

SAST, DAST, IAST: Three Ways to Test Code, and Why You Need All of Them

Wed, Aug 19, 10:00 AM EDT · <https://scottslab.io/posts/sast-dast-iaast-code-testing>



TL;DR

SAST reads code without running it, DAST attacks it while it runs, IAST instruments it from the inside, and SCA inventories what you borrowed. Each is blind to what the others see, so they're a set, not a bracket. The part nobody tells you: SAST's real cost is false positives, DAST's is authentication and coverage, IAST's is that it only sees paths your tests reach. None of the four find broken access control, which has sat at the top of the OWASP Top 10 for years.

There are three main ways to test code for security flaws, and the mistake is treating them as competitors. They're a set. Each one is structurally blind to a class of bug the others catch, and knowing *which* blindness you're accepting is the entire skill.

Static: reads the code, never runs it

SAST parses your source (often into an abstract syntax tree, then a data-flow graph) and looks for a path from a *source* (untrusted input) to a *sink* (something dangerous: a SQL string, a shell exec, a file path). It needs no running application, so it can run on a feature branch before anything is deployed, cover code paths no test exercises, and point at the exact line.

Its cost is false positives, and they're not a rounding error. SAST can see that data flows from a request parameter into a query, but it often can't prove whether the sanitiser in between actually works, so it flags the path anyway. A tool that cries wolf on 200 lines gets muted, and a muted tool is worth nothing. The practical move is to tune ruthlessly on day one, fail the build only on high-confidence rules, and report the rest without blocking. A SAST rollout that blocks builds on everything gets switched off within a month. That's the failure mode, not the tool.

SAST is also blind to anything that only exists at runtime: your actual configuration, your reverse proxy, the environment variable that turns debug mode on in production.

Dynamic: attacks the running thing

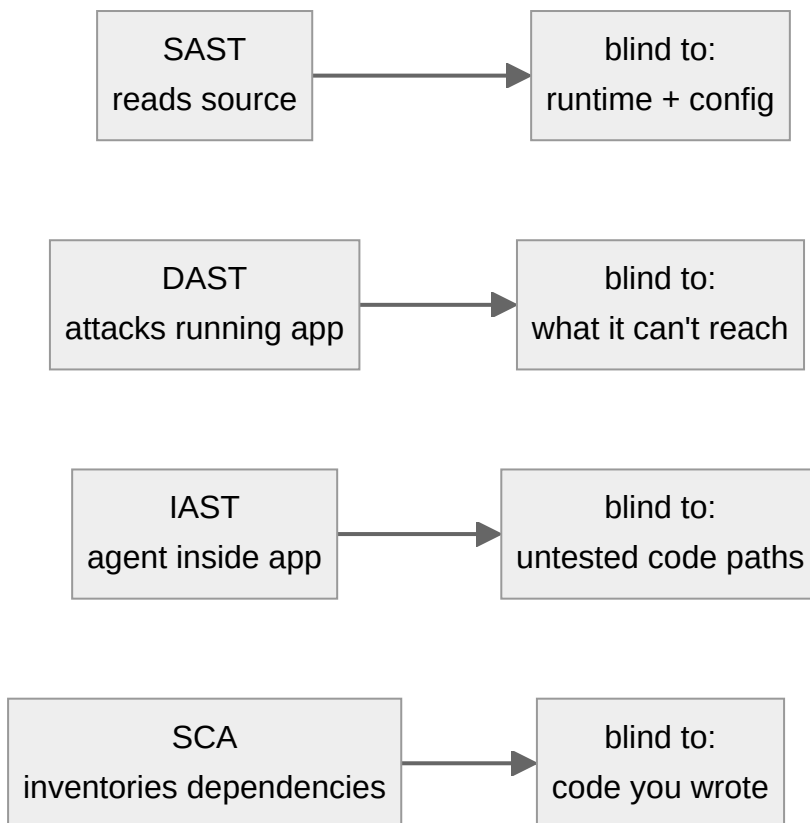
DAST runs the application, sends it input, and reads the output. It doesn't care what language you wrote it in, because it only sees what an attacker sees. That's exactly its value. It finds the misconfigured header, the verbose error page, the endpoint you forgot was exposed. A finding from DAST is almost always real, because it *did the thing*.

Its cost is coverage. DAST has to find your attack surface before it can test it, and two things break that badly: **authentication** and **client-side rendering**. If the scanner can't log in, it tests your login page and nothing else. So configuring authenticated scanning is the single highest-value thing you'll do with a DAST tool. And a single-page app that builds its UI in JavaScript is largely invisible to a crawler that only reads HTML; you point it at your API routes directly instead.

Interactive: watches from inside

IAST puts an agent inside the running application and watches data and control flow from within while the app is exercised. That's usually your existing functional tests or a DAST run. Because it sees both the request and the code path it triggered, it can say *this input reached this line*: a real vulnerability with the stack trace attached, and very few false positives.

Its cost is honest and often understated: **it only sees code your tests actually execute**. IAST inherits your test coverage. If 40% of your code is untested, IAST is blind to 40% of your code. It also needs runtime instrumentation, which is a real conversation with whoever owns production performance.



Interactive: watches from inside

Software Composition Analysis: what you borrowed

SCA is its own category, and it answers one question: *what open-source code am I shipping, and is any of it known-vulnerable?* It reads manifests and lockfiles, builds an inventory, and matches it against known CVEs.

That sounds mundane right up to the morning a critical vulnerability drops in a library everyone uses, and the only question that matters is "am I running it?" Teams with SCA answer in minutes. Teams without it spend three days grepping.

This is also where compliance arrived. **Executive Order 14028** made a Software Bill of Materials a condition of selling software to the US federal government, and two formats matter: **CycloneDX**, an OWASP project, security-focused with built-in VEX support; and **SPDX**, a Linux Foundation standard (ISO 5962) that reaches wider into licensing and provenance. If a customer asks for an SBOM, they mean one of those two.

The interesting recent development is **reachability analysis**. Rather than alerting on every vulnerable dependency, it traces whether your code actually calls the vulnerable function. Most flagged dependencies are never reached, so this cuts the noise dramatically, and it's the difference between a dependency report someone reads and one they close.

White box, black box, and the thing none of them find

Cutting across all of it is how much the tester can see. **White box** means full source access. **Black box** means none: the attacker's view. Neither is better; they answer different questions, and a mature program does both.

Here's the part worth internalising. Every tool above finds a bug by recognising a *pattern*: a tainted path, a bad header, a known CVE. That means none of them reliably find flaws where the code does exactly what it was told and the instructions were wrong: **broken access control**, business logic abuse, an endpoint that checks you're logged in but never checks the record belongs to you. There is no pattern to match. That class has sat at the top of the OWASP Top 10 for years precisely because scanners don't catch it.

So run all four, tune them so people still read the output, and then spend your human review time where the scanners are structurally blind. The tools tell you which of your walls have holes. They cannot tell you that you built a door where a wall belonged.