



Software Composition Analysis

The ultimate guide to SCA,
from Checkmarx



Open source code can take you anywhere. Travel safely

Enter →

```
disabled==a||b.disabled!=!a&&ea(b)==a:b.disabled==  
return ia(function(b){return b+=b,ia(function(c,d){var e  
[e]=(d[e]==c[e]))}))}function qa(a){return a&&"undefined"  
||{};f=ga.isXML=function(a){var b=a&&a.ownerDocument  
?a:ga.setDocument=function(a){var b,e,g=a?a.ownerDoc  
ument?(n=g,o=n.documentElement,p=!f(n),v!==n&&(e=n.default  
Listener("unload",da,!1):e.attachEvent&&e.attachEvent("on  
load",i))||a.getAttribute("className"))},c.getElementsByTa  
Comment("")||a,a.getElementsByTagName("*").length},c.get  
ElementsByTagName=function(a){return o.appendChild(a).id=u,  
g[h]),c.getById?d.filter.ID=function(a){var b=a.replace(  
0,0);var c,d=0,e=a.parentNode,f=b.parentNode,g=[a],h=[b];  
if(d.find.ID=function(a,b){if("undefined"!=typeof b.getEl  
emnts[])):d.filter.ID=function(a){var b=a.replace(_,aa);re  
AttributeNode&&a.getAttributeNode("id");return c&&c.va  
geElementById&&p){var c,d,e,f=b.getElementById(a);if(c  
ns)?function(a,b){var c=9==a.nodeType?a.documentElementE  
turn[f];e=b.getElementsByTagName(a),d=0;while(f=e[d++])if(c  
[f])return[]}),d.find.TAG=c.getElementsByTagName?funct  
TagName?b.getElementsByTagName(a):c.qsa?b.querySelectorA  
type||!(c.contains?c.contains(d):a.compareDocumentPosition  
ElementsByTagName(a);if("*")==a){while(c=f[e++])1==c.nodeType  
ja(function(a){c.disconnectedMatch=s.call(a, * ),s.cal  
a.compareDocumentPosition(b):1,1&d||!c.sortDetached&&b  
ElementsByClassName&&function(a,b){if("undefined"!=typeof  
ElementsByClassName(a),r[],q[],(c.qsa=Y.test(n.querySelectorAll  
"),length&q.push(":enabled",":disabled"),a.querySelectorAll  
||if(b)while(b=b.parentNode)if(b==a)return!0;return!1}  
"=<a id='"+u+"></a><select id='"+u+"'-r\\' msallowcap  
p(q.join("|")),r=r.length&&new RegExp(r.join("|"),b=Y  
querySelectorAll(["msallowcapture^='']").length&q.push(  
ed]).Length||q.push("\\\\["+K+"*(?:value|"+J+"))),a.query  
querySelectorAll(":checked").Length||q.push(":checked"  
[+-] )},ja(function(a){a.innerHTML= <a href=' disabled  
>";var b=n.createElement("input");b.setAttribute  
a.compareDocumentPosition!=b.compareDocumentPosition;re  
Comment("")||a.getElementsByTagName("*").length},c.get  
t==v&&t(v,a)?-1:b==n||b.ownerDocument==v&&t(v,b)?1:  
0;c.getById ja(function(a){return o.appendChild(a).id=u,  
,"D"),a.querySelectorAll("[name=d]").length&q.push("i  
gh))c.getById?d.filter.ID=function(a){var b=a.replace(  
ed).length&q.push(":enabled","disabled"),o.appendChild  
0;var c,d=0,e=a.parentNode,f=b.parentNode,g=[a],h=[b];  
y.test(s=o.matches||o.webkitMatchesSelector||o.mozMatches  
s);d.find.ID=function(a,b){if("undefined"!=typeof b.getEl  
emnts[])):d.filter.ID=function(a){var b=a.replace(_,aa);re  
AttributeNode&&a.getAttributeNode("id");return c&&c.va  
geElementById&&p){var c,d,e,f=b.getElementById(a);if(c  
ns)?function(a,b){var c=9==a.nodeType?a.documentElementE  
turn[f];e=b.getElementsByTagName(a),d=0;while(f=e[d++])if(c  
[f])return[]}),d.find.TAG=c.getElementsByTagName?funct  
TagName?b.getElementsByTagName(a):c.qsa?b.querySelectorA  
type||!(c.contains?c.contains(d):a.compareDocumentPosition
```

To detect potentially exploitable security vulnerabilities, organizations that create software tend to use solutions such as static, dynamic, and interactive application security testing (AST), to scan their custom and compiled code.

While such solutions are effective at what they are designed for (scanning proprietary code), they are simply not designed to examine the open source code that finds its way into your custom software.

You need something else.
Software composition analysis.

+ Contents

Introduction	4	Section 3: A technical deep dive into SCA	18
Section 1: Understanding open source software	5	- Open source detection methodologies	20
- Open source code evolves over time	7	- Signature (or file system) scanning	21
- The impact of open source code evolution	8	- Package manager inspection	21
- Open source code vulnerability	9	- Build dependency analysis	22
- Example of an attack timeline	10	- Snippet scanning	22
- Dealing with vulnerability alerts	12	- Component identification	23
- Working safely with open source components	13	- Risk metrics	23
- Managing licenses, compliance and regulatory requirements	13	- License risks	24
- Application testing	14	Section 4: What to consider when choosing an SCA solution	25
Section 2: Understanding software composition analysis (SCA)	15	Conclusion	27
- Key aspects of SCA	17	Further Reading	28

+ Introduction

Software composition analysis (SCA) is the detection and identification of open source or third-party components within an application; and the provision of detailed risk metrics on the vulnerabilities, potential license conflicts, and outdated libraries that relate to these elements.

Open source software has facilitated the rapid evolution of application development, and shortened development cycles. Its use is commonplace: many analysts report that open source makes up over 80% of the average codebase.

