BLACKDUCK®

Exploring the Threat of Malicious Packages in the npm Ecosystem

Real-World Examples and Mitigation Strategies

The Danger of Malicious Packages

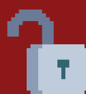
Software packages are a popular means to distribute open source and third-party software. They are often pulled from an outside source through a package manager or installer program, and typically include source code, libraries, documentation, and other files needed to build and run the software.

A *malicious package* contains malware disguised as a legitimate package. It's used to infiltrate the software supply chain via open source libraries or third-party dependencies and perform harmful action once activated. Once a malicious package's malware infects a system, it can potentially steal sensitive data, disable security software, modify or delete files, and even take over an entire system or network, spreading to other devices to further increase its damage.

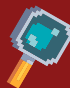
Unlike code weaknesses and vulnerabilities—which can exist in software for months or years without being exploited—a malicious package is almost always a direct and immediate threat.

A Basic Primer on CWEs, CVEs, and Malicious Packages

Malicious packages are software components or libraries that contain deliberately inserted malicious code, also known as *malware*. Common Weakness Enumerations (CWEs) and Common Vulnerabilities and Exposures (CVEs) are used to identify and classify security weaknesses and vulnerabilities in software.



CWEs are known software weaknesses. Think of them as comparable to the home protection advice that the police might issue for your neighborhood, i.e., if you're away from home, make sure your doors and windows are latched and locked, don't let mail and newspapers pile up when you're away, and so on.



CVEs identify specific vulnerabilities and track their status. They're like the FBI's Most Wanted posters that used to be displayed in post offices across the U.S.; they provide a description and warning to the public of the criminal and their crimes.

Malicious packages are like a criminal impersonating someone you would allow in your home—a repairperson or inspector, for instance—who presents you with falsified identification in order to gain entry.

How Malicious Packages Work

Four of the most common vectors for malicious code that can impact open source software and enter the software development life cycle (SDLC) include brandjacking, typosquatting, dependency hijacking, and dependency confusion.

Brandjacking

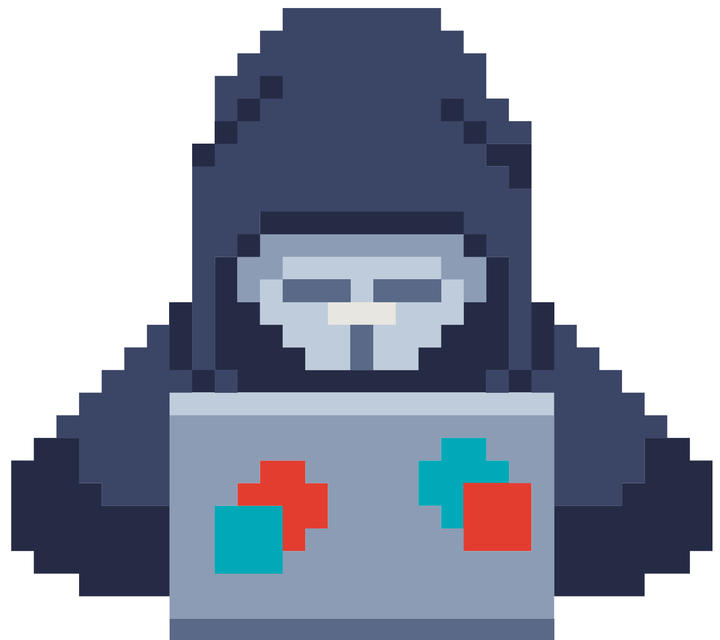
In *brandjacking*, an attacker assumes the online identity of the legitimate owner of a package, or compromises a legitimate account by stealing or otherwise gaining access to the account's user credentials, in order to distribute malware. One of the original types of malicious package attacks, brandjacking is still in wide use today.

