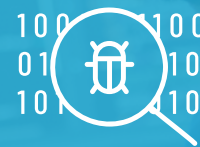




Open Source Cookbook



The Ultimate Guide to Software Composition Analysis

Software = Security

+ Introduction

Organizations who create software utilize a host of different technologies and solutions to help them ensure the code they create is void of potentially exploitable security vulnerabilities. Static, dynamic, and interactive application security testing (AST) solutions designed to scan custom and compiled code abound within many organizations today. These solutions are best for the situation at hand, but often prove insufficient when examining the open source code that finds its way into your custom software. Clearly, something else is needed.

Today, the average application is composed mostly of open source libraries and components; many analyst reports indicate these components make up more than 80% of the average codebase. As enterprises increase their adoption of open source – often to the degree of endorsement – we witness a fundamental shift in the software industry.

As a result, developers spend their time on key differentiators, proprietary features and functionality, and interoperability among custom and open source components that power the software itself, rather than recreating common software essentials from scratch. Such a transformation has accelerated development cycles and is forcing organizations to establish modernized processes to evaluate and secure both the software they create, and that which they consume.

Organizations who use open source software need solutions that are capable of detecting and identifying the open source or third-party components within their applications, and provide detailed risk metrics concerning open source vulnerabilities, potential license conflicts, and outdated libraries; industry influencers have deemed these “software composition analysis” solutions. When integrated into an organization’s CI/CD pipeline and SDLC, these solutions can enable development, security, and DevOps teams to prioritize and focus remediation efforts where they will be most effective and least costly, before potentially at-risk projects can be put into production.

+ Table of Contents

Introduction	2	SECTION 3: What is Software Composition Analysis (SCA)?	15
Why This eBook	4	- What's the Difference between SCA and SAST?	16
What You Will Learn	5	- Some Key Aspects of SCA	17
SECTION 1: Custom Code vs. Open Source Software	6	SECTION 4: Technical Deep Dive into SCA	18
- How Does Open Source Software Evolve?	7	- Open Source Detection Methodologies	21
- How Is Open Source Software Licensed?	8	Signature Scanning	22
- Is Open Source Software Vulnerable?	9	Package Managers	22
- An Attack Example	9	Snippet Scanning	23
SECTION 2: A Cookbook Analogy	11	- Component Identification	23
- Selecting Your Recipe	12	- Risk Metrics	24
Open Source Recipes Evolve Over Time	12	License Risks	24
Selecting Your Open Source Recipe	12	Section 5: Points to Consider When Purchasing an SCA Solution	26
- Understanding Your Ingredients	13	What You've Learned	28
- Prepping Your Kitchen	13	Conclusion	29
Kitchen Gadgets for Application Security Testing	13		
Keeping Your Software Kitchen up to Snuff	14		
Use Your Tools While Cooking, Not After	14		

+ Why This eBook

Open source software has facilitated the rapid evolution of application development and shortened development cycles. As with any new advancement in technology, there can be risks associated with open source components which organizations must identify, prioritize, and address. Security vulnerabilities can leave sensitive data exposed to a breach, complex license requirements can jeopardize your intellectual property, and outdated open source libraries can place unnecessary support and maintenance burdens on your development teams.

Today, organizations need deep insight into open source security vulnerabilities affecting their software, with risk severity metrics, detailed vulnerability descriptions, and remediation guidance to mitigate the risk of exploitation. This eBook is designed to help organizations, management teams, security practitioners, and developers understand Software Composition Analysis (SCA) in depth.

- + This eBook is designed to help organizations, management teams, security practitioners, and developers understand Software Composition Analysis (SCA) in depth.

+ What You Will Learn

This eBook begins with the differences between custom code and open source software. It then provides an interesting analogy for those who are new to SCA solutions to help them understand and effectively articulate the need. Then, it deeply discusses SCA and provides a technical deep-dive. Finally, it provides guidance and points to consider when evaluating and purchasing an SCA solution. After reading this eBook, and referring to it often, readers will obtain significant knowledge of approaches to effective SCA, best practices, and viable solutions for the modern software organization.

- + After reading this eBook, and referring to it often, readers will obtain significant knowledge of approaches to effective SCA, best practices, and viable solutions for the modern software organization.



