

# Securing Our Dependence on Code Reuse in Software

FEBRUARY 2023



Centre  
for Internet  
& Society

AUTHOR:

Divyank Katira

REVIEW & EDITING:

Isha Suri

CONCEPTUALISATION:

Divyank Katira and Gurshabad Grover

LAYOUT DESIGN:

Indumathi Manohar

This work is funded by the [2020 Digital Infrastructure Fund](#),  
by the [Ford Foundation](#), [Alfred P. Sloan Foundation](#), [Open So-  
ciety Foundations](#), [Omidyar Network](#) and [Mozilla Foundation](#)  
in collaboration with the [Open Collective Foundation](#).

This work is licensed under a [Creative Commons Attribution  
4.0 International License](#).

2023, [Centre for Internet and Society](#), India



# Contents

<b>Introduction</b>	<b>4</b>
<b>Methodology</b>	<b>8</b>
<b>How is software developed?</b>	<b>10</b>
<b>Securing code reuse: the status quo</b>	<b>13</b>
<b>Securing the coding process</b>	<b>15</b>
<b>Securely loading OSS code</b>	<b>20</b>
<b>Securing storage processes</b>	<b>24</b>
<b>Securing build &amp; deployment processes</b>	<b>26</b>
<b>Conclusion</b>	<b>28</b>
<b>Endnotes</b>	<b>29</b>

01

# Introduction

# What is this report and who is it for?

Dividing and breaking up a software project into smaller modules with functionality that can be reused to build other software is an increasingly common practice in software development today. Much of this reuse happens in the form of open-source software (OSS) packages, i.e. software whose source code is openly available on the internet with a permissive licence which allows for its reuse and modification. A study that analysed the composition of over 2400 commercial software applications from seventeen industries found that, on average, 78% of the code used to build them was open-source software<sup>1</sup> – indicating that code reuse is not merely supplemental, but foundational to software development processes today. Relying on domain experts to build and maintain the functionality that is ancillary to a software application's primary purpose saves effort and allows application developers to focus on their own work domains. For instance, a developer building a video conferencing application – such as Zoom – may reuse an open-source library called [ffmpeg](#) to encode and decode video streams, or another open-source component, [OpenSSL](#), to encrypt and decrypt the encoded streams as they are transmitted over the internet, rather than reimplementing this functionality from scratch.

Despite the well-known practical benefits of code reuse and its prevalence in all of the digital products and services our society relies on, several security incidents in widely used OSS projects have shown that such projects are often underfunded and under-maintained. The 'Heartbleed' vulnerability most clearly illustrates this. In 2014, a security vulnerability in the OpenSSL software library – which is widely used to encrypt web traffic – affected about one-fifth of the servers on the internet.<sup>2</sup> Malicious actors could have exploited this vulnerability to decrypt all of the data that these servers handled and even impersonated them.

In this report, we examine our infrastructural dependence on reuse of OSS components and develop an understanding of the security risks posed by the widespread reuse of code that is developed and maintained by untrusted individuals and organisations that have no obligation to provide these services or any subsequent support.

We present an analysis of common security issues in OSS packages, with a focus on the unique security issues that arise in the tooling and processes used to store, distribute and operate reused code. Finally, we survey solutions and frameworks which seek to address some of these issues on a systemic level.

This report is primarily aimed at regulators, technical decision-makers and organisations invested in furthering research in this area. It can also serve as a starting point for software developers who want to learn about the common security pitfalls of using OSS components and how they can avoid them.

# Why we need to secure code reuse

Software security refers to preventing any activity that adversely affects a software system's ability to maintain confidentiality (refers to keeping data private), integrity (preventing tampering), or availability (keeping services running).

The security of open-source software is important, in particular, because of the wide-ranging downstream effects of code reuse. A security issue in a single component can compromise the security of its manifold users, essentially reducing the security of several entities to the security of an individual entity that they all rely on. The concept of transitive dependencies is also important to understand the expansive nature of code reuse. Software projects not only rely on the packages that are directly imported into the project, but also on all of the packages imported by each of these dependencies – and so on and so forth. This creates a large and complex 'dependency graph' with tens, or even hundreds, of reused packages, as illustrated in [Figure 1](#).

The practice of code reuse isn't new – free and open-source software movements have roots in the 80s and 90s. But over the past decade, there has been a massive increase in the extent of code reuse, as well as a shift in the types of software that are reused. In an essay titled 'Our Software Dependency Problem',<sup>3</sup> Russ Cox – developer of the package manager for Google's Go programming language, attributes these changes to the integration of package managers in modern programming languages.

It is argued that the ease with which these tools allow developers to publish and reuse software packages has changed the extent of code reuse from a few high-quality, well-reputed packages to a large number of dependencies that even cover trivial tasks requiring only a dozen or so lines of code. This is evidenced by almost 2 million packages being available in the npm package manager for the JavaScript programming language.

