Xint.io

The Top Security Vulnerabilities Generated by AI Code

What AI coding agents get wrong: A vulnerability study across models, vendors, languages, and a real world app.

Table of Contents

1 Summary ..... page 2

2 Methodology ..... page 4

- What we scanned
- How we chose these for scanning
- How we scanned
- How we validated findings
- Caveats

3 Key Findings ..... page 7

- What is the most common vulnerability type in AI code?
- What is the most common SEVERE vulnerability in AI code?
- Does app size matter in the types of vulnerabilities created?

4 The Takeaway ..... page 11

5 About Xint.io and Theori .... page 12

Summary

Every analysis of AI-generated code comes back to the same conclusion: increased use of AI coding tools go hand in hand with substantial increases in serious data leaks and design flaws.

But we wanted to go one step deeper and research which types of vulnerabilities AI code is prone to produce. How severe are they, and does the type of app being built or the amount of context given to the AI make a difference? This data tells product security teams exactly which type of bugs to look for, sets them up to use AI coding tools more responsibly, and ultimately produce more secure code.

These were the questions we set out to answer. To do so, we used Xint Code to scan source code of applications built with AI code generation. We included apps built using three common AI-assisted development workflows:

- building from spec
- building from scratch
- hardening an already existing app

This led to 434 unique and validated vulnerabilities found across 28* scanned apps. Every count is a real vulnerability as all findings are post-triage and were individually re-checked against the source and then scored by CVSS severity. Non-exploitable reports were dropped from the final count. Findings were then categorized by type and analyzed for any patterns.

* 27 greenfield apps built from scratch by five models total from Anthropic (opus-4.7, opus-4.8, sonnet-4.6) and OpenAI (gpt-5.5, codex-5.3) + 1 real Laravel 12 + React 19 CMS that was AI-rewritten from legacy Gnuboard5 to be "more secure"

AI Code Is More Insecure

More PRs With More Bugs

Pull requests (PRs) per author **increased 20%** year over year, but incidents per PR increased **23.5%** and change failures increased 30% over the same period.

Source: Cortex

Increased Bug Rates

AI pull requests contain **1.7x more issues** overall and **75% more logic and correctness errors** than human-authored code.

Source: CodeRabbit

Security Vulnerabilities

AI code contains more security-related flaws, with some reports showing it is up to **2.74x more likely** to introduce XSS vulnerabilities and improper password handling.

Source: Test Collab

Higher Complexity & Maintenance Debt:

Adopting AI coding agents like Cursor has been tied to an immediate **9% increase** in code complexity, as the AI frequently optimizes for the "happy path" and ignores critical edge cases or boundary conditions.

Source: Test Collab

Silent Failures

Up to **60%** of AI code faults are "silent failures". This means the code compiles successfully and passes shallow tests, but produces incorrect results in production.

Source: Ranger

We had 4 big takeaways when it comes to vulnerabilities from AI code:

1 *What are the most common bugs in AI code?*

The most common bug in AI-generated code overall is missing rate-limiting and DoS controls (such as unbounded pagination, no rate limits, or synchronous blocking work). These were present in nearly every project we tested. They lead to code that runs in production but is highly inefficient. The impact is not data theft, but in real operation these bugs could cause runaway server cost or a server an attacker can knock over.

2 *What are the most common SEVERE bugs in AI code?*

When we narrow our focus to the most critical bugs, a different picture emerges. Secret exposures made up the largest share of the most critical bugs in AI-generated code. These are hardcoded or default secrets that allow an attacker to forge sessions or tokens. We believe this is because these are the copy-paste quick-start defaults that dominate training data and do not change whether the app runs, so a “does it work” check never surfaces them.

3 *Does app size affect the type of vulnerabilities created?*

Yes. Authorization and IDOR bugs – a type of flaw where a user gets access to data beyond their permissions – were more common in larger apps. Fine-grained user permissions hold up on small apps but break as the app grows (as measured by lines of code). Simple permissions (like stopping someone from editing another user’s post) were handled well in the small builds but fell apart at scale. This is more likely to happen as permissions complexity grows. For example, if a user manages board X but not board Y in an AI-coded app, that user will still be able to edit board Y even though they do not have permissions to do so.

4 *Has AI code improved in any aspect of security?*

Going into this research, our hypothesis was that AI-generated apps would be riddled with injection flaws (SQLi, XSS) and IDOR/BOLA-style access-control bugs, since these models are trained on large amounts of injection-prone code. What we found was the opposite on both fronts.