SECTION 1

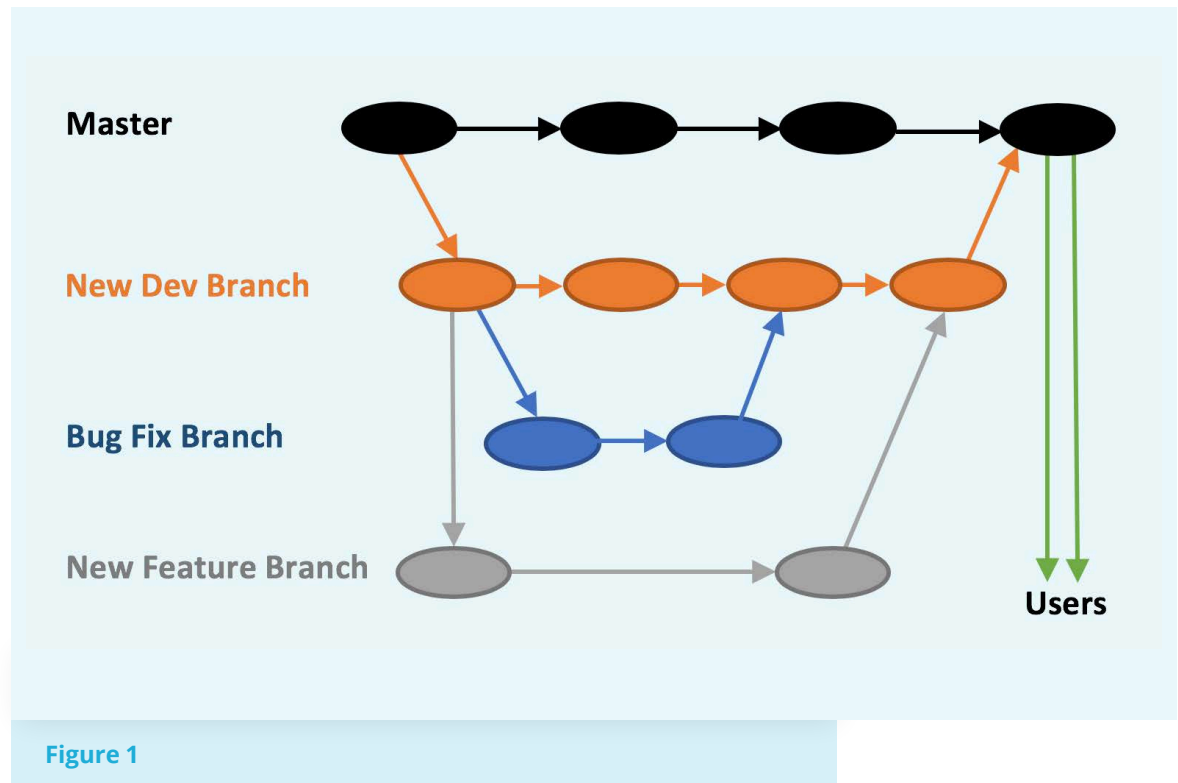
+ Custom Code vs. Open Source Software

Custom code is often referred to as proprietary code that is originally developed by a person or a team, and it's the intellectual property of an organization or that individual. This code is maintained by the creators or the owners of that code, so any innovation or enhancements to this code must be made by the person, the organization, or the company responsible. Any new versions or new releases are the responsibility of those creators, including any patches or updates required to fix any vulnerabilities. This code can be incorporated into other projects, or released individually as its own complete software.

Open source software, on the other hand, is created by developers, but often as a part of a community-driven project in which ideas and contributions are shared. This software is made available to the community as what are referred to as projects or components. This is where innovation happens organically throughout the community and in this instance, any updates, patches, and new releases are the responsibility of that community. The project or components evolve over time and each component can have licenses associated with it, which will detail any restrictions, permissions, or requirements that the project originator may choose to place upon it.

+ How Does Open Source Software Evolve?

An open source component or project begins from a certain point in time with what is called a master branch. This is the black line at the top of Figure 1. The open source community evolves and changes this master branch and the associated open source component over time. They do this by creating branches (e.g., New Dev Branch, Bug Fix Branch, New Feature Branch, etc.) off of that master branch to modify the code.



Usually, the purpose of this modification is to create new features or new functionality, to perform bug fixes, and to do various types of testing. New branches are then merged back into the master branch, as you can see by the arrows going back up into that main black line, and incorporated back into that main project. Or they may be maintained as a separate fork of the project overall. Usually, this is done when contributors or groups modifying the project intend to take it in a different direction than the one that the master branch was heading, or to suit another use case.

+ How Is Open Source Software Licensed?

Open source projects can have virtually any licensing structure. The author or originator of the component decides how that component should be licensed. They could leverage any one of thousands of existing licenses, or they could even make up their own. There are two main categories of open source licenses; copyleft licenses, often referred to as reciprocal licenses, and permissive licenses. In general, reciprocal licenses place restrictions or requirements on the distribution, attribution, or release of source code associated with the component, or the projects into which that component is incorporated. Permissive licenses generally place minimal requirements on software distribution and attribution.

