VIBE CODING SECURITY WEEKLY — MAY 26 - JUN 1, 2026

> **Download:** vibe-coding-security-weekly-jun-01-2026.pdf — printable, site-styled.

The week the number stopped being the point. Seven days after Veracode's "45% of AI-generated code contains OWASP Top-10 flaws" became the stat everyone repeats, OX Security published "62%." A builder-written explainer landed on the honest version of the same observation — "between 40 and 62%, depending on the study." Nobody can agree on the figure, and that turned out not to matter, because the field converged on something more durable: a reference breach (Moltbook — 1.5 million API tokens and 35,000 emails exposed 72 hours after launch) and a root cause that Thoughtworks finally said out loud — AI generates the path of least resistance, and the path of least resistance is rarely the secure one. The arguments this week were no longer about whether vibe-coded code is

risky. They were about who owns the control plane: the IDE, the platform, the governance layer, or the pull request.

TL;DR — The week in one paragraph

- **OX Security says 62%, May 27.** OX Security headlined "Why 62% of AI-Generated Code Is Vulnerable" — a higher number than last week's canonical Veracode 45%. Both are vendor-shaped figures; treat the exact percentage as directional. The piece's durable contribution is the **Moltbook breach** as a worked example: an AI-built social network that exposed **1.5M API authentication tokens plus 35,000 email addresses within 72 hours of launch**, because Row Level Security was never configured and no code review happened.
- **Thoughtworks names the root cause, May 28.** The "VibeSec Reckoning" write-up: when Thoughtworks' Global Marketing team tried to scale a citizen-builder prototype (Gemini + Replit AI + Claude) for 10,000 employees, work stopped twice — once on **public storage access**, once on **excessive token permissions**. The framing that stuck: "AI tools often suggest the path of least resistance. That path is not always the secure one."
- **Governance, not scanning, May 26.** In a Business Insight Journal interview, Arnica co-founder Nir Valtman put the structural problem in two numbers (Arnica's own, directional): **~59% of engineering teams now use three or more AI coding tools** simultaneously, and only **~2% of organizations have any central governance** over which tools developers use.
- **GitHub weighs PR controls, May 25.** Engineers Mario Zechner and Armin Ronacher warned that AI coding tools produce "vibe slop" — plausible "80% right" code that slips into production because review capacity doesn't scale with fleets of agents. GitHub is reportedly weighing pull-request controls in response. Moltbook is cited here too as the cautionary case.
- **AI-native attack surface, May 25 – Jun 1.** Two reminders that the entry point is moving off the code path: a Google Gemini prompt-injection flaw that leaked

private calendar data via a malicious invite (natural language, not a code exploit), and a Pluto Security breakdown of how Cursor's extension-recommendation flow can walk a developer from suggestion to install to execution of a fake extension with no review gate.

- **The practitioner response, May 27.** Soatech shipped a 12-point pre-production Cursor checklist; the recurring theme across all of it is that the IDE is fine and the *generated code* is the attack surface — citing Endor Labs' figure that **40% of AI-generated code introduces a vulnerable dependency**.

Why did "62%" show up the week after "45%" became canon?

On **May 27, 2026**, OX Security published "Vibe Coding Security: Why 62% of AI-Generated Code Is Vulnerable." One week earlier, we documented how Veracode's "45%" had become the canonical stat that founders, lawyers, and dealmakers were all reaching for. Now there are two canonical stats, and they disagree by 17 points.

A builder-written explainer the same week got to the honest framing faster than either vendor: "between 40 and 62% of AI-generated code contains security vulnerabilities, depending on the study." That is the real state of the number. Every figure in this space — Veracode's 45%, OX's 62%, the "over 30%" cited in the GitHub coverage, Endor Labs' 40%-introduce-a-bad-dependency — comes from a different corpus, a different definition of "vulnerable," and a vendor with a platform to sell. The data-studies credibility problem we keep flagging applies to all of them: the directional claim ("a large fraction of AI-generated code ships exploitable") is well-supported; the specific percentage is marketing.

