code intelligence

APPLICATION SECURITY TESTING REPORT 2020

CODE INTELLIGENCE

TABLE OF CONTENTS

INTRODUCTION

Current state of Application Security 3

NEED FOR SHIFT-LEFT TESTING

Rule of Ten 4

THE STATE OF SOFTWARE DEVELOPMENT LIFECYCLES

V-Model 5
Agile 6
DevOps 6
DevSecOps 7

EXISTING APPLICATION SECURITY TESTING APPROACHES

SAST 8
DAST 9
IAST 10
FAST 11
SCA 12
RASP 12

BUYING GUIDE FOR AST TOOLS

Basics Features 13
Integrations 13
Usability 14
Reporting 15

BEST PRACTICE

AST Landscape 2020 16

INTRODUCTION

The universe of software development is expanding increasingly with new approaches. Agile and DevOps are just two of them. Developers are working on different approaches on a variety of projects. The emphasis on software testing varies. Current developments show that software testing should be initiated as an integral part of the development process. The reality is that software bugs are still a massive problem.

> The top 606 bugs alone caused a total financial loss of USD 1.7 trillion through direct damage, hidden consequential costs, and fixed costs.

It is hard to imagine that security testing is still of secondary importance to many companies.

To shed some light on the subject, we help you to get an overview of the current state of Application Security (AppSeC). Application Security takes place in different phases of the Software Development Lifecycle (SDLC, DevSecOps). Software vendors usually rely on more than one Application Security Testing approach (SAST, DAST, IAST, ...), which has obvious advantages and disadvantages, discussed in more detail below. Since none of the approaches guarantees complete security, we have asked ourselves the following questions:

> What are the essential requirements to ensure an effective and reliable testing process?
>
> What could a better Application Security Testing solution look like?

RULE OF TEN

NEED FOR SHIFT-LEFT TESTING

Software bugs are far more relevant to costs than hardware errors. In fact, software failures account for more downtime costs than hardware failures by a factor of 3. Yet many organizations spend little effort and money to ensure software quality. Even for those companies that do extended testing to inspect their code, the effort is so complex that bugs are still inevitable. In fact, companies that do not spend the time and money upfront to correct bugs end up paying for it in downtime and corrective efforts after the application is released. In the worst case, it can cause a loss of customers or revenue.

The Rule of Ten states that the further a bug moves undiscovered into the late stages of a development process - or even to the end-customer - the higher the costs for eliminating it. The rule is well-founded by the results of several studies from the 1970s in Japan, the USA, and Great Britain, which dealt with the causes of product and quality defects. All these analyses delivered almost the same results: 70% of all product defects were caused through failure during the stages of planning, design, or preparation. Even though the studies focused on manufacturing processes, the consequences can be found in modern software development as well.

> If it takes 100€ to fix a defect at unit testing, it takes 1,000€ at system testing, 10,000€ at Acceptance Testing, and 100,000€ after release.

Organizations often use application security testing tools early in development to find and fix bugs and vulnerabilities. Currently, SAST (Static Application Security Testing) and SCA (Software Composition Analysis) tools are widely spread among development teams. But in recent months, the rise of FAST (Feedback-based / Fuzzing Application Security Testing) in particular has ensured that more and more development teams are finding bugs early in the development process.