As shown in Figure 2, some common examples of open source licenses include GPL 3.0, MIT License, or Apache 2.0. There are, however, also some examples of licenses that may be less common, but illustrate how an author can be free to create their own. The WTFPL License is completely open and the Beerware License, for example, requires that anyone who leverages a component with that license buys the author a beer.

Some Examples of Open Source Licenses

GPL 3.0

MIT License

Apache 2.0

WTFPL (Do what the F*** You Want to Public License)

Beerware License (Buy the Author a Beer)

Figure 2

+ Is Open Source Software Vulnerable?

To be extremely clear, not all open source software is vulnerable. Open source components are used everywhere and, as with custom code, there are situations in which open source components can be vulnerable. Usually this means understanding the difference between a vulnerable component and vulnerable versions of that component.

A component can contain vulnerabilities, but only in certain versions, and it's possible that newer versions may not contain the same vulnerabilities that previous versions did. However, some components may contain their own vulnerabilities that did not exist in previous versions, since a vulnerability may have just been introduced in the latest component version. It all depends on how that software is constructed and how it evolves over time.

With this in mind, a component version that may not have a vulnerability within it right now may, at some point in the future, have a vulnerability that is introduced. The past version was not vulnerable, but a newer version is. Furthermore, not every vulnerability may be exploitable, even if the component has a vulnerability. Security researchers, or even attackers, have to discover a vulnerability first, then develop an exploit to take advantage of it, in order to compromise the software. Even then, how that component is incorporated into the overall code of the application may determine whether or not a vulnerable condition can be met in order to execute the exploit.

+ An Attack Example

Concerning an attack that may be enabled by the usage of open source components, let's explore the timeline of an attack as shown in Figure 3 below.

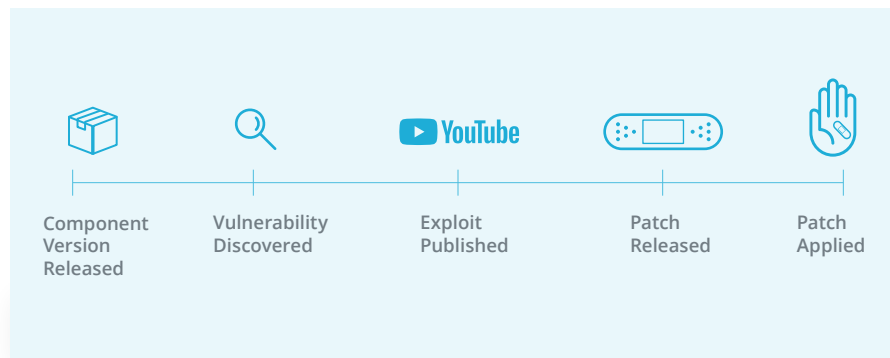


Figure 3

Imagine a timeline that represents a period of time during which an open source component exists. Maybe it's in your codebase, maybe it's not, but this component exists over time. At some point, a vulnerability is discovered. It could be the creators who discover the vulnerability, or maybe a security research team, or perhaps attackers. This vulnerability may get documented in a vulnerability database like the NVD, or those who discover it may keep the vulnerability secret for some time while people work on a patch. Regardless, some amount of time passes between when the vulnerability is discovered, and when the patch is released. Then, more time passes until that patch is applied, effectively mitigating that vulnerability.

It's possible that an exploit is discovered for this vulnerability and that exploit can be kept secret, but frequently it gets published among attacker communities or in public forums like YouTube. At that point, the world knows about the exploit and how it can be used to abuse that vulnerability. The time period between when an exploit is discovered and when the patch is applied is the window of opportunity for an attacker to infiltrate the application,

potentially compromising data, intellectual property, or simply impeding the application's performance. Clearly, the need to patch vulnerabilities quickly exists in nearly all cases.

Before moving on, let's look at an analogy that should help you understand the three primary concerns pertaining to the use of open source software.

- + It's possible that an exploit is discovered for this vulnerability and that exploit can be kept secret, but frequently it gets published among attacker communities or in public forums like YouTube.

SECTION 2

+ A Cookbook Analogy



When building applications that contain open source components, a parallel can be drawn to a software cookbook. Most meal ideas found in a cookbook have three components: a recipe, a list of ingredients, and a list of kitchen gadgets needed to get the meal just right. Some people follow the instructions step by step, while others will only follow the basics and do things a little more their own way. Regardless of how it's done, the main goal is to prepare something that is pleasantly consumable. With this in mind, let's explore the process of selecting your recipe first.

+ Selecting Your Recipe

It is often said that open source software is like a recipe. When cooking or baking, you are likely to use a combination of your own know-how, recipes you've borrowed from others, and some ready-made bits you purchase from elsewhere. That is the essence of modern application development; the code you write and the code (software) you purchase or license. Open source components are part of this equation.

