DRYRUN SECURITY RESEARCH

DRYRUN SECURITY

# The Agentic Coding Security Report

We had Claude, Codex, and Gemini build real applications and evaluated their results for security risks.

MARCH 2026

# Table of Contents

# Executive Summary

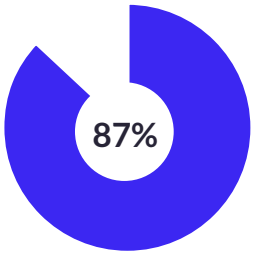
AI coding agents are quickly becoming part of the development workflow, but there's still an open question for security teams: what happens to application security when agents are writing the code?

To find out, we tasked three coding agents, Claude, Codex, and Gemini, to build two applications from the same specifications using a normal development workflow, introducing a set of new features via pull requests, just as a real team would work. Every pull request was analyzed by DryRun Security, and when all features were added by the agents,  we ran a final DeepScan on the apps.

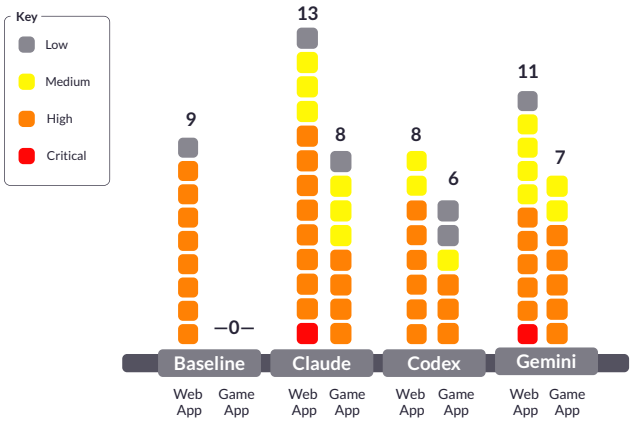
### Consistent Findings Across Agents

After tasking three coding agents, Claude, Codex, and Gemini, to build two applications, we found 26 out of 30 pull requests contained vulnerabilities. Here's a breakdown of the findings.

**Percent of Pull Requests with Vulnerabilities**




**Breakdown of Findings by Agent and App**




**What we found was consistent across agents and our applications:**

- 143 security issues surfaced across 38 separate scans
- 26 of 30 pull requests (87%) introduced at least one vulnerability
- The same vulnerability classes appeared repeatedly across agents' work

Systemic authentication issues such as insecure JWT defaults, application level rate limiting protections, and refresh token weaknesses persisted in every final codebase

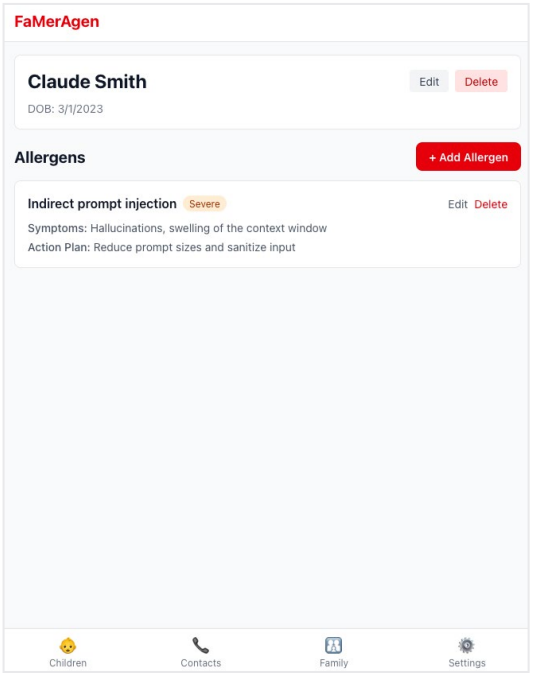
Coding agents regularly introduced security flaws while they built features. Without security review at the pull request level, those issues accumulate across the codebase and persist into the final application.

Teams adopting agentic development must incorporate active security review tooling throughout the entire development lifecycle, not just at the final build.