However, there can be risks associated with using open source components that organizations must identify, prioritize, and address:



Security vulnerabilities can leave sensitive data exposed to a breach



Complex license requirements can jeopardize your intellectual property



Outdated open source libraries can place unnecessary support and maintenance burdens on your development teams

Organizations therefore need insight into open source security vulnerabilities within their software, including risk severity metrics, detailed descriptions, and remediation guidance, and that's what software composition analysis solutions should deliver.

+ Understanding open source software



A screenshot of a browser's developer tools, specifically the Elements tab. The interface shows a large, complex DOM tree with many nodes highlighted in various colors (blue, red, green) to indicate different selection states or scopes. The tree is highly detailed, showing numerous nested elements like `<div>`, ``, and ``. Each node displays its tag name, class names, ID, and some internal state information. A tooltip is visible over one of the nodes, providing specific details about its attributes or properties. The overall appearance is that of a standard browser developer environment used for inspecting and modifying web page structures.

+ Understanding open source software



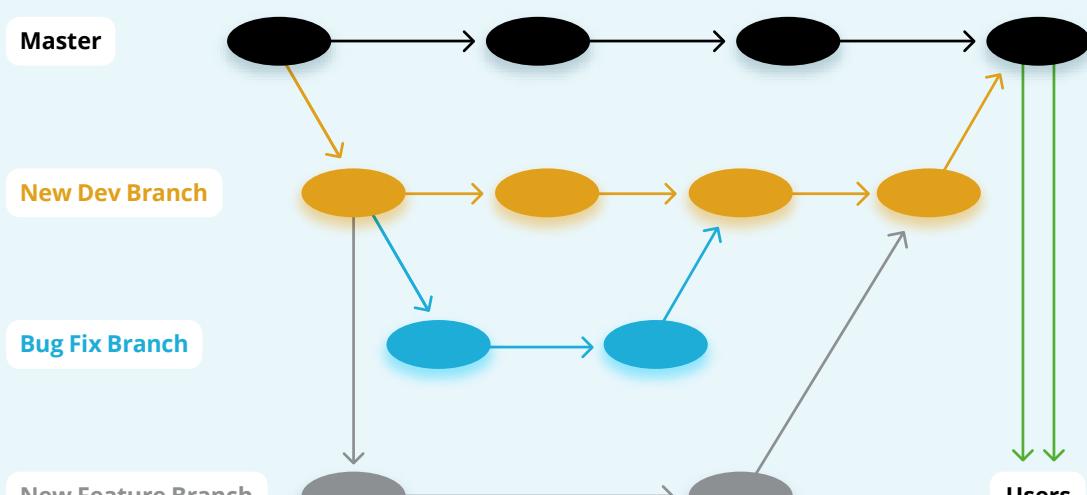
Custom (or proprietary) code has been originally developed by a person or a team, and is the intellectual property of that organization or individual.

Custom code is maintained by the creators or owners of that code, so any innovation or enhancements to it must be made by those responsible, including new releases, patches and any updates required to fix vulnerabilities. Custom code can be incorporated into other projects or released as a complete software application.

Open source code is also created by developers, often as a part of a community-driven project through which ideas and contributions are shared.

This code, or software, is made available to the community as components or projects. Because the innovation happens organically within the community, updates, patches, and new releases are the responsibility of that community. As a project or component evolves over time, it can have associated licenses that detail any restrictions, permissions, or requirements that the project originator has chosen to place upon it.

+ Open source code evolves over time



An open source component or project begins with a master branch, which is evolved and changed by the open source community as they add more branches to modify the code. This is usually with the aim of adding new features or functionality, performing bug fixes, and undertaking testing.

The new branches are then usually merged into the master branch, to become part of the main project. Or, they are maintained as a separate fork, when contributors or groups modifying the project intend to take it in a new direction, or change it to suit another use case.

Understanding open source software

The impact of open source code evolution

While two components may share a name, they may differ greatly. While one vendor's take on an open source component may be fundamentally the same as another's, they may have made some minor changes to suit their needs.

As each component branch (known as a distribution, or distro) goes through changes, this can eventually create significant differences between versions of what might have started out as the same component. These differences can potentially introduce additional maintenance and development costs, or expose you to unexpected security and compatibility issues.



+ Open source code vulnerability



Open source components are used everywhere and, as with custom code, there are situations in which open source components can be vulnerable. It's important to understand the difference between a **vulnerable component** and **vulnerable versions** of that component.

A component can contain vulnerabilities, but only in certain versions. It all depends on how that software is constructed and how it evolves over time.



Newer versions may not contain the same vulnerabilities that the original or previous versions did.



Others may contain their own vulnerabilities that did not exist in previous versions.