Open Source Recipes Evolve Over Time

No recipe is perfect from the start. Changes to open source components can manifest themselves as new versions, and often are the result of development efforts in parallel to the current main project (referred to as branches or forks). As previously discussed, branches often include development activity for new features, bug fixes, and testing, and can be incorporated back into the main project. These branches and forks can be the result of collaborative evolution, or simply one contributor's ambitious activity. Regardless, with each new version of a component, we are slightly more removed from the origin of the component.

Selecting Your Open Source Recipe

As with cooking, when incorporating open source components into applications, it's important to understand the origin and evolution of what you're baking into your software. Carefully review your open source component versions, and evaluate the community's activity in order to have the greatest chance possible to predict the potential technical debt you may inherit.

While two components may share a name, they may differ greatly depending on the vendor that provides them. For example, while one vendor's take on an open source component (what the tech community refers to as a distribution, or distro) may be, fundamentally, the same as another's, they may have made some minor changes to suit their needs (i.e., compare Red Hat Enterprise Linux and Ubuntu, as they pertain to Linux). These changes can have implications on function or compatibility, and should be considered when selecting open source components for your applications.

Each distro experiences forks in which both major and minor changes occur. This can create a significant deviation, over time, among two different types of what had otherwise been considered the same component. It can introduce additional maintenance and development costs for you in the future, or expose you to unexpected security and compatibility issues if you don't fully understand the nuances of each. Beyond the concept of a software recipe, let's next look into the software ingredients that make up your applications.

+ Understanding Your Ingredients

Open source components can have security vulnerabilities, and some are more severe than others. It's important to understand if there are any vulnerabilities in the components you select, because when those components get baked into your applications, they can be potential open doors for attackers. Depending on the source or origin of that component, you may get notified when they're discovered, or you may not. In keeping with our analogies to recipes and food, you can consider these helpful notifications of risk similar to recalls on certain food items.

As an example, components from Red Hat or Apache may yield helpful alerts when vulnerabilities are discovered, or when patches are available to remedy such vulnerabilities. Components from community-driven development groups may not have such proactive alerting, making it your responsibility to identify and fix these risks, whether you have the support of the community or not.

It's important that you understand how you will identify the vulnerabilities that may be present in the components you select, how you will discover new risks as they emerge after you've baked them into your applications, and how you will get your developers the information they need to fix those issues. Now that we've briefly discussed software recipes and software ingredients, in order to get the meal just right, we'll focus our attention on your software "kitchen" where everything is assembled. Let's explore this concept next.

+ Prepping Your Kitchen

As with cooking, software development requires a well-equipped "kitchen" with tools, methods, and processes firmly established to create stable and secure software from an eclectic mix of custom code and open source components. Perhaps the "refrigerator" and "cabinetry" are your code repositories, which may be internal resources, commercial solutions you've licensed, or public resources like GitHub, for example. Maybe your "pots and pans" are package managers and build tools. Maybe your "oven" is your testing or staging environment. Perhaps your "utensils" are IDEs, and your "gadgets" might be your application security testing solutions. Regardless, your developers are making use of a rather complex software development environment, with many aspects which must be configured and maintained to produce "edible" software.

Kitchen Gadgets for Application Security Testing

The tools and gadgets your developers, security, and DevOps teams use are instrumental to the performance, stability, and security of the software your organization publishes. Because your software recipes use a mix of custom code and open source components, the application security testing gadgets you use need to be purpose-built to identify, triage, and remediate any issues in whichever type of software and code they examine.

The tools and gadgets your developers, security, and DevOps teams use are instrumental to the performance, stability, and security of the software your organization publishes.

For many organizations, their software kitchen is stocked with these essential applications security testing tools:

Static Application Security Testing (SAST)

Interactive Application Security Testing (IAST)

Software Composition Analysis (SCA)