Other package managers for popular languages such as Java, PHP, Python, and .NET each offer over 300,000 packages.<sup>4</sup> Russ Cox further opines that "the situation [of software reuse] goes mostly unexamined" and that today "we are trusting more code with less justification for doing so".



# 02

## Methodology

We analysed security issues in OSS components from an openly available dataset covering nine package managers across nine programming languages, with a focus on how OSS is stored, managed, built and deployed.

Subsequently, we surveyed solutions and frameworks which seek to collectively secure code reuse in the broader OSS ecosystem.

To understand the most common types of security issues in OSS software, we manually analysed nearly **6500 vulnerabilities** in the Snyk Open Source Vulnerability Database<sup>5</sup> between **January 2017 and September 2021**.

This dataset contained descriptions of vulnerabilities in open-source software covering the following package managers and programming languages:

- cocoapods (for the Objective C programming language),
- Composer (PHP),
- Go (Go),
- hex (Erlang),
- Maven (Java),
- npm (JavaScript),
- NuGet (.NET),
- pip (python),
- and RubyGems (Ruby).

The vulnerabilities were annotated to distinguish which part of the software supply chain they affected, i.e. the development, transfer, storage and build processes. These were aggregated by vulnerability type to compile the types of issues that affect code reuse. We also surveyed academic literature, news reports, incident reports, mailing list archives, issue trackers, and technical write-ups by OSS developers and security researchers describing their work to compile a list of ways in which code reuse can be targeted.

In our survey of solutions in the space, we heavily reference the work done by industry bodies, coalitions and non-profits such as the Open-Source Security Foundation (OpenSSF), the Core Infrastructure Initiative, the Linux Foundation's in-toto project, and the Internet Security Research Group (ISRG).

## Limitations

While we have included metrics, such as the number of vulnerabilities of a particular kind and their severity scores where relevant, care must be taken while ranking and prioritising issues based on them. Software security is notoriously hard to measure, and the issues visible in our dataset only represent a subset of all vulnerabilities i.e. the ones that have been discovered. Security is also highly contextual, a seemingly harmless low severity vulnerability could have a higher impact (and vice versa) depending on how a particular software uses the vulnerable code. For instance, a security issue in a cryptographic library may appear critical, but it will not have much of an impact if that library is being used to shuffle a deck of cards in a single-player game of solitaire.<sup>6</sup>

# How is software developed?

To understand how code reuse practices are vulnerable to security threats, we need to first understand relevant parts of the software development process. As there is no standard way of developing software, we describe the most common workflow.

In practice, individual organisations use variations of this workflow as they see fit – they may conduct some of the automated processes described below manually, or may omit steps, such as reviews, entirely.

**Coding:**

Once the requirements and design of a piece of software are laid out, a software developer will start coding it. As they go about their work, they will likely encounter open-source packages which implement some of the functionality that they need, and may choose to reuse this code. They then load the OSS code onto their computer and integrate it with the functionality they were working on. When coding is complete, they will 'compile' or 'build' the software.

**Compile and test:**

Software written in most programming languages undergoes a 'compilation' or 'build' stage in which source code is converted into instructions that computers can understand and execute. Even programs written in interpreted languages, which are not compiled, but directly translated into instructions as the software executes, undergo some form of bundling or packaging process before they are consumed. This process, of converting code into the format that it will finally be consumed in, is known as the build process. Any tests on the software that were developed during the coding process will be run once the build is complete.

**Review:**

After the software is built and satisfactorily tested, the developer submits it for review. The review includes both manual and automated checks on the code. In the manual code review, other developers on the team go through the changes to the code and give any suggestions. The automated checks (commonly known as Continuous Integration or CI) load any OSS dependencies, build the software independently, and conduct tests on the code to ensure it integrates correctly with the rest of the codebase.

**Storage:**

After review, the code is submitted to a storage repository. The storage repository typically runs some version control software, such as git or Subversion, which handles storage of source code, tracks different versions, and manages any conflicts in contributions from multiple developers.

**Deployment:**

Finally, when software is ready to be released, a build server i.e. an independent computer tasked with building the software, will load its OSS dependencies, build the software, test it again, and deploy it for consumption. If the software being developed is also open-source, it may be reused by another developer, and this process repeats itself.

A diagrammatic representation of the software development process is shown in [Figure 2](#).

