

Building Security In: The Secure SDLC and Test Coverage

Fri, Aug 21, 6:00 PM EDT · <https://scottslab.io/posts/secure-sdlc-and-test-coverage>



TL;DR

Bolt-on security fails because the expensive mistakes are architectural, and no amount of scanning at the end undoes a design decision. Threat modelling is the cheapest security activity there is, because it happens before anything is built. And coverage numbers lie: 100% line coverage only proves every line ran, not that anything was checked. Mutation testing is how you find out whether your tests would actually notice a bug.

"Bolt-on security doesn't work" is one of those phrases that gets repeated until it stops meaning anything. Here's the concrete version: the expensive mistakes are **architectural**, and a scanner at the end of the pipeline cannot undo a decision made in a design meeting six months earlier.

A tool can tell you a query is built by string concatenation. No tool will tell you that you built a system where the tenant identifier arrives from the client and is trusted. The first is a bug. The second is a design, and by the time it's running it's in your data model, your API contracts, and every integration anyone built on top.

Threat modelling: the cheapest security you will ever do

If security has to move earlier, the earliest useful activity is **threat modelling**: sitting down before the code exists and asking what an attacker would do.

The reason to bother is economics. At design time, changing the answer costs a conversation. After launch it costs a migration, a customer comms plan, and probably an incident.

STRIDE is the usual scaffolding: Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege, walked over each component and each boundary the data crosses. It isn't magic. Its value is that it forces the question *per trust boundary* rather than in general, and "in general" is where these conversations go to die.

You don't need a ceremony. Draw how data moves, mark where it crosses from something you control to something you don't, and ask the six questions at each crossing. An hour at a whiteboard is worth more than a year of scanning, because it is the only activity on this list that can change the *shape* of the thing.

The mitigations that keep working

Across design, development, testing and deployment, the same small set does most of the work:

Input validation, because most injection is untrusted input that got trusted. Validate on an allowlist at the trust boundary, and remember the boundary is your server. A client-side check is a usability feature, not a control.

Encrypting sensitive data, so a storage breach isn't automatically a data breach.

Least privilege, so a compromised component can't reach past its own job. This is the one that decides how bad an incident gets, and it's decided long before the incident.

Sandboxed development and test environments, so the messy, half-secure state of software under construction can't touch anything real, including production data copied into a test database "just for debugging."

None of these are clever. All of them are the difference between a contained incident and a headline.

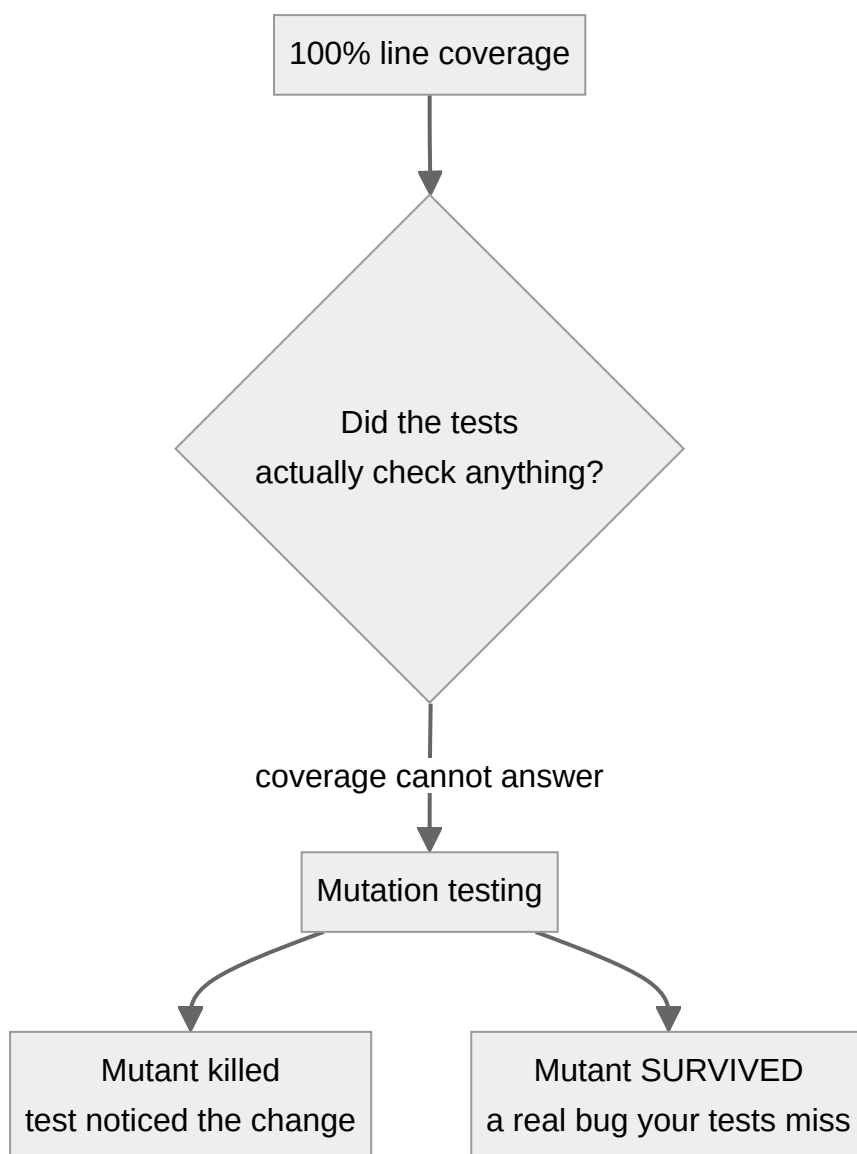
Coverage numbers lie, and here's exactly how

Test coverage analysis tells you how much of your code the tests actually exercised. The levels climb in granularity: **use case**, **function**, **line** (or statement), and **branch**. Branch coverage asks whether *both* outcomes of every decision were taken, which is why it's the honest one. Line coverage counts an `if` as covered when only the true path ever ran.

Now the part that matters. Coverage measures **execution**, not **verification**.

A test can call a function, run every line in it, assert nothing meaningful, and pass. Your report says 100%. Delete the function's entire body and the test may still pass. That number is not lying about what it measures. It's just measuring something much weaker than people believe it measures.

Mutation testing is the tool that closes the gap. It deliberately introduces small faults into your code: flip a comparison, change a boundary, remove a call. Then it re-runs the suite. If the tests still pass, that mutant "survived", and a surviving mutant is a precise, actionable statement: *there is a change to your code that your tests do not notice*. It's slower than coverage and infinitely more informative, and running it once on your most security-sensitive module is genuinely uncomfortable in a useful way.



Coverage numbers lie, and here's exactly how

The target isn't 100% of anything. It's knowing which paths nobody tested, because those are the paths where bugs live. It's also knowing which paths *are* tested but not actually checked, which is

the larger and better-hidden category.

Where the maturity models fit

If you want to measure the programme rather than the code, **OWASP SAMM** and **BSIMM** both exist for that, and they answer different questions. SAMM is prescriptive: here are practices, here's what maturity looks like, here's your gap. BSIMM is descriptive: here is what a large sample of real firms actually do, so you can see where you sit against your peers.

Use one when someone asks "are we doing enough?", because that question can't be answered by a scanner. Don't mistake either for security itself. A maturity score describes your process, not your attack surface.

The through-line is the same as everywhere else in this field: the tools find the bugs, and the design decides how bad the bugs get. Spend accordingly.

Written by Scott S. — Contact: scott@scottschlangen.com — scottslab.io — <https://scottslab.io/posts/secure-sdlc-and-test-coverage>