BLACKDUCK®

2025 Open Source Security and Risk Analysis Report

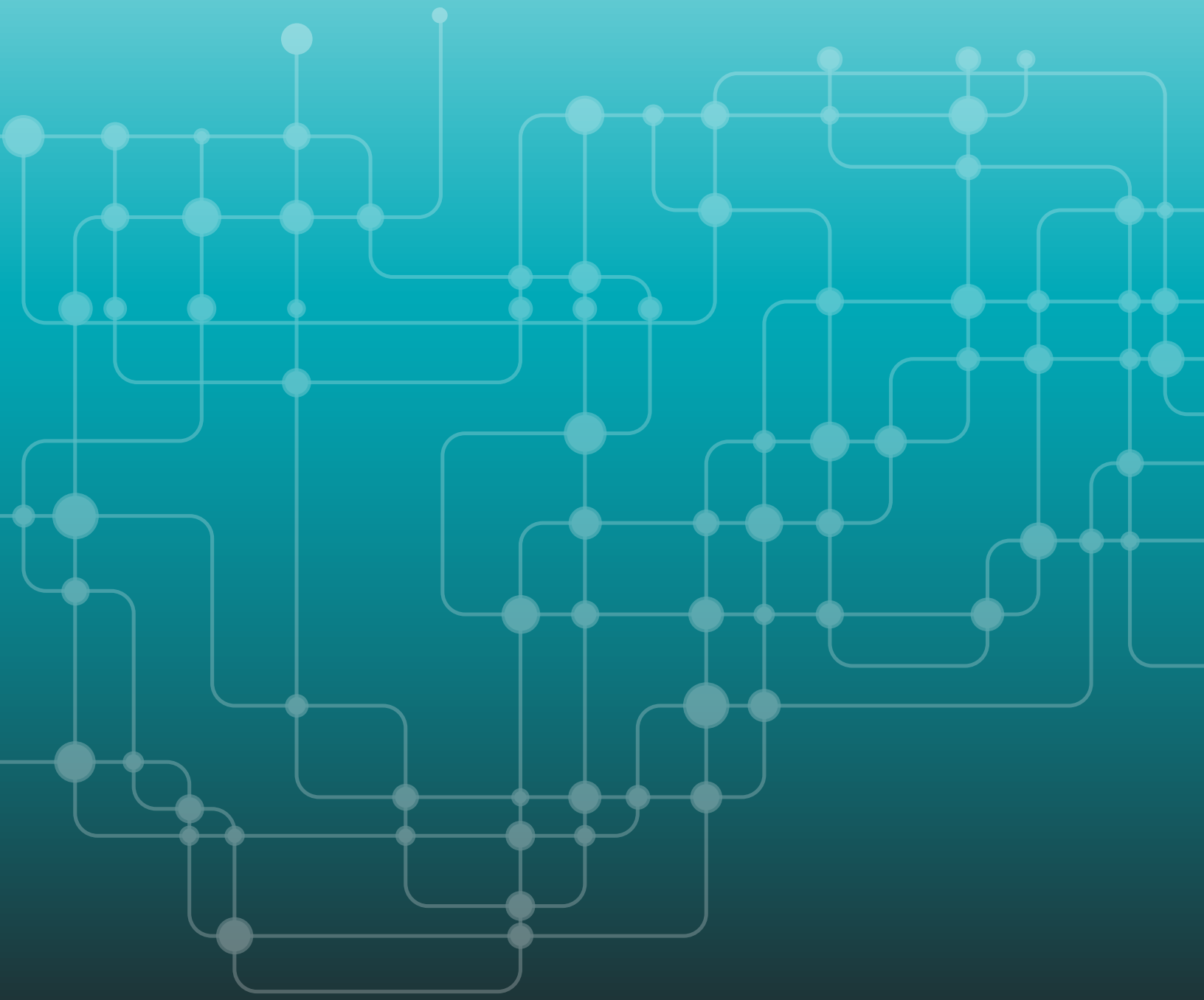


Table of contents

Welcome to the 2025 OSSRA Report 1

Who Should Read This Report 1

What You'll Learn and Why It Matters 2

About This Report's Data and Black Duck Audits 3

Our Findings at a Glance 4

Looking at Open Source Risk and Vulnerabilities 7

Software Security Begins with Visibility into Your Code 7

Understanding Risk Management and Gaining Visibility into Your Code 8

Enhancing Software Security and Transparency with SCA and SBOMs 8

Analyzing the Impact of a Vulnerability 11

Log4j and Equifax: Two Lessons on the Need for Visibility into Your Code 12

The Top High- and Critical-Risk Vulnerabilities 13

What the Data Tells Us 18

Industry-Specific Insights 18

Open Source Licensing 19

How Conflicts, Variants, and Lack of Licenses Create Risk 19

The Impact of Transitive Dependencies on License Conflicts 20

The Top 10 Open Source Licenses of 2024 20

What Are Permissive, Weak Copyleft, and Reciprocal Open Source Licenses? 21

How to Manage Open Source License Risk with SCA 21

Industry Perspectives on License Conflicts 22

If You Anticipate an M&A 23

Maintenance and Operational Factors Impacting Risk 25

Conclusion: The More Things Change 27

Key Recommendations 28

Welcome to the 2025 OSSRA Report

Open source software (OSS) has revolutionized application development, providing a vast repository of prebuilt components that offer numerous benefits such as cost savings, flexibility, and scalability. However, with all those benefits comes risks that every organization using open source needs to be prepared to acknowledge and address.

The 2025 "Open Source Security and Risk Analysis" (OSSRA) report details key findings from Black Duck® audit data, including security vulnerabilities, licensing issues, component maintenance, and industry trends. Our analysis shows that open source is ubiquitous, and that it can introduce significant risk unless properly identified and managed.

> *"He will win who has prepared himself."*
>
> *—Sun Tzu*

Who Should Read This Report

The findings of this report will be beneficial for a variety of readers, particularly those involved in **securing the software supply chain,** as well as those directly involved in **software development, security and risk management,** and **merger and acquisition (M&A) activities.**

Developers will gain insights into the types of vulnerabilities that we found prevalent in open source software, such as cross-site scripting (XSS) and denial-of-service (DoS) vulnerabilities. For example, the OSSRA report highlights the importance of following input validation and sanitization techniques, which can help developers build more-secure applications.

Further, this report identifies the most common open source components containing vulnerabilities, which will aid developers in making informed decisions when selecting open source libraries and frameworks. For example, development teams should be aware that our data shows that jQuery, jackson-databind, and the Spring Framework often include vulnerabilities that require regular management and patching.

OSSRA 2025 also emphasizes the risks associated with using out-of-date components and the need for all organizations to implement a process for timely updates. As one example, 90% of audited codebases were found to have open source components more than four years out-of-date. Outdated components magnify security risk, provide attackers with an expanded attack surface, and create compliance and compatibility issues. The presence of older open source also suggests that developers are not taking advantage of software improvements and are relying on code that is no longer being maintained.

Readers with a security focus can leverage the data presented in OSSRA 2025 to improve their vulnerability management processes. For example, the report identifies the top common vulnerabilities and exposures (CVEs) found in our audits, as well as their relationship to common software weaknesses (CWEs).

Risk management professionals can use OSSRA data to inform their strategic decisions about open source software adoption and risk mitigation. The ability to compare vulnerability percentages and other metrics across industries can help risk managers pinpoint areas where their organization is performing well or needs improvement.

The OSSRA data, primarily derived from analysis of M&A targets' code, provides key insights for professionals involved in merger and acquisition transactions into the kinds of issues they may be taking on in their own transactions, such as common open source license conflicts, the security posture of the target company, and potential operational challenges that could impact the target's IP value.

What You'll Learn and Why It Matters

There's much more open source in your software than you think: Ninety-seven percent of the codebases we evaluated contained open source, with an average 911 OSS components found per application. From an industry perspective, the percentages ranged from 100% in the Computer Hardware and Semiconductors, EdTech, and Internet and Mobile Apps sectors, to a "low" of 79% for Manufacturing, Industrials, and Robotics.

Open source codebases are getting bigger and more complicated: Our data shows that the number of open source files in an average application has tripled in just the last four years. One of the reasons behind this is the use of "transitive dependencies"—open source libraries that other software components rely on to function. Open source frequently uses other open source. Our audits found that 64% of open source components identified in our scans were transitive dependencies, most nearly impossible to locate or track without using an automated tool. Finding all instances of a transitive dependency can be like searching for a needle in a haystack when you lack an up-to-date inventory of third-party code.

Where all this open source is coming from: Our audits show that the majority of open source is being downloaded from package manager repositories. Over 280,000 of the nearly 1 million OSS components found in our audits originated from one such repository—npm, a massive public database of JavaScript packages.

Whether you think of open source as "free" or not, it comes at a cost: The odds are better than 80% that an application your organization is using right now contains high- or critical-risk open source vulnerabilities, with nearly half of those introduced by transitive dependencies.

Transitive dependencies present licensing and maintenance issues as well as security challenges: Our audits found that over half the codebases contained license conflicts, many caused by a transitive dependencies' incompatibility with another component's license. Nearly 30% of component license conflicts found in our audits were caused by transitive dependencies.

Static application security testing (SAST) and dynamic application security testing (DAST) can help identify coding errors: These testing methods can find errors such as input validation and sensitive information exposure, and mistakes like not encrypting important data when it's being sent over the internet, outdated or weak encryption methods, and failing to properly protect passwords or other secret information.

Every organization using web applications and services should be evaluating them with software composition analysis (SCA) and DAST tools: Development and security teams need to implement a multifaceted security approach integrating DAST, SAST, and SCA to achieve the comprehensive security coverage modern software demands. Our findings indicate that if such a full-spectrum approach were applied, potential exposure to critical vulnerabilities would be markedly reduced.

About This Report's Data and Black Duck Audits

This report uses data from the Black Duck Audit team's evaluation of anonymized findings from 1,658 analyses of 965 commercial codebases across 16 industries during 2024 (see note below).

Black Duck offers a range of services including open source audits tailored to diverse needs and objectives. Open source audits leverage a combination of automated tools, comprehensive databases, and expert analysis to provide a thorough assessment of an organization's OSS usage. Built over two decades, the Black Duck KnowledgeBase™, a key component of these audits, contains data on millions of open source components, including their licenses, vulnerabilities, and potential risks. Sourced and curated by the Black Duck Cybersecurity Research Center (CyRC), the KnowledgeBase includes data on more than 7.8 million open source components from over 31,000 forges and repositories.