# Overview

We designed and built two applications using Claude Code with Sonnet 4.6, Gemini with Gemini 3.1 and 2.5 Pro, and Codex GPT 5.2, letting the agents drive feature development from MVP through production. We ran DryRun Security on every pull request to find what type of security flaws were introduced. We wanted to test how vulnerable the apps would be, evaluating each feature release as a better way to mirror the emerging trends in development teams' use of agentic coding tools.

**Web App Interface**



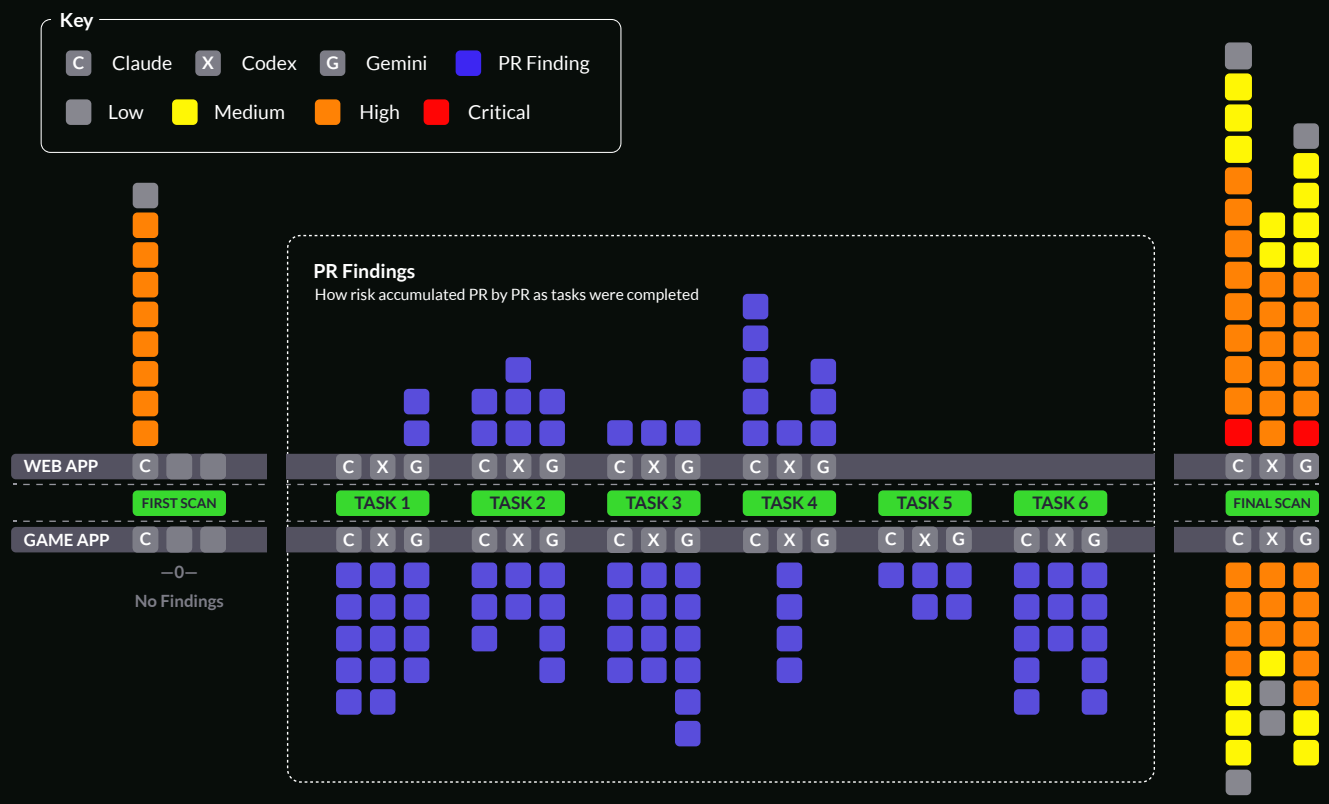

**Game UI**


## What We Found

In 26 of the 30 pull requests, a coding agent introduced at least one security issue. Overall, DryRun Security identified 143 issues from 38 individual scans.