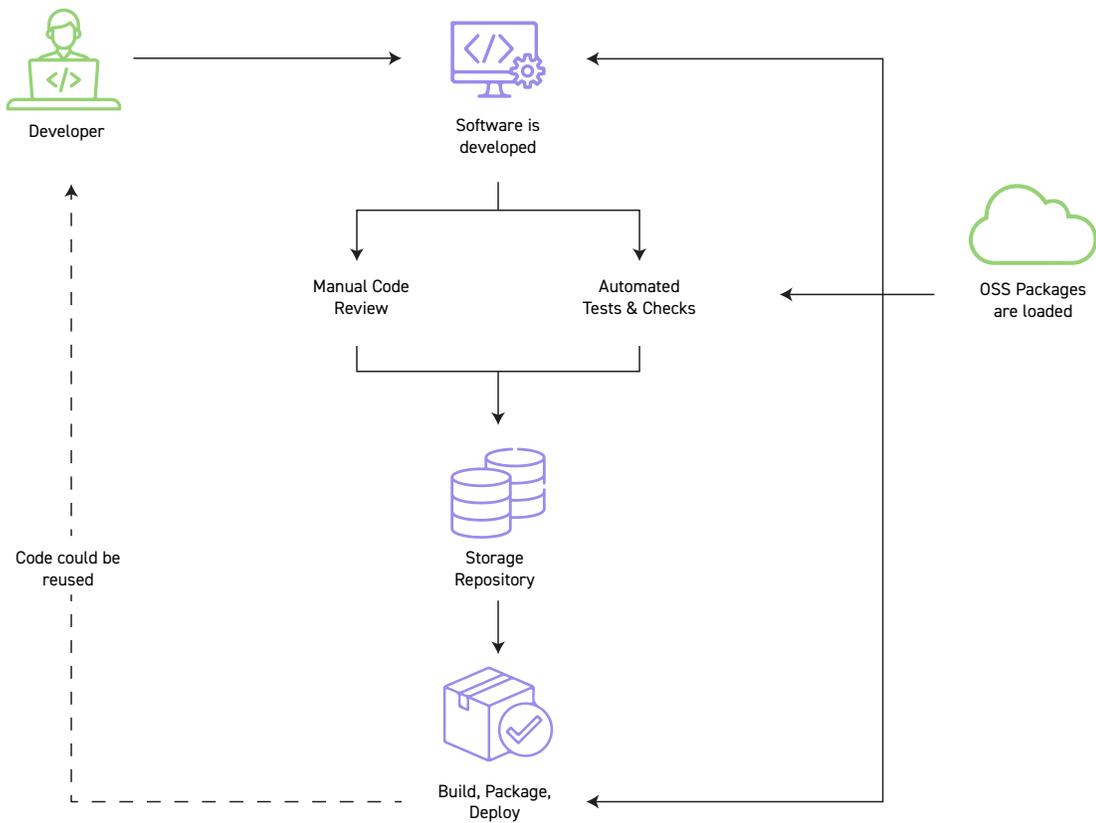


Figure 2: How software is developed

# 04

## Securing code reuse: the status quo

In this section, we describe how organisations go about securing code reuse today. Current approaches to securing code reuse are limited in that they are geared towards best practices that individual organisations follow, such as auditing code for security issues and keeping tabs on their dependency graph. They are also reactive in nature; issues are mitigated and remediated by involved entities once they come to light, with little to no focus on proactive, collective measures to secure this shared infrastructure.

### **Code audits:**

Code audits are a standard practice for organisations that develop software once they reach a certain size. It entails a periodic manual review of all code (excluding OSS code) and configurations by security experts as well as automated checks to catch common security issues.

However, for open-source projects, many of which are run by volunteers, the cost of manual code reviews can be prohibitive. Additionally, only the largest organisations have the resources to review OSS libraries they depend on along with any subsequent updates. This leaves the majority of OSS usage unreviewed and vulnerable.

### **Vetting and tracking OSS dependencies:**

There is an increasing focus on maintaining an inventory of dependencies, tracking vulnerabilities in them, and installing timely updates. A 2021 Executive Order by the US government requires vendors supplying software to the federal government to include a “Software Bill of Materials”. This is essentially a list of components used to build software and can be used to keep track of vulnerabilities in third-party components.<sup>7</sup> This regulatory acknowledgement of code reuse security issues has spawned a number of tools and standards to help organisations maintain and exchange lists of their software dependencies.

Security vendors have also developed tools to help organisations check if any of their dependencies have known vulnerabilities by cross-checking vulnerability databases. These tools are typically paid and not accessible to all organisations.

### **Intervention from package managers and code repositories:**

Vulnerabilities have been mitigated in the past through manual intervention by package managers (like npm, PyPI) and code storage repositories (GitHub, Gitlab). In some cases, package managers and storage repositories have stepped in to restore packages that have been tampered with to a previous working state. Package managers have also used their administrative power to remove malicious packages from their listings. For instance, when the author of the popular ‘colors’ and ‘faker’ packages intentionally sabotaged their code, npm reverted the packages to previous stable versions to prevent people from accidentally using them and Github suspended their developer account.