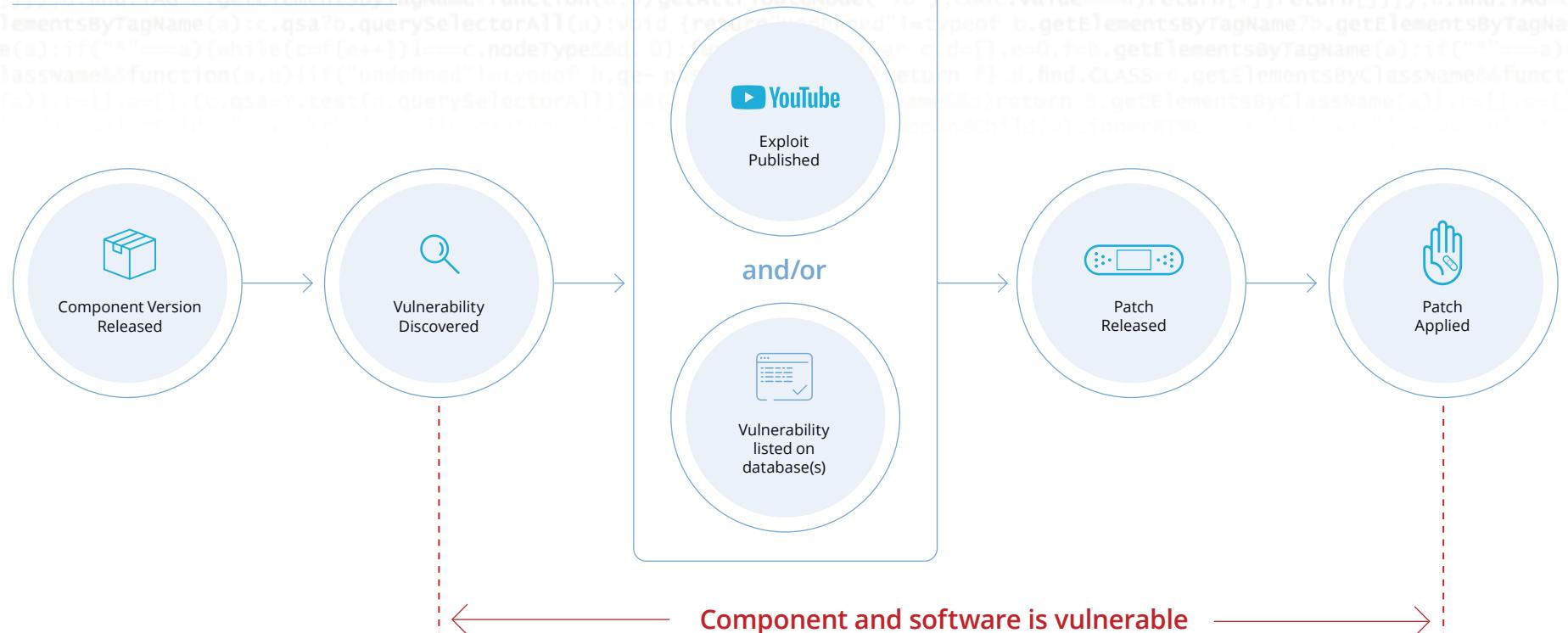


A component version with no vulnerability may have a vulnerability introduced into it in the future.



Not every open source component will be vulnerable, and not all vulnerabilities may be exploitable.

Understanding open source software

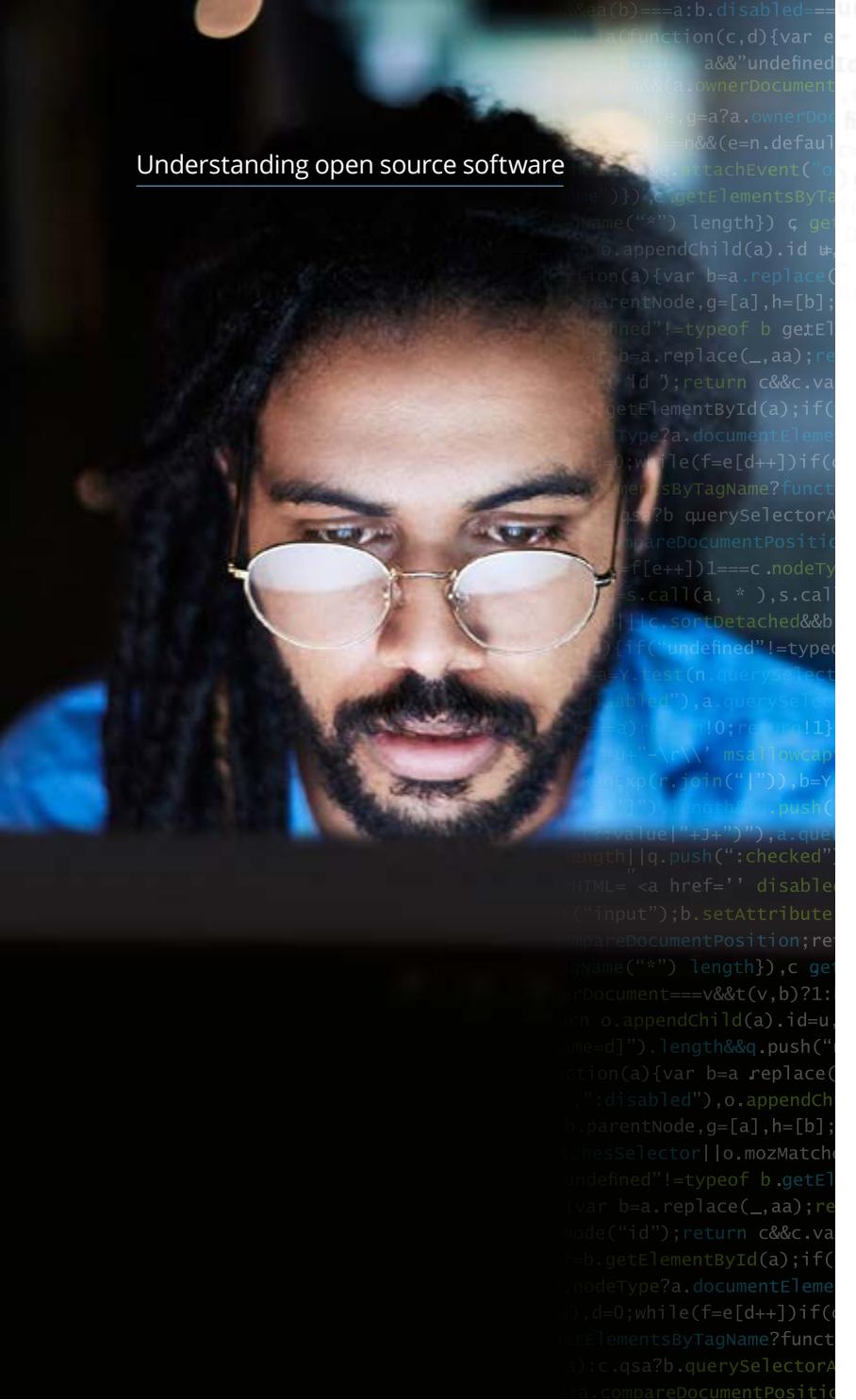


Example of an attack timeline

Attackers first have to discover a vulnerability, 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 the exploit can be executed.