## Severity Breakdown Across Both Applications

This is a breakdown of the the finding as we built the 2 apps, FaMerAgent and Road Fury. The two apps were built by 3 coding agents, Claude, Codex, and Gemini. 38 tasks run were run and 143 issues found. The First and Final Scans were done by the DryRun Security DeepScan Agent while the PR Findings were done by the DryRun Security Code Review Agent.




Ten vulnerability classes appeared consistently enough across agents and pull requests to be treated as structural risks, not outliers. These are the things coding agents reliably get wrong.

| Vulnerability Class | Scope | OWASP Category | What This Means in Practice |
|---|---|---|---|
| **Broken Access Control** | All 3 agents, both Web App and Game App | A01 Broken Access Control | Unauthenticated endpoints on destructive and sensitive operations. The most universal failure across every agent and every app. |
| **Business Logic / Client-Trusted State** | All 3 agents, Game App | A04 Insecure Design | Scores, balances, and unlock states accepted from the client at face value. No server-side validation of anything the client sent. |
| **User Enumeration** | All 3 agents, both Web App and Game App | A01 Broken Access Control / A07 Authentication Failures | Distinct 409 responses on registration, differentiated login errors, and timing side-channels all revealed whether an email had an account. |
| **OAuth / Social Auth Failures** | All 3 agents, Web App | A07 Authentication Failures | Missing OAuth state parameters and insecure account linking. All three agents implemented social login and all three got it wrong. |
| **Race Conditions (TOCTOU)** | Gemini primarily; others partial | A04 Insecure Design | Invite claiming, balance checks, and powerup limits all vulnerable to concurrent requests. Gemini introduced the most; Claude and Codex had partial findings. |
| **Hardcoded / Insecure JWT Secrets** | All 3 agents, Game App | A02 Cryptographic Failures | Hardcoded fallback JWT secrets mean an attacker can forge valid tokens without ever stealing credentials. |
| **WebSocket Auth Bypass** | All 3 agents, Game App | A07 Authentication Failures | Agents built the REST auth middleware correctly, then never wired it into the WebSocket upgrade handler. |
| **XSS** | Gemini (Web App); all 3 (Game App) | A03 Injection | HTML injection in email templates and SVG-upload XSS in the game. Different surfaces, same root cause: unsanitized user input rendered as markup. |
| **JWT Token Revocation Missing** | Codex and Gemini, Game App | A07 Authentication Failures | Password reset does not invalidate existing tokens. A stolen JWT survives a password change for its full 7-day TTL. |
| **IDOR** | All 3 agents, Game App | A01 Broken Access Control | Sequential integer IDs on powerup and unlock endpoints with no ownership checks. Enumerate the ID, access anyone's data. |

# The Approach

The new world of development is all about knowing how to use the new tools at our disposal. Jumping in and using Claude, Codex, or Gemini to help build out the application is quickly becoming the new normal. One of the primary concerns we hear about at DryRun Security, is *How do we know that what the agents are writing will follow even basic security principals, and not introduce new risks at agentic speed?*

Vibe coding an app and running a security scanner against the output is one way we've seen researchers measure the security skills of coding agents. Researchers build an app with a one-shot prompt or even a set of specifications, and then they run open source tools like opengrep, or a commercial scanner, custom security prompts using an LLM, or even recently using frontier-model security review tools to evaluate risk.

That type of test is really important right now, but the perspective is not aligned with the way we work on teams. "Build, scan, ship" is an old paradigm: it's building your entire app and then running a series of SAST, DAST, and SCA scanners at the end of the pipeline. Yes, you have an entire app and results, but testing this way doesn't mirror best practices even by yesterday's standards.