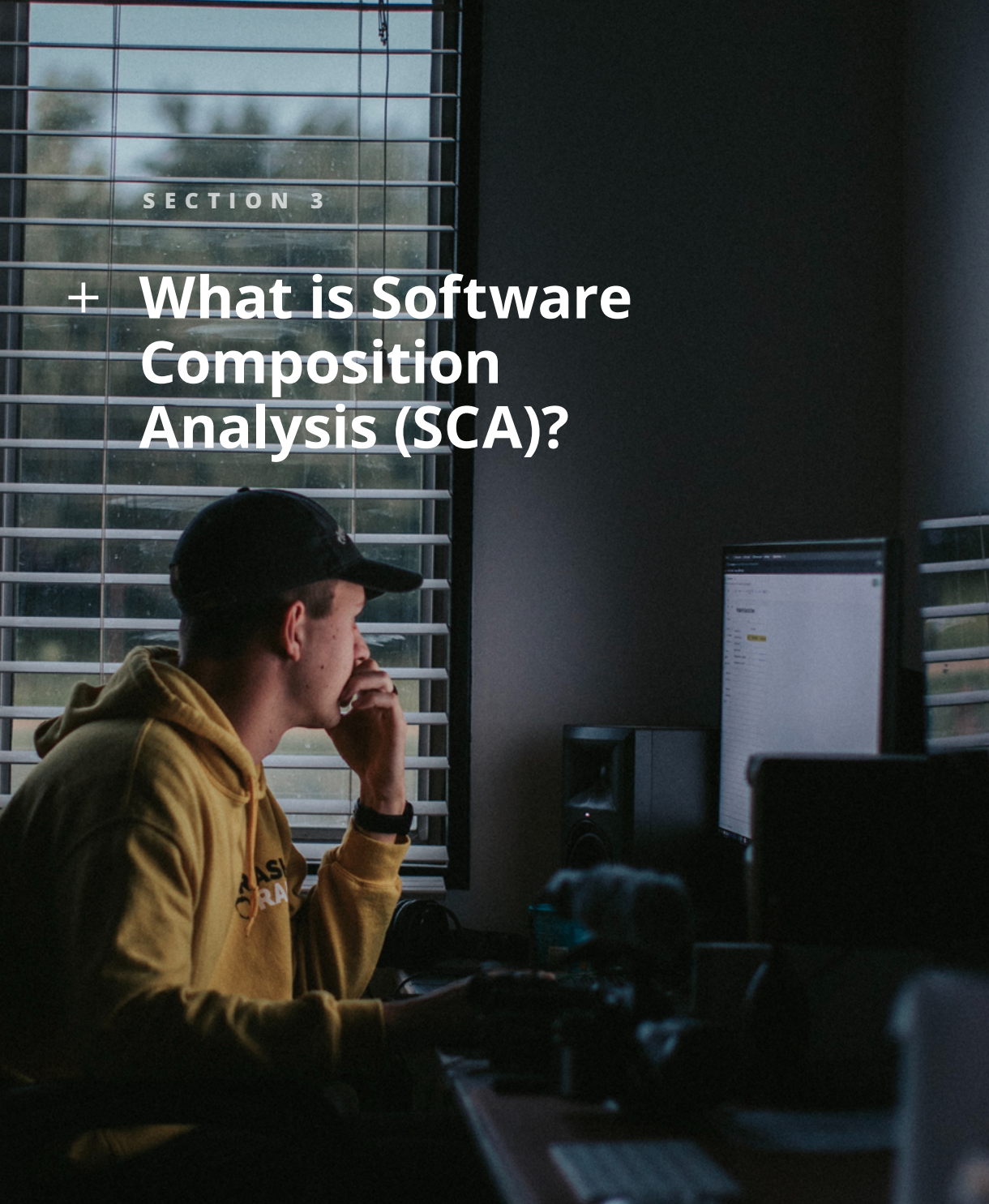
Keeping Your Software Kitchen up to Snuff

Organizations who create software are often subject to external and internal standards and requirements. It may be that your organization has committed to upholding SLAs to customers and internal stakeholders. It may be that your organization is subject to data protection requirements (e.g., E.U. GDPR, PIPEDA, etc.). In any case, it's important that you understand which standards and regulations your organization is subject to, and ensure you have the software security testing solutions in your arsenal to be able to support your efforts.

Use Your Tools While Cooking, Not After

Lastly, take a look at your software kitchen (your development environment, CI/CD pipeline, SDLC, and DevOps practices) and evaluate how you have integrated those necessary technologies along the way. You shouldn't wait until you've taken the steak off the grill and plated it to check the temperature. You shouldn't wait to add your egg/sugar mixture to your warming milk when making ice cream. And you shouldn't wait until the security testing phase to identify vulnerable open source components within your software. Instead, use the right gadgets (AST solutions) while you're cooking, not after the dish is complete.

The purpose of the aforementioned analogy was to help those who are new to the concept of open source components and how they're used in today's modern software development environments. If organizations utilize open source components, they must have a way of analyzing the composition of their software to ensure the components they're using are safe and licensed appropriately. In order to do that properly, software composition analysis (SCA) solutions are a critical resource. In the next section, we'll explore SCA at length.



SECTION 3

+ What is Software Composition Analysis (SCA)?

SCA is the market-defined term for analyzing software, discovering open source components and third-party libraries within it, and identifying the risks associated with them. SCA focuses on measuring two main types of risks. There's security risk (these are open source vulnerabilities) and license risk (these can be conflicts between open source licenses or could be non-compliance with the requirements outlined within those licenses). Sometimes, there may be a third, non-standard category of risk that explores community activity surrounding the component.