A Black Duck open source audit typically involves the following steps:

- **Codebase submission:** An organization provides Black Duck with access to the codebase to be audited. This includes source code, binaries, and other relevant artifacts.
- **Automated analysis:** Black Duck utilizes its suite of automated tools, including its SCA solution, to scan the codebase and identify all open source components and those components' dependencies, including transitive dependencies, through advanced string search capabilities.
- **Expert review:** Black Duck's team of open source experts reviews the results of the automated analysis, validates the findings, and ensures completeness and accuracy.
- **Report generation:** Black Duck generates a comprehensive set of reports that provide a detailed Software Bill of Materials (SBOM) of all open source components, their associated licenses, known security vulnerabilities, and potential operational risks. The reports also detail the issues cataloged in the SBOM.
- **Remediation guidance:** Black Duck provides guidance on how to address the identified issues, such as updating vulnerable components, resolving license conflicts, and mitigating operational risks.

Note: Several improvements to how the Black Duck Audit team evaluates and presents audit data were implemented during 2024. Notably, a single submitted customer codebase is now split into multiple analyses called "projects." The new technique provides a more granular approach to analyzing codebases and offers several benefits to customers including more-detailed reports and more-accurate component identification and dependency tracking. The changes also affect how audit data is presented. For example, while for simplicity's sake we still refer to "codebases" in the OSSRA, at a more granular level those codebases entail the analysis of 1,658 individual projects from the 965 codebases submitted to Black Duck in 2024.

Our Findings at a Glance

1,658 projects scanned by Black Duck audits

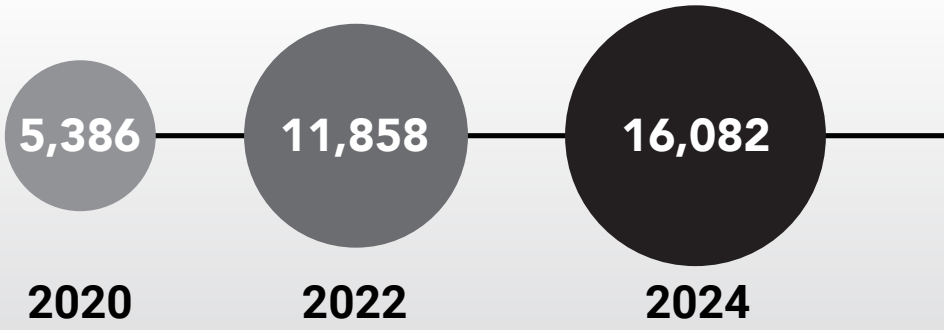
97% of the codebases contained open source

70% of scanned code had its origin in open source

An average **911** OSS components were found per application

64% of OSS components were transitive dependencies

The number of open source files in an average application has tripled in the last four years.



Over 280,000 of the open source components found in our audits originated from the npm repository. **Most OSS packages found in our scans were written in JavaScript.**

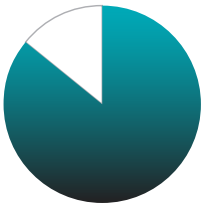
Originating repository	Language	No. of components found in 2024 audits
npm	JavaScript	282,521
yarn (JavaScript)	JavaScript	162,327
pnpm (JavaScript)	JavaScript	24,069

But not all. For example, use of Rust package repositories has increased considerably as developers respond to memory safety issues in C and C++.

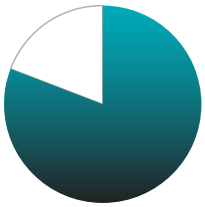
Originating repository	Language	No. of components found in 2024 audits
Cargo	Rust	33,327
Nuget	C#, Visual Basic, F#, WiX, C++, Q#	29,818
go_mod	Go	24,069
Maven	Java	14,097
packagist	PHP	6,112
Gradle	Java, C, JavaScript	4,615

Our Findings at a Glance

Vulnerabilities and Security



86%
of risk-assessed codebases contained vulnerable open source



81%
of risk-assessed codebases contained high- or critical-risk vulnerabilities

8 of the top 10
high-risk vulnerabilities were found in jQuery

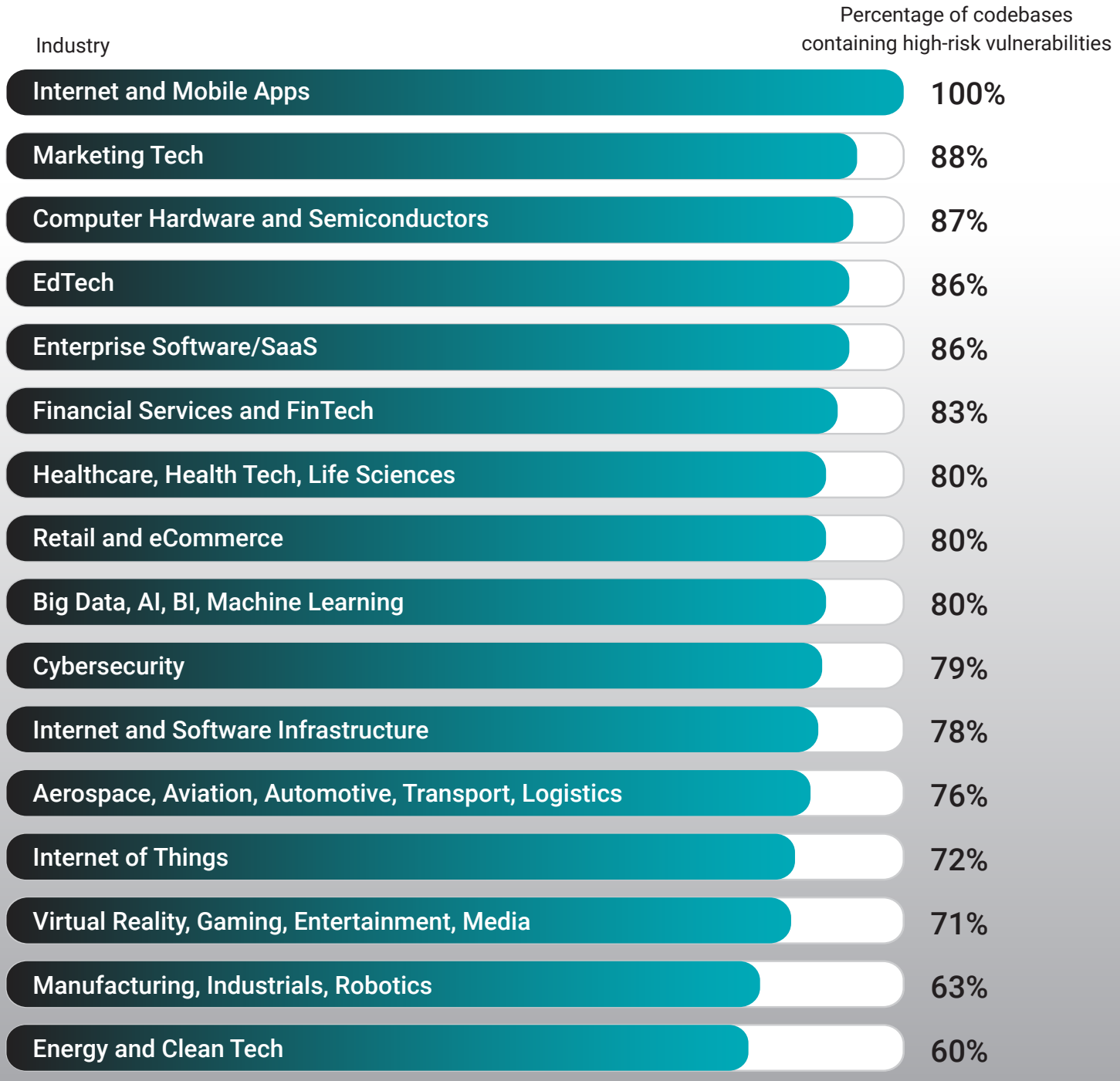


Figure 1: Codebases Containing High-Risk Vulnerabilities by Industry

Our Findings at a Glance

Licensing

56% of all codebases had license conflicts

33% of all codebases had OSS components with no license or customized license language, typically comments by the developer about how the software is to be used

Maintenance and Operational Risk

91% of all codebases contained outdated OSS components

90% of all codebases contained components more than 10 versions behind the most current version

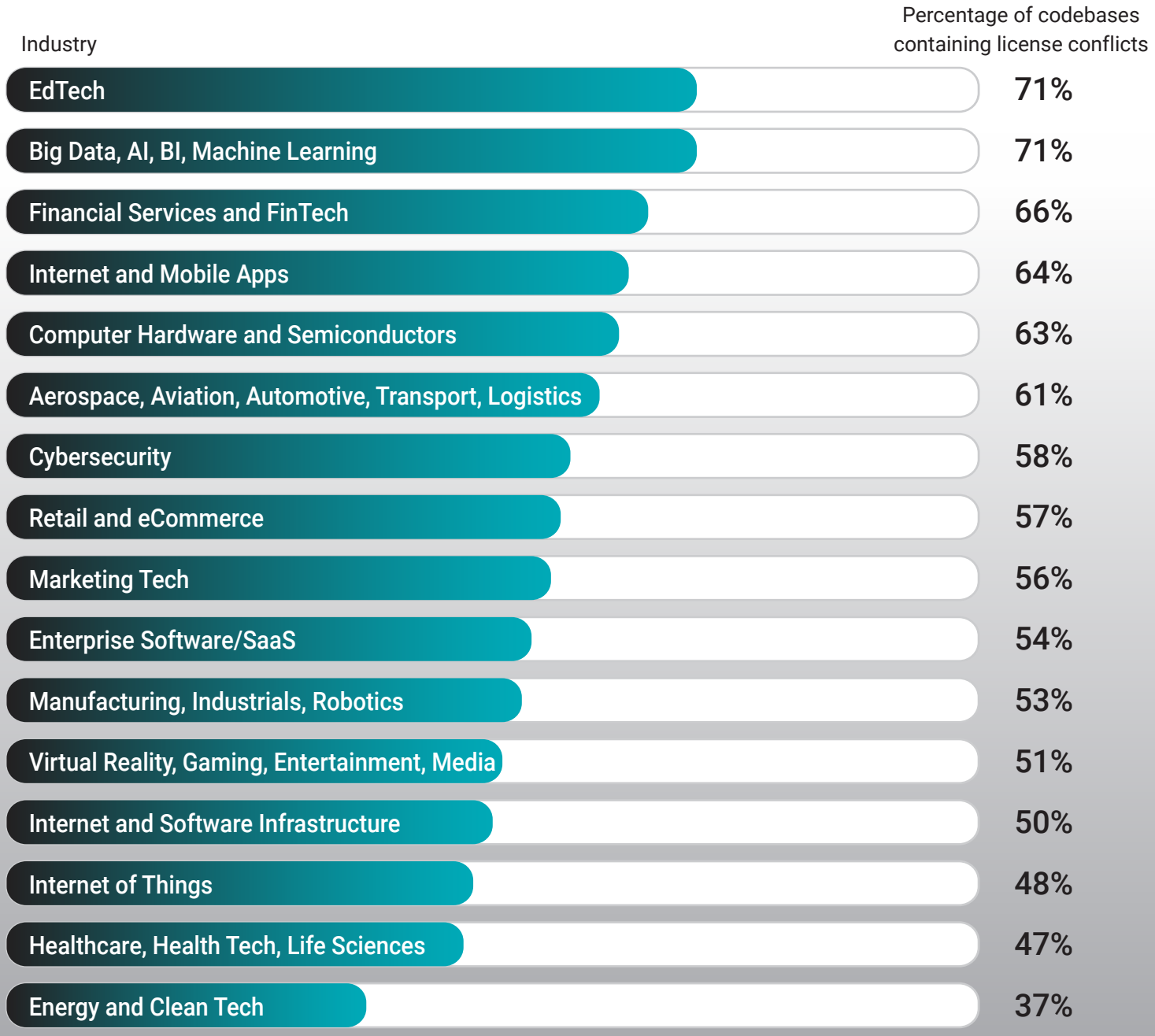


Figure 2: Codebases Containing License Conflicts by Industry

Looking at Open Source Risk and Vulnerabilities

All Black Duck audits examine open source license compliance. Customers can opt out of the vulnerability/operational risk assessment portion of the audit at their discretion. During 2024, the Black Duck Audit team conducted vulnerability/operational risk assessments on 901 customer codebases. The data in this section and the "Maintenance and Operational Factors Impacting Risk" section are based on those assessments.