What makes the OX piece worth reading anyway is that it anchors the abstraction to a named incident. The **Moltbook breach** (reported as January 2026): a founder built an AI social network "without writing a single line of code himself," and within **72 hours of launch** the app had exposed **1.5 million API authentication tokens**

and 35,000 email addresses. The cause was not a sophisticated exploit. Row Level Security policies — the entire authorization model on a Supabase-style backend — were never configured, and because the whole app emerged from conversational prompts, no code review ever happened to catch it. This is the exact failure mode we keep documenting in the naked-databases and RLS baseline work: the app passes every functional test and serves real users while the door stands open.

What to do: stop arguing about the percentage and adopt the worked example as your tabletop. If your app uses RLS-based authorization, the only number that matters is how many of your tables have a policy that is missing or set to `using (true)`. That is a query you can run today, not a study you have to trust.

What did Thoughtworks actually find when it tried to ship a citizen-builder app?

On **May 28, 2026**, the "VibeSec Reckoning" recounted a concrete internal case: Thoughtworks' Global Marketing AI applications team was asked to scale a **video-assembly prototype** — built by a non-technical "citizen builder" using **Gemini, Replit AI, and Claude** — into something 10,000 employees could use. The work stopped cold twice, and both times for the same underlying reason.

- **Security risk #1 — public storage access.** The AI's suggested path put assets in publicly readable storage.
- **Security risk #2 — excessive token permissions.** The AI provisioned credentials with far more scope than the task required.

In both cases the model recommended the configuration that was easiest to get working, and in both cases it took a human asking the right question to catch it. The sentence that names the whole problem: *"AI tools often suggest the path of least resistance. That path is not always the secure one."*

This is the most important framing shift of the week, because it replaces the unhelpful "AI writes buggy code" narrative with a precise mechanism. The model is not malicious and not incompetent. It is optimizing for "does this run," and the shortest route to "this runs" is public buckets, broad tokens, disabled RLS, and `cors: *`. Those are the same defaults that produced Moltbook. Thoughtworks' proposed fix is the one worth copying: **give the agent your security rules as context from the first prompt, then validate the output with deterministic checks in the workflow** — so the insecure default never survives to staging, instead of being caught by whoever happens to review.

What to do: treat "the AI took the easy path" as a predictable risk, not a surprise. Pre-load your `.cursorrules` / `CLAUDE.md` / system prompt with explicit "never make storage public, never grant a token broader than the operation, never disable RLS" rules — and then assume the model will violate them anyway and gate on a deterministic scan. Context reduces the rate; the gate is what actually stops it.

Why is everyone suddenly talking about governance instead of scanning?

In a **May 26** interview, Arnica co-founder Nir Valtman framed the AI-coding security problem as one of scale and chaos rather than code quality, and backed it with two numbers (Arnica's own, so directional): roughly **59% of engineering teams now run three or more AI coding tools at once** — Cursor, Copilot, Claude Code, in some mix — and only about **2% of organizations have any central governance** over which tools their developers are allowed to use.

If those proportions are even roughly right, they explain why the conversation moved off scanning this week. A scanner finds a vulnerability in one repository. It does nothing about an organization where dozens of developers are each routing proprietary code through a different, unvetted assistant with no inventory of which model touched what. Valtman's term for the fix is "developer-native governance" — security that lives inside the workflow rather than as a gate bolted on after. The

distinction that matters for buyers: *the AI-BOM question (which model touched this code, with what data residency) is now a governance question, not a code-review question.*

What to do: before you buy another scanner, write down the list of AI coding tools actually in use across your team and where each one sends your code. If you can't produce that list, you are in the 98% Valtman is describing, and no per-repo tool closes that gap.

What is "vibe slop," and why is GitHub weighing pull-request controls?

On **May 25**, coverage of comments from engineers **Mario Zechner and Armin Ronacher** crystallized a term that is going to stick: "vibe slop." The argument is that AI coding tools let users skip design, testing, judgment, and ownership while generating plausible-but-flawed code at scale — and the dangerous output is not the obviously-broken code (you catch that) but the **"80% right" code that slips into production** because human review capacity does not scale with fleets of agents churning out changes.