While these individual and reactive approaches to secure code reuse deployed today are an important first step, there is a need to focus on more proactive and collective efforts. In sections 5 to 8 below, we describe the unique security issues that arise in the tooling and processes used to store, distribute and operate code reuse, and survey efforts that approach these problems at an infrastructural level to collectively address them.

# 05

## Securing the coding process

Most security vulnerabilities are introduced during the coding process. We describe the most common types of vulnerabilities we encountered during our examination of the Snyk vulnerability database, and other writeups and reports in this section.

<p>● <b>Human error:</b></p>	<p>Computer programming is prone to human error. A large majority of security vulnerabilities that we encountered during our analysis were common programming mistakes which allowed malicious actors to compromise the security of the software in question. The most common ones are explained in <a href="#">Table 1</a>.</p>
<p>● <b>Memory safety issues:</b></p>	<p>Programming languages like C and C++ require programmers to manually manage a program's memory. This process is highly error-prone, and lapses can be exploited by attackers to inject and execute their own code in a program's memory, essentially allowing them to take over the entire program or even the device. Memory safety issues were underrepresented in our dataset as only one (Objective C) of the nine programming languages covered is vulnerable to them. However, large portions of our digital infrastructure, including the most popular operating systems, web browsers, databases, and encryption and networking libraries, are written in memory unsafe programming languages such as C and C++. Year after year, between 60 and 90% of all vulnerabilities in critical software such as Android, MacOS, iOS, the Linux kernel, Firefox and Chrome, have been memory safety issues.<sup>8</sup></p>
<p>● <b>Typosquatting &amp; masquerading:</b></p>	<p>Typosquatting is a way to introduce malicious dependencies into software by creating packages whose names correspond to common typographical mistakes in popularly used packages. Such attacks rely on developers accidentally referencing the malicious package, which typically includes all of the functionality of the original library along with some malicious embedded code that is triggered under certain conditions. For instance, a malicious Python package named "Collored" mimics the popular "colored" library, which is used to add colours to terminal windows.<sup>9</sup></p> <p>A variation of this attack is known as masquerading, wherein a malicious library mimics the functionality of a well-known library and uses a similar-sounding name which developers may confuse for the original one. For example, 'tools-for-discord' is a malicious package that masquerades as a legitimate one.<sup>10</sup></p> <p>This type of attack is very common – we encountered 408 instances of typosquatting in our dataset, and several more in news reports. Mitigating such attacks requires manual intervention from the package managers to remove the offending packages as and when they are reported to them.</p>

VULNERABILITY TYPE	OCCURRENCES	DESCRIPTION
<i>Cross-site Scripting (XSS)</i>	1048	This vulnerability allows an attacker to inject malicious code into a victim's web browser when a legitimate website is loaded and can be used to steal credentials or perform actions on a victim's behalf.
<i>Malicious Package</i>	625	This refers to third-party software packages which have been injected with malicious code. These are discussed in detail in the sections below.
<i>Denial of Service (DoS)</i>	512	Denial of Service vulnerabilities affect the availability of data or services that the software has been tasked with.
<i>Information Exposure</i>	344	Refers to vulnerabilities that accidentally reveal private information.
<i>Cross-site Request Forgery (CSRF)</i>	284	Allows attackers to trick web users into performing unwanted actions on websites.
<i>Directory Traversal</i>	264	Directory traversal allows attackers to access files that they are not authorised to view.
<i>Remote Code Execution (RCE)</i>	252	Gives attackers the ability to remotely take over a computer and execute commands on it.
<i>Regular Expression Denial of Service (ReDoS)</i>	247	Attackers can make a computer hang by supplying a specifically crafted input to software that contains this vulnerability.
<i>Prototype Pollution</i>	232	Allows attackers to tamper with a program's internal representation of objects. It can be used to gain access to data and take over a program's flow.

Table 1: Ten most common vulnerabilities in our dataset.

We discuss ways to collectively minimise security issues stemming from coding mistakes below:

## 01

### Using Memory Safe Languages

As discussed, memory management issues are the root cause of several high severity issues in our digital infrastructure. Memory unsafe programming languages remain in heavy use for systems programming as they are highly performant and offer a high degree of control. There are also historical reasons – they integrate easily with existing operating systems and system libraries which were also written in the same languages. There are efforts underway to replace the use of memory unsafe languages with modern programming languages, such as Rust, which are memory safe and offer comparable performance. These languages entirely eliminate the possibility of memory safety issues by either managing memory automatically or conducting checks at compile time to ensure secure memory use. The [Prossimo](#) project, led by the Internet Security Research Group (ISRG), is working on rewriting some of our most critical software dependencies, such as encryption libraries and parts of the Linux operating system, in memory safe languages. However, this is a massive effort – decades of internet infrastructure building needs to be redone before we are able to minimise memory safety issues to satisfactory levels.