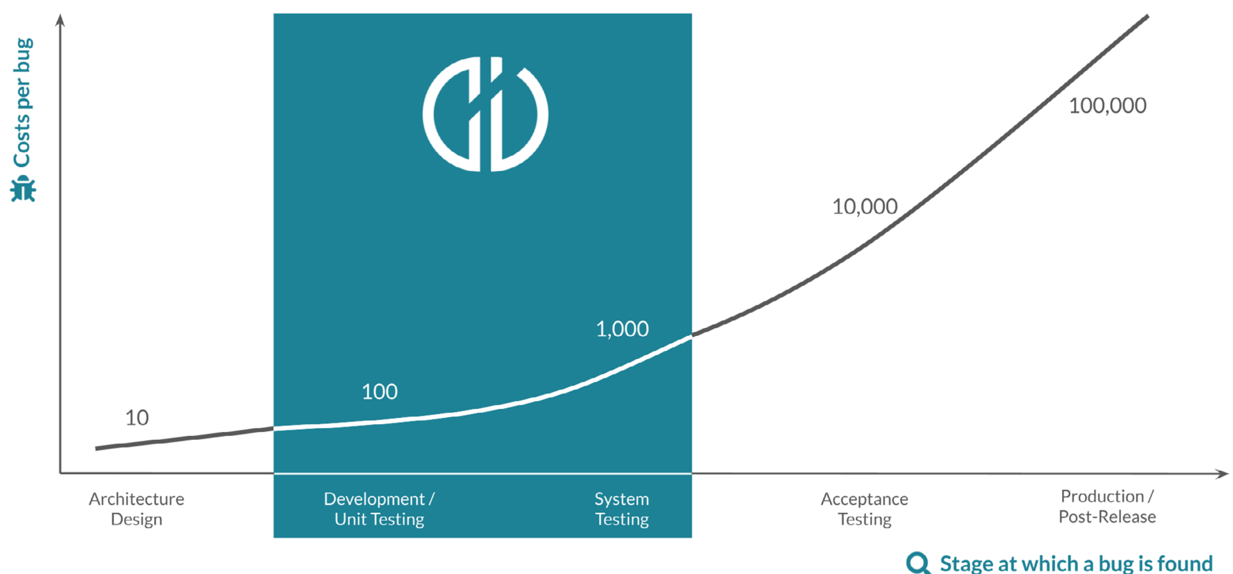

IMAGE: Rule of Ten

SOFTWARE DEVELOPMENT LIFE CYCLE

THE STATE OF SOFTWARE DEVELOPMENT LIFECYCLES

The Software Development Life Cycle (SDLC) is a process used to design, develop and test software with a high-quality standard. The SDLC aims to produce applications that meet customer expectations and reach completion within the estimated budget. It consists of a detailed plan describing how to develop, maintain, replace and alter or enhance the application.

The SDLC defines a methodology for improving the quality of software and the overall development process. In recent years, the methodology for the SDLC experienced a shift from the classical V-model to DevSecOps. The following section will highlight the advantages and disadvantages of the different SDLC methods and provide guidance for best practice.

V-MODEL

The V-model is an SDLC model where the execution of processes happens in a sequential manner in a V-shape. It is an extension of the waterfall model and also known as Verification or Validation model. For every single phase in the software development cycle, there is a directly associated testing phase on the other side of the "V". The model advocates a highly-disciplined process and only allows the team to start the next development stage after the completion of the previous one.

Unfortunately, the V-Model comes with several downsides for modern software development:

- High risk and uncertainty for development costs and time
- Difficult to go back and change a functionality after proceeding to the next stage
- Poor model for long and complex projects
- Software can not be run until the end of SDLC

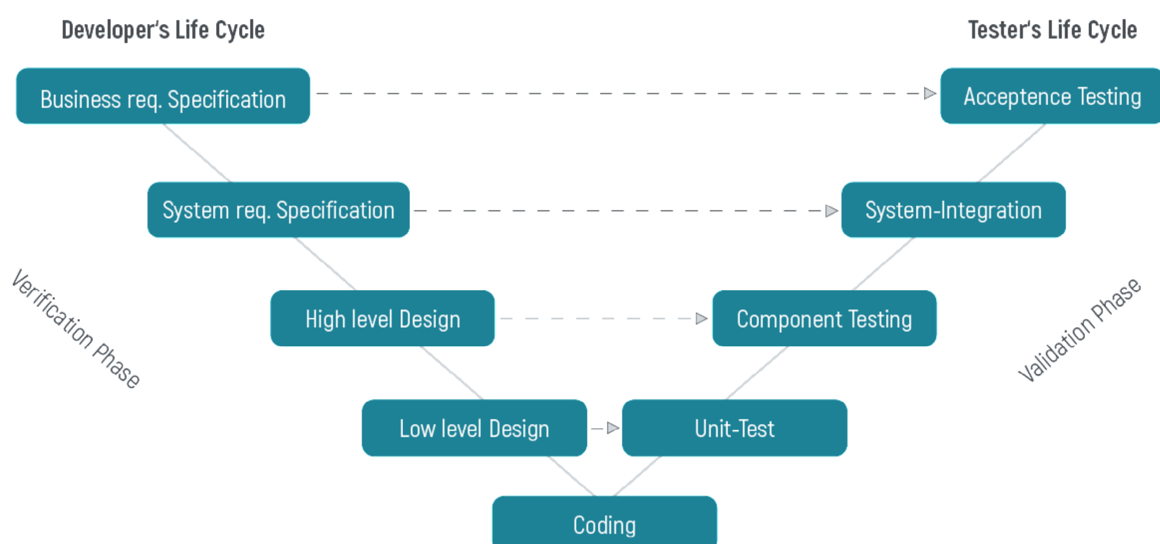

IMAGE: V-Model

SOFTWARE DEVELOPMENT LIFE CYCLE

THE STATE OF SOFTWARE DEVELOPMENT LIFECYCLES

AGILE

Within software development, Agile Software Development is defined as a set of methodologies that enables development teams to deliver results smarter and faster. Agile Software Development is based on an iterative approach, where requirements and solutions evolve through collaboration between self-organizing cross-functional teams. Its goal is to provide a high-grade management system to meet the organization's goals and the customer's needs. Kanban and Scrum are two of the most widely used Agile methodologies.

