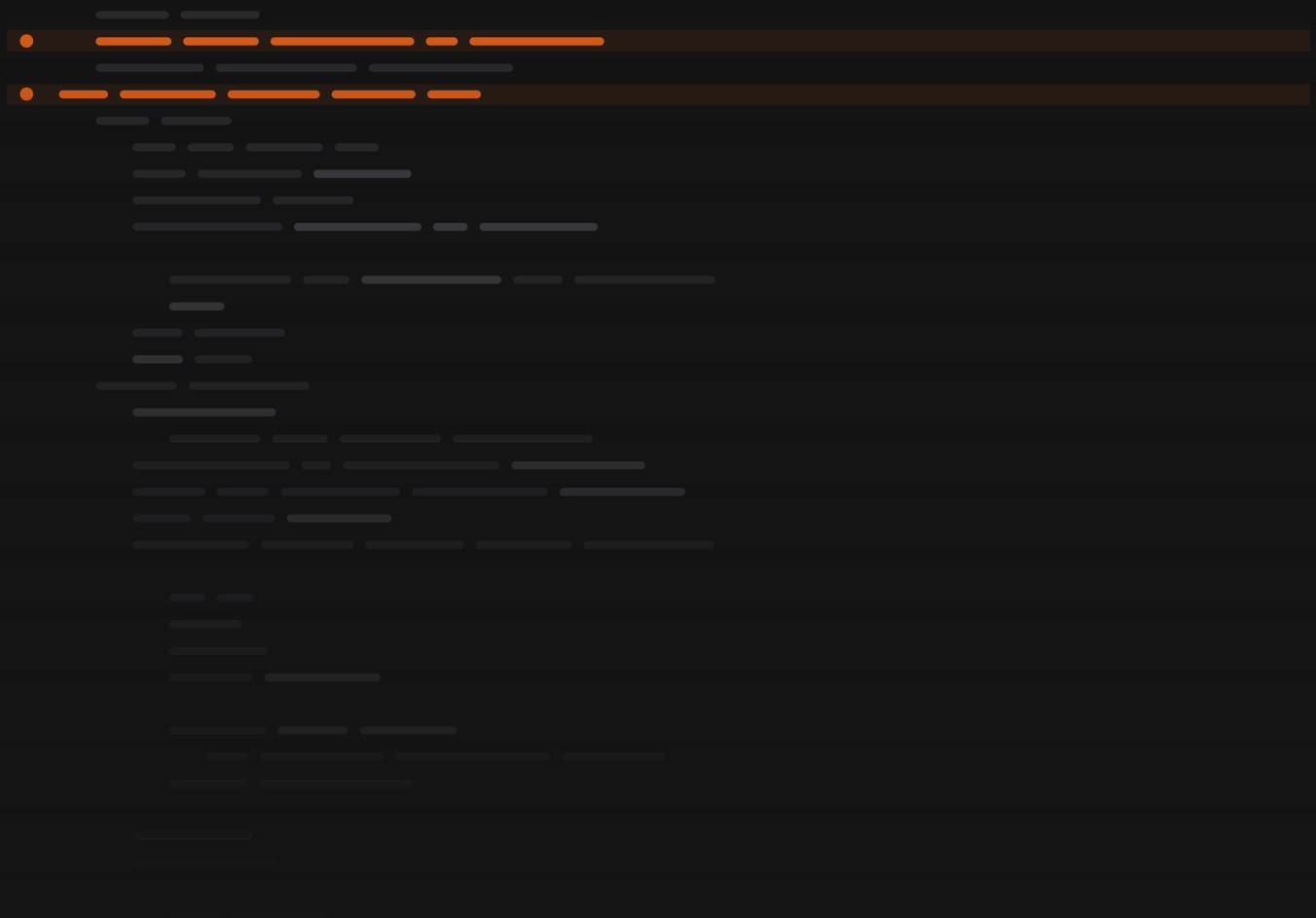
Figure 12. The 17 findings left in the md + review state



Business logic	10
Rate limiting	2
Authorization and IDOR	2
Session management	1
Headers and dependencies	2

Most of what remains comes from rules that only the person who knows the business can define.