## 02

### Collective Code Auditing

This approach suggests that instead of individual organisations conducting security audits on their software dependencies independently, they can pool together resources to conduct public audits of popular shared libraries. The Open Source Security Foundation's Alpha-Omega project is one such effort. It aims to select 200 projects per year and conduct security assessments on them.<sup>11</sup>

## 03

### Open Bug Bounties

Bug bounties are a way of crowdsourcing security audits. Security researchers and ethical hackers are encouraged to find issues in software and are rewarded with a monetary 'bug bounty' for responsibly disclosing it to the project maintainers. Open bug bounty programs are a way of funding such activity for open-source projects. The [Internet Bug Bounty](#) program, funded by a few tech companies, rewards researchers with a bug bounty for reporting issues to open-source projects and offers a portion of it to the maintainers of the project in which the vulnerability is discovered. The program covers 17 projects and has resolved over 700 security reports at the time of writing. The European Commission's Open Source Programme Office also runs one such program.<sup>12</sup>

## 04 Fuzzing OSS projects

Fuzz testing is an automated method of discovering vulnerabilities in software. In this method, computer programs are supplied with a large number of inputs and monitored for any abnormal behaviour or crashes. Google's OSS-Fuzz project, which it runs in collaboration with the Core Infrastructure Initiative and OpenSSF, conducts fuzz testing on over 500 open-source projects and has discovered over 8000 security issues.

## 05 Large-scale package analysis

The OpenSSF's [Package Analysis](#) project is developing tools to automatically scan package repositories and identify malicious variations in the names and code of the packages. So far, it has discovered over 200 malicious packages in the npm and PyPI package managers.

## 06 Identifying a set of 'critical' OSS components

Given the limited resources available for securing open-source projects, there is a need to identify a set of dependencies that are most critical for security to focus efforts on. There are a number of factors to consider while deciding on this:

- **Usage:** Security issues in OSS projects that are more popular can have a greater impact.
- **Functionality:** Not all dependencies are equal when considering security. A vulnerability in a library that handles network connectivity for software applications will have a much greater security impact than one that helps convert units.
- **Maintenance status:** A project that is already well maintained may not require additional resources.

The [Civil Infrastructure Platform](#) project seeks to identify a core set of building blocks which can be used for civil infrastructure projects. OpenSSF's [Alpha-Omega](#) project is working to select OSS projects that are critical for security and plans to collaborate with maintainers to help secure them. Initiatives like the [Digital Public Goods](#) registry can also help prioritisation efforts.

# 06

## Securely loading OSS code

Even though most OSS maintainers are honest in practice, OSS code is essentially under the control of untrusted individuals or organisations who are under no obligation to develop or maintain it. Maintainers can change the expected functionality of a package, insert malicious code, or withdraw the code entirely. In this section, we discuss vulnerabilities which allow attackers to supply code that is different from the expected source code.

OSS libraries are loaded at multiple points during the software development workflow – on the developer's computer during development, and on the build server during the testing and build stages. This compounds the harm that any intrusion into the OSS code can cause, as it can be used to compromise multiple parts of the software development workflow.

OSS code may be loaded from the internet or from a repository of OSS libraries maintained within the organisation. The latter configuration is more secure, as it reduces the possibility of interference by third parties.

The common types of vulnerabilities associated with loading OSS code are:

### **Dependency confusion**

Unlike typosquatting and masquerading attacks, which rely on tricking developers into accidentally referencing malicious packages, dependency confusion attacks are targeted towards tooling. In such attacks, malicious packages use names that are identical to private packages that organisations may use. The malicious packages declare a high version number, confusing package management tools, which default to using the newer versioned malicious package over the older version of an identically named private package that the developer intended to use.

This novel attack was reported in early 2021 by a whitehat hacker who used it to gain access to the networks of 35 large companies.<sup>13</sup> Its use has since been reported in real-world attacks<sup>14</sup> and we also found four instances in our dataset.

Mitigating such attacks also requires manual intervention from the package managers who remove the offending packages as and when they are reported to them.

### **Code sabotage and withdrawal**

An effect of using open-source packages that are under the control of a third party is that their authors can rename, withdraw, relicense, and even sabotage the code. As a result, we have seen many vulnerabilities that exploit this level of control to inject malicious code into software.