- Injection barely showed up: Recent models default to safe patterns out of the box, reaching for prepared statements and ORMs and sanitizing inputs on their own.
- IDOR/BOLA was also far less common than expected: For the straightforward cases, the models consistently added the right authorization (authz) ownership checks. This suggests the foundational labs have genuinely improved in these areas.

Methodology

We sought to get a data-backed picture of which vulnerability classes AI models miss or introduce, correcting for model, vendor, language, reasoning effort, or kind of app. The greenfield design holds the product spec constant and varies only the generator. The addition of the live, in-production app (Gnuboard7) is meant to reflect the real-world question, "when AI is asked to harden a large existing app, what slips through?"

The two greenfield archetypes (a stateful, multi-user CRUD app with file handling and object ownership and a mixed anonymous/authenticated API service with bearer auth) were selected because they are compact but security-dense. Even these small, generated apps include authentication, authorization, resource-handling, and request-forgery surfaces while staying small enough to scan many model/effort/language variants.

What we scanned

- **A spec-driven build:** we wrote a careful spec for a URL shortener and had the model implement it to see how secure the output is when requirements are well defined. Built by Anthropic (opus-4.7, opus-4.8, sonnet-4.6).
- **A from-scratch build:** we just asked, like a casual vibe coder, "make me a simple board," with no security guidance. Built by both vendors, Anthropic (opus-4.7/4.8, sonnet-4.6) and OpenAI (gpt-5.5, codex-5.3).
- **An AI-hardened production-grade app:** we took a real, mature codebase (Gnuboard7, a widely used PHP board) whose old bugs are largely fixed, and had AI re-architect and harden it (Gnuboard7, on Laravel + React).

Why we chose these for scanning

- To see how secure an app is when it is produced from a well-written spec. This reflects real world scenarios where you have an experienced developer overseeing an AI-generated application.
- To see the security you get when a normal user just says "build this, add that" and throws a board together with no security thought. This reflects the "vibe coder" scenario wherein a non-technical user builds an app from scratch with gen AI.
- To see how secure the result is when AI is used to harden an existing app that once had many vulns but is now mostly cleaned up.

How we scanned

Each app had a single scan through Xint, an autonomous pentesting platform for both runtime as well as source code analysis.

On average each scan took about 30 minutes + 10 minutes for validation. For comparison, it would have taken a human pentester at least a day per app.

How we validated findings

Originally the scans resulted in over 8,800 findings.
These were the steps we took to triage:

1 First, Xint is able to automatically collapse duplicate detections into 513 distinct findings, so the findings we analyzed do not include duplicate counts (Xint auto-collapsed 94% of duplicates).

2 Then for each unique finding, we used the trigger conditions and step-by-step reproduction instructions that are part of the Xint output to create a POC for 85% of the findings.

3 Through this, we were able to prove exploitability for 434 of the findings and did not include the rest in our final analysis. This is in line with the Xint platform's general <25% false positive rate compared to traditional SAST scanners which are normally 70-80%.

From 8,827 raw detections to 434 verified findings

Greenfield + g7 combined. Xint de-dups, then POC-verification proves exploitability.

8,827 raw detections

↓ Xint auto-collapses duplicates –94%

513 distinct findings

↓ POC-verified exploitable 85% proven

434 verified findings

Where the 434 splits by cohort

196 greenfield (45%)	238 g7 (55%)

= the number shown in the greenfield charts
(chart_greenfield_classes, cross-model views)

Raw: reports table (g7 6,667 + greenfield 2,160). Distinct: deduped vulns (g7 253 + greenfield 260). Verified: g7 238 + greenfield 196 = 434.

For reproducible artifacts reach out to research@xint.io.

Caveats

Exploitability

The 434 findings are real and verified but are not exploit-proven (196 from greenfield apps and 238 from g7). Verdicts are code-evidenced, deliberately skeptical, but not run as live exploits. Some are gated behind preconditions (admin/extension trust boundaries for Gnuboard7). Each was re-checked against the cited source and non-exploitable reports were dropped (greenfield and Gnuboard7 both had about ~4% of initial findings removed; we verified Gnuboard7 findings twice with 237/238 agreement).

Severity

CVSS scores are from the Xint scanner. Severity is not exploitability for every class. The largest class, synchronous-DoS, is real on Node but usually conditional on data volume. We recommend treating the High-by-CVSS DoS count as an upper bound on real-world impact.

Sample size & balance

Per-model n is small (OpenAI 3-4 projects each, sonnet-4.6 2 on simple_board), so cross-vendor rates are suggestive, not significant. OpenAI built simple_board only, so the vendor comparison is restricted to that app. Effort tiers differ by vendor and are not equated. Gnuboard7 is n=1 app - a real-world contrast, not a population rate.

