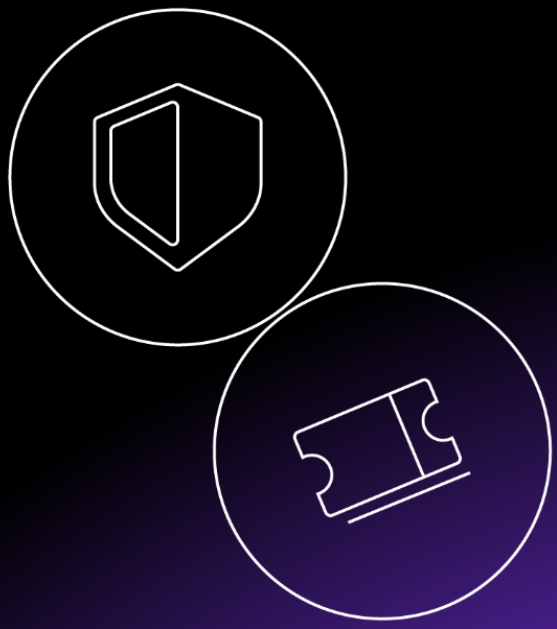
ENDOR LABS

STATION 9 BY ENDOR LABS



Top 10 Open Source Software (OSS) Risks

By the Station 9 Research Team at Endor Labs

Authors

Henrik Plate,
Security Researcher, Endor Labs

Dimitri Stiliadis,
CTO & Co-Founder, Endor Labs

Varun Badhwar,
CEO & Co-Founder, Endor Labs

Ron Harnik
VP Marketing, Endor Labs

Reviewers & Contributors

Anshu Gupta,
VP Security, Fast

Colin Anderson,
CISO, Ceridian

Gerhard Eschelbeck,
CISO, Kodiak Robotics

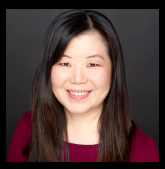
Maarten Van Horenbeeck,
CISO, Adobe

Niall Browne,
CISO, Palo Alto Networks

Shanku Niyogi,
VP Product Management, Databricks

Rachit Lohani,
CTO, Paylocity

Ralph Pyne,
CISO, Apollo.io



Kathy Wang,
CISO, Discord

Clint Maples,
CISO, F-500 company

Arkadiy Goykhberg,
CISO, Branch Insurance

Jonathan Meadows
MD Cybersecurity, Citi

Talha Tariq,
CISO, HashiCorp

Selim Aissi,
CISO, Blackhawk Network

Justin Dolly,
CSO, Sauce Labs

Ody Lupescu,
VP Security, Ethos

Yassir Abousselham,
CISO, UiPath

Motivation & Methodology

80% of code in modern applications is code you didn't write, but rely on through open source packages. Open source has clearly won as the method to deliver incredible value quickly, while leveraging the work of others, and hopefully contributing back so that others may benefit from your work as well. The selection, security, and maintenance of these open source dependencies are crucial steps towards software supply chain security.

This document introduces the Top 10 Risks introduced through the dependency on open source components throughout the software development process, e.g., the use of application frameworks like Spring Boot or libraries like Apache Log4j.

This list aims at addressing operational as well as security risks, and the target audiences are both engineering and security leaders and practitioners seeking to prioritize their open source governance programs.