- After a component version is released, a vulnerability is discovered within it. It could be the creators who discover the vulnerability, maybe a security research team, or perhaps attackers.
- The vulnerability may be documented in a vulnerability database (like the NVD), or those who discover it may keep it secret for some time while working on a patch, or an exploit.
- Once the patch is released, there may be a lag before it is applied everywhere the component has been used.
- In the meantime, an exploit may be developed and used secretly, or (as is often the case) published among attacker communities or on public forums such as YouTube so anyone can use it.

Understanding open source software



The time between the exploit being discovered and the patch being 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.



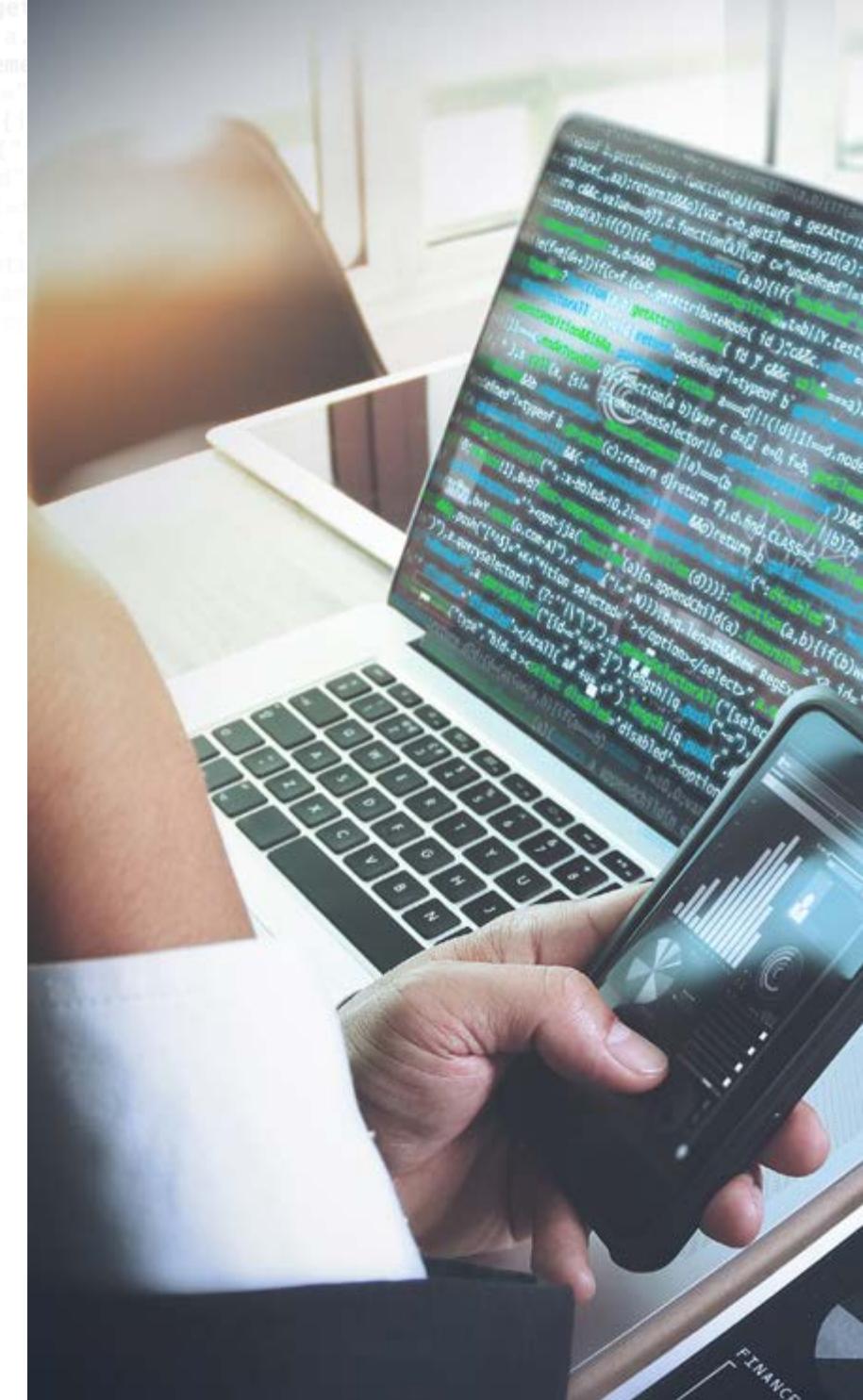
Clearly, there is a need to patch vulnerabilities quickly.

Understanding open source software

+ Dealing with vulnerability alerts

Depending on the source or origin of the component, you may be notified when they're discovered, or you may not. For example, components from Red Hat or Apache may yield helpful alerts when vulnerabilities are discovered, or when patches are available to remedy them.

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.



+ Working safely with open source components

You need the right tools, methods, and processes firmly established to create stable and secure software from your mix of custom code and open source components. Your developers work in a complex software development environment, with many aspects which must be configured and maintained to produce viable software.



Managing licenses, compliance and regulatory requirements

It's not just vulnerabilities that you will need to check your code for. Organizations that create software are often subject to external and internal standards and requirements, such as customer SLAs, internal specifications, and data protection regulation. Open source projects may also have licensing restrictions or requirements, determined by the author or originator of the component.

Open source projects can have virtually any licensing structure. There are two main categories of open source licenses: reciprocal (or copyleft) licenses, and permissive licenses.

