SNYK CHEATSHEET

snyk

12 best practices for developer-first static application security testing (SAST)

Static application security testing (SAST) plays a major role in securing the software development lifecycle (SDLC). Unlike dynamic application security testing (DAST), where you need the system running to interact with it, SAST works at the source code level prior to compiling. SAST can address issues at the earliest stages of development. Let's take a look at some best practices when implementing a developer-first SAST solution.

01 Use static code analysis

Static code analysis (SCA) finds issues early on in the SDLC, discovers problems you have no test cases for during dynamic testing, and improves readability and layout. Still, these tools are often slow, results are hard to understand, and riddled with false alarms. These drawbacks may be why recent surveys suggest that there is still a third of developers not using static analysis at all

02 Select robust SCA tools that fit your need

There is a variety of SCAs out there. When you research the specifics of a tool, here are some dimensions to look at:

- **Focus:** Some tools focus on security, others on performance or style.
- **Complexity:** Relatively simple tools like linters and higher complexity tools like semantic scanners each find different types of issues.
- **Developer friendliness:** Some tools can be easily incorporated into a developer workflow via IDE plugins, while other tools can take longer to provide results or require manual result import into developer tooling.
- **Accuracy:** SCAs can harm more than they help when the suggestions are riddled with false positives.
- **Runtime:** Some tools run in real time, some take hours to run over significant amounts of source code
- **Languages and frameworks covered:** Different SCAs work with different languages and libraries. Typically, libraries that make use of language features without any transpiler will benefit from static code analysis without specifically being supported. Run your libraries through your tools and check for yourself, or choose SCAs that cover as many languages and libraries as possible.
- **Actively developed and supported:** Finally, while the types of issues do not change frequently (take the OWASP Top Ten as an example), it makes sense to check for actively developed tools. And having support available is always preferred.

The mix of tools should reflect the needs of your project at hand. Remember, SCAs are not all the same.

03 Embed SCA across the SDLC

There are several positions in the developer workflow, where static program analysis can be included:

- **IDE:** Most support plugins for linters and Snyk Code provide plugins for JetBrain's IntelliJ and Microsoft's Visual Studio Code (coming soon to Snyk Code). This provides direct feedback right where and when you work on your code. Make sure these tools run in useful performance margin (not blocking the developer's machine, but still providing actionable feedback).
- **Compiler messages:** Make sure to review the compiler messages, preferably in the IDE directly
- **Before pull request:** Some IDEs (such as IntelliJ) provide a list of built-in static program checking tools that might not be suitable for a constant run during standard developer work as they take a bit longer to finish. But running them before a pull request or during a code review is advisable.
- **During pull requests/code review:** After a pull request is made, all the tool goodness from before should be rerun to inform the reviewer. Now's the right time to run tools that can take a bit longer. If you do so, make sure the reviewer knows what to expect. Some tools take extensive time to analyze pull requests. As a reviewer, you need to be aware to wait for the result.
- **Daily builds:** Again, rerun your tests. Remember that a

  daily build needs to complete within 6 to 8 hours (a.k.a. the night shift).
- **Before deployment:** Run a full test suite combining static and dynamic tests.

04 Test your SAST

From time to time, test your SAST and see that findings actually lead to a reaction within the CI/CD pipeline. It is common to find tools that are ineffective do to how long they take to run. So even though they run correctly, their findings are never used because the CI/CD pipeline has already moved on.

05 Keep your scanning engines up to date

Optimally use online services (SaaS) as those are maintained by the vendors. If you need to use in-house solutions, have a precise regiment for updates.

06 Synergize your analysis tools

Combine tools so that they can add value to each other (refer to “do” #2). Use the most generic and general tools early in the process, since These tools normally run faster and have an actionable output. Most of the time, it makes sense to use the more specific tools later in the process as those tools tend to be resource-hungry and have a longer runtime. In general, the sieve should be finer later in the process.

07 Scan first, manually test later

SCA is cheap in comparison to manual tests as it runs automatically, needs minimal to no additional work, and does not even need a running application. You should try to saturate the number of quality improvements available by static code analysis before starting manual tests.

08 Prioritize and fix findings

When starting SAST with a legacy project, the number of findings and false positives can be overwhelming. There are two ways of doing it:

- Prioritize and handle technical debt on a steady, workable stream.
- Handle it all at once. Make it the focus for a sprint and make it fun (bug hunting trophies for the team; pizza and party to celebrate victory).

This leaves the question of how to prioritize. Normally, your tools provide prioritization categorization which can be rough (severe to low impact) or based on a calculated index. Our suggestion is to use this as an input and build your own final prioritization index. Your index can include things like customer demand, team capabilities, waiting times, product strategy, and lots more that are important for you.

09 Don’t overload your developers

Fixing thousands of suggestions at once is a project in itself. Especially when introducing SAST on legacy code can lead to error fatigue and in the end, nothing is achieved.

10 Shift left, but not too much

Test-driven development (TDD) means that a developer grows the final code over various stages of obviously insufficient implementations. Static code analysis can actually hinder that phase when overstating the obvious. So, shift left on your quality but not too much.

11 Tools need to be actionable

It is one thing to point out a possible fault, but it is only the start of the remediation process. Developers need to understand the reasoning, background information, and ideas on how to remedy it. A brief error message with a file name means lots of work. It’s better if the tool shows an argumentation to follow using the real code, pointing to external resources, and even examples from open source projects on how to fix the issue.

Snyk Code offers built-in best practices

Snyk thought a lot about the list above and Snyk Code is our platform addition to address the list. Snyk Code provides static application security testing which is extremely fast. It fits into the developer workflow and can be used directly in your IDE. Snyk Code provides accurate suggestions based on a unique AI-based algorithm. The suggestions are actionable, easy to understand, and well-explained. Snyk Code is also easy to integrate into your CICD process by using the Snyk CLI. Just book a demo without any obligation and ask us any questions you have.

12 Build KPIs around fixes applied, not the number of open bugs

A good KPI for SAST is not to count the number of possible issues. It drives the behavior to collect a larger and larger number of open bugs, which overwhelms developers and actually leads to the opposite you want to achieve. Think about a KPI that actually counts the number of applied fixes and their severity.

[Learn About Snyk Code]

snyk