CSAI Foundation

Cloud Security Alliance AI Safety Initiative

Vibe Coding Security Debt: AI-Generated Vulnerabilities at Scale

Risk Patterns, Real-World Incidents, and Organizational Guidance

Unofficial AI-assisted Research

2026-04-06

Key Takeaways

- The Georgia Tech Vibe Security Radar confirmed 74 AI-linked CVEs through March 2026, with a roughly 6x increase in monthly new CVEs from January to March 2026 alone — and researchers estimate the true number is 5–10x higher in undetected cases. [7]
- Between 45% and 70% of AI-generated code samples fail security tests depending on the methodology and tools evaluated, with authorization flaws, missing access controls, and hardcoded credentials as the dominant failure patterns. [2][5]
- Real-world consequences are material: a single Firebase misconfiguration in one AI-generated platform exposed 406 million records in January 2026, and a compilation of 20 or more documented AI application breach incidents confirms the pattern extends broadly across the sector. [24]
- AI coding tools introduce a new class of supply-chain risk through package hallucination — approximately 19.7% of AI-suggested dependencies in Python and JavaScript are non-existent names that attackers can register as malicious packages. [14]
- Developer confidence in AI-generated code is systematically miscalibrated: over 75% of developers in one survey believed AI-generated code is more secure than human-written code, even while 56% admitted it frequently introduces security issues. [4]
- Existing CSA frameworks — particularly the AI Organizational Responsibilities guidance, LLM Authorization Practices, and the MAESTRO threat model — provide directly applicable controls for organizations managing this risk today.

Background

"Vibe coding" — a term coined by researcher Andrej Karpathy in February 2025 and named Collins Dictionary's Word of the Year for 2025 [1] — describes an emerging development practice in which programmers generate entire applications or functional subsystems through natural-language prompts, with minimal hands-on review of the resulting code. The phenomenon has moved rapidly from novelty to industry norm: 25% of Y Combinator Winter 2025 startups had codebases that were 95% or more AI-generated [25], and Claude Code alone accounted for more than 4% of all GitHub commits by early 2026. [7]

The appeal is straightforward. Tools like Cursor, GitHub Copilot, Claude Code, OpenAI Codex, Lovable, Replit, and Devin compress development timelines dramatically, enabling solo developers or small teams to produce functional multi-service applications in hours rather than weeks. The productivity gains are real, particularly for prototyping, proof-of-concept work, and early-stage startups operating under competitive time pressure.

The security implications of this shift, however, are only now becoming clear through empirical evidence. The core problem is structural: large language models produce code by predicting plausible continuations of patterns from their training data. They do not reliably reason about threat models, do not consistently apply authorization logic across multi-role systems, and do not spontaneously anticipate how a given code path will interact with the rest of a system under adversarial conditions. Security controls that require explicit, system-wide intent — such as row-level database access policies, API authentication middleware, or cross-service authorization checks — are exactly the category where AI code generators perform worst. They are also the controls most likely to be invisible to developers who lack a security background and are relying on AI tools to handle implementation details.

CSA's own research identified this dynamic in July 2025, finding that 62% of AI-generated code solutions contain design flaws or known security vulnerabilities, even when using the latest foundational models. [3] The underlying failure modes are consistent across tools and languages: insecure patterns repeated from training data, dangerous optimization shortcuts such as use of `eval()` for dynamic execution, missing access control implementations, and subtle logic errors in multi-step authorization flows.

Security Analysis

The Vulnerability Profile of AI-Generated Code

Multiple independent assessments have characterized the vulnerability landscape of AI-generated code in quantitative terms, and the results are consistent enough to describe a pattern rather than an isolated finding.