No external/human baseline

No non-AI reference projects exist in the data. Results are framed as relative prevalence (across models/vendors/languages/cohorts) and vs. an expected-secure baseline of zero - not "X% vs an industry baseline."

Taxonomy

Classes are the scanner's `bug_type_desc.name`, consolidated into auditable buckets (e.g. *Output/Protocol Injection* to XSS; *Cryptographic Weaknesses* to Credentials; *Structural Code Quality* + *Supply Chain* to Debug/Dependency-RCE). For Gnuboard7, the scanner's "Authenticity and Endpoint Spoofing" findings are request-integrity/spoofing issues (e.g. an integrity-less install) and appear as "Request Authenticity / Spoofing"; "Access Boundary / Traversal / SSRF" spans clickjacking/redirect in the toy apps and traversal/SSRF in Gnuboard7.

Causal language

Every "Why" is an explicitly labeled hypothesis about mechanism (code locality, runtime statefulness, training-data idioms), not a proven cause.

Key Finding #1

What is the most common vulnerability type in AI-generated code?

Missing controls for rate-limiting and DoS are the most common flaw, and the ones that actually bite in production.

Resource-exhaustion/DoS was the largest class of flaws found, regardless of severity, at 21% of findings (93 of 434), with authorization/IDOR (20%) and access boundary/traversal/SSRF (12%) in second and third place, respectively.

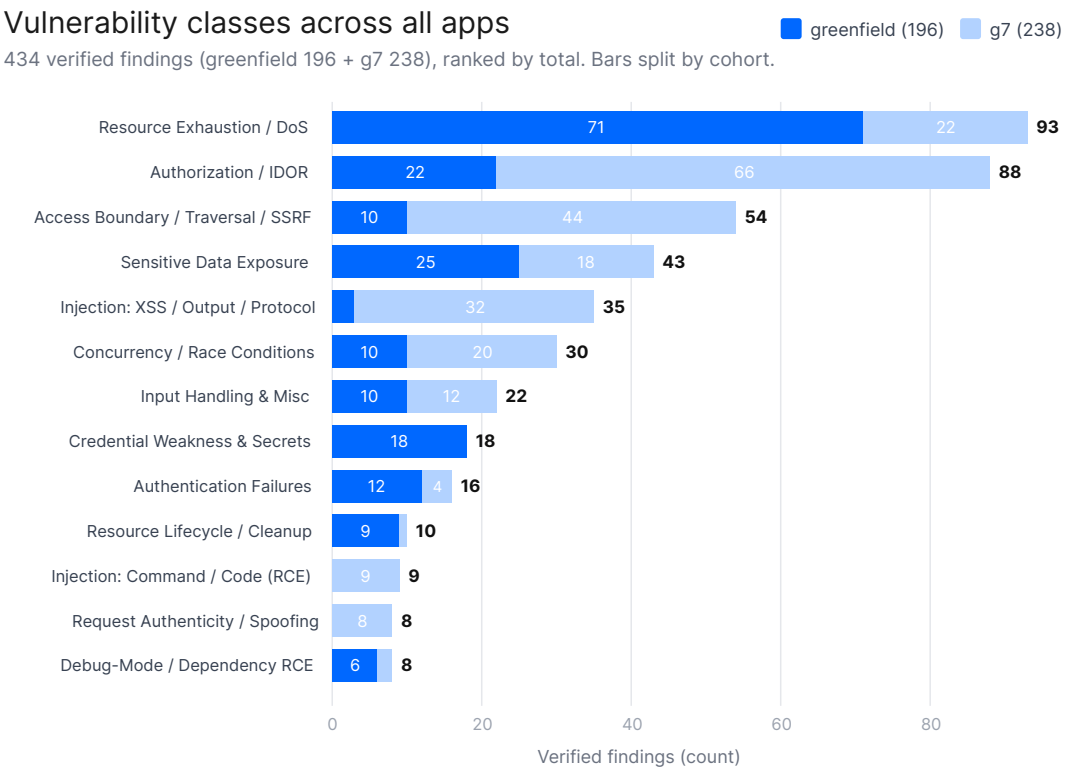
Resource exhaustion/DoS bugs were present in nearly every project (unbounded pagination, no rate limits, synchronous blocking work). By CVSS most are High, though exploitation is typically *conditional* (the attacker must first inflate data, or it depends on a single-threaded runtime.

Why

Though the code compiles, efficiency and performance are cross-cutting operational concerns overlooked by models when a developer only asks for features. "Don't let this be called a million times" is rarely in the prompt or the example code the model learned from, so it emits the working endpoint and omits the guardrail. The impact is not data theft, but in real operation it means runaway server cost or a server an attacker can knock over.