The structural response under discussion is **GitHub weighing pull-request controls** — gates that assume a meaningful fraction of incoming PRs are machine-authored and need a different review posture than a human's. This is the same insight as Thoughtworks' from the other direction: the bottleneck has moved from *writing* code to *reviewing* it, and the review layer was sized for human throughput. Moltbook shows up in this coverage too, as the canonical example of "80% right" — a working, deployed, user-serving app that was one missing policy away from a mass exposure.

What to do: if your team merges AI-authored PRs, decide explicitly what review a machine-authored change requires that a human-authored one might skip —

dependency provenance, secret scanning, auth-path diff review. “We review everything the same” is the policy that lets vibe slop through.

How are the attacks actually getting in?

Two write-ups this week were useful reminders that the entry point is sliding off the code path entirely.

Gemini calendar prompt injection (May 25). IntelligenceX documented a Google Gemini flaw where a **malicious calendar invite** carried injected instructions that caused private calendar data to leak — an attack carried entirely in natural language, with no traditional code exploit involved. This is the indirect prompt injection pattern reaching a mass-market surface: the attacker doesn’t need your code, just a field your AI assistant will read and obey. (The same write-up references a Cursor IDE RCE tracked as **CVE-2026-22708**, said to abuse shell built-ins to manipulate environment variables; that detail is single-sourced here, so treat the CVE specifics as unconfirmed pending a primary advisory.)

Cursor extension trust (Jun 1). Pluto Security walked through how Cursor’s strength — recommending tools based on your open files, `package.json`, error logs, and build output — becomes a risk amplifier. Their worked example: Cursor detects a linting error and suggests installing **“FastLint Pro,”** a fake extension targeting Cursor users; the flow goes suggestion → install → execution with **no code review and no security approval anywhere in the chain**. It is the MCP / tool-spec injection and Agent Skills attack surface problem wearing an IDE’s clothes: implicit trust in machine recommendations, no governance on what gets executed.

What to do: the common thread is that your AI tools will read attacker-controlled inputs (calendar invites, repo artifacts, error logs) and act on them with your privileges. Audit which of your AI surfaces can *install or execute* based on untrusted context, and put an approval gate in front of execution specifically — that is the step both attacks exploit.

The week's smaller stories

- **Soatech's 12-point Cursor checklist, May 27.** Soatech published a pre-production Cursor review checklist covering Privacy Mode, `.cursorrules` audits, dependency lock files, and code-level hardening. Notable citation: per Endor Labs, **40% of AI-generated code introduces a vulnerable dependency**, and Cursor can't detect typosquatting in package names it suggests. The IDE-level checks are Cursor-specific; checks 7–12 apply to any AI-generated code.
- **MCP security guidance keeps shipping, May 22.** obot.ai's "MCP Security Best Practices: The Complete 2026 Guide" is part of a steady stream of MCP-hardening content as Model Context Protocol becomes the default agent-to-tool plumbing — and a default new attack surface.
- **Social engineering the model, not the code, May 25.** "Hackers are learning to exploit chatbot personalities" ran in multiple syndications: a new class of attacker whose skill is conversational manipulation rather than code, steering a model into violating its own guardrails. Same theme as the Gemini and Cursor stories — the human-language layer is the soft target.
- **A "Claude Code source leak / infostealer" rumor is circulating — don't trust it yet.** Several content-farm domains (frost7.com, sharpdrillbits.com) published near-identical "Malware Alert: Hackers Exploit Claude Code Leak" posts on **May 28** claiming Claude Code's source leaked to GitHub with infostealer-laced repos. The articles are grab-bags that splice in unrelated items (an FBI FISMA breach, botnet takedowns, a Cisco source-code theft) and cite no primary source. Treat this as SEO chum until a credible advisory appears — flagging it precisely so you can recognize and discount it if it reaches your feed.