### **Rogue packages and protestware**

The authors of OSS libraries can intentionally insert malicious code or otherwise change the expected functionality to disrupt software that relies on it. For instance, in early 2022, the author of the 'colors' and 'faker' javascript packages, with thousands of users each, introduced a change which displayed gibberish on users' screens and rendered the functionality of the packages unusable. The author had indicated that they no longer wanted their free work to support corporations.<sup>15</sup> There were also instances of 'protestware' during the Russian invasion of Ukraine in 2022. Experts compiled a list of 21 packages which showed users messages about the war, and some even tried to remove files from users' computers if they were based in Russia or Belarus.<sup>16,17</sup>

## Withdrawal and rename

Authors may also rename or withdraw packages, which can cause unwanted effects to downstream consumers. In 2016, the author of a popular library called 'left-pad' unpublished the package, breaking the projects of people who directly and indirectly relied on it. An author changing their account username to lower-case on a code storage repository has caused similar issues.<sup>18</sup> Once a package is withdrawn from a particular repository, its name can also be occupied by a malicious package that takes its place.<sup>19</sup> We encountered 33 instances of such 'use-after-free'<sup>20</sup> malicious packages in our dataset.

Instances of code sabotage and withdrawal have been mitigated in the past by intervention from code storage repositories and package managers, who have used their administrative power to restore the code to a previous working version.

## Tampering dependencies in transit

Another point of potential compromise of an OSS library is while it is being transmitted over the internet. We encountered 16 vulnerabilities where OSS packages were downloaded over the internet without using encryption, allowing network intermediaries to tamper with the code in transit. Such an attack can only be executed by powerful actors who can target and intercept network traffic.

# Emerging Solutions

## 01

### Ensuring the integrity and availability of code

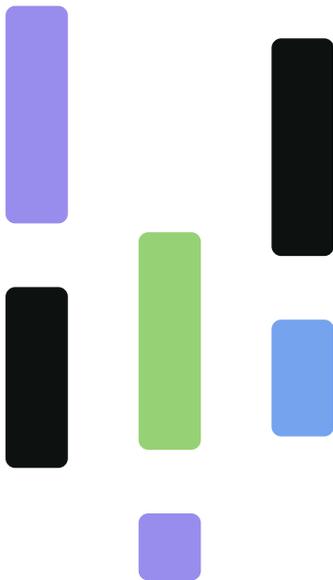
Instances of code withdrawal, sabotage and dependency confusion attacks have demonstrated the need for tooling that ensures the integrity and availability of code. The purpose of integrity checks is to ensure that OSS packages being loaded match the expected source code, and have not been tampered with in any way. Availability entails ensuring that code is available for use when it is needed, and access to it cannot be removed or impeded by any third party.

The [sigstore](#) project, affiliated with OpenSSF, is developing standards and infrastructure to digitally sign and verify open-source software. It not only aims to verify the integrity of code but also includes other attestations about the code such as information authenticating its developer and version. [The Update Framework](#), from the Cloud Native Computing Foundation, also seeks to provide similar functionality.

However, there is a need to make such features available in existing tooling and enabled by default. Google's Go programming language is a success story in this area. It has implemented integrity checks into its built-in package management system, Go Modules. It maintains availability of packages by providing a proxy service that stores and serves packages even if they are withdrawn or removed from the source control system. It also automatically verifies the integrity of downloaded packages by calculating 'cryptographic hashes' – which are essentially fingerprints of packages – and checking them against a tamper-evident log of hashes, ensuring that no third party, including the operator of this service's infrastructure, can change the contents of the code once it is installed by a developer.<sup>21</sup>

## 02 Package version pinning

Version pinning, i.e. specifying up-front the exact version of the OSS package to be used, is another improvement which can be incorporated into package management systems to load OSS dependencies more securely. Some popular package managers, such as npm and yarn for the javascript programming language, default to using a versioning system which automatically updates dependencies when minor revisions are released. When the author of the 'colors' npm package published a sabotaged version, several of the tens of thousands of packages that depended on it, both directly and indirectly, included the newer version automatically and broke.<sup>22</sup> Version pinning ensures that updates to packages are intentional, and bad updates are not automatically included in software.



# 07

## Securing storage processes

Code can also be manipulated at its point of storage. Vulnerabilities in the tools used to store source code and publish OSS packages can be exploited to inject malicious code into software applications. The common types of vulnerabilities associated with storing OSS code and packages are discussed in this section.



### **Vulnerabilities in storage tools and package managers**

Most development processes use version control systems such as git and Subversion to store source code, track changes to files, and coordinate contributions from multiple developers.<sup>23</sup> Package managers are used to modularise and share open-source packages with others. Vulnerabilities in such tools can be used to inject malicious source code into software, effectively bypassing any manual and automated review processes performed on the code. For instance, in 2021, Github was made aware of a vulnerability in its npm package manager that allowed malicious actors to update any package with code of their choice.<sup>24</sup>

### **Credential compromise**

Another way in which attackers can tamper with OSS packages is by gaining access to the authors' accounts on various online code storage repositories and package management tools. During the compromise of the popular javascript package 'ua-parser-js', the author's account was hijacked and used to publish malicious versions of the package.<sup>25</sup> In our dataset, we encountered three malicious packages which were published from stolen accounts.