Veracode's 2025 GenAI Code Security Report tested over 100 LLMs across 80 curated coding tasks in Java, Python, C#, and JavaScript, finding that 45% of AI-generated code samples failed security tests against the OWASP Top 10. [2] Java was the highest-risk language at a 72% failure rate; JavaScript and Python ranged between 38% and 45%. Cross-site scripting (CWE-79) proved the worst single category — only 12–13% of AI-generated code written for XSS-prone tasks was secure, representing an 86% failure

rate for that specific vulnerability class. Notably, Veracode found that security performance has remained largely flat across successive model generations, suggesting that scale and capability improvements in underlying models do not automatically translate into more secure code output. [2]

Checkmarx's independent evaluation placed the overall insecurity rate higher, finding that up to 70% of AI-generated code across multiple developer assistants was insecure. [5] A December 2025 study by CSO Online evaluated five major agentic coding platforms — Claude Code, OpenAI Codex, Cursor, Replit, and Devin — building identical test applications from standardized prompts. That study found 69 vulnerabilities across 15 generated applications, with the most common critical class being API authorization logic failures and broken business logic — vulnerability types the researchers attributed to AI's lack of intuitive understanding of system-wide intent. [15]

One finding from the CSO Online study complicates the picture in an important way: the agentic tools tested produced zero exploitable SQL injection or cross-site scripting vulnerabilities, in contrast to the higher rates Veracode observed in direct LLM API testing. This divergence likely reflects training improvements in recognizing and avoiding classic injection patterns, while authorization logic — which varies by application and requires holistic reasoning — remains the persistent weak point. The practical implication is that AI-generated code is not uniformly weak; it has specific, characterizable failure modes that security programs should target.

A 2025 iterative degradation study introduced an additional dynamic: when GPT-4o was asked to modify code up to 40 times in succession — a pattern not uncommon in iterative AI-assisted development workflows — the resulting codebase contained 37% more critical vulnerabilities than the initial output after only five iterations, as reported by secondary coverage. [17] Each revision introduces fresh opportunities for the model to omit or modify security-relevant logic, and the cumulative effect compounds in ways developers do not observe in isolation.

Real-World Breach Incidents

The theoretical vulnerability rates documented in controlled studies have found direct expression in a series of significant breach incidents. A compilation by security researchers tracking 2025–2026 AI application breaches identified more than 20 documented incidents, the majority linked to vibe-coded or AI-assisted applications deployed without adequate security review. [24]

CVE-2025-48757, disclosed in May 2025, exemplified the systemic nature of the problem. The vulnerability affected Lovable-generated applications that failed to implement Supabase Row Level Security (RLS) policies — an access control mechanism that must be explicitly configured to prevent public database access. Security researchers found 303 vulnerable endpoints across 170 applications in a scan of 1,645 Lovable apps, representing roughly 10% of the platform's publicly accessible applications.

Because Lovable embedded the Supabase anonymous key in client-side JavaScript, unauthenticated attackers could read and write arbitrary database tables containing PII, financial records, developer API tokens, and administrative credentials. [8] The root cause was not a flaw in any single application but a design pattern in the AI tool itself: Lovable did not implement RLS unless explicitly instructed, while users lacked the security knowledge to know it was required.

The same pattern of missing database access controls appeared in the Moltbook incident (January 2026), in which an application whose founder publicly stated he "didn't write a single line of code" exposed 4.75 million records, including 1.5 million API tokens and 35,000 user email addresses. [11] The Tea App experienced two distinct breaches in July 2025: an unprotected Firebase storage instance exposed tens of thousands of images including government-issued identity documents, and a second breach three days later exposed over a million private messages through an API endpoint that performed no access control. [10][24] The Base44 AI vibe coding platform — subsequently acquired by Wix — suffered an authentication bypass vulnerability that exposed user accounts and session data; Wiz Research's disclosure characterized it as a critical weakness in the platform's identity verification logic. [9] The Chat & Ask AI platform suffered an incident in January 2026 in which a Firebase instance configured with `allow read: if true` by default exposed 406 million records belonging to approximately 25 million users. [24]

A January 2026 study by CovertLabs scanning 198 iOS AI applications found that 196 of them — 98.9% — contained security misconfigurations exposing sensitive data across millions of users. [23] A concurrent Cybernews analysis of 38,630 Android applications with AI functionality found 197,092 unique hardcoded secrets — API keys, credentials, and tokens embedded directly in application code — in 72% of the applications analyzed. [12]