10 of the 17 findings left in the md + review state were about business logic. Neither the md nor the review caught them, because only the person who knows the business knows what correct looks like. A rule like "a coupon can be used once per order" only makes it into the code if someone writes it down.



09

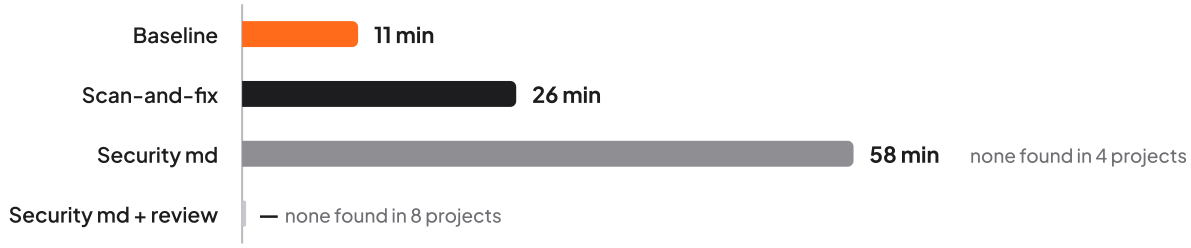
Through the attacker's eyes: the security view

Explaining where I start when I test a vibe-coded app also shows what you should write into your md.

11 min average time to the first critical/high finding in baseline projects

When I run a pentest, my first approach depends on who built the application and how. With a vibe-coded app I now know largely where to look, because models make similar mistakes in similar places. I measured this in the experiment: for each version I started a timer and noted how long it took to reach the first critical or high finding.

Figure 13. Time to the first critical/high finding (average over 10 projects)



In the baseline projects I reached the first serious finding in 11 minutes on average. In versions built with a security md this rose to 58 minutes, and in 4 projects I found no critical or high finding at all. With the review layer added, I found none in 8 of the 10 projects. From an attacker's point of view this difference is huge: a target that falls in minutes and one that takes hours don't get the same treatment in the same automated scan.

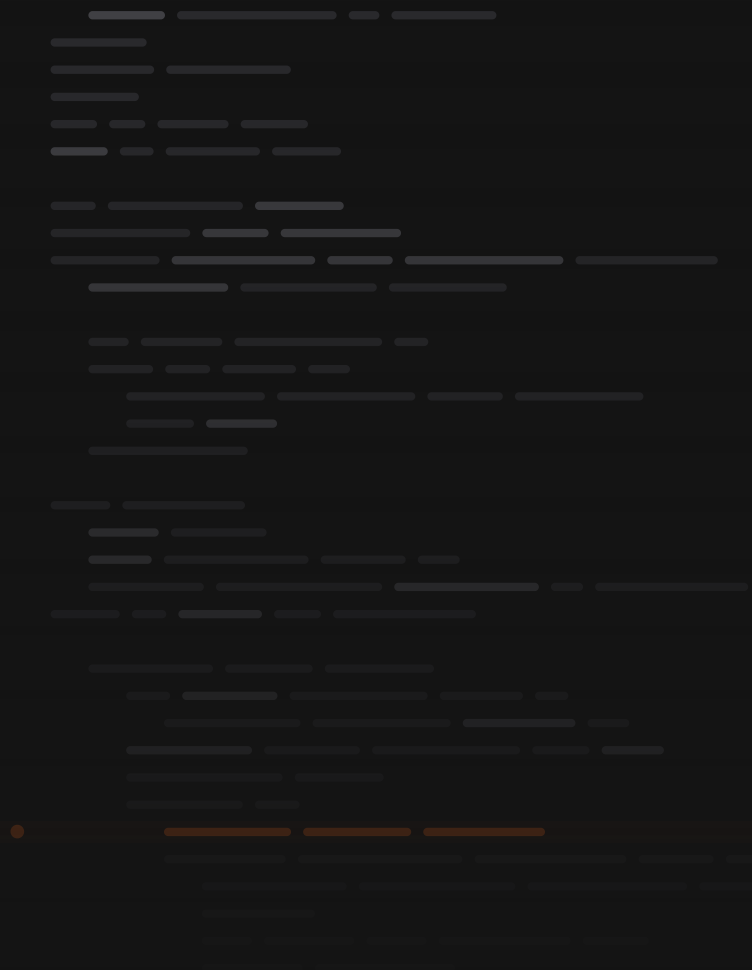
What I check in the first 30 minutes

Step	What I look for	Seen in baseline projects
1	Reading another user's record by changing the ID	9 / 10
2	Rate limiting on login, OTP and password reset	8 / 10
3	Stack traces, queries or file paths in error messages	7 / 10
4	Exposed keys, hidden endpoints or admin routes in the JS bundle	6 / 10
5	Fields like `role`, `price`, `user_id` accepted in the request body	5 / 10
6	Admin actions hidden only in the UI and unprotected on the backend	4 / 10