Software Security Begins with Visibility into Your Code

86%

of the codebases contained at least one vulnerability

81%

of the codebases contained high- or critical-risk vulnerabilities

Maximum number of unique vulnerabilities found in a single codebase

3,548

Mean number of unique vulnerabilities per codebase

154

Components	Percentage of codebases containing the component
jQuery	32%
jQuery UI	16%
Bootstrap (Twitter)	15%
Spring Framework	12%
Lodash	12%
Netty Project	11%
jackson-databind	9%
Apache Tomcat	8%
Python programming language	5%
TensorFlow	1%

Figure 3: Top 10 Components Containing High- or Critical-Risk Vulnerabilities

Understanding Risk Management and Gaining Visibility into Your Code

Effective open source risk management is not about finding and fixing every vulnerability—a Sisyphean task if ever there was one. Rather, risk management is about gaining the knowledge necessary to make informed decisions regarding risk to your code. For example, once a vulnerability is identified, you can assess its severity, likelihood of exploitation, and potential impact on your systems. Likewise, not every license conflict or code quality issue may be a high priority for every organization. Focusing on the most critical issues is essential for efficient open source risk management.

But you must first be aware of those issues to address them. Insight into your code is key to prioritizing remediation efforts. To gain that insight, organizations are increasingly adopting SBOMs, comprehensive inventories of all software components and their dependencies.

> *"Less certainty requires more inquiry."*
>
> *—Erik Seidel*

Enhancing Software Security and Transparency with SCA and SBOMs

An SBOM is a formal record containing the details and supply chain relationships of all the components used in building software. It also serves as an inventory of all the constituent parts of a software application, including open source libraries, third-party modules, frameworks, and their associated metadata, such as licenses and versions. SBOMs provide transparency into the software's composition, enabling organizations to understand what's running in their environment and ultimately enabling security teams to understand risk, track dependencies, and audit software.

A study by the Linux Foundation found that organizations that generate SBOMs are better able to understand dependencies across components in an application, monitor components for vulnerabilities, and manage OSS license compliance. And a report by Gartner® highlights that SBOMs improve the visibility, transparency, security, and integrity of proprietary and open source code in software supply chains. Many customers at the end of the software delivery pipeline now make an SBOM a requirement in their vendor contracts.

In short, SBOMs are essential for organizations to ensure the security, compliance, and overall health of their software applications, providing benefits for

- **Risk management:** Identifying and managing risks in the software supply chain.
- **Vulnerability management:** Quickly identifying and mitigating known vulnerabilities.
- **License compliance:** Ensuring compliance with open source and third-party licenses.
- **Software quality:** Identifying outdated or unsupported components.
- **Mergers and acquisitions:** Assessing legal and intellectual property (IP) risks associated with software components during mergers and acquisitions.

- **Secure software development:** Enhancing secure software development practices as recommended by the U.S. Cybersecurity and Infrastructure Security Agency (CISA), the SLSA (Supply Chain Levels for Software Artifacts) Framework, and the NIST SSDF.
- **Effective software development, deployment, and maintenance:** Facilitating efficient software development processes and life cycle management.
- **Consistent and readable dependency profiles:** Providing a standardized and easily understandable representation of application dependencies.
- **Standardized dependency listing and automation:** Ensuring consistency in the way dependencies are listed and making automation easier.

SBOMs improve the visibility, transparency, security, and integrity of proprietary and open source code in software supply chains.

How SCA Tools Generate SBOMs

SCA tools generate an SBOM by

- **Code scanning:** SCA tools scan the source code or binary files of a software project to identify all the components and dependencies. These scanners utilize a variety of scanning methods, including
  - **Manifest scanning:** Checks manifest files (e.g., package.json or Cargo.toml) for the dependencies listed.
  - **Binary scanning:** Checks compiled binaries for any third-party code it can trace back to a specific library.
  - **Hybrid scanning:** Uses a mix of manifest and binary scanning to ensure that no dependency slips through.
  - **Snippet scanning:** Analyzes smaller parts of files or lines of code ("snippets") and matches them against a database of known full open source components.
- **Dependency analysis:** SCA tools analyze the relationships between components, including direct and transitive dependencies.
- **Vulnerability and license identification:** The tools compare the identified components against vulnerability databases and license repositories to identify potential security risks and license compliance issues.
- **SBOM generation:** Based on the information gathered from the analysis, SCA tools create a comprehensive SBOM in a standardized format, such as SPDX or CycloneDX.
- **Continuous monitoring:** SCA tools often provide continuous monitoring capabilities to keep the SBOM up-to-date. As new vulnerabilities or updates for open source components become available, the tool can update the SBOM accordingly, ensuring its accuracy over time.

For example, Black Duck® SCA can be integrated into the software development life cycle to generate SBOMs as software is developed. Black Duck SCA also allows users to import third-party SBOMs so that those components can be added to relevant projects, continuously analyzed for risk, and added to any reports or SBOMs generated as part of the application life cycle. At the end of the software delivery pipeline, the SBOM for the software that has been analyzed can be exported into an industry-standard file format.

Tracking Transitive Dependencies

Transitive dependencies occur when a software component depends on another component, which in turn depends on other components. These dependencies can create complex relationships that are difficult to track manually. SBOMs generated by SCA tools can help teams identify and track all transitive dependencies, providing organizations with a complete picture of their software supply chain.

Addressing Vulnerability, License and IP, and Code Quality Issues

SBOMs enable organizations to proactively identify and address vulnerabilities before a security breach occurs. By analyzing the SBOM, security teams can identify vulnerabilities in third-party components and determine whether they are up-to-date and properly configured. This proactive approach helps reduce the risk of security breaches and ensures that software is built on a solid foundation.

SBOMs can be used to ensure that all components meet compliance requirements and help organizations track the licenses associated with their software components, creating a more open and scalable path for managing license compliance risks and meeting necessary obligations. SBOMs can also help organizations identify outdated or unsupported components that may pose security risks or performance issues. This information enables organizations to prioritize updates. By identifying and addressing code quality issues early on, organizations can reduce the risk of security breaches, improve software performance, and enhance maintainability.

SBOM Adoption

The benefits offered by SBOMs make them an increasingly essential component of secure software development practices. While SBOM adoption is still evolving and the rate of adoption varies across industries and organizations, there is a growing recognition that SBOMs are a valuable tool for managing software supply chain risks and ensuring software security. A 2022 study by the Linux Foundation found that 78% of organizations were expected to produce an SBOM that year. And according to research conducted by Censuswide in 2023, 60% of large enterprises (over $50 million in annual revenue) require an SBOM from their vendors.

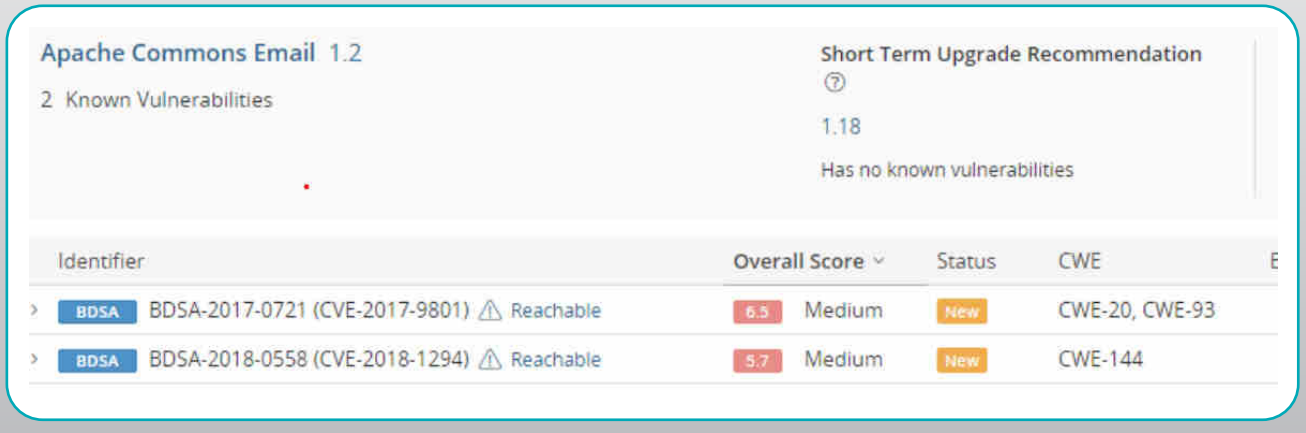
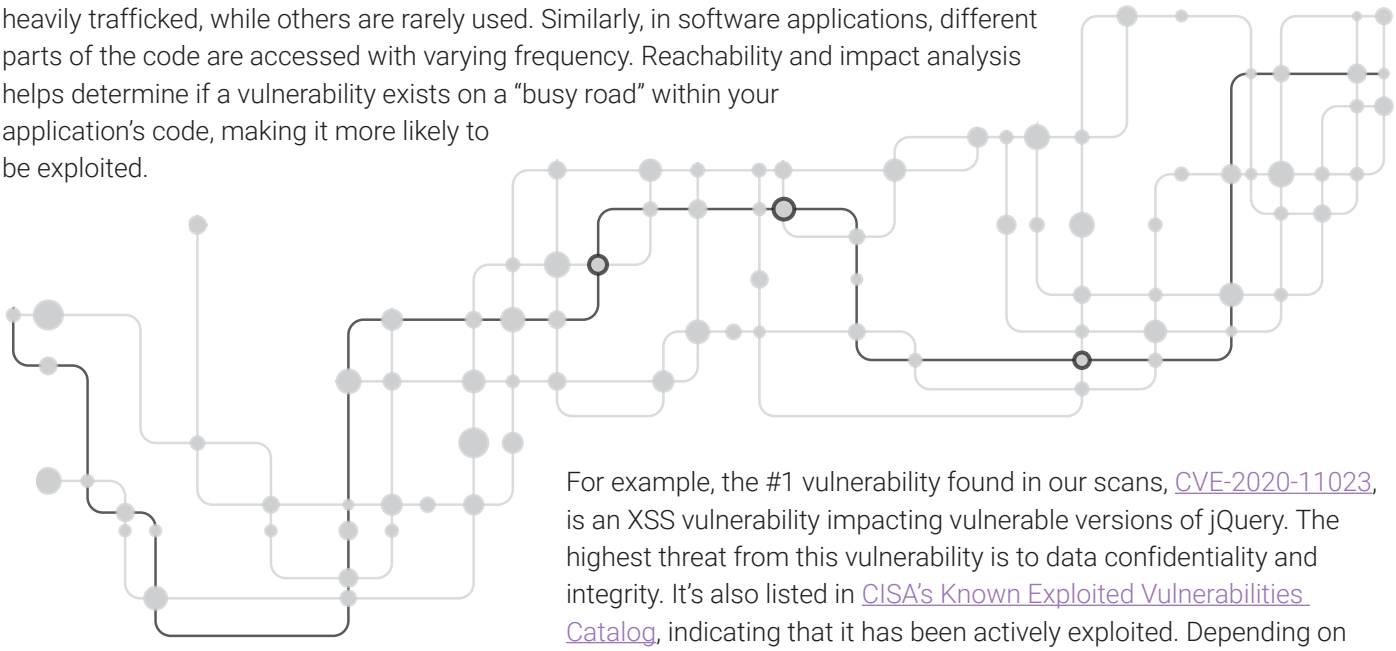