At the tooling layer, CVE-2025-54135 (designated "CurXecute") demonstrated that the development tools themselves introduce attack surface: this vulnerability allowed attackers to instruct the Cursor AI development tool to execute arbitrary commands on a developer's machine through a connected Model Context Protocol (MCP) server. [17] CVE-2025-53109 exposed arbitrary file read and write capabilities on developers' machines via Anthropic's file-system MCP server, with advertised access restrictions failing to function as documented; the vulnerability was disclosed by Cymulate as part of a coordinated advisory covering two related symlink sandbox escape issues. [26] These vulnerabilities target the development environment rather than the deployed application, representing a distinct risk vector that extends the attack surface to the developer workstation.

Supply Chain Risk: Package Hallucination

A third risk vector has emerged from a structural characteristic of how language models generate code: the tendency to reference package names that do not exist. A March 2025 study analyzing 576,000 generated Python and JavaScript code samples from 16 LLMs found that 19.7% of AI-suggested package dependencies were hallucinated — references to packages that have no corresponding published package in PyPI or npm. [14] Open-source models hallucinated at approximately 22%; commercial models at roughly 5%. Critically, 43% of hallucinated package names were produced consistently across multiple queries — meaning the same non-existent package names were suggested repeatedly and predictably. [14]

Researcher Seth Larson of the Python Software Foundation coined the term "slopsquatting" to describe the attack pattern this creates: a threat actor identifies commonly hallucinated package names, registers those names in public package registries with malicious payloads, and waits for developers using AI tools to inadvertently install them. [14] As of the writing of this note, no confirmed slopsquatting exploitation at scale has been publicly documented, but the preconditions are well established — the hallucination rates are measurable, the package names are consistent and predictable, and the registration of such names on PyPI and npm requires minimal barrier. Security researchers at Socket have characterized the attack surface as "common, repeatable, and semantically plausible." [14]

Miscalibrated Developer Confidence

A consistent finding across multiple studies is that developer confidence in AI-generated code is not well-calibrated to actual security outcomes. Snyk's AI Code Security Report found that over 75% of developers surveyed believed AI-generated code is more secure than human-written code — while 56% of the same respondents admitted that AI-generated code sometimes or frequently introduced security issues into their codebases. [4] Fewer than 25% of developers reported using software composition analysis tooling on AI-generated code suggestions, and roughly 80% admitted to bypassing security policies at some point in their AI-assisted development workflow. [4] This combination — elevated confidence, reduced verification — creates the conditions in which vulnerabilities are introduced and remain undetected until exploitation.

Recommendations

Immediate Actions

Organizations should treat AI-generated code as unverified input, not trusted output. This means integrating static application security testing (SAST) and software composition analysis (SCA) as mandatory gates in the CI/CD pipeline for any codebase that uses AI-assisted generation, not as optional post-development steps. Any code path touching authentication, authorization, data persistence, or external API integration deserves particular scrutiny — these are the categories where available assessments consistently show AI tools underperforming compared to security requirements.

Development teams should audit existing AI-generated applications specifically for missing database access controls, particularly Supabase RLS policies, Firebase security rules, and equivalent mechanisms in other backends. The Lovable CVE and the Moltbook incident both demonstrate that this class of misconfiguration is common, consequential, and technically straightforward to detect and fix once identified.

The presence of hardcoded secrets in AI-generated code is common enough to treat as a default assumption pending scan results. Secret scanning should run immediately against any AI-assisted codebase that has not been subjected to prior review, with particular attention to client-side code that may embed backend API keys or service credentials accessible in browser or mobile application bundles.

Short-Term Mitigations

Organizations should implement software bills of materials (SBOMs) for AI-generated components and integrate dependency scanning that can flag packages with minimal publication histories or unusual registration timing — early indicators of slopsquatting. Developers using AI coding tools should be instructed to verify that any AI-suggested package dependency exists in the relevant registry before accepting it.