Having been around for over a decade, SCA originally focused on license compliance for software and embedded technologies like hardware and chipsets. But, with the growing popularity of open source, software security started becoming the biggest use case for SCA. Now, SCA is evolving to extend its influence across the AST portfolio, with some SCA solutions integrating and correlating data with SAST solutions to better-assess exploitability and examine if vulnerable components are actually called by the application. With its current trajectory, SCA is poised to mark the next great wave of secure software development.

+ What's the Difference between SCA and SAST?

SAST examines source code directly to look for weaknesses or vulnerabilities in the code that might be able to be exploited. Any vulnerabilities that are identified within the code are going to need to be taken out of the application. Therefore, developers will need to rewrite the pieces of source code to remove any vulnerabilities. As you might expect, this takes time and effort, and the SAST analysis itself can be a lengthy process, depending on the size of the codebase being analyzed.

With software composition analysis, because it's looking for open source components within code and not examining source code itself, it needs to be able to detect and identify any open source component versions within the software, match those identified versions against a database of vulnerabilities, and then, any vulnerabilities that are identified within that software are going to need to be patched or replaced. This means either changing the version of the component that is within the software to one that does not have vulnerabilities, or perhaps replacing the entire component itself.

+ Some Key Aspects of SCA

Any SCA solution must include some basic capabilities if it's going to have a chance at helping organizations achieve their goals for secure application development. It must be able to accurately detect and identify open source components and component versions in use within software. It must be able to provide insight into any vulnerabilities associated with those components and component versions, as well as any licenses that may apply to them. It must provide actionable risk insight and remediation guidance. It must allow organizations to configure and enforce policies against the analysis results. And, finally, it must be able to integrate with any tools that your organization is using in its SDLC or CI/CD pipelines, and deliver the aforementioned insight and results into the hands of the people who need it, in the manner in which it is most helpful to them.

Depending on the vendor, some SCA solutions might also include additional functionality as follows:

- Ability to identify if a vulnerable open source component version is actually exploitable
- Metrics associated with component bugs and community activity
- Correlation of analysis results with other application security testing solutions

Regardless of the approach/capabilities of the chosen SCA solution, most of them operate in similar ways. We'll explore further.

- + It must be able to provide insight into any vulnerabilities associated with those components and component versions, as well as any licenses that may apply to them.



SECTION 4

+ Technical Deep Dive into SCA

Software composition analysis happens in three major steps:



1. Detection

In order to identify open source and measure its risk, we must first detect its presence. Open source detection is the process of finding open source components within software and codebases. Some approaches to detection yield a high number of false positives and take a long time to accomplish, while others yield higher accuracy in a shorter amount of time, yet require slightly more up-front configuration.



2. Identification

Next, SCA has to identify the open source components it has detected; you have to know what there is before you can determine if what's there is safe. This usually requires a database of open source component information to reference for identification of the detected components. Usually, the information you get back during this process includes basic component version information. Sometimes, this data also includes information about where the component version came from, if it's a particular distribution of the open source component, or some other specific metadata.



3. Risk Metrics

Finally, you have to include risk metrics based on what you detected and identified in the first two steps. This is almost always security information and license data. Again, you need a database of information which you can reference your findings against. This time, that database is going to include vulnerability and license data rather than component metadata, and may include data exclusive to the solution vendor itself, backed by their research team (if they have one). Usually, security risk is ranked on severity as defined by standards like CVSS2.0 or CVSS3.0. Sometimes, SCA customers can even customize risk metrics on their own; of course, this has its own pros and cons once you start deviating from standardized metrics.



Open Source Detection

- The process of finding open source components within software or a codebase
- Can be accomplished in multiple ways



Component Identification

- Requires a database of open source component information to reference for identification
- Usually includes version
- Sometimes includes component or library source data



Risk Metrics

- Requires database(s) of security vulnerabilities and licenses to reference
- Usually ranked by standards
- Sometimes can be customized

Figure 4 summarizes the steps that must be taken, as discussed earlier.


Figure 4

+ 1. Open Source Detection Methodologies

How do SCA solutions detect open source components? It's important to understand that not all SCA solutions take the same approach to detection. Some solutions may perform what's known as signature scanning in which the SCA solution will scan the application and generate unique SHA1 hash signatures of the open source components it detects. You can think of these as unique fingerprints of the specific component versions in use. The SCA solution then tries to match these hash signatures against a database of previously scanned open source components and their corresponding "fingerprints."

Often, SCA solutions will look at the files that are used by package managers and build tools when compiling applications. In this case, it will determine the specific component version in use from the package manager declarations. You can think of this as examining what the developers say is in the software.