Why this week's stories rhyme

Last week the open question was the number. This week the field answered it by changing the subject. Four moves, one shared premise:

- **The breach got a name.** Moltbook (1.5M tokens, 35K emails, 72 hours, missing RLS) is now the reference incident that OX, the GitHub coverage, and downstream writeups all reach for — the way Orchids/BBC anchored last week. A named breach does more work than a percentage.
- **The root cause got named.** Thoughtworks said it plainly: AI ships the path of least resistance, and that path is public buckets, broad tokens, and disabled RLS. Moltbook is that sentence in production.
- **The fix moved upstream.** Thoughtworks (rules-as-context plus deterministic gates), Valtman (developer-native governance), and GitHub (PR controls) are three versions of the same idea: stop catching insecure defaults in review, stop them in the workflow.
- **The attack surface moved off the code path.** Gemini's calendar injection and Cursor's extension flow are the same exploit — trust a machine that read attacker-controlled input — and neither needs a code-level vulnerability to work.

The settled fact: the disagreement is no longer about whether AI-generated code is its own risk. It is about who owns the control plane — the IDE vendor (Cursor checklists), the platform (Replit/Supabase defaults), the governance layer (Arnica), or the pull request (GitHub). The number was never the open question. The owner is.

Manual checklist — 10 things to verify yourself

01. **Run the Moltbook test on your own app: count tables whose RLS policy is missing or `using (true)`.** That single query tells you more than any 45%-vs-62% study. Anon-key-readable rows are the exact failure that exposed 1.5M tokens in 72 hours.
02. **Inventory every AI coding tool in active use across your team, and where each sends your code.** If you can't produce the list, you're in the ~98% with

no central governance that Valtman describes. The AI-BOM is now a governance artifact, not a nice-to-have.

03. **Pre-load your `.cursorrules` / `CLAUDE.md` / system prompts with explicit "never public storage, never over-scoped tokens, never disable RLS" rules — then gate on a deterministic scan anyway.** Context lowers the rate; the gate is what stops the path of least resistance.
04. **Audit token and credential scopes the AI provisioned.** Thoughtworks' second showstopper was excessive token permissions. Check every service account and API key your vibe-coding tool created against the operation it actually performs.
05. **Check your storage buckets for public-read defaults.** Thoughtworks' first showstopper was public storage access — the easiest path the model could take. Grep your infra for publicly readable buckets and CDN paths.
06. **Decide what review a machine-authored PR requires that a human one doesn't.** Vibe slop is the "80% right" change that passes a normal review. Dependency provenance, secret scanning, and auth-path diffs are the checks that catch it; "we review everything the same" doesn't.
07. **Put an approval gate specifically in front of *execution and installation* triggered by AI recommendations.** The Cursor "FastLint Pro" flow and the Gemini calendar injection both exploit the install/execute step. Suggestion is fine; auto-execution on untrusted context is the hole.
08. **Audit which AI surfaces read untrusted input (calendar invites, repo artifacts, error logs, issue text) and can act on it with your privileges.** That is the indirect-prompt-injection attack surface, and it does not require any code-level CVE.
09. **Verify dependency lock files and typosquatting protection on AI-suggested packages.** Per Endor Labs, 40% of AI-generated code introduces a vulnerable dependency, and the IDE won't catch a package name that's a misspelling of a real one.
10. **Treat the "Claude Code source leak" malware posts as unverified until a primary advisory exists.** Recognize the content-farm pattern (no source, spliced-in unrelated breaches) so a rumor doesn't drive a real incident-response page-out.

Related coverage

- Vibe Coding Security Weekly — May 25, 2026 — Apple pulls 'Anything', Cursor Composer 2, OpenAI/Windsurf, the 45% becomes canon, Orchids/BBC
- Vibe Coding Security Weekly — May 18, 2026 — B1KEY 1,764-app audit, Cursor Bugbot, Mini Shai-Hulud, Apple WWDC framework
- Vibe Coding Security Weekly — May 11, 2026 — RedAccess 380K-app scan, Replit Security Agent, Vercel Deepsec
- The RLS Baseline: What a Default Vibe-Coded App Exposes — the missing-RLS failure mode behind Moltbook, measured on a real build
- Naked Databases — the pattern category Moltbook belongs to: backend reachable, authorization never configured
- Indirect Prompt Injection — the class the Gemini calendar attack falls under
- MCP & Tool-Spec Injection — the governance gap the Cursor extension flow exploits
- Your CLAUDE.md Is Attack Surface — implicit trust in machine recommendations as a threat model
- The Integration Layer Is the Real Security Gap — why per-tool scanners miss the path-of-least-resistance defaults