The OpenSSF Security-Focused Guide for AI Code Assistant Instructions recommends configuring AI tools with explicit security prompting — instructing the model to use parameterized queries, enforce input validation, apply least-privilege access patterns, and generate corresponding test cases for security-relevant logic. [6] The Guide also endorses Recursive Criticism and Improvement (RCI), a technique in which the developer prompts the AI to review its own output and iteratively improve the security properties of generated code, citing measurable security improvement within two revision cycles. [6] Organizations should codify these prompting patterns in team standards rather than leaving them to individual discretion.

Human review requirements for AI-generated code should be tiered by risk: code that touches authorization logic, external API boundaries, data persistence, or cryptographic operations should require review by a developer with explicit security training, not only a peer code review from a teammate who may also be relying on AI tools to evaluate AI output. NIST SP 800-218A provides a government-aligned baseline for integrating these controls into secure software development practices for GenAI-assisted development. [16] Palo Alto Networks Unit 42 has published complementary operational guidance for configuring and hardening AI coding tool environments at the enterprise level. [22]

Strategic Considerations

The Vibe Security Radar tracking project at Georgia Tech SSLab provides a real-time window into AI-attributed CVEs across public repositories. Organizations that contribute to or consume open-source dependencies should monitor this data as part of their vulnerability intelligence program, particularly as the monthly CVE count has grown from 6 in January 2026 to 35 in March 2026 — a trajectory suggesting the problem is accelerating, not stabilizing. [7]

The fundamental strategic challenge is that AI coding tools dramatically lower the barrier to producing functional code while not lowering — and in some respects raising — the bar required to evaluate that code for security properties. This creates a skills gap dynamic that will persist and likely widen: organizations will accumulate AI-generated security debt faster than they can develop the human capacity to review it. Sustainable management of this debt requires investment in tooling that does not depend on manual review as the primary control — automated security testing integrated into deployment pipelines, dependency verification at build time, and runtime monitoring that can detect anomalous data access patterns consistent with missing authorization controls.

CISO-level framing should treat vibe-coded applications as a distinct asset class with its own risk posture, separate from traditionally developed software. Existing security questionnaires and vendor assessment processes do not yet routinely ask whether applications were AI-generated or whether AI-specific security controls were applied during development. Updating these processes to capture AI development methodology is a necessary governance step.

CSA Resource Alignment

This research note connects directly to several established CSA frameworks and publications that provide implementation-level guidance for the risks described above.

The CSA document *AI Organizational Responsibilities: AI Tools and Applications* (January 2025) addresses secure LLM application development practices, including prompt injection defense, output evaluation and guardrails, third-party supply chain management for AI components, and integration of security gates into CI/CD pipelines. [18] The companion document *Securing LLM-Backed Systems: Essential Authorization Practices* (August 2024) provides specific guidance on sandboxing AI-generated code execution and implementing human-in-the-loop approval processes for autonomous code generation, directly applicable to the agentic coding tools that produced many of the incidents documented here. [19]

The CSA MAESTRO framework (Multi-Agent Execution and Security Threat Reasoning and Orchestration) provides a structured threat model applicable to AI coding agents operating with tool access, file system permissions, and external service connections — the attack surface demonstrated in CVE-2025-54135 and CVE-2025-53109. Organizations deploying agentic coding environments should map their tool permissions and execution contexts against MAESTRO's control recommendations.

The CSA Cloud Controls Matrix (CCM) and AI Controls Matrix (AICM) provide control families applicable to AI-assisted development environments, particularly in areas covering change management, secure development lifecycle, cryptographic controls, and third-party dependency management. The AICM, as a superset of CCM, should be the primary reference for organizations operating AI-integrated development environments. The *State of Cloud and AI Security* report (September 2025) documented the broader security posture trends in cloud-based AI deployments that provide context for the governance gaps exploited in the incidents described above. [20]

The *State of AI Security and Governance* report (December 2025) documented that governance is the strongest predictor of AI security readiness, yet 72% of respondents were not confident in their ability to secure AI systems. [21] The vibe coding vulnerability surge is in part a governance failure — organizations adopted AI development tools without equivalent investment in the security processes and tooling required to manage the risk those tools introduce.