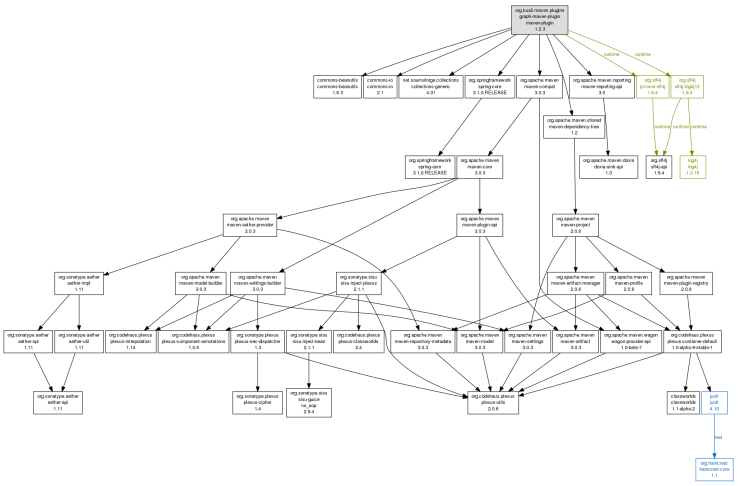
This report does not mean to criticize open source projects. It is well-known that open source is oftentimes more performant and secure than proprietary software. And it is also clear that open source software comes as-is, without warranties of any kind, and any risk of using it being solely on downstream users. Once an organization decides to rely on open source code, its security becomes their responsibility. Tools like the OpenSSF scorecard project, and now this Top 10 OSS Risks framework, are here to help organizations make better decisions about the software they rely on.

Top 10 Open Source Software Risks

Dependency Management 101

Let us quickly review some basic concepts of dependency management using a simple example. Just skip this section if you're familiar with dependency management.

The root node of the dependency graph displayed below represents the example project. The child nodes of the root represent 9 open source components the project "directly" depends on. Those components, however, depend on other components as well, all of which become "transitive" or "indirect" dependencies from the perspective of the top-level project.



Direct dependencies are consciously selected by the project developers, e.g., through the declaration in a manifest file such as package.json (Node.js/npm) or pom.xml (Java/Maven) file. Package managers take care of downloading and installing those direct dependencies from 3rd party package repositories to the developers workstation or a CI/CD system. When doing so, package managers also identify transitive dependencies, resolve potential version conflicts and install them locally. In other words, plenty of components are downloaded in an automated fashion in order to make sure all code required by the example project (and more) is present on the developer machine.

Risk Overview

Every single open source dependency of a software development project has plenty of properties - both the component (i.e. the code or binary artifact downloaded) and the corresponding open source project with all its stakeholders (e.g., contributors, maintainers) and systems (e.g., source code management, build systems).

The following list provides an overview about 10 problematic properties, which can result in significant security or operational risks for downstream consumers. Security risks can result in the compromise of system or data confidentiality, integrity or availability. Operational risks can endanger software reliability on the one hand, but can also increase the efforts and investments required to develop, maintain or operate a software solution.

The deficiencies and risks will be described in more detail below, together with examples and possible mitigation strategies.

Risk	Description	Category
OSS-RISK-1 Known Vulnerabilities	A component version may contain vulnerable code, accidentally introduced by its developers. Vulnerability details are publicly disclosed, e.g, through a CVE. Exploits and patches may or may not be available.	Security
OSS-RISK-2 Compromise of Legitimate Package	Attackers may compromise resources that are part of an existing legitimate project or of the distribution infrastructure in order to inject malicious code into a component, e.g, through hijacking the accounts of legitimate project maintainers or exploiting vulnerabilities in package repositories.	Security
OSS-RISK-3 Name Confusion Attacks	Attackers may create components whose names resemble names of legitimate open-source or system components (typo-squatting), suggest trustworthy authors (brand-jacking) or play with common naming patterns in different languages or ecosystems (combo-squatting).	Security
OSS-RISK-4 Unmaintained Software	A component or component version may not be actively developed any more, thus, patches for functional and non-functional bugs may not be provided in a timely fashion (or not at all) by the original open source project	Ops
OSS-RISK-5 Outdated Software	A project may use an old, outdated version of the component (though newer versions exist).	Ops
OSS-RISK-6 Untracked Dependencies	Project developers may not be aware of a dependency on a component at all, e.g., because it is not part of an upstream component's SBOM, because SCA tools are not run or do not detect it, or because the dependency is not established using a package manager.	Security, Ops
OSS-RISK-7 License and Regulatory Risk	A component or project may not have a license at all, one that is incompatible with the intended use by a downstream consumer, or one whose requirements are not or cannot be met by a downstream user.	Ops
OSS-RISK-8 Immature Software	An open source project may not apply development best-practices, e.g., not use a standard versioning scheme, have no regression test suite, review guidelines or documentation. As a result, a component may not work reliably or securely.	Ops
OSS-RISK-9 Unapproved Change (Mutable)	A component may change without developers being able to notice, review or approve such changes, e.g., because the download link points to an unversioned resource, because a versioned resource has been modified or tampered with or due to an insecure data transfer.	Security, Ops
OSS-RISK-10 Under/over-sized Dependency	A component may provide very little functionality (e.g. npm micro packages) or a lot of functionality (of which only a fraction may be used).	Security, Ops