- Reciprocal licenses generally 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.

Common examples of open source licenses include GPL 3.0, MIT License, and Apache 2.0. There are also some examples of licenses that illustrate how an author can be free to create their own. These include the WTFPL license (Do What the **** You Want To Public License), which is completely open, and the Beerware License, which requires that anyone who leverages a component with that license buys the author a beer.

It's essential to be aware of the licensing requirements that your organization – and your application – is subject to, and ensure you have the software security testing solutions in your arsenal to help you ensure compliance on an ongoing basis.

Understanding open source software

Application 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. When your software uses a mix of custom code and open source components, the application security testing you use needs to be purpose-built to examine, identify, triage, and remediate any issues across all types of code.

Organizations tend to use static application security testing (SAST), dynamic application security testing (DAST), interactive application security testing (IAST), and software composition analysis (SCA).

SAST

Reviews source code to identify the sources of vulnerabilities

DAST

Is a so-called ‘black-box’ testing method that looks at functionality and tests by performing attacks – it does not look at the source code

IAST

Is a combination of both SAST and DAST methodologies

SCA

Identifies open source code and components, matching them with known vulnerabilities, such as those listed on the National Vulnerability Database (NVD)

SAST examines source code directly to look for weaknesses or vulnerabilities in the code that could be exploited. This analysis can be lengthy process, depending on the size of the codebase being analyzed, and can generate huge volumes of results that need to be addressed. Any identified vulnerabilities must be removed, and those code components rewritten. This can take considerable time and effort.

SCA does not examine the source code itself – it looks for open source components within it. It needs to be able to detect and identify open source components within the software and match those identified versions against a database of vulnerabilities. Any vulnerabilities that are identified within that software then need to be patched or replaced.

Application testing is part of the development process

Review 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 the security testing phase to identify vulnerable open source components within your software. Use the right solutions during the process, not afterwards.

In summary: if you use open source components, you must have a way of analyzing the composition of your software to ensure the components you’re using are safe and licensed appropriately. To do this properly, software composition analysis (SCA) is a critical resource.

+ Understanding software composition analysis (SCA)



+ Understanding software composition analysis (SCA)



Software composition analysis is the standard term for analyzing software, discovering open source components and third-party libraries within it, and identifying the associated risks.

SCA focuses on measuring two main types of risks: security risk (open source vulnerabilities) and license risk (noncompliance, or conflicts between open source licenses). Sometimes, there may be a third, non-standard category of risk that explores community activity surrounding the component.

When it first appeared around a decade ago, SCA originally focused on license compliance for software and embedded technologies, such as hardware and chipsets. However, with the rapid growth in the use of open source code, software security has become its biggest use case, and SCA is now evolving to extend its influence across application security testing (AST), with some SCA solutions integrating and correlating data with SAST solutions to better assess exploitability and examine if vulnerable components are actually being used by the application.

SCA is essential to secure software development.

Understanding software composition analysis (SCA)

+ Key aspects of SCA

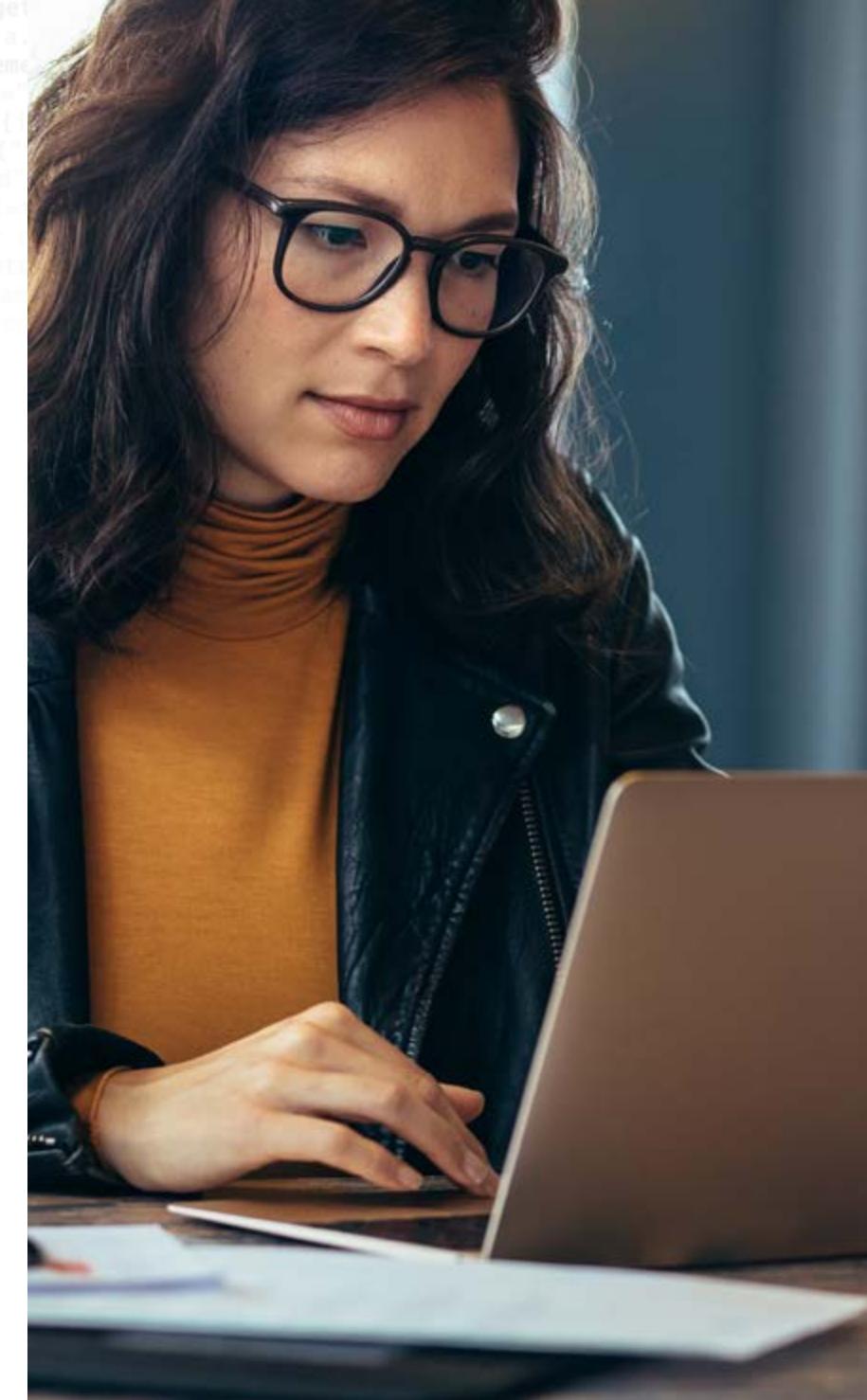
When an effective SCA solution is integrated into an organization's continuous integration/continuous delivery (or development) pipeline (CI/CD) and software development lifecycle (SDLC), it enables development, security, and DevOps teams to prioritize and focus their remediation efforts where they will be most effective and least costly, before any potentially at-risk projects go into production.