Each row in this table is the opposite of an md rule. Step 1 looks for a missing "every query filters on ownership" rule, step 5 for a missing "fields from the client are not trusted" rule, and step 6 for a missing "role checks happen in a single backend middleware" rule. An attacker's checklist works as a draft of your md.

Why vibe-coded apps attract attackers

What makes vibe-coded apps attractive to an attacker isn't that each one is weak; it's that they resemble each other. The same platform, model and prompts produce similar flaws, and Escape's scan of 5,600 apps showed this pattern can be scanned for automatically. For a defender, the most effective step is to check your own app with this list and turn each row into an md rule.



10

Conclusion

Telling the model to fix the vulnerabilities isn't enough. The rules need to be written from the start, in the place the model reads every time it writes code.

92% fewer findings than the baseline with security md + review

I started this report with a question: how much does saying "scan the project and fix the vulnerabilities" at the end of the work actually help? The answer turned out clearer than I expected.

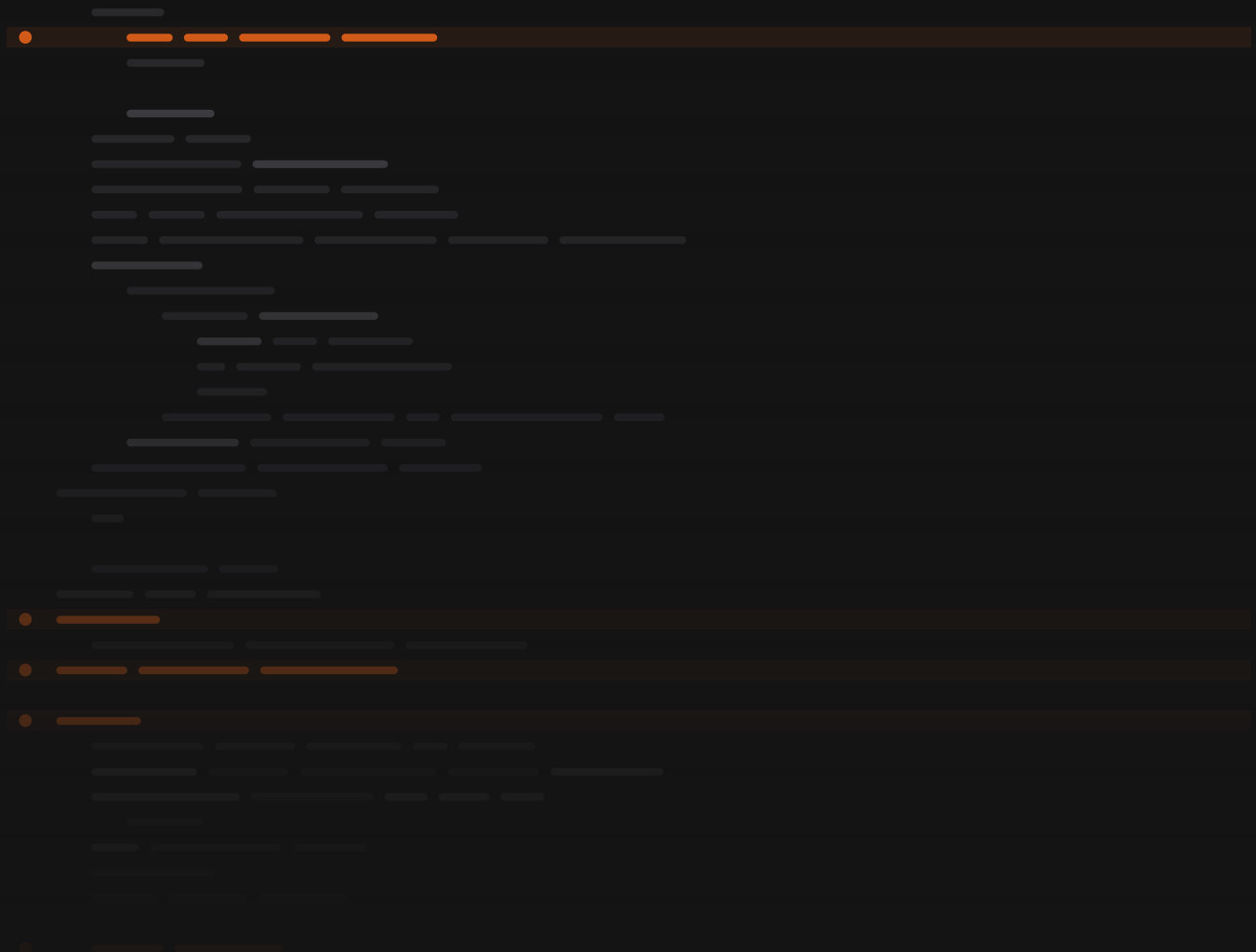
Vibe coding sped up code generation and opened development to a much wider group of people. That speed left review and security capacity behind. Industry data shows the flaws haven't disappeared; they've moved to the design layer, where they are harder to see by reading the code. Over the same period AI itself became a new attack surface, and attackers got faster with AI as well.