Figure 4: Analyzing the Impact of a Vulnerability with Black Duck SCA

Analyzing the Impact of a Vulnerability

When an SCA tool identifies vulnerabilities in your application, how can you decide which vulnerability to focus on first? While the presence of a vulnerability indicates a potential weakness, it doesn't necessarily mean it can be easily exploited—that is, will be used by malicious actors to compromise systems, applications, or networks.

Imagine a complex network of roads connecting various cities. Some roads might be heavily trafficked, while others are rarely used. Similarly, in software applications, different parts of the code are accessed with varying frequency. Reachability and impact analysis helps determine if a vulnerability exists on a "busy road" within your application's code, making it more likely to be exploited.



For example, the #1 vulnerability found in our scans, CVE-2020-11023, is an XSS vulnerability impacting vulnerable versions of jQuery. The highest threat from this vulnerability is to data confidentiality and integrity. It's also listed in CISA's Known Exploited Vulnerabilities Catalog, indicating that it has been actively exploited. Depending on the specific circumstances, these factors might drive organizations to assign a higher priority to addressing this vulnerability than to other vulnerabilities in their codebases.

Exploitability depends on various factors, including the availability of exploit code, the complexity required to exploit the vulnerability (that is, the level of difficulty an attacker would face), and the potential rewards an attacker might gain from exploitation. For example, the Heartbleed vulnerability, discovered in 2014, was a significant security flaw in the OpenSSL cryptographic library that impacted millions of websites and servers because it was easily exploitable and allowed attackers to steal sensitive information. One notable victim was the Canada Revenue Agency (CRA), which reported that hackers exploited the vulnerability to steal Canadian social insurance numbers. The CRA was forced to shut down its online services temporarily to address the issue and extend tax filing deadlines. CloudFlare, a web security company, estimated that revoking and reissuing SSL certificates for its customers would cost the certificate issuer around $400,000 per month.

How SCA Tools Can Help

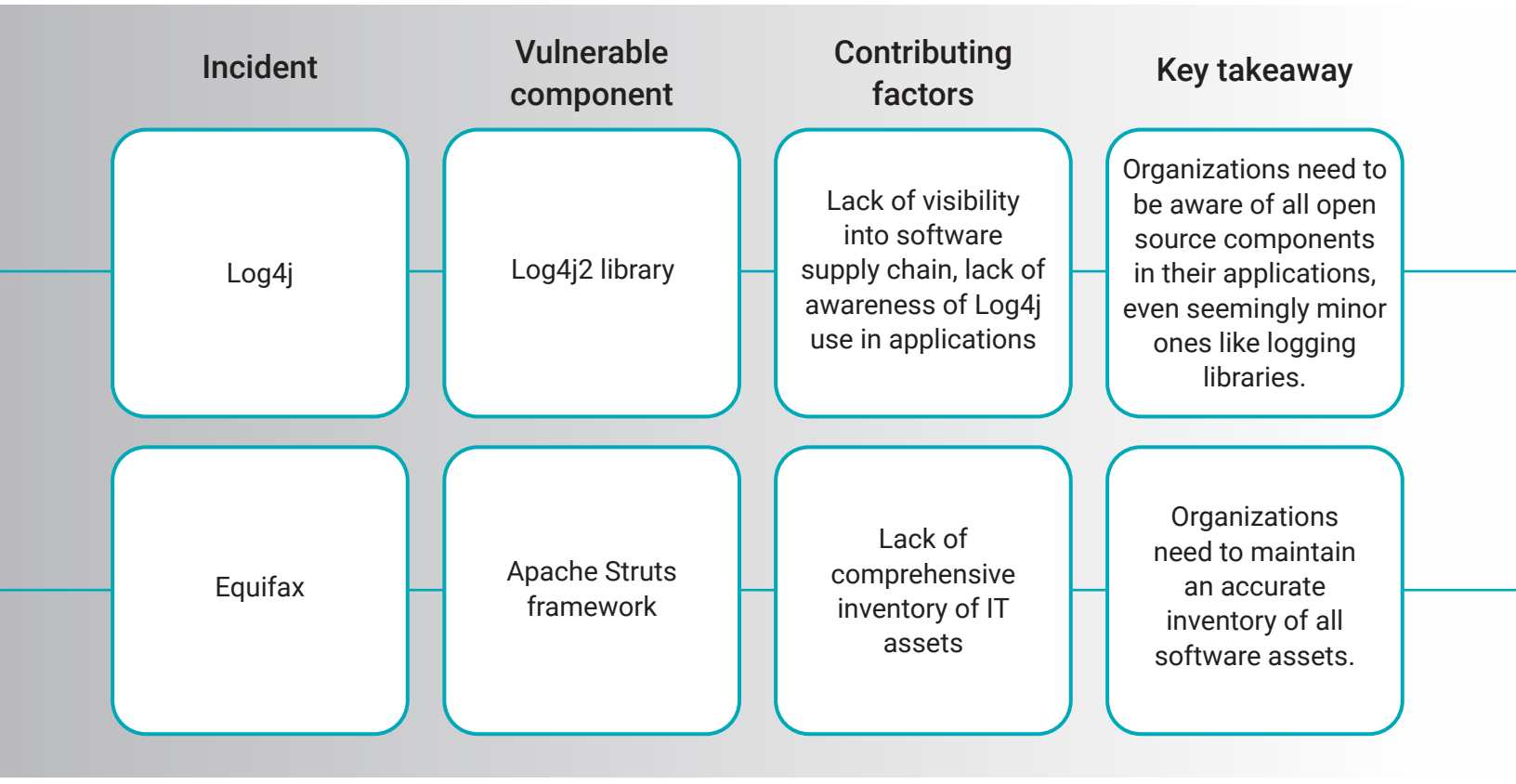
SCA tools can prioritize vulnerabilities based on various factors, including severity, exploitability, and potential impact. This helps organizations focus on the most critical vulnerabilities, optimize their remediation efforts, and reduce the very real problem of alert fatigue by filtering out irrelevant issues. Simply identifying vulnerabilities is insufficient; their sheer scale makes it necessary to have an intelligent way of understanding which ones need to be fixed first.

Black Duck SCA prioritizes vulnerabilities based on factors such as exploitability, remediation guidance, severity scoring, and call path analysis. Black Duck can determine if vulnerable code is more likely to be invoked, and flags those vulnerabilities as reachable, indicating that these vulnerabilities are a higher priority for remediation.

Log4j and Equifax: Two Lessons on the Need for Visibility into Your Code

Although somewhat lost in the contemporary media frenzies surrounding them, the Log4Shell vulnerability of 2021 and the Equifax breach of 2017 are both reminders of the importance of visibility into the open source you're using. In both cases, a lack of awareness of the open source components in use were contributing factors to the severity of the incidents.

In the case of Log4j, many development teams didn't know where, how, or even if their applications were using the open source logging utility, often because it was buried several dependencies deep within an application and invisible to basic code reviews. When CISA issued a directive to federal agencies to locate all instances of the Log4j library in their software, check if their systems were vulnerable to the Log4Shell exploit, and patch affected servers—all within 10 days—many teams found themselves spending their December holidays in the office desperately trying to locate vulnerable versions of Log4j as well as forestalling a potentially catastrophic security breach.



Although the explanation for the Equifax Apache Struts vulnerability exploit in 2017 has been simplified over the years into *the person who was responsible for communicating the need to apply the patch did not communicate to the right level,* the full reasons, as detailed in the Congressional report on the incident, are much more complicated.

Equifax had been on an acquisition spree for several years, with each acquisition adding to the complexity and opacity of the company's technology infrastructure. One part of that patchwork IT infrastructure was a web-based dispute and disclosure application that became the primary target of the breach.

"It is critical for an organization to know what assets are present within its IT environments to make accurate and informed risk determinations—such as when, and how, to patch a vulnerable system."

—The Equifax Data Breach, Majority Staff Report, 115th Congress, December 2018

As Equifax's then-CIO explained in his testimony, Equifax did not have a clear picture of the software used by that application. The lack of visibility into its software inventory was a known issue, documented two years earlier in an Equifax internal audit. As the audit report related, "A comprehensive IT asset inventory does not exist nor does accurate network documentation. [...] The lack of an accurate asset inventory makes it difficult to ensure all assets are adequately patched and configured. [...] Without a firm understanding of the status of all IT assets, ensuring the security and stability of Equifax systems is extremely difficult."

Adding to the problem was that the application was—like many of Equifax's older systems—a "legacy" in IT-speak. By 2017, there were few people left in Equifax who were familiar with that particular web application's inner workings, to the point that the person nominally in charge of its oversight hadn't known it contained Apache Struts software.

