THE PROBLEM

# Two hundred warnings, and one that matters.

A release deadline, a tool nobody trusts, and no record of who decided what.

**Thursday afternoon.** The new release goes out tomorrow. The automated checking tool has produced two hundred warnings, and someone has to say which of them should stop the release.

Most tools give a warning a name, a severity and a file. They do not say whether anyone has looked at the code since, whether the warning still matches the code that will ship, or who decided it was real. So engineers learn to wave warnings away, and the one that matters looks exactly like the rest. If it is real, customer data can be read or changed.

Afterwards, nobody can show which warnings were checked, against which version of the code, or who accepted the risk of shipping. The decision lives in a chat thread and a ticket that just says “reviewed”.

What the release meeting needs is a short list of warnings that each point to an exact line, show how thoroughly they have been checked, and name the person who agreed. That is what Assure builds, and the next three pages follow Thursday through it.

WHAT ASSURE FOUND

# One line, pinpointed.

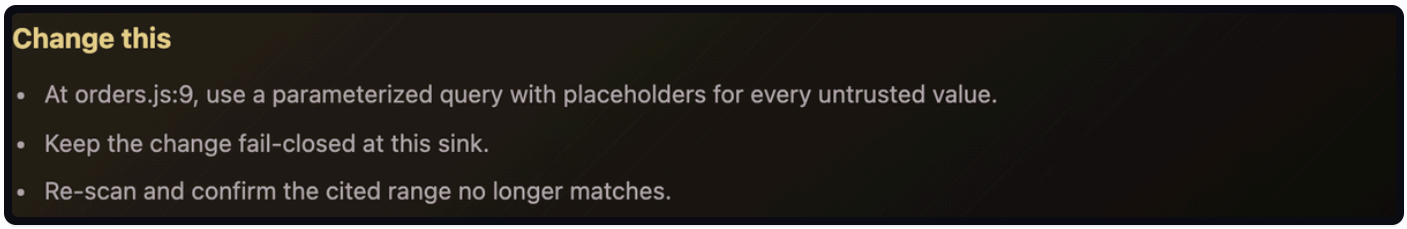
The warning, the explanation, and the change it asks for.

These screens come from Assure checking a small practice project with a flaw we planted, and a larger practice payments project that comes with it.




The exact line in the orders file, frozen as it was when checked.

Assure found the planted flaw on its own. On line 9 of the file that handles orders, whatever a visitor types as their customer number is pasted straight into a database command. An attacker could type something that changes the command, and read or alter other customers' data. Assure rated it critical and explained it in four plain steps: how an attacker would get there, what the line does, what happens if it ships, and what to change.



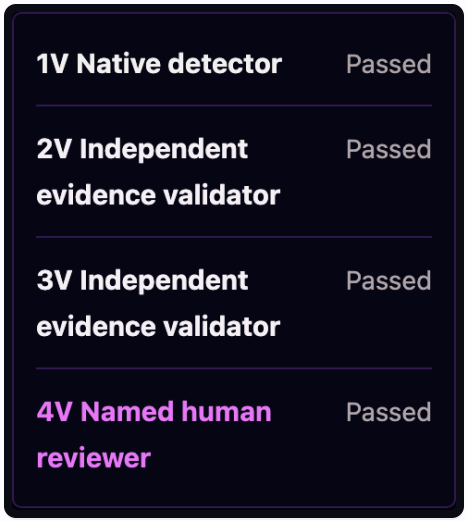

The fix, tied to that line.

The fix is specific: change that line so the visitor's input is kept separate from the command. Assure tells the engineer what to change, but it never edits your code itself. Afterwards Assure checks the same line again and reports whether the warning still matches. A quiet re-check is not proof the problem is fixed, and it is not a test of the live service.

HOW IT CHECKS ITSELF

# Three checks, and a name at the end.

Three automatic checks, a name that can still be missing, and what else Assure records about the code.

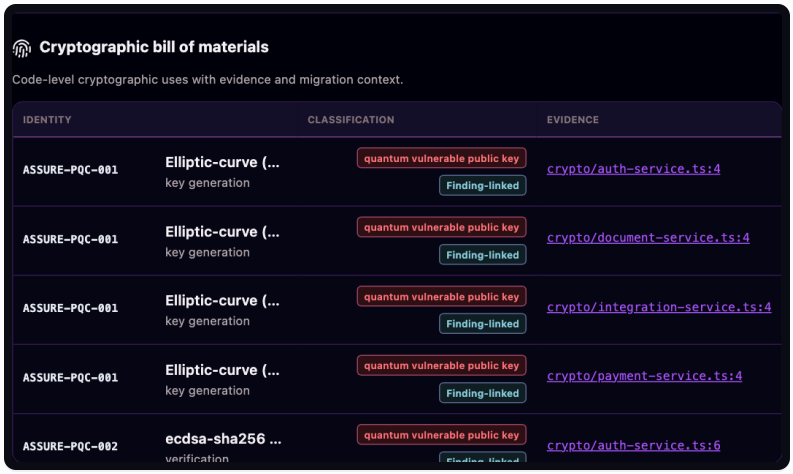



Each warning shows how far it has been checked.

**First,** an automatic check spotted the problem. **Second,** an independent check re-read the same line from a frozen copy of the code, to confirm it was really there. **Third,** another check tried to rule the warning out using Assure's own rules for spotting false alarms, and could not. These three are automatic and use no AI, so the same code always gives the same answer.

**Then a person** on your team can accept or reject it. Until someone does, the warning has no human sign-off. For this one, we recorded the reviewer's sign-off ourselves, with a written reason.

Agreeing a warning is real and fixing it are separate jobs: a warning can have a reviewer's sign-off and still have nobody assigned to change the code. And an empty list is not treated as good news: Assure records which kinds of checks ran and which did not, so nobody mistakes "nothing found" for "nothing wrong".




Places where the practice payments project uses encryption, each tied to a file and line.

Assure also lists every place the code uses encryption. In the practice payments project it found 20, such as creating keys and signing documents, and flagged each as a kind future quantum computers are expected to break. It also listed the 15 outside software packages the project relies on. Both lists help with audits and with planning which code to update first.

USING IT TODAY

# The same Thursday, with Assure in place.

What changes for the release meeting, where Assure stands, and how to try it safely.

Picture Thursday again, with Assure in place. The release meeting does not start from two hundred warnings. It starts from the ones that held up, like the orders flaw: the exact line, the three automatic checks it passed, the fix, and the name of the person who agreed it was real. The decision to ship or wait is made on that, and it is on record.

Assure is an engineering candidate, not a finished product. It reads software written in two programming languages, JavaScript and TypeScript, and has only been tried on practice projects. It is not yet approved for your source code, for installation at your site, or for production use.

So today, try it on practice projects that resemble your software, with flaws your engineers plant themselves. That shows what it finds and what it misses, without touching your own code. First, ask your engineers whether your software is written in JavaScript or TypeScript; if it is not, a trial will not tell you much about your own releases.

Assure reads code as it was when checked. It does not test the running system, so it cannot prove a live service is safe, and it does not fix code; your engineers do.

The hosted demonstration uses practice data and needs a sign-in: Google sign-in for approved email addresses, or an access code. The screens in this guide come from a copy of Assure running on a single computer, which creates its own access code when it is set up.