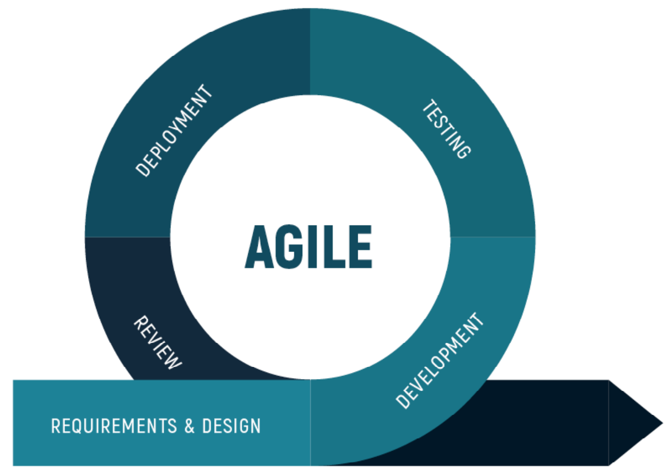

IMAGE:Agile

The benefits of Agile include a fast-responding approach to changes, faster review cycles, greater flexibility in releasing new features, and less up-front work for development teams. Nevertheless, Agile does not provide real answers to the challenges of modern application security but most solutions in the area of DevSecOps are fitting well into Agile Software Development.

DEVOPS

DevOps is a set of practices that combines technical and cultural aspects to help developers and IT operations teams to build, test, and release software faster and more efficiently. "In the DevOps ideal, developers receive fast, constant feedback on their work, which enables them to quickly and independently implement, integrate, and validate their code, and have the code deployed into the production environment.", according to The DevOps Handbook.

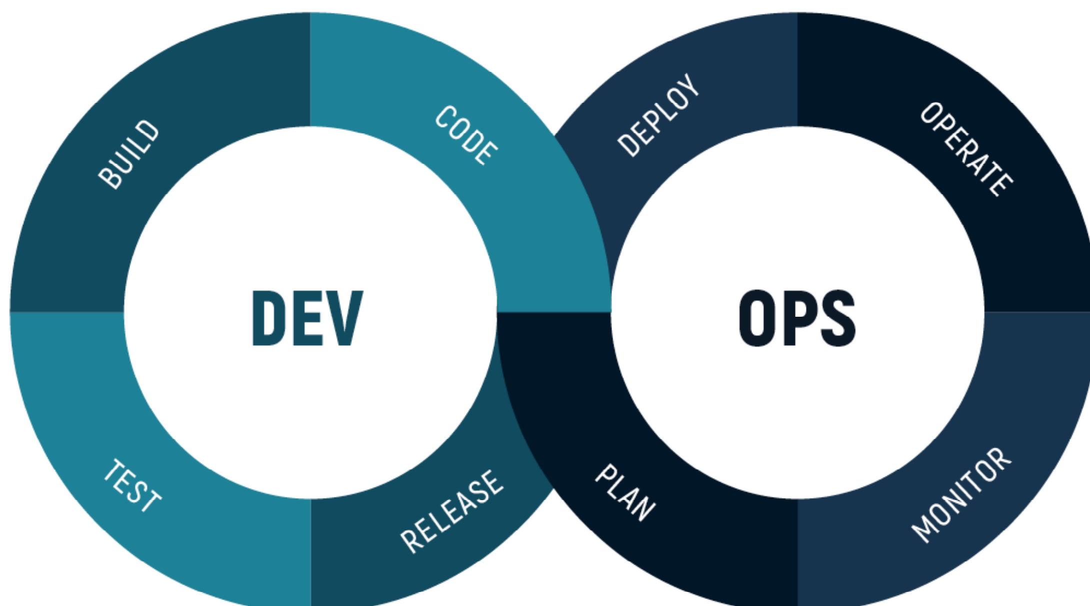

IMAGE:DevOps

THE STATE OF SOFTWARE DEVELOPMENT LIFECYCLES

DEVSECOPS

DevSecOps is about introducing security earlier in the software development life cycle (SDLC), thus minimizing vulnerabilities and bringing security closer to IT and business objectives. It adds a security perspective to the idea of DevOps and involves the integration of security testing technologies into continuous integration/delivery (CI/CD) workflows. The simple premise of it is that everyone in the SDLC is responsible for the security of the application. The main benefits of DevSecOps are the reduction of misadministration and mistakes from the start. Also the high degree of automation reduces the need for security architects to manually configure security consoles. DevSecOps should definitely be considered a "best practice" in 2020.