An SCA solution must be able to:

- accurately detect and identify open source components and component versions in use within software
- provide insight into vulnerabilities associated with those components and component versions, as well as any licenses that may apply to them
- provide actionable risk insight and remediation guidance
- allow organizations to configure and enforce policies against the analysis results
- integrate with tools that your organization is using in its SDLC or CI/CD pipelines
- deliver insight and results to relevant people, in the format that is most helpful to them

Some SCA solutions might include additional functionality:

- the ability to identify if a vulnerable open source component version is exploitable
- metrics associated with component bugs and community activity
- correlation of analysis results with other application security testing solutions



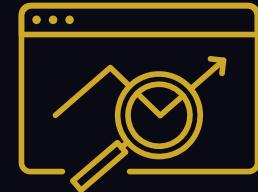
SECTION 3

+ A technical deep dive into SCA



+ A technical deep dive into SCA

Software composition analysis happens in three major steps:



1

Detection

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 complete, while others yield higher accuracy in a shorter amount of time, with slightly more up-front configuration.

2

Identification

Next, SCA identifies the open source components it has detected by referencing a database of open source component information. The output usually includes basic component version information, and may include details of where the component version came from, plus other metadata.

3

Risk metrics

The solution produces risk metrics based on what has been detected and identified – this is almost always security information and license data. This also involves checking against a reference database (or databases) covering vulnerability and license data. It may include data exclusive to the solution vendor, if they have a security research team.

A technical deep dive into SCA

Open source detection methodologies

Not all SCA solutions take the same approach to detection. For example, some solutions will perform **signature scanning**, which generates a SHA-1 hash signature of each open source component it detects. These alphanumeric strings uniquely represent individual components, like fingerprints. The SCA solution then tries to match these hash signatures against those listed in a database of previously scanned open source components.

Some SCA solutions will look at the files created by your build tools and package managers: **package manager inspection**. This can determine the specific version of each component being used – it's the equivalent of checking what the developers say is in the software.

Finally, an SCA solution may also conduct **build dependency analysis**, examining the software after development. This identifies any dependencies that have not been declared but were brought into the application during the build process – such dependencies could present a potential risk by introducing vulnerabilities into software.



Signature (or file system) scanning

The main benefit of signature scanning is its ability to produce a large number of results. This can be perceived as the most complete or comprehensive representation of all the open source components within a codebase.

Signature scanning can detect any non-declared components that may not have been included in the package manager files, or can be used if the software was built without the use of package managers.

However, the scanning process can take a long time, consumes a lot of compute power, and produces a large volume of results that need to be reviewed – often including a significant number of false positives. When time is short and production deadlines are getting closer, this methodology may cause you more headaches than it resolves.

Package manager inspection

When an SCA solution uses package managers to detect open source components, you will see a few more benefits. The results tend to be highly accurate, with very few false positives, and with less noise and fewer junk results, it is easier for your developers or security teams to review the output and prioritize their efforts.

These scans also tend to be a lot faster, and this methodology is more suitable for DevOps thanks to its integration with the CI/CD tools that developers are already using.

A technical deep dive into SCA

Build dependency analysis

If open source components have not been declared, or if the software is built without the use of package managers (as we often see in some legacy applications), package manager inspection may not identify all open source elements within the analyzed codebase. This is why solution providers often pair package manager inspection with build dependency analysis.