Detailed Descriptions & Mitigations

Digging into the risks in more detail

Regarding mitigations strategies or controls, it is to be noted that they can differ significantly in regards to effectiveness and implementation/operational costs. Open source consumers should select them according to the specific security, regulatory and business requirements at hand, similar to the maturity levels proposed by OWASP SCVS or SLSA.

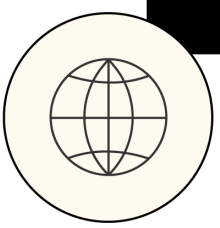
For example, building components from the sources of an open source project avoids many security issues of build and distribution environments. However, this control also comes with significant implementation costs such that it can hardly be recommended as a default, must-have control.

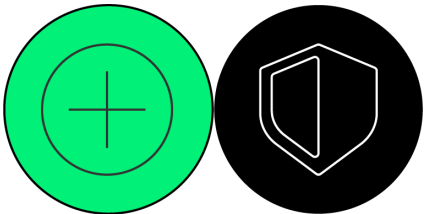
1.

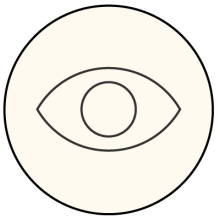
OSS-RISK-1

Known Vulnerabilities

Description	A component version may contain vulnerable code, accidentally introduced by its developers. Vulnerability details are publicly disclosed, e.g, through a CVE. Exploits and patches may or may not be available.
Risk(s)	The vulnerability may be exploitable in the context of the downstream software, which could compromise the confidentiality, integrity or availability of the respective system or its data.
Actions	• Regularly scan for vulnerabilities in all OSS versions used • Prioritize findings to optimize resource allocation, e.g., by using SAST tools to determine whether vulnerable code can be executed in the context of the dependent software.
Examples	• CVE-2017-5638 in Apache Struts (which caused the Equifax data breach) • CVE-2021-44228 in Apache Log4j (Log4Shell)
References	• OWASP Top 10 A06:2021 – Vulnerable and Outdated Components







2.

OSS-RISK-2

Compromise of Legitimate Package

Description

Attackers may compromise resources that are part of an existing legitimate project or of the distribution infrastructure in order to inject malicious code into a component, e.g, through hijacking the accounts of legitimate project maintainers or exploiting vulnerabilities in package repositories.

Risk(s)

Malicious code can be executed on end-user systems or on systems belonging to the organization that develops and/or operates the dependent software (e.g., build systems or developer workstations). The confidentiality, integrity and availability of systems and the data processed/stored thereon is at risk.

Actions

There's no single action to detect and prevent the ingestion of compromised packages. Organizations should consult emerging standards and frameworks like the OpenSSF Secure Supply Chain Consumption Framework (S2C2F) to inform themselves about possible safeguards, which should be selected and prioritized according to individual security requirements and risk appetite.

Example actions include:

- Verify component provenance according to SLSA
- Build component from the sources (yourself or a trusted 3rd-party)
- Review code manually and/or automatically
- Retrieve all components from a secured internal store (such binary repositories host home-made binaries and mirror external components)

Examples

- Event-stream: This attack on a legitimate component targeted users of Copay Bitcoin wallets.
- The SolarWinds Cyber-Attack

References

- Risk Explorer for Software Supply Chains - Subvert Legitimate Package (AV-001)
- OpenSSF Supply chain Levels for Software Artifacts (SLSA)
- MITRE ATT&CK Compromise Software Dependencies and Development Tools
- CICD-SEC-3: Dependency Chain Abuse

3.