Ryan O'Leary, Chief Security Research Officer at White Hat, says the following about teams that implemented DevSecOps:

> "Our average customer takes 174 days to fix a vulnerability found when using dynamic analysis in production. However, our customers that have implemented DevSecOps do it in just 92 days. If we look at vulnerabilities found in development using static analysis, an average company takes 113 days, while the DevSecOps companies take just 51 days. [It's] a pretty drastic improvement, [...]. In addition, vulnerabilities that were found and fixed in just 10 days for an average customer were just 15 percent of the total number of vulnerabilities ultimately fixed. For DevSecOps companies, 53 percent of vulnerabilities found were fixed in just 10 days."

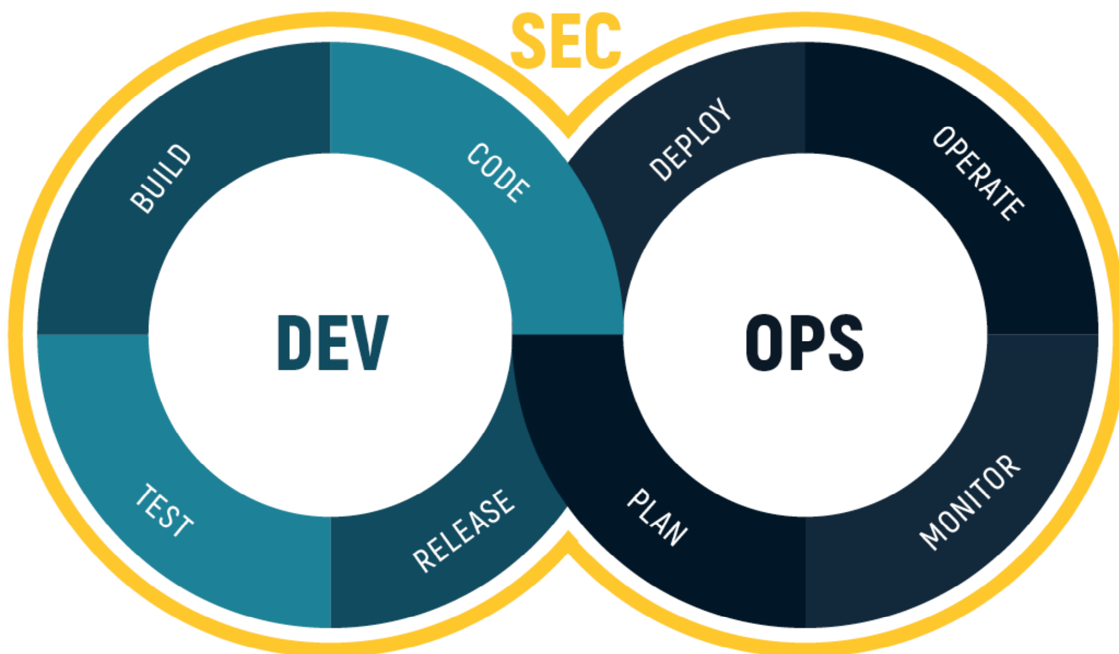

IMAGE:DevSecOps

EXISTING APPLICATION SECURITY TESTING APPROACHES

In today's software testing industry acronyms like SAST, DAST, or IAST are omnipresent, with FAST and SCA being the most recent trend in 2020. This section will first give you a short recap of the current application security testing approaches and discuss the strengths and weaknesses of the available approaches.

SAST

SAST, or Static Application Security Testing, has been around for many decades. In SAST, the analyzer scans the source code without actually executing it. The code is then traversed for suspect patterns using heuristics. Code fitting specific patterns, which could indicate potential vulnerabilities, are then presented to the user. Since SAST tools do not execute the code, they can be used at any stage of the software development process.

The fundamental disadvantage of SASTis that it produces large numbers of false positives (code that does not actually contain vulnerabilities). In practice, large projects can easily have hundreds of thousands of warnings and even in toy examples can produce thousands of warnings. This leads to tremendous usability issues and most developers and testers strongly reject these tools. A common coping strategy is to outsource the analysis of the warnings, thus defeating the purpose of running the tools in-house.

Many SAST companies now offer heuristics to reduce the number of false positives, however, since these heuristics are also based on static analysis they suffer from the same advantages and disadvantages and do not change the fundamental problem of SAST.

SAST strengths	SAST weaknesses
Offers high code coverage	Produces too many false positives so that usability is well below recommendable levels
Can be performed at the early stages of software development, since it does not require the application to be built completely	Misses security vulnerabilities and produces False Negatives / Positives
	Requires access to the source code ("white-box testing")
	Cannot discover runtime issues
	Not well suited to track issues where user input is involved
	Has difficulty with libraries and frameworks found in modern apps

EXISTING APPLICATION SECURITY TESTING APPROACHES

DAST

DAST, or Dynamic Application Security Testing, has also been known for several decades. Here, the analyzer searches for security vulnerabilities and weaknesses by executing the application. The software under test is executed using predefined or randomized inputs. If the behavior of the application differs from predefined correct responses or the program crashes, there is an error or bug in the application. The main advantage of dynamic testing is that there are virtually no false positives since real program behavior is analyzed, which makes the results a lot more useful to testers.