References

[1] Kaspersky. "What Is Vibe Coding and What Security Risks Does It Bring?." Kaspersky Blog, 2025.

[2] Veracode. "2025 GenAI Code Security Report." Veracode, 2025.

[3] Cloud Security Alliance. "Understanding Security Risks in AI-Generated Code." CSA Blog, July 9, 2025.

[4] Snyk. "AI Code Security Report." Snyk, 2023. (Survey data on developer confidence and security practices around AI-generated code.)

[5] Checkmarx. "Security in Vibe Coding." Checkmarx Blog, 2025.

[6] OpenSSF Best Practices Working Group. "Security-Focused Guide for AI Code Assistant Instructions." OpenSSF, August 1, 2025.

[7] Infosecurity Magazine. "AI-Generated Code Vulnerabilities Are Tracked by Georgia Tech." Infosecurity Magazine, March 2026.

[8] SecurityOnline. "CVE-2025-48757: Lovable's Row Level Security Breakdown Exposes Sensitive Data Across Hundreds of Projects." SecurityOnline, May 2025.

[9] Wiz Research. "Critical Vulnerability in Base44 Authentication." Wiz Blog, July 29, 2025.

[10] Barracuda Networks. "Vibe Coding and the Tea App Breach: Why Security Can't Be an Afterthought." Barracuda Blog, December 22, 2025.

[11] Barrack AI. "Every AI App Data Breach: 2025–2026." Barrack AI Blog, 2026.

[12] Cybernews Research. Cited in: Barrack AI. "Every AI App Data Breach: 2025–2026." Barrack AI Blog, 2026. (Original Cybernews study: January 2026.)

[13] Escape. Cited in: Barrack AI. "Every AI App Data Breach: 2025–2026." Barrack AI Blog, 2026. (Original Escape scan: October 2025.)

[14] BleepingComputer. "AI Hallucinated Code Dependencies Become New Supply Chain Risk." BleepingComputer, March 2025.

[15] CSO Online. "Output from Vibe Coding Tools Prone to Critical Security Flaws, Study Finds." CSO Online, December 2025.

[16] NIST. "Secure Software Development Practices for Generative AI and Dual-Use Foundation Models: NIST SP 800-218A." NIST CSRC, July 26, 2024.

[17] Autonoma. "Vibe Coding Security Risks." Autonoma Blog, 2025–2026. (Secondary coverage aggregating iterative degradation study findings and CVE-2025-54135 disclosure.)

[18] Cloud Security Alliance. "AI Organizational Responsibilities: AI Tools and Applications." CSA, January 2025.

[19] Cloud Security Alliance. "Securing LLM-Backed Systems: Essential Authorization Practices." CSA, August 2024.

[20] Cloud Security Alliance. "The State of Cloud and AI Security 2025." CSA, September 2025.

[21] Cloud Security Alliance. "The State of AI Security and Governance." CSA, December 2025.

[22] Palo Alto Unit 42. "Securing Vibe Coding Tools." Unit 42 Blog, 2025–2026.

[23] CovertLabs. Cited in: Barrack AI. "Every AI App Data Breach: 2025–2026." Barrack AI Blog, 2026. (Original CovertLabs Firehound study: January 2026.)

[24] Barrack AI. "Every AI App Data Breach: 2025–2026." Barrack AI Blog, 2026.

[25] Garry Tan (Y Combinator CEO). "Statement on YC W25 AI-Generated Codebases." X (formerly Twitter), March 6, 2025. Widely cited in press coverage including TechCrunch, March 6, 2025.

[26] Cymulate Research Team. "CVE-2025-53109 / CVE-2025-53110: EscapeRoute – Anthropic MCP Filesystem Server Sandbox Escape." Cymulate, 2025.