Sources

- OX Security — Vibe Coding Security: Why 62% of AI-Generated Code Is Vulnerable — May 27, 2026
- Business Insight Journal — Interview With Nir Valtman (Arnica) — May 26, 2026
- LavX — The VibeSec Reckoning: When AI-Assisted Development Meets Enterprise Security — May 28, 2026
- NewsBang — GitHub Weighs Pull-Request Controls as Engineers Warn 75% AI-Written Code Can Turn to Slop — May 25, 2026

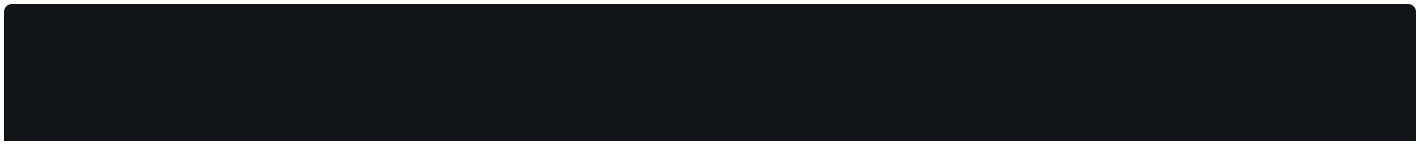
- IntelligenceX — Google Gemini Prompt Injection Flaw Exposed Private Calendar Data via Malicious Invites — May 25, 2026
- Pluto Security — Cursor Security Issues in AI Coding Tools and Execution Flows — June 1, 2026
- Soatech — Cursor Security Review: The 12-Point Pre-Production Checklist — May 27, 2026
- berniepng — Your AI Wrote That Code. Did It Also Leave the Door Open? — May 25, 2026
- obot.ai — MCP Security Best Practices: The Complete 2026 Guide — May 22, 2026
- trendyfii — Hackers are learning to exploit chatbot 'personalities' — May 25, 2026

This digest is compiled from public reporting. VibeEval is not affiliated with OX Security, Arnica, Thoughtworks, GitHub, Google, Cursor, Pluto Security, Soatech, Endor Labs, Replit, Supabase, Anthropic, obot.ai, or any other organization cited. Numbers from vendor or marketing-shaped writeups (OX's 62%, Veracode's 45%, Arnica's 59%/2%, Endor Labs' 40%, the "over 30%" in the GitHub coverage) are attributed and directional. The "Claude Code source leak" malware claim is unverified and flagged as such. Questions? Contact our team.

/ MORE UPDATES

Keep reading

- 2026.05.29 THOUSANDS OF VIBE-CODED APPS ARE LEAKING DATA — AND GOOGLE INDEXED THEM.
- 2026.05.25 VIBE CODING SECURITY WEEKLY — MAY 19 - MAY 25, 2026
- 2026.05.18 VIBE CODING SECURITY WEEKLY — MAY 12 - MAY 18, 2026
- 2026.05.13 2026 AI CODING SECURITY REPORT: THE DATA BEHIND THE VIBE-CODING BREACH WAVE



/ NEXT STEP

STOP GUESSING. SCAN YOUR APP.

Join the founders who shipped secure instead of shipped exposed. 14-day trial, no card.

START FREE SCAN →

/ NEWSLETTER

GET THESE WEEKLY

One email, every Monday. Five vibe-coding security stories from the past seven days — dated, sourced, opinionated. Unsubscribe anytime.

/ PLATFORM REVIEWS

Is Lovable Safe?

Is Cursor Safe?

Is Bolt Safe?

All platforms →

/ ALTERNATIVES

Snyk Alternative

Burp Suite Alternatives

All alternatives →

LATEST

Weekly Digest (May 25)

Weekly Digest (May 18)

Lovable May 2026 Report

Lovable Apr 2026 Report

Lovable BOLA Vulnerability

Vercel/Context.ai Breach

Agent Skills Security

All updates →

AI Pentesting

LEGAL

Trust & Security

Privacy

Terms

DPA

Refunds

PROGRAMS

Get paid

VIBE-EVAL.COM