An interesting feature of DAST is that it also can be used on software for which the tester does not have the source code. In this case, DAST treats the application as a black box and only looks at in- and outputs. This feature has led many to incorrectly use the terms black-box testing and DAST interchangeably. Black-box testing is a subcategory of DAST. Another common misconception about DAST is that it is only used during the testing phase of development. While DAST does require that the program be executable, beyond that DAST can be used at any time during the software development lifecycle (SDLC), including during early development.

However, DAST also has some disadvantages. Since DAST executes the program with random inputs, it cannot guarantee code coverage and it has poorer runtime properties than SAST solutions. Black-box DAST solutions also have the disadvantage that there is nothing to guide the generation of random inputs making it very inefficient and under most conditions incapable of finding bugs buried deep within the code. It also requires manual effort to understand the stack traces produced by crashes and map them onto source code to fix the problems later. Some DAST solutions can address these problems, however, unlike the very simple black-box DAST solutions, they suffer from high complexity and require significant expertise to use.

DAST strengths	DAST weaknesses
Produces virtually no false positives	Requires working application to be tested
Can discover runtime issues	Needs special testing infrastructure and expertise
Can discover issues based on user interaction with the software	Often executed towards the end of the software development cycle, due to poor performance
Does not require access to the source code	Does not cover all code paths

EXISTING APPLICATION SECURITY TESTING APPROACHES

IAST

IAST, or Interactive Application Security Testing, is a marketing term and is often described as combining the benefits of SAST and DAST. Another feature claimed by IAST is that it is integrated into the SDLC and the CI/CD chain instead of only being used in the testing phase. This feature gives rise to the "I" in IAST.

For instance, Gartner defines IAST as follows:

> "Interactive application security testing (IAST) uses instrumentation that combines dynamic application security testing (DAST) and static analysis security testing (SAST) techniques to increase the accuracy of application security testing. Instrumentation allows DAST-like confirmation of exploit success and SAST-like coverage of the application code, and in some cases, allows security self-testing during general application testing. IAST can be run stand-alone, or as part of a larger AST suite, typically DAST."

There are several distinct ways this can be interpreted. Firstly, a DAST solution is used to test warnings produced by SAST tools to weed out the false positives. This would be very desirable but, to the best of our knowledge, no tool can actually do this at scale with any scientific rigor and thus we consider this to be snake oil. Alternatively, it can be interpreted as a DAST solution that utilizes the source code to improve performance, such as fuzzers that use instrumentation to improve code coverage.

These are highly successful tools but they all fall in the DAST category, since DAST is not restricted to black-box testing. The "interactivity" feature also is not excluded from DAST, since dynamic testing can be done as soon as the code is executable. So we see the term IAST mainly as a marketing term, which describes a sub-category of DAST to explicitly feature the fact that the DAST tool is integrated into the CI/CD pipeline. Cutting through the marketing hype, this is still an important distinction to make, since fixing bugs early in the SDLC is definitely a desirable goal.

However, current IAST solutions still have a major drawback: they either rely on the definition of good test cases triggering a high code coverage (passive) or rely on randomization as used in dumb fuzzing and well-defined patterns generated by the DAST engine. This was state of the art until the rise of Feedback-based Application Security Testing (FAST) in 2020.

EXISTING APPLICATION SECURITY TESTING APPROACHES

FAST

Feedback-based Application Security Testing (FAST), is also a subcategory of DAST and is currently being developed on the basis of feedback-based (also called coverage-guided) fuzzing techniques. Old DAST solutions and black-box fuzzing approaches have the fundamental drawback missing actual code coverage information when executing a given input. As a result, they rely on brute force, pattern-based, and random approaches to generate inputs in the hope of triggering crashes vulnerabilities. In other words, they are only able to find shallow bugs due to the limited code coverage they can achieve. But fuzzing has developed enormously in recent years, so it is not without reason that it is referred to as "modern fuzzing" in 2020.

Technology leaders such as Google and Microsoft already use modern fuzzing and technologies to automatically test their code for vulnerabilities.

> For example, with the help of oss-fuzz over 16,000 bugs have been discovered in Google Chrome and 11,000 bugs in 160 open-source projects.

> In 2019, fuzzing found more bugs at Google than any other technology.

This clearly illustrates the effectiveness of feedback-based fuzzing to uncover bugs and vulnerabilities.