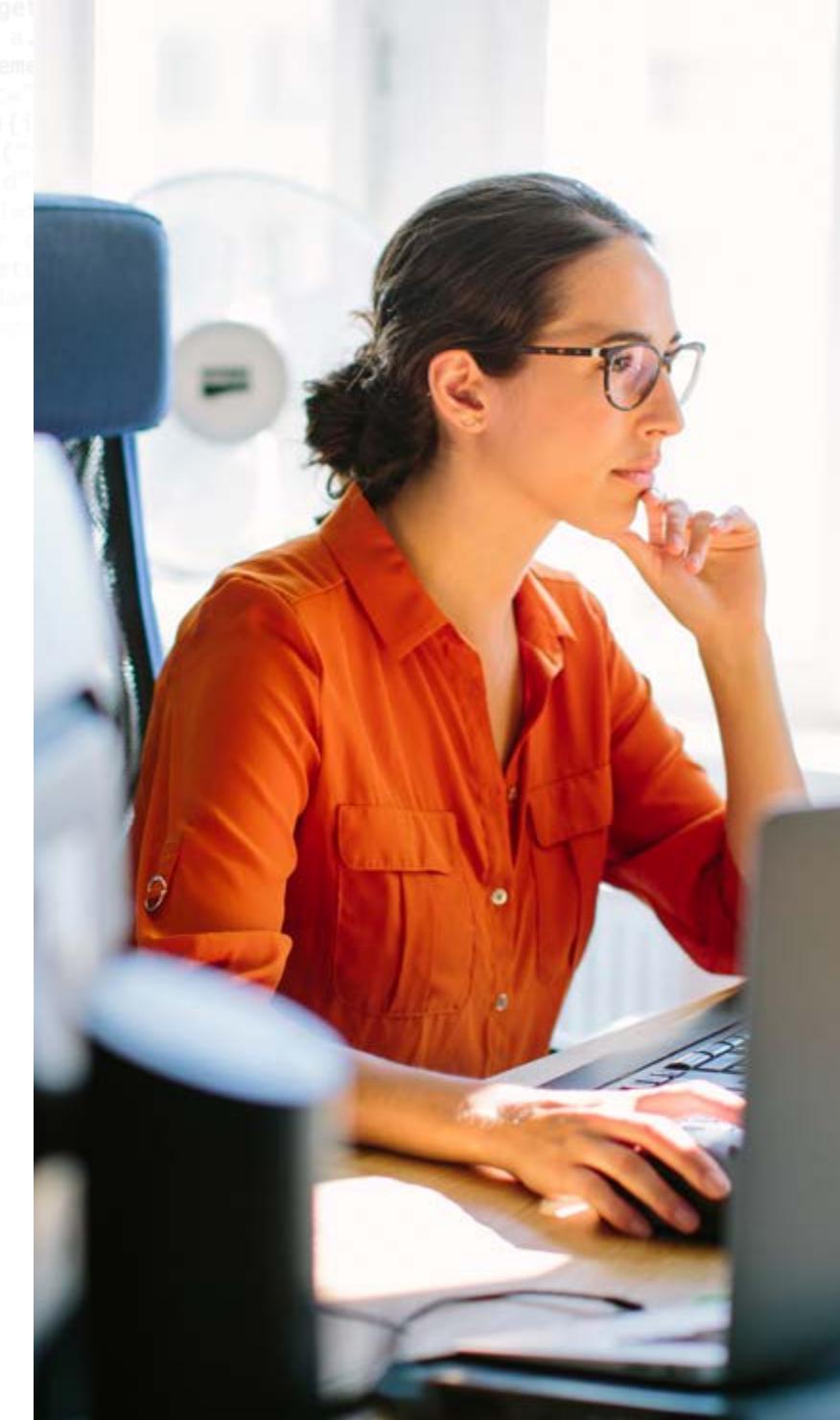
Build dependency analysis detects any non-declared dependencies that have been incorporated during a build, or any dependencies of dependencies (transitive dependencies).

Snippet scanning

Snippet scanning is similar to signature scanning. Rather than looking at entire open source components, the SCA solution performs a signature scan of smaller subsets of code, referencing a database of previously scanned and documented component segments.

Snippet scanning can help identify potential license requirements, license conflicts, or risk of noncompliance resulting from 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.

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



+ Component identification

After open source components have been detected using one or several methods, they need to be identified. Often, component metadata is referenced against a database of open source components maintained by the solution vendor. Such databases contain information from various code repositories and sources, such as GitHub, Maven Central, and many others.

If a match for the detected component is found in the database, its information is displayed by the SCA solution. This is where the risk of false positives is greatest, and where selecting the right detection method can have the greatest positive impact on the quality and actionability of your results.

+ Risk metrics

Once open source components have been detected and identified, the SCA solution needs to generate associated risk metrics. This is essential for prioritizing where to focus your efforts, in order to improve your risk posture.

Firstly, 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.

Security risk often has standardized scoring, usually, CVSS2.0 or CVSS3.0, but it's important to recognize that risk metrics are not always standardized. The determined severity or priority associated with risk metrics can vary by SCA solution vendor. Where non-standard scoring is used, it's usually possible to adjust sensitivity to risk, based on criteria associated with the project that's being scanned. This is not a standard capability for all SCA solutions and does prevent comparison of the risk profile of one project against the relative risk profile of another.

License risks



Security risk metrics are among the most common criteria for organizational policy rules. License risks, however, are highly contextual, and can vary, depending on how the application is deployed.

- Is it an internal application that's on company servers and not for public or commercial use?
- Is it an external-facing application?
- Is it a commercial application?