Lastly, there's dependency resolution in which an SCA solution examines the software following the build process to find any dependencies that were not declared, but which were brought into the application during the build process. Any dependencies that were not declared for the build process can obviously introduce unknown vulnerabilities into software.



+ The SCA solution tries to match the hash signatures against a database of previously scanned open source components and their corresponding "fingerprints."

Signature Scanning

The main benefit of signature scanning (sometimes known as file system scanning) is its ability to produce a large number of results, which many people consider to be the most complete or comprehensive representation of all the open source components within a codebase. This type of scanning can detect any non-declared components that may not have been included in the package manager files, or if software was built without the use of package managers. This methodology does have some downsides though. The scanning process can take a long time, consume a lot of compute power, and produces a large number of results that need to be reviewed. Often, these results include many false positives. When time is short and you're approaching your production deadlines, this methodology may cause you more headaches than it resolves.

Package Managers

When an SCA solution uses package managers to detect open source components, we see a few more benefits. These results tend to be highly accurate and produce false positives at very low rates. This tends to result in less noise or fewer junk results, obviously making it easier for organizations, developers, or security teams to review these results and prioritize their efforts. These scans tend to be a lot faster, and this methodology is more suitable for DevOps by way of its integration with the CI/CD tools that developers are using.

Package manager inspection, however, may not identify all open source within the analyzed codebase if components are not declared, or if the software is built without the use of package managers, as we often see in some legacy applications. This is why solution providers often pair this methodology with build dependency analysis. This is going to detect any non-declared dependencies that get incorporated during a build, or any dependencies of dependencies, which we consider transitive dependencies.

+ A quick way to think of the pros and cons of each detection method might be to ask yourself: "Do I need to find everything," "do I have time to review a lot of results," or "do I need to generate highly accurate results quickly?"

Snippet Scanning

Similar to signature scanning is snippet scanning, in which the SCA solution performs a signature scan not of entire open source components, but instead looking for smaller subsets of open source component code that match previously scanned and documented component segments within a database. This could be as long as just a few lines of code, looking at unique hash natures of those smaller code snippets. Snippet scanning can help identify any potential license requirements, license conflicts, or risk of noncompliance that are the result of a developer copying a small piece of code from a larger body of work. This is predicated on the results of a snippet scan being matched to an original open source component.

As you may expect, this process takes quite a long time and consumes a lot of compute resources. The results of snippet scanning can be very noisy, with a very long list of potential matches with a low certainty of exact matches and a high prevalence of false positives. And, of course, these snippet results are virtually worthless at identifying vulnerabilities since a vulnerability does not necessarily need to exist within a small snippet of code. This type of scanning usually only benefits license- oriented use cases.

+ 2. Component Identification

After we detect open source components using one or several of the methods that we just discussed, we must identify them. So how does component identification work? Often, component metadata is referenced against a database of open source components maintained by the solution vendor.

These databases contain information from various code repositories and sources like GitHub, Maven Central, and a wealth of others. If a component is found in this database that matches the detected component, its information is displayed in the SCA solution. This is where the risk of false positives is greatest, and this is where the detection methods that we just discussed have the greatest impact on the quality and actionability of the results.

+ 3. Risk Metrics

After we've identified our open source components, we need to generate some risk metrics associated with them – a necessary step to prioritizing where we focus our efforts to improve our risk posture. How does the process of generating open source risk metrics work? Well, first, identified component versions are checked against databases of vulnerabilities and licenses. Security and license risks are reported back to the SCA tool's analytics UI (user interface) and associated with the components that were analyzed in the codebase.

It's important to understand that risk metrics are not always standardized and the severity or priority associated with these risk metrics can vary by the SCA solution vendor. Security risk often has standardized scoring available for organizations to choose from; usually, this is CVSS2.0 or CVSS3.0 scoring. In this case, the user can adjust the sensitivity to risk based on various criteria associated with the project that's been scanned. As mentioned earlier, this is not a standard capability for all SCA solutions, and eliminates the ability to evaluate the risk profile of one project against the relative risk profile of another.

License Risks

Security risk metrics are among the most common criteria to impact any policy rules that organizations choose to put in place. License risks, however, are highly contextual and can vary depending on the deployment model of the application for example:

- An internal application that's on company servers and not for public or commercial use
- An external-facing application
- A commercial application

In addition, other components or licenses within the application may impact license risk. This is known as license conflict. The presence of both permissive and reciprocal licenses within the same application can tend to lead to some complicated results, as an example, and any royalties or attribution requirements placed upon them are also a concern. All of these things can determine the severity of your license risk.