For example, the jQuery open source project has more than 4 million weekly downloads from npm, the world's largest software registry. In February 2022, a malicious npm package called "jquery-lh" was downloaded by more than 100 million users. Because "jQuery" was in the package name, it looked legitimate to many people, so they downloaded the malicious package without checking its provenance.

Typosquatting

In *typosquatting*, an attacker publishes a malicious package with a name similar to a popular package—usually slightly misspelled—in the hope that a downloader will unintentionally fetch the malicious version. For example, at a quick glance, there doesn't seem to be a difference between these commands.

```
$ npm install electron
$ npm install electorn
```



However, the first one installs a legitimate software package from npm, while the second, which transposed two letters of the legitimate package's name, calls a malicious version.

Dependency hijacking and dependency confusion

A dependency is software that is required for another piece of software to function, making the main piece of software "dependent" on the first. Software dependencies can be either internal or external, and either between components of the same software application or between different software applications.

Dependency hijacking and *dependency confusion* are similar types of software supply chain attacks. Although the terms are sometimes used interchangeably to describe any attack using software dependencies as a vector, the difference between the two is the manner in which the attacker gains access to the target application's dependencies.

- In dependency hijacking, an attacker creates a package with the same name as a legitimate package and then uploads it to a public repository. When the victim pulls in the package, the attacker's malicious code is installed instead of the legitimate dependency. In 2018, for example, a malicious package was published to npm and added as a dependency to the widely used "event-stream" package, leading to it being downloaded more than 8 million times in less than three months.
- In dependency confusion—also known as a substitution attack—a malicious actor publishes a package to a public registry with the same name as a package used internally by a company, but with a higher version number. This causes an automatic download of the bogus version as the company's package manager attempts to keep dependencies up-to-date. Using this method in February 2021, a researcher, after gaining permission from the various organizations targeted for his security tests, managed to breach over 35 major companies' internal systems, including that of Microsoft, Apple, PayPal, Shopify, Netflix, Yelp, Tesla, and Uber.

Examples of Malicious Packages in the npm Ecosystem

Attackers have used a variety of strategies to introduce malicious packages into the npm ecosystem. Here are some notable examples.

2018

As discussed in the dependency hijacking example, in 2018, a popular package called "event-stream" was found to contain malicious code after its original maintainer transferred ownership to a new maintainer. The new maintainer introduced a malicious dependency designed to steal funds from bitcoin wallets. This highlights the importance of a project owner vetting volunteer project maintainers before allowing them access to popular projects, as well as the need for developers to perform thorough code reviews of downloaded packages.

2019

In 2019, a malicious package called "flatmap-stream" was discovered to contain code capable of stealing sensitive data from users' environments. This package had more than 2 million weekly downloads before its malicious intent was discovered by a security researcher during a routine code review. Again, this emphasizes the importance of regular code review and the need to verify package authenticity.

2021

In 2021, a malicious version of the popular "ESLint" package was uploaded to the npm repository following the compromise of an ESLint maintainer's account. The malicious package included code that would execute a backdoor on the user's system when the package was installed. The compromise was apparently made possible when the maintainer reused the same password on multiple accounts and didn't have two-factor authentication enabled on their npm account. This demonstrates the need for project owners and maintainers to be as security-conscious as downloaders.

2022

In 2022, ReversingLabs researchers discovered evidence of a widespread typosquatting software supply chain attack involving malicious JavaScript packages offered via the npm package manager. The npm modules that the ReversingLabs team identified were collectively downloaded more than 27,000 times. The incident stresses the need for software development organizations to use tools and processes that assess supply chain risks that could impact their projects' security.

The Four Major Open Source Ecosystems

Although there are four major open source ecosystems, this eBook focuses on JavaScript, the largest and most popular of them, and npm, the default package manager for JavaScript's runtime node.js. It should be emphasized that JavaScript and npm are not less secure than the other three ecosystems, but their ubiquity has made them popular with malicious actors.