Action item for Dev/ProdSec

Don't just check to see if AI code compiles; look for how it performs and the resources it consumes in runtime.



Key Finding #2

What is the most common SEVERE vulnerability in AI code?

Secrets exposure is the top source of critical-severity bugs.

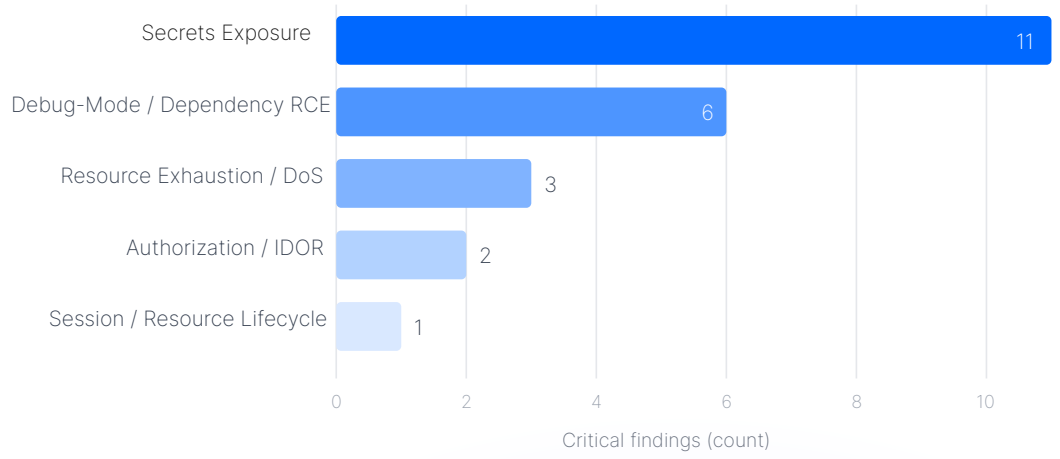
Hardcoded or default secrets (e.g. a default SECRET_KEY or JWT secret that lets an attacker forge sessions or tokens) were 11 of the 23 critical findings in the from-scratch apps (with debug-mode RCE, 17 of 23) - the single largest source of vulnerabilities identified as “critical”.

Why
These are the copy-paste quick-start defaults that dominate training data and do not change whether the app runs, so a “does it work” check never surfaces them.

Action item for Dev/ProdSec
Look for hardcoded secrets, such as API keys or PII, embedded in code produced by AI.

Critical vulnerabilities in AI-generated apps

Greenfield cohort, all models combined · 23 critical true-positive findings, grouped by class



Severity = Critical only. "Secrets Exposure" = hard-coded/default/predictable app secrets (SECRET_KEY, JWT) and weak password storage. Sensitive Data Exposure had 0 criticals.

Key Finding #3

Does app size matter in the type of vulnerabilities created?

Fine-grained authorization holds up on small apps but breaks as the app grows.