### **Malicious contributions**

The collaborative nature of open-source software entails contributions from unknown, untrusted individuals who are willing to help with a project. While most contributors are honest and contributions are typically reviewed by maintainers of projects, there is still a possibility of malicious code being introduced by contributors.

The javascript package 'event-stream', which helps developers manage streams of data, was compromised after its maintainer handed over ownership of the package to one of its contributors. The contributor introduced malicious code into the package to steal cryptocurrency from its users.<sup>26</sup>

## Emerging solutions

### **Stronger authentication policies**

A weak password chosen by a single OSS maintainer can be used to compromise several organisations that depend on their code. To remedy this, there is an emerging trend to mandate stronger authentication policies by using technologies like multi-factor authentication (MFA). As of 2022, the top 500 maintainers in the npm package manager are required to enable MFA on their accounts.<sup>27</sup>

# Securing build & deployment processes

The testing, build, and deployment processes in the software development workflow have been identified as a potential point of compromise. A vulnerability in the build server or the tools that orchestrate and execute these processes can be exploited to inject malicious code into software. Injecting malicious code at this stage of the software development lifecycle can go unnoticed as security reviews and audits typically happen before this step. The common types of vulnerabilities that affect the build and deployment processes are discussed in this section.

## Build and deployment infrastructure compromise

In 2020, a supply-chain compromise on SolarWinds, which provides network monitoring tools to thousands of government and corporate customers, was attributed to a backdoor which was inserted into one of its products during the build process.<sup>28</sup> This was a fairly sophisticated state-linked attack with more than a thousand developers working on developing the malware<sup>29</sup> and went undetected for months. Attacks on build and deployment infrastructure are not trivial to carry out as this infrastructure is typically kept on a private network that is not connected to the internet, but such intrusions can be very stealthy.

In our analysis of the vulnerability dataset, we found a total of 437 vulnerabilities in various build and deployment tools. This includes continuous integration tools such as Jenkins and Buildbot, automated deployment tools like Puppet, Chef and Ansible, and tools to orchestrate larger deployments of multiple servers like Kubernetes and Consul. Out of these 144 were critical and high severity issues, 244 were medium and 49 were considered low severity.

## Compiler, operating system, hardware compromise

Further down the supply chain, attackers can exploit or insert vulnerabilities in programming languages and their compilers, the operating systems that servers use, or even the physical hardware. Vulnerabilities in any of these components can be used to compromise the build and deployment processes, and insert malicious code into software applications. While compilers and operating systems were out of scope for our analysis, many of them are open-source software and are vulnerable to the issues described in this report.

# Emerging solutions

## Reproducible builds

Similar to integrity checks on code which verify that code has not been tampered with during loading, reproducible builds aim to ensure the integrity of the build process. Reproducible builds require that the build process is deterministic, i.e. it produces the exact same output each time it is run. If two independent systems can run the build process and produce identical outputs, it provides some assurance that the build process was not compromised or tampered with.

However, there are many technical challenges to developing reproducible build processes. Firstly, all the inputs to the build, including all source code, OSS dependencies, compilers, toolchains, etc. need to be declared up-front. Secondly, all sources of non-determinism i.e. things that may change between subsequent builds, need to be removed from the build process. This includes things like timestamps, certain optimisation techniques which produce different code each time, and code signatures present in the build.<sup>30</sup> The [Reproducible Builds](#) project is working towards solving these technical challenges and building software tooling that allows for reproducible builds.

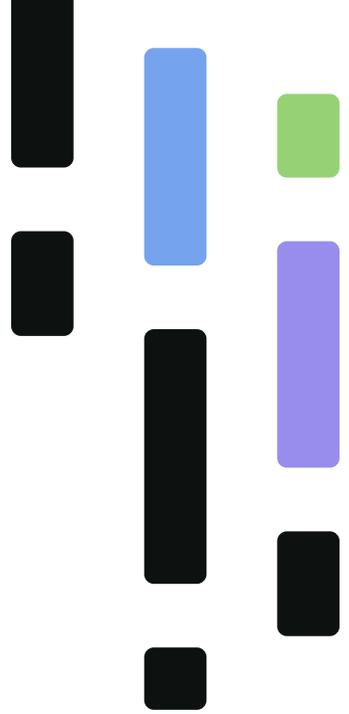
# Conclusion

The state of security of code reuse has been compared to that of the web before the widespread adoption of the encrypted HTTPS protocol in the 2010s.<sup>31</sup> A majority of web traffic was unencrypted, allowing any internet intermediary to view or modify users' web traffic as they wished. The development of easy-to-use encryption made freely available, reversed this trend, leading to a large majority of websites supporting encryption. Our analysis of vulnerabilities in open-source software demonstrates a similar state of neglect.