State-of-the-art fuzzing techniques instrument the program being tested so that the fuzzer gets feedback about the code covered when executing each input. This feedback is then used by the mutation engine as a measure of the input quality. At the core of the mutation engine are genetic algorithms using code coverage as a fitness function. Generated inputs resulting in new code coverage survive and are used in the next iterations of mutations. The net effect of this process are inputs that maximize code coverage and thus increase the probability of triggering bugs.

Despite these enormous advancements, the full potential of FAST has barely been explored yet. Apart from the use of genetic algorithms to optimize code coverage, a wealth of other techniques can be used to significantly improve the effectiveness of DAST and current FAST fuzzers such as CI Fuzz.

FAST strengths	FAST weaknesses
Produces virtually no false positives	Requires a working application to be tested
Highly automated - Feedback mechanisms guide the input generators to maximize the code coverage anth thus find vulnerabilities with minimal human effort	Covers significantly more code than traditional SAST & DAST, but cannot guarantee full code coverage (as any other practical tool)
Automatically maximizes the code coverage	
More effective and efficient than traditional DAST / IAST and thuscan be integrated seamlessly into CI/CD	

EXISTING APPLICATION SECURITY TESTING APPROACHES

SCA

Software Composition Analysis (SCA) is similar to SAST, however, the main goal of SCA is to identify all open source components and dependencies in a codebase and to map that inventory to a list of current known vulnerabilities. Here, there are various possibilities to detect those components, starting with a static analysis of the source code (including build systems scripting) up to binary file scanning and dynamic linking of libraries at run time.

SCA strengths	SCA weaknesses
Can be performed at the early stages of software development, since it does not require the application to be built completely	Only checks for security issues in OSS components and dependencies not the proprietary code
Can give an overview over the open source components in use (including licensed)	Doesn't check for misusing APIs or usage of deprecated functions
Can give alerts for productive code if issues are found in existing components	

RASP

Runtime Application Self-Protection (RASP) works with instrumentation similar to IAST, however, in this case, the instrumentation is added to the production code. The goal is to detect and prevent actual attacks during run time.

RASP strengths	RASP weaknesses
Can be used in almost any development process	High performance impact and computing overhead in production code taking more resources and slower response time
No manual effort required once integrated into the deployment pipelines	Compromised reliability due to complexity: each false positive detection could lead to limited functionality on the customer's side rather than your own developers.
DAST / IAST / FAST tools could learn from the productive inputs and thus increase the code coverage in the CI process (not widely adopted though)	

BASICS FEATURES

AST tools have to meet some basic requirements to ensure an effective and reliable testing process:

- Easy-to-use for developers (without demanding security domain knowledge)
- Support for used programming languages and frameworks
- High degree of automation for deployment and analysis of source code
- High code coverage to ensure discovery of critical bugs deep in the source code
- Low false positives rate to prevent manual developer effort
- Integration into common CI/CD pipelines, bug tracker, cloud and container development tools

INTEGRATIONS

In order to stay competitive and achieve faster go-to-market, organizations are focusing on low-overhead AST tools, which involves a rise of software-as-a-service (SaaS) solutions. AST tools must seamlessly integrate with existing development and DevOps tools, which means that the tools must fit with the tools that developers are already using or will use in the future (e.g CI/CD pipelines, IDEs, container, etc.). If AST tools slow down the development lifecycle, teams will not use them.

Generate CI/CD Script

IMAGE: Easy Integration into CI/CD Pipelines (Tool: CI Fuzz)

USABILITY

Developers need AST tools that help them do their job without getting in their way or creating extra manual effort because otherwise, they will get frustrated and not use AST tools efficiently. To ensure a pleasant experience for the developers, it is crucial that no false positives are caused. FAST tools are particularly suitable for this, as they do not generate any false positives. In order to enable developers without security knowledge to detect vulnerabilities, it is important that the entire process runs as automated as possible. Furthermore, a good AST tool educates the developer to learn from his mistakes and thus avoid repetitive bugs.