OSS-RISK-3

Name Confusion Attacks

Description	Attackers may create components whose names resemble names of legitimate open-source or system components (typo-squatting), suggest trustworthy authors (brand-jacking) or play with common naming patterns in different languages or ecosystems.
Risk(s)	Malicious code can be executed on end-user systems or on systems belonging to the organization that develops and/or operates the dependent software (e.g., build systems or developer workstations). The confidentiality, integrity and availability of systems and the data processed/stored thereon is at risk.
Actions	Prior to installing/using a component: • Check code characteristics (pre/post installation hooks, encoded payloads, etc.) and project characteristics (source code repository, maintainer accounts, release frequency, number of downstream users, etc.) for leading risk indicators. Note that some component metadata is not verified by package repositories, thus, can easily be forged by attackers. • Verify signatures (if any)
Examples	• Colourama: Typo-squatting attack on the legitimate Python package “colorama” to redirect Bitcoin transfers to an attacker-controlled wallet.
References	• Risk Explorer for Software Supply Chains (overview about attack vectors with plenty of references to real-world attacks, safeguards, etc.) • OpenSSF Supply chain Levels for Software Artifacts (SLSA) • MITRE ATT&CK Compromise Software Dependencies and Development Tools • CICD-SEC-3: Dependency Chain Abuse

4.

OSS-RISK-4

Unmaintained Software

Description

A component or component version may not be actively developed any more, thus, patches for functional and security bugs may not be available.

Risk(s)

The patch development may need to be done by downstream developers with potentially less experience and knowledge regarding the affected component. This can result in increased efforts and longer resolution times. During that time, the system remains exposed.

Actions

- Check project liveliness and health, e.g., the number of active maintainers/contributors, release frequency or the number of issues and pull requests opened/closed. Note, however, that little activity can also be a sign of maturity. Projects that are considered feature-complete and mature will see less activity than projects under active development, and still receive timely patches in case of problems.
- Search for information on a project's maintenance or support strategy, e.g., the presence and dates of LTS versions
- Check the project page for explicit mentions of maintenance status (e.g., an archived GitHub project)

Examples

- Gorilla Web Toolkit
- Adoptoposs or Jazzband
  (Web pages used by projects to find new co-maintainers)
- Spring Boot support time frames

References

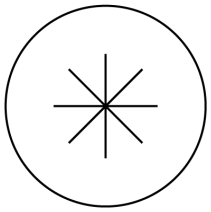
- OWASP Top 10 A06:2021 – Vulnerable and Outdated Components
- Bus factor
- CWE-1104: Use of Unmaintained Third Party Components
- CWE-1329: Reliance on Component That is Not Updateable

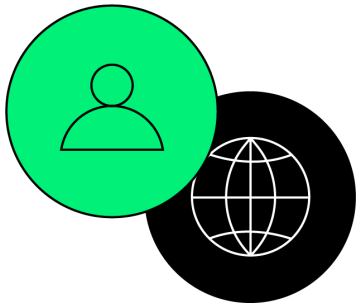
5.

OSS-RISK-5

Outdated Software

Description	A project may use an old, outdated version of the component (though newer versions exist).
Risk(s)	Falling too much behind the latest releases of a dependency can make it difficult to perform timely updates in emergency situations, e.g., when a vulnerability is disclosed for the version in use. Old releases may also not receive the same level of security assessment as recent versions, esp. whether they are affected by vulnerabilities. If a new version is syntactically or semantically incompatible with the current version in use, application developers may require significant update/migration efforts to resolve the incompatibility.
Actions	Keep dependencies up-to-date, e.g., by using tools that create merge/pull requests with update suggestions, and making dependency updates recurring backlog items
Examples	• Spring Boot support time frames
References	• OWASP Top 10 A06:2021 – Vulnerable and Outdated Components





6.