Open-source code is largely unreviewed for security issues, lacks adequate systemic safeguards to prevent tampering of code, and does not present users with the tools to verify whether the software they are consuming matches the expected source code.

Our survey of emerging solutions indicates that while there are ways to fix many of these issues, the solutions are not operating at a large enough scale to address the problem. This suggests that market forces and existing data protection regulations have failed to sufficiently incentivise organisations to address the problem of securing code reuse. As both public and private sectors derive massive value from software while reusing open-source code as their foundation, there is a need for regulation to ensure that they contribute to maintaining this infrastructure, instead of relying on them to do so voluntarily.

# Endnotes



- 1 Synopsys Inc . "2022 Open Source Security and Risk Analysis Report," [synopsys.com](https://synopsys.com)
- 2 Paul Mutton, "Half a million widely trusted websites vulnerable to Heart-bleed bug," [news.netcraft.com](https://news.netcraft.com)
- 3 Russ Cox, "Our Software Dependency Problem," [research.swtch.com](https://research.swtch.com)
- 4 Erik DeBill, "Module Counts," [modulecounts.com](https://modulecounts.com)
- 5 Synk, "Vulnerability DB," [security.snyk.io](https://security.snyk.io)
- 6 Alex Gantman, "SBOM: Good Intentions, Bad Analogies, and Ugly Outcomes," [againsthimself.medium.com](https://againsthimself.medium.com)
- 7 Joseph R. Biden Jr., "Executive Order on Improving the Nation's Cybersecurity," [whitehouse.gov](https://whitehouse.gov)
- 8 Alex Gaynor, "What science can tell us about C and C++'s security," [alex-gaynor.net](https://alex-gaynor.net)
- 9 Ax Sharma, "Careful Out There: Open Source Attacks Continue to Be on the Uptick," [blog.sonatype.com](https://blog.sonatype.com)
- 10 Ravie Lakshmanan, "25 Malicious JavaScript Libraries Distributed via Official NPM Package Repository," [thehackernews.com](https://thehackernews.com)
- 11 Bruce Schneier, "Finding Vulnerabilities in Open Source Projects," [schneier.com](https://schneier.com)
- 12 Nikos Vaggalis, "European Union Will Pay For Finding Bugs In Open Source Software," [i-programmer.info](https://i-programmer.info)

- 13 Alex Birsan, "Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies," [medium.com](https://medium.com)
- 14 Ravie Lakshmanan, "Over 200 Malicious NPM Packages Caught Targeting Azure Developers," [thehackernews.com](https://thehackernews.com)
- 15 Owen Williams, "Open source developers, who work for free, are discovering they have power," [techcrunch.com](https://techcrunch.com)
- 16 Bruce Schneier, "Developer Sabotages Open-Source Software Package," [schneier.com](https://schneier.com)
- 17 Brian Krebs, "Pro-Ukraine 'Protestware' Pushes Antiwar Ads, Geo-Targeted Malware," [krebsonsecurity.com](https://krebsonsecurity.com)
- 18 Andrew McAndre "Rename back to Sirupsen/logrus," [github.com](https://github.com)
- 19 Thomas Claburn, "You can resurrect any deleted GitHub account name. And this is why we have trust issues," [theregister.com](https://theregister.com)
- 20 Term originally used by Ohm, Marc, et al. "Backstabber's knife collection: A review of open source software supply chain attacks."
- 21 Filippo Valsorda, "How Go Mitigates Supply Chain Attacks," [go.dev](https://go.dev)
- 22 Russ Cox, "What NPM Should Do Today To Stop A New Colors Attack Tomorrow," [research.swtch.com](https://research.swtch.com)
- 23 Wikipedia, "Git," [en.wikipedia.org](https://en.wikipedia.org)
- 24 Mike Hanley, "GitHub's commitment to npm ecosystem security," [github.blog](https://github.com/blog)
- 25 Lawrence Abrams, "Popular NPM library hijacked to install password-stealers, miners," [bleepingcomputer.com](https://bleepingcomputer.com)
- 26 "Details about the event-stream incident," [blog.npmjs.org](https://blog.npmjs.org)
- 27 "Top-500 npm package maintainers now require 2FA," [github.blog](https://github.com/blog)
- 28 CrowdStrike Intelligence Team, "SUNSPOT: An Implant in the Build Process," [crowdstrike.com](https://crowdstrike.com)
- 29 Simon Sharwood, "Microsoft says it found 1,000-plus developers' fingerprints on the SolarWinds attack," [theregister.com](https://theregister.com)
- 30 SLSA, "Frequently Asked Questions," [slsa.dev](https://slsa.dev)
- 31 The Kubelist Podcast, "Ep. #20, Sigstore with Dan Lorenc of Google," [heavybit.com](https://heavybit.com)



Centre  
for Internet  
& Society