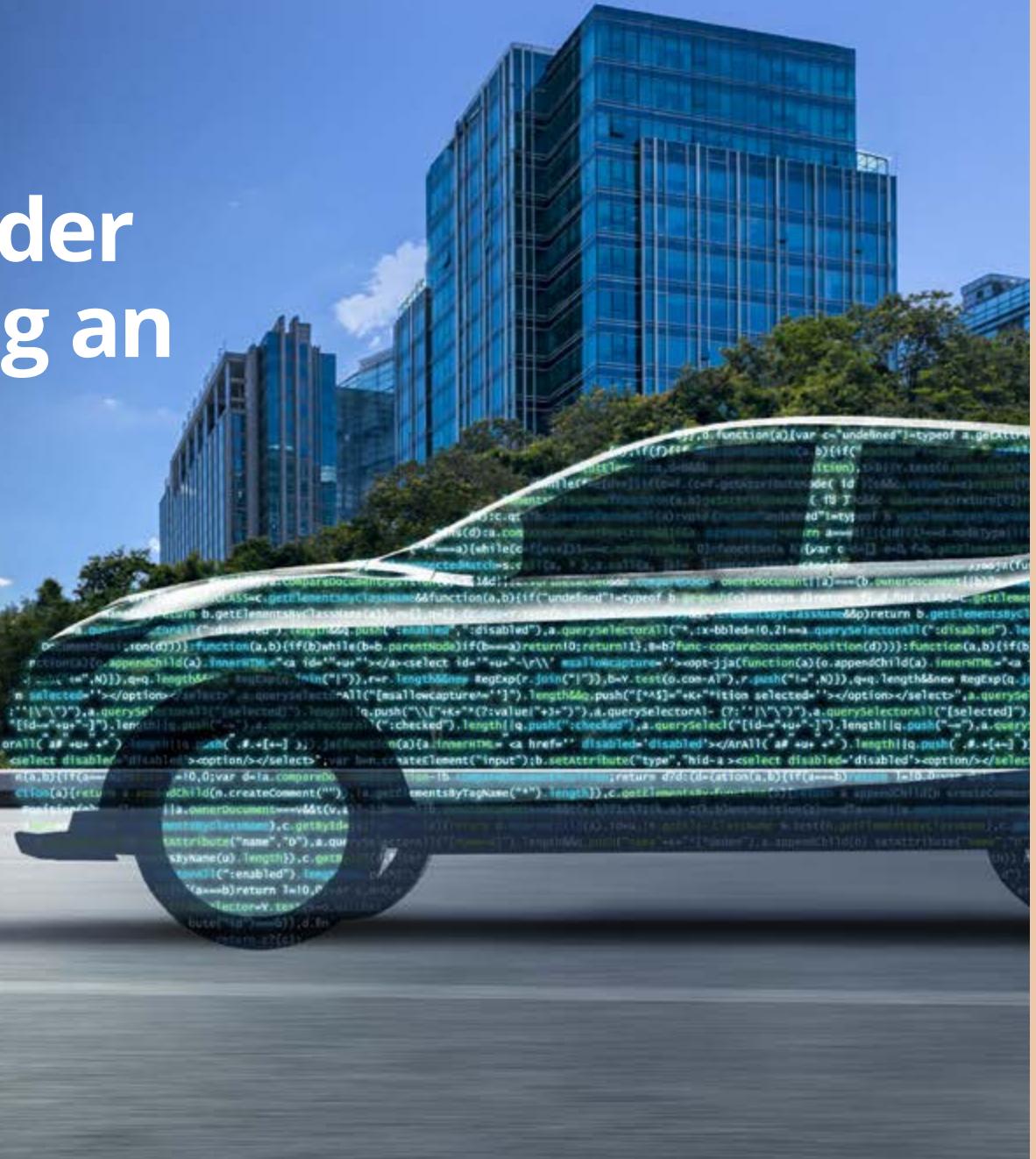


Other components or licenses within the application may also impact license risk: this is known as license conflict. If an application uses open source components with both permissive and reciprocal licenses, this can lead to some complicated results, and any royalties or attribution requirements may be a concern. These various factors can determine the severity of your license risk.

While there is no standardized measure of license risk, it's generally accepted that licenses which cost money, restrict use, or require sharing intellectual property from the associated codebase are all generally considered to be higher risk.

In terms of SCA, license risk is usually most relevant to embedded devices or chipset manufacturers. This is particularly relevant to use of the internet of things (IoT), and to tier-one and tier-two automotive industry suppliers, system integrators, and other organizations where it can be hard to access, replace, or update the software, or the device on which the software sits. This tends to be the case in industries where potential loss of intellectual property due to license conflicts or noncompliance can be devastating.

+ What to consider when choosing an SCA solution



+ What to consider when choosing an SCA solution



Focus on solutions with higher accuracy and fewer false positives.

Comprehensive results can be good, but only if you have the time to review and verify them all. It's also worth noting that risk metrics aren't standardized across vendors, and can vary in severity or priority.

Highly consider vendors whose solution is supported by a dedicated security research team.

Make sure the vendor is proactively finding zero day or non-public vulnerabilities, and enhancing their existing security records.

Look for vendors that provide a comprehensive list of any publicly reported vulnerabilities in open source components.

These need to be accompanied by appropriate remediation guidance.

Ensure the solution will fully support the requirements of your security, legal, and engineering teams.

It should enable them to configure and enforce policies against the analysis results.

Prioritize solutions that are part of a complete application security testing (AST) portfolio.

Or those that complement what you're currently using.

Certify available integrations with your package managers, build tools, code repositories, issue management solutions, and so on.

Give priority to solutions which enable cross-product synergy, which will help to prioritize your remediation efforts and enhance the accuracy and actionability of the analysis.

Verify that the solution integrates with the tools you're already using in your SDLC or CI/CD pipelines.

It must enable you to automatically trigger scans, share results, and reduce time-to-remediation.

Validate that the solution supports unified user management, project creation, and scan initiation capabilities for multiple testing technologies.

Solutions that do this will yield the greatest efficiencies and reduce your total cost of ownership.

+ Conclusion

Open source components are not going to disappear any time soon. Organizations therefore need to use SCA as part of their software security strategy.

The key to implementing SCA successfully is to select a solution that can be integrated with your software development tools, that supports internal and external standards for risk tolerance and compliance, and gets detailed insight promptly into the hands of the 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. This trend is likely to increase due to the prevalence and accessibility of open source components, and the (historically) inadequate documentation, evaluation, and monitoring of the risks they contain. Clearly, software composition analysis solutions are needed now, and will be required well into the future.



+ Further reading:



Datasheet

Download the Checkmarx CxSCA datasheet



eBook

2020 Gartner Magic Quadrant for Application Security Testing



Website

Discover next generation open source security: Checkmarx CxSCA





About Checkmarx

Checkmarx is the global leader in software security solutions for modern enterprise software development. Checkmarx delivers the industry's most comprehensive Software Security Platform that unifies with DevOps and provides static and interactive application security testing, software composition analysis, and developer AppSec awareness and training programs to reduce and remediate risk from software vulnerabilities. Checkmarx is trusted by more than 40 of the Fortune 100 companies and half of the Fortune 50, including leading organizations such as SAP, Samsung, and Salesforce.com.