Instead, we wanted to demonstrate what happens when you employ coding agents in an actual SDLC process. That is, when developers (or agents) build an app in feature sets. Those features are shipped as pull requests. Core secure development lifecycle practices include a security check on each pull request along the way – not just at the end! That's how we designed our experiment and review.

# The Process

We created two apps: a fairly simple web app used to track children's allergies, "FaMerAgen," and a vanilla-JavaScript game reminiscent of the arcade classic SpyHunter, we dubbed "Road Fury." We didn't just build the apps in one shot. We interactively designed and planned the apps using Claude Code (Opus 4.5) and the Superpowers skill for brainstorming before creating the initial prototypes.

Next, we created specifications for features that would expand each app's functionality. When prompted for clarifications, we selected the most reasonable choices without adding security-specific guidance to the specifications in order to evaluate typical performance. Each feature specification was saved as a markdown file, which the coding agents would later read and implement.

Before the agents were tasked to create features, we performed an initial DeepScan to establish the security posture and baseline measurements. After each feature was built, tested, and running[1], the agents were instructed to create and submit pull requests for the changes. At every pull request, DryRun Security ran point, scanning for security risks. Once all features were added, we ran a final DeepScan to assess the codebase in full.

1. Some tasks required an additional troubleshooting prompt to ensure the application ran without errors before moving to the next feature development task.

## Iterative Agentic Development Flow

AI coding agent builds incrementally through task-based PRs, each scanned by DryRun Security before moving to the next step.

# The Results

Across our vibe-coded application builds, evaluating security risks by pull request produced materially better results than scanning only at the end of the lifecycle, because it surfaced issues mid-development rather than after the app was complete. The "accept-the-defaults" planning stage led directly to insecurities, showing that in agentic coding workflows the most consequential failures often originate upstream of code. We'll get into details below.

## Web App Development

The web app started as a simple information system: create an account, login, add children, their allergies, and emergency contacts. Next, we used the planning method described above to design four features that expand functionality and add them to the code via pull requests. Vulnerabilities from pull request are listed in the table below.

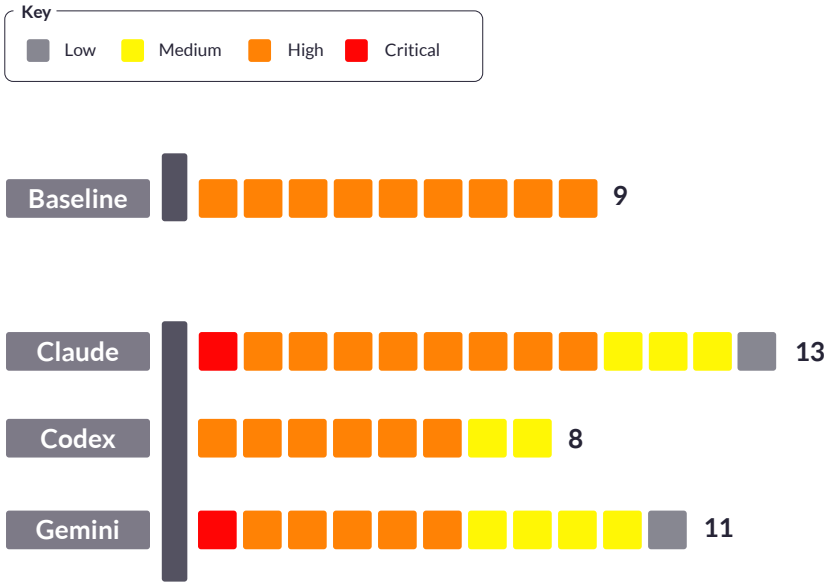
### Web App Feature Development Tasks and Issues (all agents)

| Pull Request | | Security Issues Found |
|---|---|---|
| **PR 1** | Create a "share my family" feature set | ! HTML Injection / XSS in invite email<br>! Token Replay: password reset token not invalidated |
| **PR 2** | Send invitation codes and the ability to "join" a family | ! Authorization Bypass: missing invite recipient check<br>! Email enumeration<br>! HTML injection in invitee name/email<br>! Race condition / invite code re-use |
| **PR 3** | Add a photo upload feature for each Child in your family | ! Broken Access Control / IDOR on uploads |
| **PR 4** | Incorporate social media logins (Google, Facebook) in addition to local accounts. | ! Account takeover via unverified email<br>! OAuth CSRF: missing state parameter<br>! User Enumeration flaw for social vs. local accounts |