The Equifax case illustrates another aspect of the lack of visibility into open source that any business dependent on other companies' code would do well to remember: While you might have a good handle on your own code, do you know what's in theirs?

The Top High- and Critical-Risk Vulnerabilities

Before diving into the specific high-risk vulnerabilities we found in our audits, let's clarify what the terms CVEs, CWEs, and BDSAs mean. A CVE is a standardized identifier for publicly known cybersecurity vulnerabilities. When a vulnerability is discovered, it is assigned a unique CVE ID, which allows security professionals and developers to quickly and easily refer to and track that specific vulnerability. The National Vulnerability Database (NVD) utilizes the CVE standard as its foundation for identifying and describing vulnerabilities.

On the other hand, a CWE is a community-developed list of software and hardware weakness types. CWEs serve as a common language for describing security weaknesses, aiding in their identification, mitigation, and prevention. As Figure 5 shows, reviewing a list of the most common CWEs behind the vulnerabilities found in our scans can be instructive.

For example, given that over 70% of the overall open source vulnerabilities we found were linked to improper input validation (which could lead to injection and XSS exploits), development and security teams might want to focus their efforts on using appropriate validation techniques for their own code, and implement a regimen of regular testing of third-party code with SAST and DAST to catch vulnerabilities early and continuously.

CWE	Percentage of codebases with vulns linked to CWE	Description
CWE-20	71%	**Improper Input Validation:** The software does not validate or incorrectly validates input that can affect the control flow or data flow of the program. This can lead to vulnerabilities like buffer overflows, SQL injection, and cross-site scripting.
CWE-400	70%	**Uncontrolled Resource Consumption:** The software does not properly control the allocation and release of system resources, such as memory, CPU time, or disk space. This can lead to DoS attacks.
CWE-200	60%	**Exposure of Sensitive Information to an Unauthorized Actor:** The software exposes sensitive information, such as passwords, credit card numbers, or personal data, to unauthorized actors. This can happen through various means, such as insecure storage, unencrypted transmission, or improper access controls.
CWE-79	56%	**Improper Neutralization of Input During Web Page Generation (Cross-Site Scripting):** The software does not neutralize or incorrectly neutralizes user-supplied input before including that input in an HTML page. This allows attackers to inject malicious scripts that can steal user data or take control of their browser.
CWE-185	48%	**Incorrect Regular Expression:** The software specifies a regular expression in a way that causes data to be improperly matched or compared. Regular expressions should be subjected to thorough testing techniques such as equivalence partitioning, boundary value analysis, and robustness testing.
CWE-770	48%	**Allocation of Resources Without Limits or Throttling:** The software allocates resources without limits or throttling, which can allow an attacker to consume excessive resources and cause a DoS.
CWE-80	44%	**Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS):** Similar to CWE-79, this weakness specifically focuses on the failure to neutralize HTML tags that can be used to inject scripts.
CWE-1321	39%	**Improperly Controlled Modification of Object Prototype Attributes:** The software allows attackers to modify the prototype of an object, which can affect all instances of that object and lead to unexpected behavior or privilege escalation.
CWE-1333	36%	**Inefficient Regular Expression Complexity:** The software uses a regular expression that is too complex, which can lead to excessive CPU consumption and DoS attacks.
CWE-502	31%	**Deserialization of Untrusted Data:** The software deserializes untrusted data, which can allow attackers to execute arbitrary code or manipulate application logic.

Figure 5: Top CWEs Found in Codebase Scans

The Black Duck Security Advisories (BDSAs) noted in this report are a Black Duck-exclusive vulnerability data feed sourced and curated by our CyRC. BDSAs offer deeper coverage for a wide set of vulnerabilities than is available through the NVD. While providing more timely and detailed vulnerability insights, including severity, impact, and exploitability metrics, BDSAs also provide actionable remediation guidance to save time by providing details on fixed versions, patch information, exploits, and workarounds where available.

The CyRC team provides detailed vulnerability guidance over and beyond what the NVD typically provides in CVE records. BDSAs are also cross-checked and validated against possibly affected component versions, resulting in additional and more accurate mappings for components and versions affected by a given vulnerability.

Let's take a closer look at the top high- and critical-risk open source vulnerabilities we found in our scans.

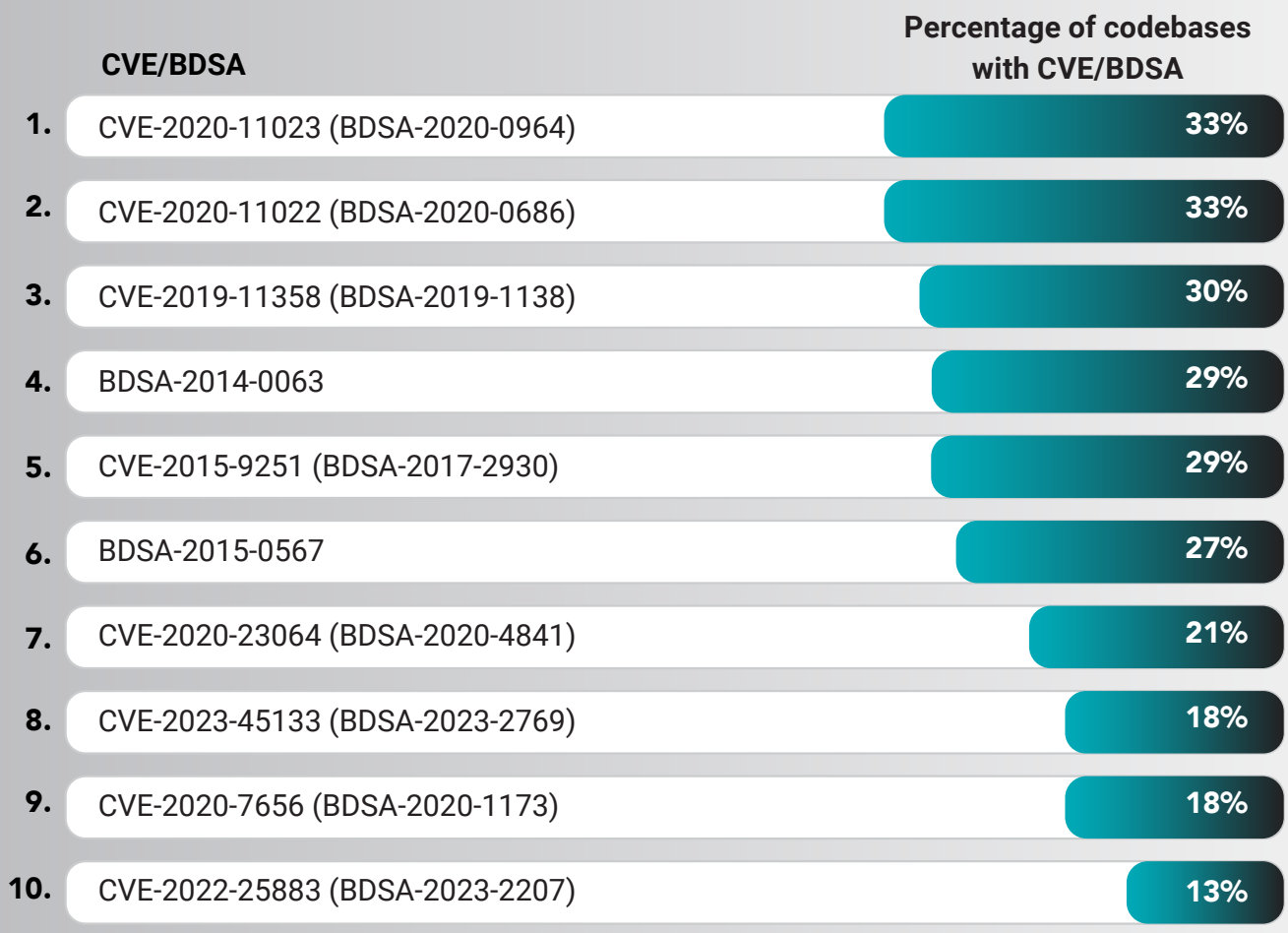


Figure 6: Top High- and Critical-Risk Vulnerabilities Found

1. CVE-2020-11023

CVE-2020-11023 was found in a third (32.6% to be exact) of the scanned codebases. It is an XSS vulnerability that affects jQuery versions greater than or equal to 1.0.3 and before 3.5.0 (at the time this report was released, the current stable release of jQuery was 3.7.1, released in August 2023). Notably, CVE-2020-11023 is listed in CISA's Known Exploited Vulnerabilities Catalog, indicating that it has been actively exploited.

The vulnerability allows untrusted code to be executed by manipulating how jQuery handles HTML containing <option> elements from untrusted sources. This vulnerability has a wide-ranging impact, affecting various systems that utilize jQuery including Debian Linux, Fedora, Drupal, and Oracle products.

The CWE associated with CVE-2020-11023 is CWE-79: Improper Neutralization of Input During Web Page Generation (Cross-Site Scripting). This CWE, as we'll see with many other CVEs in this list, highlights a common security issue in which user-supplied input is not properly sanitized before being included in web pages. This failure to neutralize potentially malicious input can allow attackers to inject and execute malicious scripts in users' browsers.

2. CVE-2020-11022

CVE-2020-11022 is another XSS vulnerability in jQuery, affecting versions greater than or equal to 1.2 and before 3.5.0. This flaw enables attackers to inject JavaScript code into web pages by exploiting how jQuery's DOM manipulation methods handle input. Unlike CVE-2020-11023, this vulnerability is not listed in CISA's Known Exploited Vulnerabilities Catalog, suggesting it might pose a lower immediate risk.

The CWE associated with CVE-2020-11022 is also CWE-79: Improper Neutralization of Input During Web Page Generation (Cross-Site Scripting).

3. CVE-2019-11358

CVE-2019-11358 involves prototype pollution in jQuery versions before 3.4.0. Prototype pollution occurs when an attacker can manipulate the prototype of an object, potentially affecting all objects that inherit from that prototype. This can lead to unexpected behavior, denial of service, or even arbitrary code execution.

The CWE associated with CVE-2019-11358 is CWE-1321: Improperly Controlled Modification of Object Prototype Attributes (Prototype Pollution). This CWE is a child of CWE-913: Improper Control of Dynamically Managed Code Resources, which broadly categorizes weaknesses related to managing code resources during program execution.

4. BDSA-2014-0063

This is an older vulnerability, first raised as an issue in January 2014 and relating to a potential XSS vulnerability in jQuery caused by a lack of user-supplied input validation. It does not have an associated CVE.

The vulnerability could allow an attacker to inject arbitrary web scripts and steal a victim's session cookies. The vulnerability was mitigated in jQuery 3.0.0-rc1. However, the mitigation does not sanitize malicious input and will still allow scripts to be executed. The default behavior of the parser is changed such that if the context is unspecified or given as null/undefined, a new document is created. This delays execution of parsed HTML until it is injected into the document, allowing the opportunity for tools to traverse the created DOM and remove unsafe content after the function call.