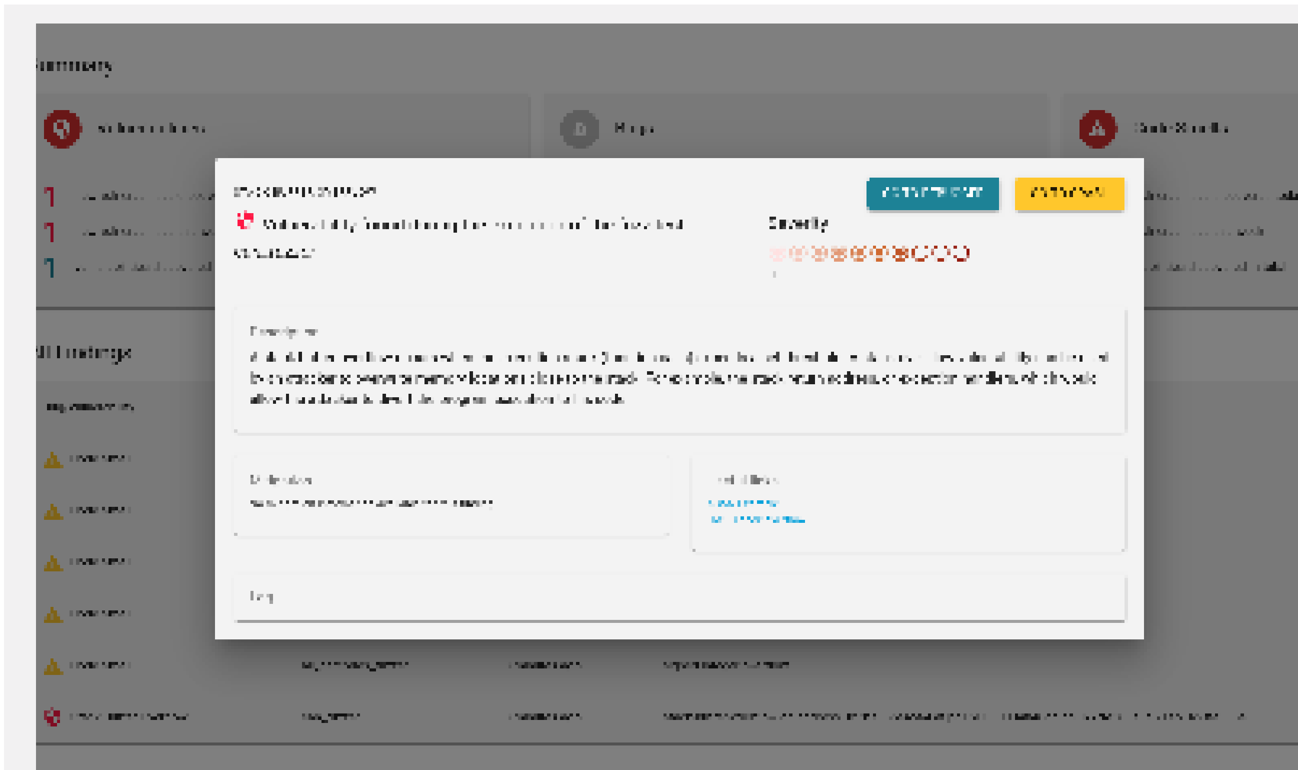

IMAGE: Developers View of a Finding incl. Vulnerability Description, Risk Evaluation and Education Links (Tool: CI Fuzz)

REPORTING

AST tools have to provide extensive and comprehensive reports for both managers and developers. These should not only include technical details like the reached code coverage or the severity of the found bugs but also executive-level dashboards which can help CISO's or product leads to make strategic decisions about upcoming security activities. For example, they must provide trend data and compliance information in relation to industry standards (e.g. OWASP Top 10, CWE Top 25). But most importantly the reporting must be customizable for the organizations' needs.