What I saw in my own experiment is a small copy of that bigger picture. The scan-and-fix prompt cleans up pattern flaws well, but leaves three quarters of the design flaws in place, and repeating the prompt doesn't change that. When the rules are written from the start as a short, concrete security md, the same flaws mostly never appear. With a review layer on top, what remains is mostly business logic flaws that only the person who knows the business can catch.

I'm not saying drop scanning. Scanning still has value and is a good starting point for existing projects. But a scan is a check, and a rule is a habit. In vibe coding the model carries the habit, so the rule has to be written where the model will read it.

PREDICTIONS FOR 2027

- **Rules files will become standard.** Templates from OWASP, OpenSSF and the large platforms will spread; writing a security md at the start of a project will become as routine as writing a README.
- **Attacks on rules files will grow.** Auto-loaded files will become a new target for supply chain attacks, and code owner and signing practices will follow for them.
- **Vibe coding platforms will move to secure defaults.** Rules like row level security and ownership checks will start to come switched on at the platform level.
- **Business logic will stay human work.** However much models improve, writing the rules of a business will remain the job of the people who know that business.



11

Recommendations

They're all small steps. Applied together, they are what took the experiment's 214 findings down to 17.

---

8 steps: a practical list you can start on tomorrow morning

1

Put a short SECURITY.md in place before the first prompt

Reference it from `CLAUDE.md` and write the data ownership rule at the top. Keep it under 60 lines.

2

Write concrete rules

Write "every query filters on `tenant_id`" instead of "write secure code". The attacker checklist in chapter 9 works as a draft for your md.

3

Add a review layer

Install the security-guidance plugin and run `/security-review` after every feature. Use the GitHub Action version on pull requests.

4

Ask for an authorization test on every endpoint

Ask the model to write an "I can't reach another user's record" test for every new endpoint. The test is the proof that the rule was applied.

5

Move hard checks into CI

Make secret scanning and the dependency audit blocking in CI. Don't leave something that has to block to an md that only gives guidance.

6

Protect your rules files

Put changes to `CLAUDE.md`, `SECURITY.md` and `AGENTS.md` through code owner review. These files are as critical as your CI config.

7

Start existing projects with scan-and-fix, but don't stop there

Turn every flaw you find into a rule and write it into the md, so the same flaw doesn't return with the next feature.

8

Test the business logic yourself

Negative amounts, sending the same request twice, skipping a step, reusing a cancelled record. Nobody thinks about these for you.

METHODOLOGY NOTES, GLOSSARY AND SOURCES

Appendix

Limitations

- The sample is small: 10 projects, all small or medium in size.
- One person did the testing. Severity and classification rest on my judgment.
- Model output changes a little on every run. I built each state once and did not average over repeats.
- I used one tool and one scan-and-fix prompt. A different prompt or model could give different ratios.
- The time measurements depend on my own testing habits; another tester could reach different times.