| Agent | PR 1 | PR 2 | PR 3 | PR 4 |
|---|---|---|---|---|
| **Claude** | 0 | 2 | 1 | 5 |
| **Codex** | 0 | 3 | 1 | 2 |
| **Gemini** | 2 | 2 | 1 | 3 |

## DeepScan Review

The results below show risks the DryRun Security DeepScan Agent found after all features were implemented by the agents, as compared to the initial Baseline DeepScan.




| Agent | Critical | High | Medium | Low | Total | Net Change |
|---|---|---|---|---|---|---|
| **Baseline DeepScan** | 0 | 8 | 0 | 1 | 9 | |
| **Claude** | 1 | 8 | 3 | 1 | 13 | +4 |
| **Codex** | 0 | 6 | 2 | 0 | 8 | -1 |
| **Gemini** | 1 | 5 | 4 | 1 | 11 | +2 |

### Discussion

**Codex** Codex finished with the fewest vulnerabilities, but one of the remaining gaps was a temporary token bypass that persisted in the final codebase.

**Claude** introduced the most vulnerabilities that remained unresolved in the final scan. Its work yielded a 2FA-disable bypass not seen in other agents' work.

**Gemini's** work kept the temporary token bypass issue throughout while also introducing OAuth CSRF and invite bypass issues.

Universally unresolved across all three agents, compared to the initial codebase's posture included lack of rate limiting, poor JWT secret management, and refresh token replay issues that were present

in some form in every final scan. These represent systemic risks that persisted through development. Ironically, rate limiting middleware was defined in every codebase but no agent ever wired it up.

## Game Development

The racing game was built with a design to mimic a simple arcade game: think an infinite scrolling top down driver with 2D sprites for obstacles, enemy cars, and power-ups. Then we designed features that required adding a data store, an API, and even a websocket service. Each feature was submitted as a pull request and scanned by DryRun Security, with the results shown below.

### Game Feature Development Pull Requests

| Pull Request | | Security Issues Found |
|---|---|---|
| **PR 1** | Add a high score board | ! Race condition (client-side trust)<br>! Auth bypass to edit and delete scores |
| **PR 2** | Create an upgrade store | ! Client-controlled score function<br>! IDOR for unlock mechanism |
| **PR 3** | Add a player login and save game system, including forgot password flows | ! XSS in email<br>! User enumeration<br>! Poor token management / revocation controls |
| **PR 4** | Create multiplayer functionality, including a lobby and join codes | ! No websocket authorizations<br>! Weak seed for room codes |
| **PR 5** | Support a “Custom Power Up” feature | ! TOCTOU on Power Up Unlock and Balance check<br>! XSS in SVG upload<br>! IDOR for uploaded Images |
| **PR 6** | “Productionalize” the application to prepare for a docker-based deployment | ! Auth Bypass (DELETE Scores) |

| Agent | PR 1 | PR 2 | PR 3 | PR 4 | PR 5 | PR 6 |
|---|---|---|---|---|---|---|
| **Claude** | 5 | 3 | 4 | 0 | 1 | 5 |
| **Codex** | 5 | 2 | 4 | 4 | 2 | 3 |
| **Gemini** | 4 | 4 | 6 | 0 | 2 | 5 |

Again, the pull requests were modeled by taking an MVP product idea and adding features to it, just as a developer might do in building out their project.