+ It's important to understand that risk metrics are not always standardized and the severity or priority associated with these risk metrics can vary by the SCA solution vendor.

```

getById?(d.filter.ID-function(a){var b=a.replace(,aa);return mentsById
l.find.ID-function(a,b){if("undefined"!-typeof b.getElementsBy- function(a)
})):d.filter.ID-function(a){var b=a.replace(,aa);return Id&&p){var a
outeNode&&a.getAttributeNode("id");return c&&c.value==b}},d. function(
tElementsById&&p){var c,d,e,f=b.getElementsById(a);if(f){if find.ID funct
rn[f];e=b.getElementsByName(a),d=0;while(f=e[d++])if(c=f,(c=f.getAttrib
)}return[]}}),d.find.TAG=c.getElementsByTagName?function(a,b)getAttribute
e?b.getElementsByTagName(a)+c.qsa?b.querySelectorAll(a):void 1{return us
ByTagName(a);if("=""==a){while(c=f[e++])l==c.nodeType&&d, 0):function(
mentsByClassName&&function(a,b){if("undefined"!-typeof b.ge- push(};ret
lassname"

```

There is no standardized measure of license risk. But there is a common

expectation that licenses that cost money, restrict usage scenarios, or require sharing intellectual property from the associated codebase are all generally considered to be higher risk. License risk, as a result of software composition analysis, is usually most relevant to embedded devices or chipset manufacturers; you can think of this as relevant to industries like the Internet of Things (IoT), or tier-one and tier-two automotive industry suppliers, system integrators, and similar situations where it can be hard to access, replace, or update the software or the device on which the software sits. These tend to coincide with industries where potential loss of intellectual property due to license conflicts or non-compliance can be devastating.

When organizations consider purchasing a commercially-available SCA solution, there are several factors that must be measured. Let's explore some points to consider.



SECTION 5

+ Points to Consider When Purchasing an SCA Solution



Look for solutions that provide a comprehensive list of any publicly reported vulnerabilities in the components, accompanied by remediation guidance for those vulnerabilities. Give special consideration to solutions backed by a security research team which performs primary security research to find zero-day or non-public vulnerabilities, and enhance existing security records.

Take note of available integrations with package managers, build tools, code repositories, issue management solutions, and so on. These are not only the mechanisms which help generate accurate and impactful results, but they are those which enable direct and timely proliferation of critical insight to the people and teams who need it to minimize risk. This is critical in today's agile, DevOps, CI/CD landscape.

Ensure the solutions you explore can support your internal and external compliance requirements by way of policy controls. Perhaps your security team has unique standards for projects depending on their attributes. Your legal teams may have established rules to protect valuable intellectual property. Your engineering teams may have preferred software libraries that ensure compatibility and functionality across your software portfolio. Whatever the case, the SCA solution you select should be a key supporter of these stakeholders' needs.

If you can, focus on solutions with higher accuracy and fewer false positives, often by way of the detection methodology. You and your security, DevOps, and development teams have precious little time to parse through lengthy catalogs of potential risks to arrive at a pared-down list of true issues. Comprehensive results can be good, but only if you have the time to review and verify them.

Give priority to solutions which enable cross-product synergy, such as leveraging SAST to verify that an open source component identified as containing a vulnerability is actually being called by the application's source code. Capabilities like this will help prioritize your remediation efforts and enhance the accuracy and actionability of the analysis itself.

Consider solutions which are part of a complete AST portfolio, or which complement those you're currently using. Give special attention to solutions which allow unified user management, project creation, and scan initiation capabilities for multiple testing technologies, as these will yield the greatest efficiencies and reduce total cost of ownership.



+ What You've Learned

We began this eBook with a brief discussion about custom code vs. open source software in light of component evolution, licenses, and vulnerabilities. Then, we moved on to a cookbook analogy that should have helped you better understand the caveats of open source software usage. We next dove deeply into Software Composition Analysis (SCA), how it compares to SAST solutions, key aspects of SCA, and the various detection methodologies and approaches. Finally, we covered risk metrics and ended with points to consider when purchasing an SCA solution.

+ Conclusion

The increasing reliance on open source software will not likely change any time soon. More developers and organizations will continue to use open source simply because it makes the most sense for reasons we've discussed. As a result, organizations must add SCA to their software kitchen (so to speak) to complement the AST tools that are most likely already in use. The real key is to select an SCA solution that can be fully integrated with your software development tools, supports

internal and external standards for risk tolerance and compliance, and gets detailed insight into the hands of people who need it.

Many security experts expect to see an uptick in cybercriminals exploiting vulnerable open source libraries to gain access to sensitive and valuable data, largely due to the prevalence and accessibility of open source components and the (historically) inadequate documentation, evaluation, and monitoring of the risks they contain. Clearly, SCA solutions are needed now and will be required well into the future.