As with several other of our top 10 vulnerabilities, the CWE associated with BDSA-2014-0063 is also CWE-79: Improper Neutralization of Input During Web Page Generation (Cross-Site Scripting).

5. CVE-2015-9251

CVE-2015-9251 is an XSS vulnerability in jQuery affecting versions before 3.0.0. (At the time this report was released, the current stable release of jQuery was 3.7.1, released in August 2023.)

This vulnerability arises when cross-domain AJAX requests are made without specifying the dataType option, allowing malicious JavaScript responses to be executed. While the primary CWE associated with CVE-2015-9251 is CWE-79: Improper Neutralization of Input During Web Page Generation (Cross-Site Scripting), it's worth noting that it is also related to CWE-94: Improper Control of Generation of Code (Code Injection) and CWE-74: Improper Neutralization of Special Elements in Output Used by a Downstream Component (Injection).

6. BDSA-2015-0567

This is another older vulnerability without a CVE, this time with jQuery vulnerable to arbitrary code execution. Versions of jQuery that use an unpatched UglifyJS parser are vulnerable to arbitrary code execution through crafted JavaScript files. Ultimately, this can allow attackers to run rogue code. The vulnerability was fixed in 1.12.0 and in 2.2.0. The CWE associated with this BDSA is CWE-1395: Dependency on Vulnerable Third-Party Component.

7. CVE-2020-23064

CVE-2020-23064 is an XSS vulnerability found in jQuery versions 2.2.0 through 3.x before 3.5.0. This vulnerability allows attackers to execute arbitrary code by exploiting the handling of the <options> element. It has a CVSS v3 base score of 6.1, indicating a medium severity level.

The CWE associated with CVE-2020-23064 is CWE-79: Improper Neutralization of Input During Web Page Generation (Cross-Site Scripting).

8. CVE-2023-45133

CVE-2023-45133 is a vulnerability in Babel, a popular JavaScript compiler. It affects the @babel/traverse package and all versions of babel-traverse. This vulnerability can lead to arbitrary code execution during the compilation process, when handling malicious code crafted to exploit this flaw.

CVE-2023-45133 is associated with two CWEs: CWE-697: Incorrect Comparison and CWE-184: Incomplete List of Disallowed Inputs. These CWEs indicate that the vulnerability stems from improper validation and handling of specific inputs during the compilation process.

9. CVE-2020-7656

CVE-2020-7656 is yet another XSS vulnerability in jQuery, affecting versions prior to 1.9.0. This vulnerability occurs because the load method fails to properly handle and remove <script> HTML tags that contain whitespace characters, potentially allowing malicious scripts to be executed.

The CWE associated with CVE-2020-7656 is CWE-79: Improper Neutralization of Input During Web Page Generation (Cross-Site Scripting).

10. CVE-2022-25883

CVE-2022-25883 is a regular expression denial-of-service (ReDoS) vulnerability in the semver package used in certain Node.js systems. This vulnerability affects specific versions of the package and can be triggered when untrusted user data is processed as a range. ReDoS attacks exploit vulnerabilities in regular expressions, causing excessive processing time and potentially leading to denial of service by consuming system resources.

The CWE associated with CVE-2022-25883 is CWE-1333: Inefficient Regular Expression Complexity. This vulnerability can significantly impact system performance and stability, potentially disrupting services and applications.

What the Data Tells Us

Understanding CWEs is crucial for both developers and security professionals. CWEs provide a standardized way to categorize and describe software weaknesses, enabling better communication and collaboration in addressing security risks. By understanding common weaknesses, developers can implement secure coding practices to prevent vulnerabilities, and security teams can effectively identify and mitigate potential threats.

The prevalence of CWE-79 in our results, and all the CVEs related to cross-site scripting exploits of jQuery vulnerabilities, highlights the critical importance of input validation in web development. Failing to properly sanitize user input can have serious consequences.

jQuery is not inherently insecure. In fact, it is a well-maintained open source library with a large community of users, developers, and maintainers. But according to our audits, jQuery was the component most likely to have vulnerabilities—indeed, nearly a third of all the codebases we scanned were found to have vulnerable jQuery components—even though each of those vulnerabilities impacted outdated versions of jQuery and had available patches. It is important for users of jQuery—and indeed of all open source—to be aware of the potential security risks associated with outdated versions of software, and to take steps to address those risks.

For developers, our data emphasizes the need to prioritize input validation and sanitization techniques to prevent cross-site scripting and other injection attacks. Utilizing security analysis tools such as Coverity® Static Analysis and Black Duck® Continuous Dynamic (a production-safe DAST tool) to identify potential vulnerabilities arising from inadequate checks on user-submitted data, like form inputs or API parameters, can help ensure that only expected data formats and values are accepted, thereby mitigating risks like SQL injection, cross-site scripting, and other injection attacks.

Staying up-to-date with security advisories and promptly patching vulnerable software is essential to minimizing the risk of exploitation. Regularly updating libraries and frameworks, such as jQuery and Babel, is crucial to ensuring that systems are protected against known vulnerabilities.

Industry-Specific Insights

As shown in Figure 1 on page 5, from a vulnerability perspective, high-risk industry sectors include Internet and Mobile Apps (100% of codebases scanned from this sector contained high-risk vulnerabilities), Marketing Tech (88% of codebases contained high-risk vulnerabilities), Computer Hardware and Semiconductors (87% of codebases contained high-risk vulnerabilities), and the EdTech and Enterprise Software/SaaS sectors (86% of codebases contained high-risk vulnerabilities).

It's worth noting that the lowest percentage of codebases containing high-risk vulnerabilities of all 16 sectors was 60% in the Energy/Clean Tech industry. Overall, well over 50% of each respective industry's codebases contained high-risk vulnerabilities, making them attractive targets of opportunity for exploitation.

Open Source Licensing

> *"To be a programmer requires that you understand as much law as you do technology."*
>
> *—Eric Allman*

All Black Duck audits examine open source license compliance. During 2024, the Black Duck Audit team conducted 965 audits. The data in this section is based on those assessments.

- Percentage of codebases with license conflicts: 56%
- Percentage of codebases containing open source with no license or a custom license: 33%

Effective open source management requires licensing as well as security compliance. You know the open source components and libraries you're using are governed by licenses, but do you know those license details? Perhaps more importantly, are your executives and legal counsel aware of those details in relation to your proprietary software?

Even one noncompliant license in your software could result in legal issues, loss of lucrative intellectual property, time-consuming remediation efforts, and even delays in getting a product to market. In the case of our audit results, 56% of customer codebases had license conflicts, opening them up to those potential scenarios.

How Conflicts, Variants, and Lack of Licenses Create Risk

In the U.S. and other jurisdictions, creative work (including software) is protected by exclusive copyright by default. No one can legally use, copy, distribute, or modify that software without explicit permission from the creator/author in the form of a license that grants the right to do so.

In the context of open source software, a declared license conflict arises when the license of an open source component clashes with the overall license declared for the entire project or codebase. This often happens when a component with a restrictive license, like the GNU General Public License (GPL), is included in a commercial project, potentially requiring the entire project's source code to be released. The severity of a declared conflict can vary; it might apply to the whole project or just specific files, depending on the scope of the license. A component license conflict occurs when two open source licenses within a project are incompatible with each other.

Because of the way most open source licenses are written, conflicts can arise even when only a small piece of licensed code—a snippet—is included in a larger work. Historically, this has occurred when a developer cut and pasted from an open source project with a problematic license. Today, with the rise of generative AI models trained on open source, a snippet may have been appropriated by an AI tool without regard for licensing.

Variants or customized versions of standard open source licenses can also place undesirable requirements on the licensee and require legal evaluation for possible IP issues or other implications. The JSON license is often used as an example of a customized license. Based on the permissive MIT license, the JSON license adds the restriction that "The software shall be used for good, not evil." While laudable, the ambiguity of this statement leaves its meaning up to interpretation, and many counselors would advise against using software so licensed, especially in the context of M&A scenarios. Since 2016, the Apache Foundation has disallowed software with this license to be used in any of its projects.

Thirty-three percent of the 2024 audited codebases were using code with either no discernible license or a customized license. It's not uncommon for developers to make code publicly available that has code without discernable terms of service or mention of software terms. It's also not unusual for developers to grant permission to use their code either by modifying or augmenting standard license terms—"good, not evil" for example—or adding usage, obligation, and restriction terms in their code comments. Often, these types of modifications require legal review.

The Impact of Transitive Dependencies on License Conflicts

Nearly 30% of component license conflicts found in our audits were caused by transitive dependencies (that is, a component needed by a direct dependency and the overall software to function). If a transitive dependency uses a strong restrictive license like the GPL, it can potentially affect the licensing of the entire application, even if the direct dependency has a more permissive license. Many restrictive licenses often require derivative works to also be licensed under the same terms.

The Top 10 Open Source Licenses of 2024

License	Percentage of scanned codebases containing license	Risk*	OSI approved
MIT License	92%	Low	Yes
Apache License 2.0	90%	Low	Yes
BSD 3-Clause "New" or "Revised" License	85%	Low	Yes
BSD 2-Clause "Simplified" License	74%	Low	Yes
ISC License	61%	Low	No
Generic Public Domain	57%	Varies by Usage	Yes
GNU Lesser General Public License v2.1 or Later	48%	High	Yes
The Unlicense	47%	Low	Yes
Creative Commons Zero v1.0 Universal	46%	Varies by Usage	No
Mozilla Public License 2.0	45%	Medium	Yes

* Risk classifications are guidelines and should not be used for decisions about using the open source software. Consult your corporate policies and/or legal teams for guidance regarding license compliance.

Figure 7: Top 10 Open Source Licenses

What Are Permissive, Weak Copyleft, and Reciprocal Open Source Licenses?

Low-Risk: Permissive Licenses

Permissive licenses generally do not have many limiting conditions. Rather, they usually require that you keep the copyright notice in place when you distribute your own software. This means you can use and change the open source software if you keep the copyright notices intact. MIT and Apache licenses, the two most popular licenses currently in use, are in this category. We rate permissive licenses as low-risk licenses.

Medium-Risk: Weak Copyleft Licenses

Copyleft licenses generally include a reciprocity obligation stating that modified and extended versions are released under the same terms and conditions as the original code. Weak copyleft licenses usually require you to make any modifications to the source code available under the same terms of the given license. Some of these licenses explicitly define what a modification is. For instance, a license might cite copying unmodified open source code into proprietary code as a modification. To comply with the license obligations, you would have to release the source code (original, modified, and newly added). Popular open source licenses in this category include the Mozilla public license. We rate semipermissive licenses as medium-risk licenses.