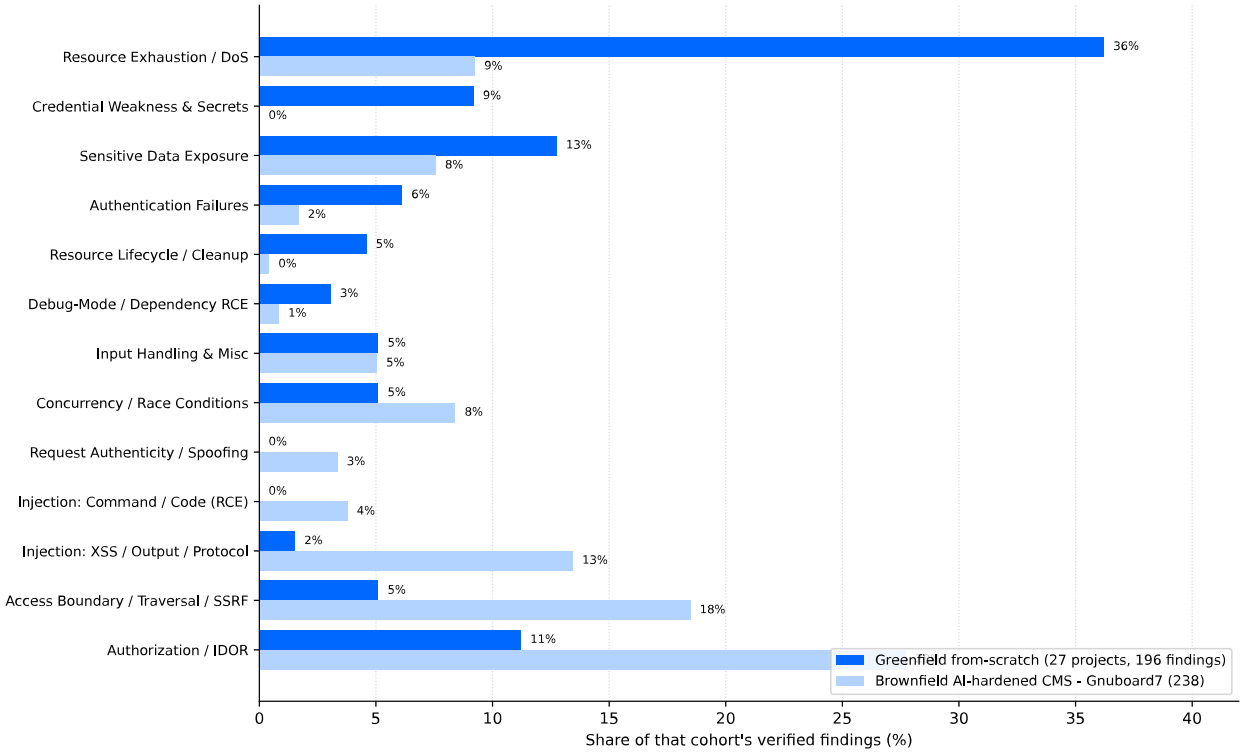
The simple cases (like stopping someone from editing another user's post) were handled well in the small builds but fell apart at scale. IDOR - a type of flaw where a user gets access to data beyond their permissions - was only 11% of findings (22 of 196) in the small spec/from-scratch apps, but the largest share of flaws at 28% (66 of 238) in the large G7 app. At some point of complexity, the AI would no longer check which users had access down to the granular object level.

Why
A same-user ownership check is local and easy, but in a big app the rule spans hundreds of endpoints and depends on runtime identity plus resource state, so a few sites always slip through.

Action item for Dev/ProdSec
Double check granular object permissions as the amount of endpoints in an application increases.

The failure mode flips with app type

From-scratch apps fail at cross-cutting guardrails (DoS); an AI-hardened real-world CMS fails at access control and injection - classes that scale with feature surface



The Takeaway

Developers are pushing out more code than human teams can adequately review and secure. Not only is the volume of code generated too large, but AI code is becoming less discernible to human readers. This doesn't mean organizations need to choose between AI or security - they just need AI enablement that goes beyond bug discovery.

In one week, a single researcher used Xint to scan 28 apps, isolate and validate nearly 500 unique bugs (including 20+ critical bugs), and then categorize them by type and severity. For this specific research he did not remediate the bugs but on average, using just the outputs from a standard Xint report, teams are able to patch each bug in less than 30 minutes.

Now that they know what to look for, our researchers can also use Xint's Operator Prompts feature to tell Xint to, for example, "pay close attention for API keys and other hard coded secrets" when scanning an application they know has been heavily touched by AI. They also know to tell Xint to hone in on object-level user permissions as Xint can maintain complex relationships even as the codebase gets bigger.

Yes, you can embrace AI agentic coding to scale your development. But without arming your product security teams with a tool like Xint that scales bug discovery, validation, and remediation, then you're shipping code that takes more resources to run and is easier for AI-enabled attackers to exploit.

About Xint.io

Xint.io is the award-winning autonomous pentesting platform built by the security researchers at Theori, the most decorated team of whitehat hackers in the world. Xint offers both blackbox as well as whitebox pentesting with results in less than 12 hours, including: full trigger conditions, exploit impacts, and suggested remediations so teams can quickly triage and patch the most critical vulnerabilities. Xint is working with organizations with the highest defensive security needs, including DARPA and Samsung.

About Theori

Theori is an offensive cybersecurity firm dedicated to solving the industry's most complex security challenges. Founded in 2016 by Carnegie Mellon alumni, our elite team of white hat hackers is trusted by global technology leaders and government agencies, backed by 70+ international hacking competition wins including a record four consecutive DEF CON CTF championships.

We offer a comprehensive security ecosystem: Xint (AI-powered application security testing), aprism (LLM security guardrail) and offensive security consulting. Certified to ISO/IEC 27001:2022 and ISO/IEC 27017:2015.

Globally Recognized Offensive Security Experts

Top-3 in the Dept of Defense DARPA Challenge

Winner of the database category

9x champion of the largest competition in the world

5x Codegate champion

4x consecutive winner

Trusted by organizations with the highest security standards