OSS-RISK-6

Untracked Dependencies

Description	Project developers may not be aware of a dependency on a component at all, e.g., because it is not part of an upstream component's SBOM, because SCA tools are not run or do not detect it, or because the dependency is not established using a package manager.
Risk(s)	Flying under the radar, the respective component cannot be checked or monitored for any of the other deficiencies.
Actions	• Evaluate and compare SCA tools regarding their capability to produce accurate bills of materials, both at coarse-granular level (e.g., dependencies declared with help of package management tools likes Maven or npm) and fine-granular level (e.g., artifacts like single files included "out of band", i.e., without using package managers)
Examples	• Incomplete SBOMs received for upstream components or produced by SCA tools • Inclusion of 3rd-party code in a managed (tracked) dependency, e.g., code snippets, 3rd-party source code files (copied as-is into the dependency's sources) or 3rd-party compiled code (e.g., platform-specific binaries or re-bundled Java archives/class files) • Not using a package manager at all • IDE plugins, build scripts, test dependencies or other developer tools, though not included in the dependent software itself, still pose security and operational risks.
References	• OWASP SCVS V1 Inventory and V2 Software Bills of Materials • Fossology

7.

OSS-RISK-7

License and Regulatory Risk

Description

A component or project may not have a license at all, one that is incompatible with the intended use by a downstream consumer, or one whose requirements are not or cannot be met by a downstream user. A component may also violate license terms independent from downstream use, e.g., if it is licensed as GPL but includes files licensed under the original (4-clause) BSD license. A component may also conflict with legal and regulatory requirements, e.g., related to FedRAMP certification or export control.

Risk(s)

It is important to use components in compliance with their license terms. The absence of a license or non-compliant use can result in copyright or license infringements, which the copyright holder can take legal action against. The violation of legal and regulatory requirements can constrain or hamper addressing certain verticals or markets.

Actions

- Identify acceptable licenses for the intended use of the component in the software under development, considering, for instance, how the component is linked, the software's deployment model (cloud, on-premise/device) and the intended distribution scheme.
- Comply with requirements stated in the open source licenses
- Avoid components without license
- Scrutinize component files for multiple and/or incompatible licenses

Examples

- Free Software Foundation, Inc. v. Cisco Systems, Inc. (2008)

References

- OSI Licenses & Standards
- SPDX License List
- Reuse Software
- GPL-Incompatible Free Software Licenses

8.

OSS-RISK-8

Immature Software

Description	An open source project may not apply development best-practices, e.g., not use a standard versioning scheme, have no regression test suite, no development or review guidelines or no documentation. As a result, a component may not work reliably or securely (in the sense of having security weaknesses that result in exploitable vulnerabilities).
Risk(s)	The dependency on an immature component or project comes with operational risks. The dependent software may not work as expected and result in runtime reliability issues, or its use may be overly complex and expensive for the dependent software development organization. For example, a component or project may lack documentation, may not use or comply with an established versioning scheme (which can result in breaking changes during component updates), or may not have a test suite to discover regressions introduced through pull/merge requests. Such cases can increase the effort of developers depending on such components.
Actions	• Check whether a project follows development best-practices, e.g., the presence, quality and up-to-dateness of project documentation and release notes, the presence of badges to indicate test coverage or the presence of CI/CD pipelines to detect regressions. • A proxy for checking project maturity may also be the number of downstream dependents.
Examples	• None
References	• OpenSSF Best Practices Badge Program • Common Weakness Enumeration

9.

OSS-RISK-9

Unapproved Change (mutable)

Description

A component may change without developers being able to notice, review or approve such changes, e.g., because the download link points to an unversioned resource, because a versioned resource has been modified or tampered with or due to an insecure data transfer.