High-Risk: Reciprocal/Copyleft Licenses

Some popular open source licenses, such as the GNU General Public License v2.0 or later and GNU Lesser General Public License v3.0 or later, are quite restrictive. Depending on how you integrate open source software with your proprietary software, you may face significant risk. In the worst-case scenario, you may be required to release your proprietary software under the same license—royalty-free. We rate restrictive licenses as high-risk licenses.

How to Manage Open Source License Risk with SCA

If you build packaged, embedded, or commercial SaaS software, open source license compliance should be a key concern for your organization. You need to determine the license types and terms for the open source components you use and ensure that they are compatible with the packaging and distribution of your software. Even companies whose software is not a commercial product and only used internally are still subject to the license terms of the open source components in their software.

The first step to managing risk is using an automated SCA tool to create an up-to-date, accurate SBOM of all open source components in your software, the versions in use, and their associated licenses. Compile the license texts associated with those components so that you can flag any components not compatible with your software's distribution and license requirements, or not compatible with licenses that may be used by other components in your software. It is important to ensure that the obligations of all licenses have been met, as even the most permissive open source licenses still contain an obligation for attribution.

Black Duck SCA enables development, security, and compliance teams to manage the risks that come from the use of open source. Black Duck's multifactor open source detection and KnowledgeBase of over 7.8 million components can provide an accurate SBOM, including licensing information, for any application or container. And although most open source components use one of the most popular licenses, Black Duck provides an extra layer of information with data on over 2,500 other open source licenses that could potentially impose restrictions on the software your team writes. Tracking and managing open source with Black Duck helps you avoid license issues that can result in costly litigation or compromise your valuable intellectual property.

Industry Perspectives on License Conflicts

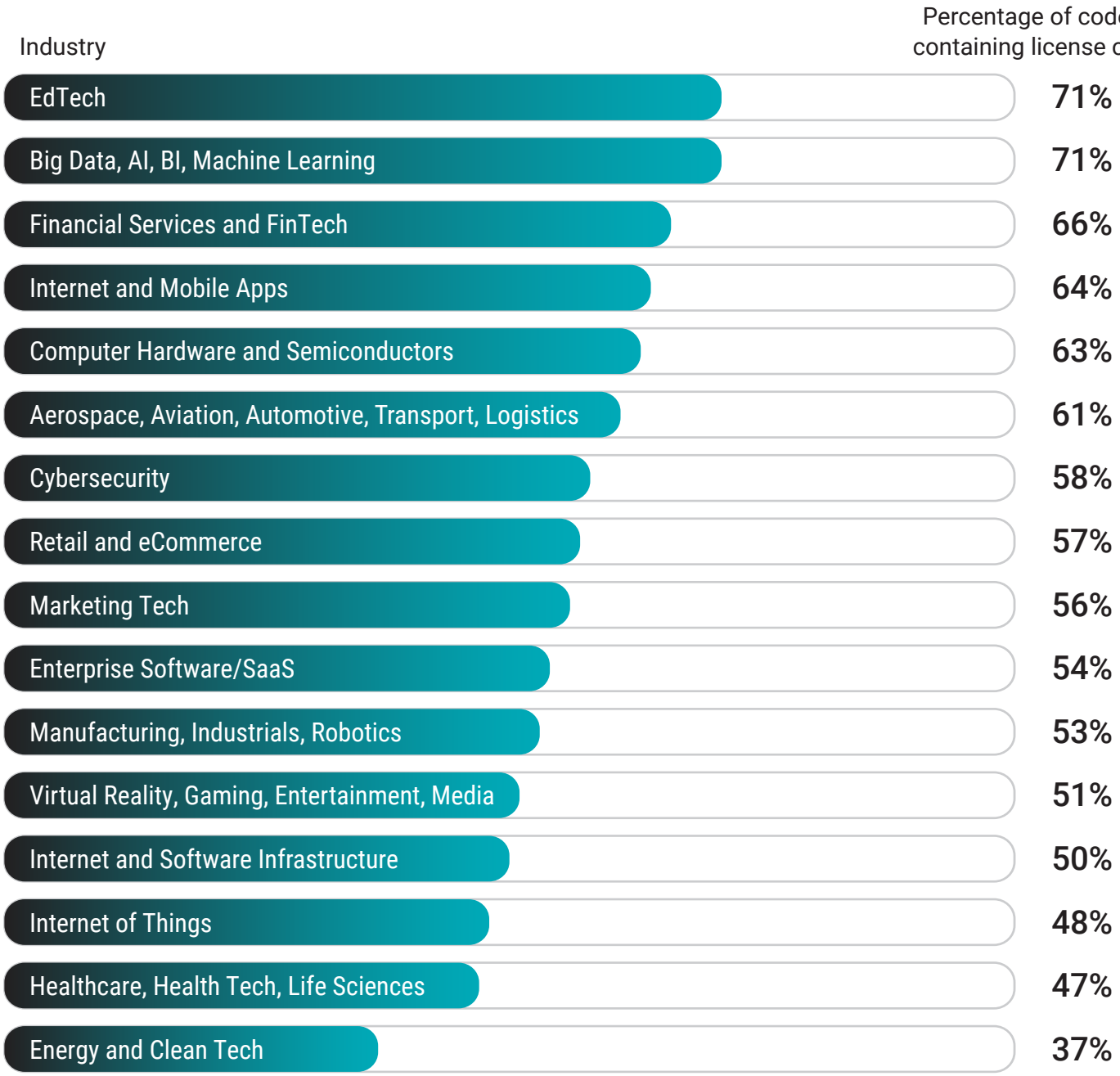


Figure 2: Codebases Containing License Conflicts by Industry

As noted on page 6 of this report, Figure 2 (reproduced above) highlights the prevalence of license conflicts within codebases across various industries. It should serve as a signal to businesses on the importance of proactive license identification to avoid costly legal and operational challenges arising from license conflicts within their software.

High-Risk Sectors

- Tech-heavy industries like Big Data and AI, Financial Services and FinTech, and Computer Hardware exhibit a significantly higher percentage of codebases with license conflicts. This is likely due to these industries' heavy reliance on software and services, and those applications themselves relying on open source components.
- Many of these industries also tend to license and distribute their software as on-premises products. Most restrictive licenses apply specifically to software that is distributed in this manner. Other industries with lower numbers may do more subscription-based or SaaS-type deployments, which are not traditionally considered "distributing" and are not subject to the same license terms.
- EdTech also shows a surprisingly high percentage, indicating potential licensing issues within educational software and platforms. Thanks to online learning and digital educational tools, the EdTech sector has experienced rapid growth over the last several years. Many EdTech companies, especially startups and smaller organizations, also have limited resources and expertise focused on software licensing.

Moderate Risk

- Sectors including Aerospace, Cybersecurity, Manufacturing, and Enterprise Software demonstrate a moderate level of risk. While the percentages are lower than the high-risk group, they still have a considerable chance of encountering licensing problems.

Lower Risk (But Not Risk-Free)

- Industries like Healthcare and Energy show a lower percentage of license conflicts. However, this doesn't imply that they are immune to such issues. For example, Healthcare organizations rely on a wide range of software, from electronic health records and medical imaging systems to telehealth platforms and AI-powered diagnostic tools. This intricate ecosystem often involves integrating numerous third-party components and libraries, increasing the possibility of licensing issues.

If You Anticipate an M&A

If your company plans to be involved with an M&A transaction at some point, either as seller or buyer, you will want to involve your organization's IP counsel or seek outside legal advice, as understanding licensing terms and conditions and identifying conflicts among various licenses can be challenging. It's vital to get this right the first time—especially if you build packaged or embedded software—because license terms are often more explicit for shipped software and harder to mitigate after the fact. Knowing what open source code is in a company's codebase is crucial for properly managing its use and reuse, ensuring compliance with software licenses, and staying on top of patching vulnerabilities—all essential steps in reducing business risk.

If you're on the buy side of a tech M&A transaction, an open source audit should be part of the software due diligence process. A code audit enables a buyer to understand risks in the software that could affect the value of the intellectual property, and the remediation required to address those risks. An open source audit can also be invaluable for companies wanting a better understanding of the code's composition. For example, using a range of tools such as Black Duck SCA, expert auditors comprehensively identify the open source components in a codebase and flag legal compliance issues related to those components, prioritizing issues based on their severity.

An audit uncovers known security vulnerabilities that affect open source components, as well as information such as versions, duplications, and the state of a component's development activity. It also provides clues as to the sophistication of a target's software development processes. Open source is so ubiquitous today that if a company isn't managing that part of software development well, it raises questions about how well it is managing other aspects.

Acquirers need to identify problematic open source in the target's code before the transaction terms are set, and a trusted third-party audit is the best way to get a deep, comprehensive view. Identifying even permissively licensed open source is valuable, as acquirers will want to ensure they will be able to comply with the attribution requirements of those licenses. Sellers should prepare for questions about the composition of their code and how well they have managed open source security and license risk. Proactive sellers may employ an audit in advance to avoid surprises in due diligence, particularly given the amount of unknown open source in a typical company's code.

By identifying open source code and third-party components and licenses, an open source audit can alert your firm to potential legal and security issues in an M&A transaction.

The bottom line is that significant monetary and brand risk can be buried in the open source components of acquired code. Evaluating that risk as part of an acquirer's due diligence must be part of the decision-making process in an M&A transaction.

Maintenance and Operational Factors Impacting Risk

Ideally, all of us would use only open source components sustained by robust communities. After all, support from large and vibrant developer communities was one of the key benefits of OSS promised by open source champions when the software was first introduced. Dedicated communities of developers would deliver enhanced code quality and security while fostering regular improvements to the projects they were overseeing.

Unfortunately, that scenario never happened for many open source projects. As the Linux Foundation's Census III of Free and Open Source Software report (utilizing SCA data from Black Duck among other vendors) relates, much of the most widely used open source today is developed and maintained by only a handful of contributors, not the thousands or millions of developers popularly thought to be working behind the scenes. In reality, the small number of contributors working to ensure updates—including feature improvements as well as security and stability updates—decreases over time on almost all OSS projects.

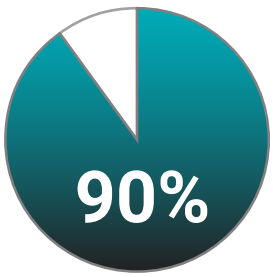
When maintainers have stopped maintaining a project, one consequence is elevated security risk, as the data from our scans shows.

Outdated Components

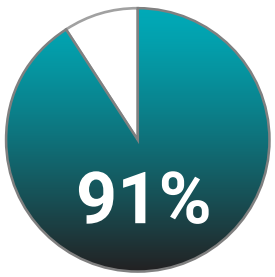
- A very high percentage of codebases—90%—contain open source components that are more than four years out-of-date. This indicates a widespread issue of outdated dependencies and could lead to security vulnerabilities and compatibility issues.