## DeepScan Review

The results below show risks the DryRun Security DeepScan Agent found in the final analysis of the game app, after all features were implemented by the agents.


| Agent | Critical | High | Medium | Low | Total | Net Change |
|---|---|---|---|---|---|---|
| ***Baseline DeepScan*** | 0 | 0 | 0 | 0 | 0 | |
| **Claude** | 0 | 4 | 3 | 1 | 8 | +8 |
| **Codex** | 0 | 3 | 1 | 2 | 6 | +6 |
| **Gemini** | 0 | 5 | 2 | 0 | 7 | +7 |

### Discussion

**Codex** finished with the cleanest final security posture, but its missing JWT revocation and stateless rate limiting represent meaningful gaps in the auth system it built.

**Gemini** introduced the most issues overall — PR 6 was the noisiest, and the final state reflects that some changes in feature code suppressed issues surfaced earlier. However, PR 4 being completely silent on

WebSocket issues is a detection gap–though it was found in the final DeepScan. Gemini finished with the most HIGH severity findings.

**Claude** was the weakest on access control issues — carrying an IDOR from PR 2 and an unauthenticated destructive endpoint from PR 1 all the way to the final version, the longest-lived unresolved findings of any agent.

All three shared the same failure: WebSocket authentication issues were present in every final codebase. It is the one finding that transcends agent differences — it reflects a structural pattern where multiplayer was bolted on without revisiting the security model.

PR 3 (Auth & Save Games) was the highest-risk task across all agents — it introduced the largest cluster of findings (JWT secrets, user enumeration, session management, client-side trust) and most of the HIGH findings that persisted through to the final analysis trace back to design choices made in PR 3.

# Why Agentic Code Creates a Different Risk Profile

Using a coding agent is not the same risk profile as using a copilot. Agents make architectural decisions across multiple tasks and often without a security reviewer in the loop. The failures in this study are not random bugs. They are a consistent pattern: the same security issues teams have been finding for more than a decade are being introduced again. Security features that do exist might not even be connected to the application or configured incorrectly. The way a prompt or design is structured can drastically alter the threat model or protections that are built by agentic coding systems downstream.

Pattern-based SAST was not built to find these issues. Regex scanners flag known-bad strings and function calls. They cannot tell you that middleware was defined but never mounted. They cannot trace how the authentication for your WebSocket connection aligns with the API layer. They will not catch a business logic flaw in how unlock costs are validated. These are judgment calls, not pattern matches.

DryRun Security Contextual Security Analysis is built for exactly this gap. It reads code the way a senior engineer reviews a PR: tracing data flows, checking trust boundaries, and reasoning about whether the authentication policies you expect to be in place actually apply everywhere they should. In the 2025 SAST Accuracy Report, DryRun identified 88% of seeded vulnerabilities across four application stacks, outperforming five leading SAST tools. The biggest gap was on logic-level findings, which is the exact category agentic coding introduces most reliably.

## What Teams Should Do

The answer is not to stop using coding agents. The answer is to stop assuming - or hoping - they are security-aware by default. Here is what teams need to practice as they pick up agentic tools for development:

1. **Scan every pull request, not just the final build.**

   Risk compounds feature by feature. Issues introduced early can create bigger problems later. PR-level scanning catches problems before they spread across the codebase.

2. **Review security during planning, not just coding.**

Many issues in this study started with design decisions. The SDLC begins with requirements and architecture. If those are weak, an agent will compound the problem quickly.

3. **Use contextual security analysis, not pattern matching.**

Many of the vulnerabilities we found were logic and authorization flaws. These are not easily caught by regex-based SAST. You need analysis that understands how your code actually works.

4. **Pair PR scanning with full-codebase analysis.**

PR scanning catches what changed. Full-codebase scans reveal what has accumulated across the SDLC. You need both.

5. **Prioritize the recurring issues now.**

Insecure JWT defaults and state management failures, failure to implement brute force protections and rate limiting, and non-revocable refresh tokens appeared across multiple agents and codebases. If your team is using coding agents today, start by checking for these now.

DryRun Security provides the security layer agentic development is missing. By analyzing code in context and reviewing every pull request as it evolves, teams can catch real application risk before it compounds across the codebase.

See what DryRun Security finds in your codebase: https://www.dryrun.security/get-a-demo