Risk(s)

Using components that are not guaranteed to be identical when downloaded at different points in time are primarily a security risk. Attacks such as on the Codecov Bash Uploader demonstrate the risk of piping downloaded scripts directly to bash, without checking their integrity beforehand. Mutable components also threaten the stability and reproducibility of software builds.

Actions

- Use resource identifiers providing guarantees (or at least some degree of assurance) to always point to the same, immutable artifact.
- Additionally, verify digests or signatures after component download and before installation/use
- Use secure protocols for connection/distribution to avoid MITM attacks

Examples

- References to non-versioned shell scripts in CI/CD pipelines (e.g., https://codecov.io/bash)
- References to Git repositories without commit identifier (e.g., https://raw.githubusercontent.com/.../main/install.sh)
- HTTP links to package repositories (e.g., CVE-2021-26291)

References

- SLSA Immutable Reference
- CWE-829: Inclusion of Functionality from Untrusted Control Sphere
- CWE-830: Inclusion of Web Functionality from an Untrusted Source
- OWASP Top 10 A08:2021 – Software and Data Integrity Failures
- Codecov Bash Uploader Security Update

10.

OSS-RISK-10

Under/over-sized Dependency

Description	A component may provide very little functionality (e.g. npm micro packages) or a lot of functionality (of which only a fraction may be used).
Risk(s)	Very small components, e.g. ones containing few lines of code only, are subject to the same supply chain risks as large ones, e.g. account take-over, malicious pull requests or CI/CD vulnerabilities, for comparably little functionality. In other words, in exchange for very few lines of code used, the consumer's security becomes dependent on the upstream project's security and development posture. Very large components, on the other hand, may have accumulated many features that are not needed in standard use-cases, but contribute to the component's attack surface. Additionally, such unused features may also bring in additional, unused dependencies (bloated dependencies).
Actions	• Redevelop the specific functionality needed internally.
Examples	• Apache Log4j (a large dependency coming with many features) • Left-pad (a small dependency)
References	• Feature creep • A comprehensive study of bloated dependencies in the Maven ecosystem

Conclusion

The first decade of open source adoption was all about speed and productivity. Developers could use open source to deliver software faster than ever before, and the communities built on open source software (OSS) became a home for amazing developers and innovation that moves at breakneck speed. Today, most companies are not able to compete in the marketplace without a heavy reliance on OSS, which also drives more and more companies to sponsor and participate in the OSS ecosystem. These changes have ushered in a new stage of maturity, and companies now have to consider how to keep relying on OSS in a safe and scalable way.

The traditional approach to OSS security has its roots in compliance, a big part of which, when it comes to OSS, is making sure that packages don't carry any known vulnerabilities. This is why CVEs quickly became the ultimate risk indicator for OSS, and remain so today. But as this exploration of the top 10 OSS risks shows, CVEs are only one vector of attack, and a lagging indicator of risk.

OSS became the bedrock of modern software development, but whether it is a weak or strong link in the software supply chain depends on which and how OSS is used. Next-generation attacks such as name confusion attacks cannot be captured by CVEs, and neither can the operational risks of unmaintained, outdated, or untracked dependencies.

Thanks to the tremendous efforts of foundations like OWSAP and the OpenSSF, maintainers and consumers have more tools than ever to evaluate and strengthen the security of open source projects. We hope that publications like this one will help organizations implement a holistic view of risk - both security and operational.

About Endor Labs

Endor Labs helps developers spend less time dealing with security issues and more time accelerating development through safe Open Source Software (OSS) adoption. Our Dependency Lifecycle Management™ Solution helps organizations maximize software reuse by enabling security and development teams to select, secure, and maintain OSS at scale. The Endor Labs engineering team includes some of the world's leading static analysis experts, including 7 PhDs and senior engineers from Meta, Uber, Amazon, and Microsoft. Endor Labs was founded by industry veterans Varun Badhwar and Dimitri Stiliadis, and is backed by Lightspeed & Dell Technologies Capital, as well as executives at companies like Palo Alto Networks, Zscaler, Zoom, Google, and more.