Inactive Components

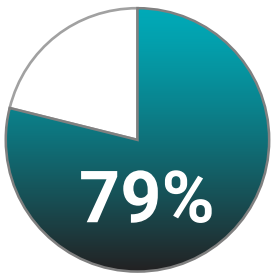
- An equally high portion of codebases—91%—have components that have not seen new development in the past two years. This suggests that many applications and web services are relying on OSS that no longer receive updates, potentially leaving them vulnerable to undiscovered or unpatched security flaws.
- Seventy-nine percent of codebases contain components with no activity for the last 24 months, while still using the latest version of the component. This suggests that even up-to-date components are not being actively maintained.
- Eighty-eight percent of codebases have components with no activity for the last 24 months and are ***not*** using the latest version of the component—an even riskier prospect for those using the components.



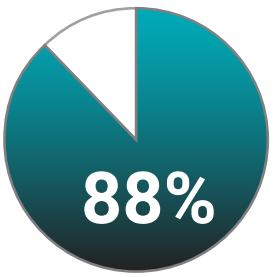
of codebases contain open source components that are more than four years out-of-date



of codebases have components that have not seen new development in the past two years



of codebases contain components with no activity for the last 24 months, while still using the latest version of the component



of codebases have components with no activity for the last 24 months and are not using the latest version of the component

Version Lag

- The majority of codebases—91%—include OSS that is not the latest available version of that particular component.
- Worse, 90% of codebases contain open source components that are more than 10 versions behind the most recent release

A failure to keep up with current releases, which often include important bug fixes and security patches, increases risk and technical debt.

There can be valid justifications for not keeping an open source component up-to-date. Major version updates can introduce significant changes that might break your existing code, especially if you've already fallen several versions behind. Sometimes the effort required to adapt your code isn't feasible. Updating can be time-consuming, requiring development time, testing, and deployment. Smaller teams or projects with limited resources might need to prioritize more critical tasks.

But, as we've noted in a decade of publishing OSSRA reports, open source is different from commercial software—not worse, not better, but different—and it requires different techniques when it comes to maintenance. For example, all organizations that use commercial software are familiar with patches and updates being "pushed" to their software, or at a minimum, receiving a notice from the vendor that an update is available for download. That's seldom the case with open source, where users are largely left to their own initiative to stay aware of a component's status.

Given that reality, how *do* you stay aware of updates?

Follow project websites and repositories: Most open source projects have websites, blogs, or repositories (like GitHub) where they announce new releases and provide changelogs. Subscribe to their mailing lists or RSS feeds to stay informed. However, with the number of open source components in a typical application today, manually tracking is impractical, so automated approaches become more critical.

Use package managers: Package managers (like npm, pip, or Bundler) often provide notifications about available updates and can automate the update process. The vast majority of open source identified in our scans originated from npm, which provides a variety of tools to upgrade packages. For example, running ***npm outdated*** will generate a list of packages that have available updates.

Utilize version tracking tools: Tools like Dependabot or Renovate can monitor your project's dependencies and automatically create pull requests with updates.

Use identification and monitoring tools: Security tools like Black Duck SCA can scan your codebase for vulnerabilities in open source components and alert you when updates with security fixes are available.

There's only one viable solution to stay aware of the open source you use. You need an accurate, comprehensive inventory of open source, as well as automated processes to monitor vulnerabilities, upgrades, and the overall health of the open source in your software.

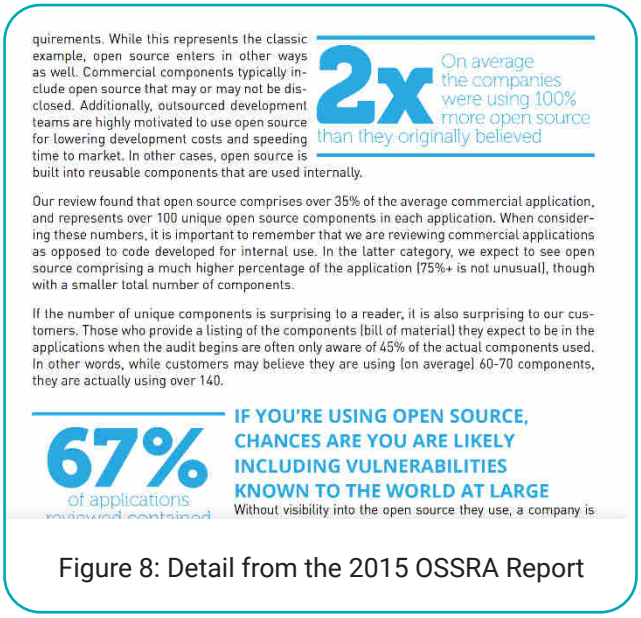
Conclusion: The More Things Change

> *"Risk comes from not knowing what you're doing."*
>
> *—Warren Buffet and Charles Munger*

For the past 10 years, a continuing theme of Black Duck's "Open Source Security and Risk Analysis" report has been "Do you know what's in your code?" The numbers have changed since 2015—in most instances, they have significantly increased—but the question remains the same.

Whatever end of the software supply chain you reside on—whether you're at the top or bottom of the funnel, whether your organization develops or uses software from different vendors, whether that software is on-premises, in the cloud, embedded, or on a mobile device—it's a near-certainty that your software contains open source code. Do you know exactly what those OSS components are and whether they pose security, code quality, or license risks?

When 97% of code contains open source, visibility into your code needs to be a priority. When 91% of codebases are using open source far behind the current version, everyone needs to do better in keeping their code up-to-date, especially when it comes to popular open source components.


quirements. While this represents the classic example, open source enters in other ways as well. Commercial components typically include open source that may or may not be disclosed. Additionally, outsourced development teams are highly motivated to use open source for lowering development costs and speeding time to market. In other cases, open source is built into reusable components that are used internally.

2x On average the companies were using 100% more open source than they originally believed

Our review found that open source comprises over 35% of the average commercial application, and represents over 100 unique open source components in each application. When considering these numbers, it is important to remember that we are reviewing commercial applications as opposed to code developed for internal use. In the latter category, we expect to see open source comprising a much higher percentage of the application (75%+ is not unusual), though with a smaller total number of components.

If the number of unique components is surprising to a reader, it is also surprising to our customers. Those who provide a listing of the components (bill of material) they expect to be in the applications when the audit begins are often only aware of 45% of the actual components used. In other words, while customers may believe they are using (on average) 60-70 components, they are actually using over 140.

67% of applications reviewed contained

IF YOU'RE USING OPEN SOURCE, CHANCES ARE YOU ARE LIKELY INCLUDING VULNERABILITIES KNOWN TO THE WORLD AT LARGE

Without visibility into the open source they use, a company is

Figure 8: Detail from the 2015 OSSRA Report

- **OSS is overwhelmingly present in modern software:** Ninety-seven percent of the commercial codebases we evaluated contained open source, with some industry codebases reaching 100%. Is your company in one of those high-risk sectors?
- **You can't manage open source manually:** The number of open source files in the average application has tripled in the last four years. Transitive dependencies are a major factor in code complexity. Sixty-four percent of open source components identified in our scans were transitive dependencies.
- **Security vulnerabilities are a pervasive risk:** The majority (81%) of assessed codebases had high- or critical-risk vulnerabilities. Many of these vulnerabilities stem from outdated open source components.
- **Many of those vulnerabilities also stem from specific coding weaknesses:** Seventy-one percent of the overall open source vulnerabilities we found were linked to improper input validation.
- **Software Bills of Materials are essential for visibility into your code**: SBOMs are critical for managing risk, vulnerabilities, license compliance, software quality, and M&A due diligence.
- **Licensing risks are a common issue:** More than half of audited codebases (56%) contained license conflicts, often due to incompatible transitive dependencies. Thirty-three percent of all codebases had OSS components with no license or a customized license.
- **Outdated components present a major challenge:** Most audited codebases (91%) contain outdated components, with 90% of the codebases containing components more than 10 versions behind the most current version.

Key Recommendations

Implement SCA: Use SCA tools to generate SBOMs, identify vulnerabilities, and manage license compliance.

Prioritize risk management: Focus on high-risk vulnerabilities and license issues that can impact the most important aspects of your business.

Regularly update OSS: Stay up-to-date with security advisories, and promptly patch vulnerable software, particularly jQuery and other popular libraries.

Establish secure coding practices: Focus on input validation, sanitization, and regular security testing of third-party code.

Monitor OSS maintenance: Stay aware of updates to open source components by tracking project websites, using package managers, and utilizing automated security services.

Create an SBOM: Develop a detailed SBOM that lists all open source components in your code, including licenses, versions, and provenance.

Integrate OSS management into your SDLC: Incorporate open source management into your secure software development framework, following best practices such as those outlined by CISA and NIST.

If you're planning an M&A, utilize Black Duck audits to vet your acquisitions: You need a trusted third party with access to the target's source code and the tools and expertise to provide the necessary insights in these high-risk situations.

As we wrote at the beginning of this report, while open source software offers numerous benefits, it also introduces significant risks that must be actively managed. Organizations need comprehensive visibility into their software supply chains, robust security practices, and a proactive approach to licensing and maintenance to avoid potential issues. Implementing SCA tools, SBOMs, and proper hygiene practices is not optional—it's a necessity in today's software landscape. By adopting our recommendations, organizations can mitigate risks and continue to leverage the benefits of open source software safely and effectively.

You need to know *without question* what's in your code.

Contributors

The 2025 "Open Source Security and Risk Analysis" report was produced by Black Duck, with contributions from our Audit, CyRC, Professional Services, and Marketing teams.

Special thanks to Nancy Bernstein, Conor Brolly, Kevin Collins, Scott Handy, Clement Pang, Merin McDonell, Mike McGuire, Phil Odence, Liz Samet, Mark Van Elderen, and Jack Taylor.

Fred Bals scripsit hoc

About Black Duck

Black Duck® offers the most comprehensive, powerful, and trusted portfolio of application security solutions in the industry. We have an unmatched track record of helping organizations around the world secure their software quickly, integrate security efficiently in their development environments, and safely innovate with new technologies. As the recognized leaders, experts, and innovators in software security, Black Duck has everything you need to build trust in your software. Learn more at www.blackduck.com.

©2025 Black Duck Software, Inc. All rights reserved. Black Duck is a trademark of Black Duck Software, Inc. in the United States and other countries. All other names mentioned herein are trademarks or registered trademarks of their respective owners. February 2025

©2024 Black Duck Software, Inc. All rights reserved. Black Duck is a trademark of Black Duck Software, Inc. in the United States and other countries. All other names mentioned herein are trademarks or registered trademarks of their respective owners. February 2025