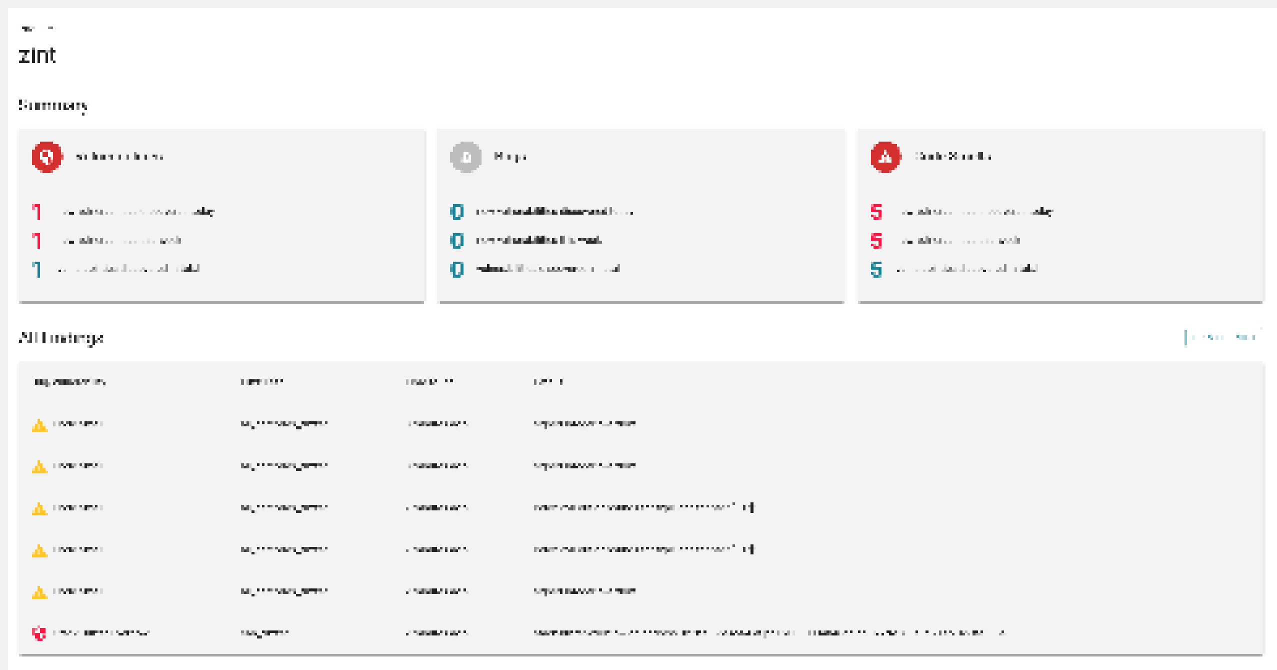

IMAGE: Customized Reporting Dashboard (Tool: CI Fuzz)

BEST PRACTICE

APPLICATION SECURITY TESTING LANDSCAPE 2020

The best practice Application Security Testing Landscape in 2020 is largely defined by modern fuzzing (FAST) technologies. FAST produces virtually no false positives, is highly automated, requires minimal manual effort, and can find bugs that stay uncovered by traditional DAST / IAST. More importantly from the development process perspective, it can be integrated seamlessly into CI/CD workflows with testing platforms like CI Fuzz. FAST is oriented on classic testing, can be performed on unit, system and acceptance testing levels, allows regression testing and is able to learn from existing test cases. This way, developers feel more familiar...

As with any other testing methodology, not all aspects of application security can be covered by FAST. Especially during the coding stage it makes sense to apply a combination of SAST and SCA. The advantage mainly lies in the ad-hoc feedback to the developer. During the development stage, often, the code is not entirely compilable/runnable, so the ability to do partial evaluation is very valuable. Finally, it still makes sense to perform a manual pentest before the actual release in order to complete the security testing process.

Another important point is the implementation of DevSecOps. DevSecOps enables developers to self-service infrastructure. Developers can easily configure networks and codify the infrastructure (infrastructure-as-code). This makes the process more reviewable and reproducible as well as faster than waiting for an infrastructure/admin team to set up stuff according to the developers requirements.

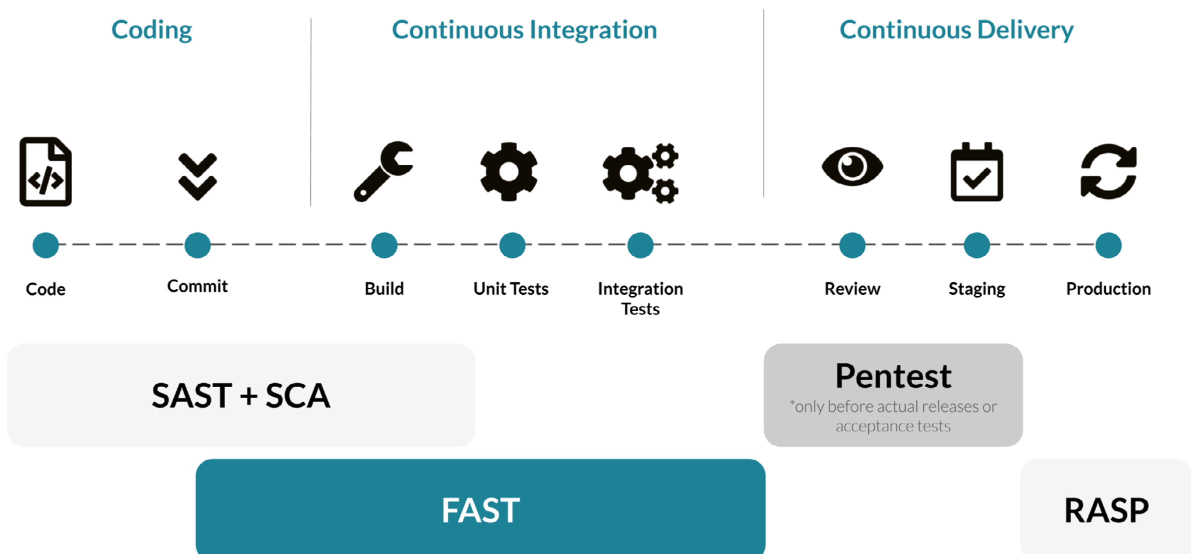

IMAGE:Best Practice AST Landscape 2020

code intelligence

Code Intelligence GmbH
Rheinwerkallee 6
D-53227 Bonn

Phone
+49 228 / 28695830

E-Mail
info@code-intelligence.de

Website
www.code-intelligence.com