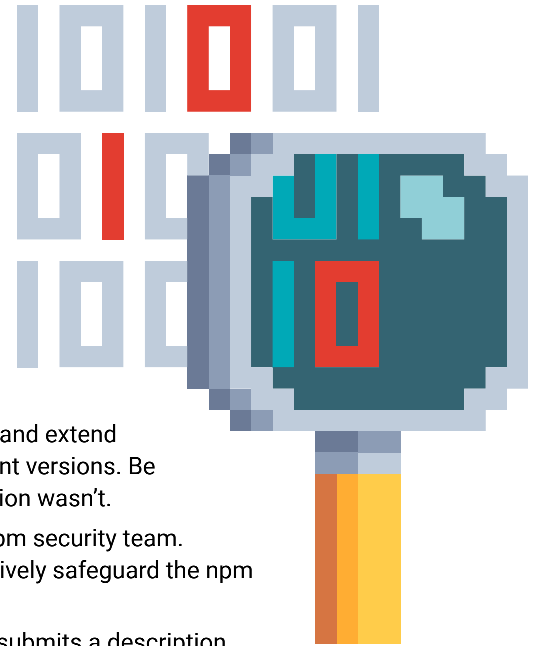
The other three major open source software ecosystems are Java, Python, and .NET. Each of these also has its own package distribution and management system: Maven Central for Java, PyPI for Python, and NuGet for the .NET family. Like npm and JavaScript, each has experienced attacks from malicious actors. Whatever the attack vector, malicious packages can pose serious risks to the integrity and security of applications in all four open source ecosystems.

Popular npm Packages

Some of the most popular npm packages include

- *lodash*: A utility library delivering consistency, modularity, performance, and extras
- *express*: A fast, minimalist web framework for node.js
- *moment*: A lightweight JavaScript date library for parsing, validating, manipulating, and formatting dates
- *async*: A utility module that provides straightforward, powerful functions for working with asynchronous JavaScript
- *request*: A simplified HTTP request client
- *chalk*: Terminal string styling done right
- *bluebird*: A fully featured promise library with a focus on innovative features and performance
- *debug*: A JavaScript debugging utility modeled after the Node.js core's debugging technique

Taking Action to Identify Malicious Code and Prevent It from Entering Your SDLC



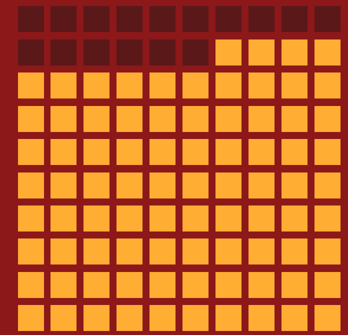
The threat of malicious packages in npm is real and requires proactive measures to mitigate the risks. Here are some strategies that developers can adopt.