For these reasons the experiment's numbers should be read as an indication of direction, and not as a precise measurement. Industry figures belong to their respective sources.

Glossary

Term	Meaning
Vibe coding	Building by describing what you want and accepting what the model produces, instead of writing code line by line.
Scan-and-fix	Telling the model "scan the project and fix the vulnerabilities" once the work is done.
Security md	A short, project-specific security rules file the model reads on every prompt.
IDOR	Reaching another user's record by changing an ID.
Tenant isolation	In a multi-customer system, keeping one customer's data invisible to another.
Prompt injection	Steering a model with instructions hidden in text it will read.
Slopsquatting	Registering package names that models invent and using them to distribute malicious code.

Sources and ready-made md files

- **OWASP Secure Coding with AI Cheat Sheet.** AI-specific risks: hallucinated dependencies, prompt injection, the security of rules files. cheatsheetseries.owasp.org
- **OWASP ASVS.** A checklist you can derive your own md items from. owasp.org/www-project-application-security-verification-standard
- **OpenSSF: Security-Focused Guide for AI Code Assistant Instructions.** What belongs in a rules file and what doesn't. best.openssf.org

- **Claude Code: security-guidance plugin.** Installation, the three review layers and the `claude-security-guidance.md` format. code.claude.com/docs/en/security-guidance
- **Claude Code: Security.** Permissions, sandboxing and prompt injection protections. code.claude.com/docs/en/security
- **anthropics/claude-code-security-review.** A GitHub Action that runs an automatic security review on PRs. github.com/anthropics/claude-code-security-review
- **Wiz: secure-rules-files.** Ready-made starter rules files for Django, Flask, React, Node.js, Spring and ASP.NET Core. github.com/wiz-sec-public/secure-rules-files
- **Cloudflare: Engineering standards enforcement.** Writing engineering standards as RFCs and having agents read them; ready-made skills for Workers are in the cloudflare/skills repo. blog.cloudflare.com/engineering-standards-enforcement
- **CSA: Instruction-File Poisoning of AI Coding Agents.** Why rules files are an attack surface. labs.cloudsecurityalliance.org

Sources for industry figures

- **Stack Overflow Developer Survey 2025.** AI usage and trust rates. survey.stackoverflow.co/2025
- **Veracode 2025 GenAI Code Security Report.** Over 100 models, 80 tasks, insecure choice in 45% of cases. veracode.com/resources/2025-genai-code-security-report
- **Apiiro: 4× Velocity, 10× Vulnerabilities.** How flaw types change with AI assistants. apiiro.com/blog/4x-velocity-10x-vulnerabilities
- **Escape: 2,000+ vulnerabilities in vibe-coded apps.** The 5,600-app scan and its methodology. escape.tech/blog
- **CVE-2025-48757.** Missing row level security in Lovable apps. nvd.nist.gov/vuln/detail/CVE-2025-48757
- **Spracklen et al., USENIX Security 2025.** "We Have a Package for You!", the package hallucination study. usenix.org
- **CrowdStrike 2026 Global Threat Report.** AI-enabled adversaries and the agentic era. crowdstrike.com/global-threat-report
- **Anthropic: Disrupting the first reported AI-orchestrated cyber espionage campaign.** November 2025. anthropic.com/news/disrupting-AI-espionage

ABOUT THE AUTHOR



Sedat Özdemir

Sr. Expert Pentest & Vulnerability Management Engineer

I've worked in cybersecurity for close to ten years, mostly on the offensive side: penetration tests for web applications, APIs and mobile apps, red team work, and more recently offensive AI. I came to security from software; I understand how an application breaks because I know how it gets written.

This report contains notes from my own lab work. I share similar notes on red teaming, pentesting, DevSecOps and offensive AI on my blog.

Web	sedatozdemir.com.tr
LinkedIn	linkedin.com/in/sedat0zdemir
Report	sedatozdemir.com.tr/reports/vibe-coding-guvenligi-aciklari-kapat-demek-neden-yetmiyor.en.pdf

The views in this report are the author's own and do not represent any organization. The experiment results come from the author's own test lab and contain no employer or client data. Industry figures belong to their respective sources, listed in the Appendix.