

5 Reasons security fixes **don't get merged**

Finding the vulnerability was never the hard part

Very few measure the step that actually changes risk: whether a fix reaches production.

As of June 2026, Snyk's own product data shows roughly six new vulnerabilities introduced for every one remediated, and agentic development keeps widening the gap. Six failure modes block the merge, and each one leaves a signature you can diagnose. [If you have not mapped where security actually runs in your pipeline, start there.](#)

1 The fix breaks the build

What is happening: The fix is right in security terms and wrong in application terms. Closing the vulnerability breaks a contract the application depends on, whether that is a version bump that changes a function signature, a sanitization wrapper that alters an output format a downstream service parses, or a patched library that drops a method the code still calls. CI catches the break, the PR goes red, and the fix returns to the queue rather than the branch.

How to tell this is you: Open your closed-but-unmerged security PRs. If they carry a failed pipeline rather than a review comment, the blocker is validation, not persuasion.

The number to check: Share of security fix PRs opened last quarter that closed with a failing pipeline and no human comment.

2 The upgrade path cascades

What is happening: One direct dependency bump can pull a chain of transitive ones with it, turning a three-line security diff into a lockfile change in the hundreds. Because review effort scales with the blast radius rather than with the vulnerability, the fix that is cheapest to generate becomes the most expensive to approve.

How to tell this is you: Compare the size of the lockfile change against the size of the source change across your last ten security PRs. A wide ratio is the cascade.

The number to check: Median count of transitive packages moved per security upgrade. Anything above your team's tolerance sits unmerged regardless of severity.

3 The fix comes from a CVE description alone

What is happening: Automated generation made fixes cheap to produce, and most of what produces them works from vulnerability metadata alone. A generator with no model of the application can only produce a fix for the general case, and the general case is rarely the one in front of it: this application has a custom wrapper, a framework-level sanitizer already sitting in the path, or a call pattern the generic patch does not account for. The reviewer reads a plausible diff that does not fit the codebase, rejects it, and trusts the next one a little less.

How to tell this is you: Read the rejection comments. "This is not how we do it here," and "we already handle this upstream" both point to a generator operating without an application context.

The number to check: Share of rejected fixes where the rejection cites application context rather than correctness.

4 The model that writes the fix also grades it

What is happening: Agentic remediation made self-grading the default arrangement rather than an edge case. When one model generates a fix, and the same model validates it, the validation inherits every assumption the generation made, so a fix built on a misread data flow gets graded by a grader that misreads it the same way. What reaches the reviewer is a confidence signal with nothing independent to back it, and reviewers learn to discount such signals.

How to tell this is you: Trace the validation path behind your automated fixes. When generation and validation resolve to the same model working from the same context, the grade only repeats what the fix already asserts about itself.

The number to check: Share of automated fixes that clear an independent deterministic check, such as a test suite or a scanner re-run, rather than a model's assessment of its own output.

5 The reviewer did not write the original code

What is happening: Code review has always had a fallback: ask the person who wrote it. That fallback disappears when an agent does. The reviewer inherits a diff against code that nobody on the team chose line by line, which matters because judging whether a fix preserves intent requires knowing the intent in the first place, and no record of it may exist.

How to tell this is you: Compare review time on security PRs touching human-authored code against those touching agent-generated code. A widening spread points here rather than at the quality of the fixes themselves.

The number to check: Median review time on security PRs touching agent-generated code, tracked month over month.

Find your bottleneck

The first two are properties of the fix as a change: it has to build, earn the review cycle, and stay inside the review budget. The last three are properties of the evidence surrounding it: the generator has to understand the application, an independent party has to verify the result, and the reviewer needs a basis for judgment that does not depend on the author. Reversing any of them means changing the fix or the evidence, not finding more.

Pull the six numbers for a single quarter and rank them. One or two will dominate, and they are rarely the two that a team expects. Work those first: merge rate governs everything downstream of detection, from backlog burndown to whether coverage expansion reduces risk or only adds findings.

In every case, the finding already existed, and in most cases, a fix did too. What was missing was a reason to merge it. **The bottleneck was never finding; it was merging the right fixes.**

Want the argument behind the numbers?

[Learn why better prioritization reorders the queue without shortening it, and what has to change for the rate to reverse.](#)