- **Verify package authenticity and names.** Double-check the reputation and trustworthiness of the package. Look for signs of fake accounts or impersonations, and verify the legitimacy of the package's source before installing it. Confirm the accuracy of package names and URLs before installing them to avoid falling victim to fake or malicious packages.
- **Review package ownership and maintenance.** Be cautious when using packages that have recently changed maintainers. Thoroughly vet the new maintainers and review any changes made to the package after the transfer. Consider using packages with a history of reliable maintainers to minimize the risk of malicious activity.
- **Consider functionality changes.** Newer versions of packages serve to patch issues and extend functionality, but be suspicious of significant changes in functionality across different versions. Be wary when one version of a package is accessing files or systems the previous version wasn't.
- **Engage with the npm community.** Report suspicious packages or activities to the npm security team. Collaborate with other developers by sharing information and experiences to collectively safeguard the npm ecosystem from malicious packages.
- **Use npm security tools.** Available beginning with npm@6, the npm audit command submits a description of the dependencies configured in your package to your default registry and asks for a report of known vulnerabilities. The npm shrinkwrap command allows you to specify the exact versions of the packages you're using, which can help prevent the installation of a malicious package.
- **Create and maintain a Software Bill of Materials (SBOM).** In the fight against software supply chain attacks, having an accurate, up-to-date SBOM that inventories open source components is critical to ensuring that your code remains high quality, compliant, and secure. A comprehensive SBOM lists all the open source components in your applications as well as those components' licenses, versions, and patch status—the perfect defense against supply chain attacks, including those using malicious packages.
- **Stay informed.** Ensure that you have the means to be informed of newly identified malicious packages, malware, and disclosed open source vulnerabilities. Look for newsfeeds or regularly issued advisories that provide actionable advice and details about issues affecting open source components in your SBOM.
- **Perform code reviews.** Examine the code of packages before including them in your project. Look for any suspicious patterns or unexpected behaviors that could indicate malicious activity. Review the package's source code and check for any known vulnerabilities.
- **Use an automated software composition analysis (SCA) tool.** It's all well and good to advise your team to create and maintain an SBOM, monitor for new malicious packages, and perform detailed code reviews of package downloads, but the reality is that few DevOps teams have the time or resources to effectively do so. An SCA tool automates the process of identification, management, and mitigation of software security issues and allows developers to focus their energies on writing code. Such tools can evaluate open source and third-party dependencies that have been pulled into codebases, identify malicious packages or code, and provide remediation guidance.

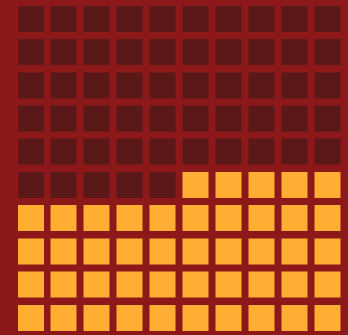
How Popular Is JavaScript?

The answer is ***very popular.*** Findings from the 2023 "Open Source Security and Risk Analysis" (OSSRA) report show that 74% of the scanned codebases use JavaScript. In fact, all of the top 10 open source components were written in JavaScript.

1. jQuery: The jQuery JavaScript library
2. lodash: A modern JavaScript utility library
3. ms.js: A tiny millisecond conversion utility for Node.js and the browser
4. visionmedia-debug: A tiny JavaScript debugging tool
5. safe-buffer: A safer Node.js buffer API
6. inherits: A simple way to do classic inheritance in JavaScript
7. punycode.js: A robust Punycode converter
8. isarray: Array#isArray for older browsers
9. readable-stream: Node.js core streams for userland
10. sindresorhus/supports-color: A way to detect whether a terminal supports color



74% of the scanned codebases in the OSSRA Report use JavaScript



45% of the scanned codebases in the OSSRA Report use jQuery

The 2023 OSSRA report also found the #1 component, the jQuery library, in nearly half—45%—of the audited codebases.

But like in Hollywood and high school, problems sometimes accompany popularity: according to OSSRA report audits, jQuery was also the component most likely to be vulnerable. Nearly half—47%—of the codebases contained a vulnerable jQuery component, with the most likely vulnerability being CVE-2020-11023 (BDSA-2020-0964), a cross-site scripting issue in jQuery versions 1.0.3 to 3.5.0. This vulnerability could allow an attacker to inject malicious code into a website using jQuery.

Conclusion

The rise of malicious packages in npm poses a significant threat to the security and integrity of JavaScript applications. Thorough code review, package authenticity verification, regular dependency updates, the use of npm security tools, package naming verification, and community collaboration are all proactive steps developers can take to protect their systems.

For added protection, you should consider implementing one of the many SCA tools available today. Look for one that will automate the process of creating and maintaining a full SBOM of open source and third-party code, will continuously monitor for and advise of new security threats as they appear, and can be used to scan your code regularly for security, license compliance, and operational risk issues.

By doing so, developers can safeguard their projects from malicious packages and contribute to a safer and more secure npm ecosystem.

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.

©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. August 2024