# Lesson 10: Supply Chain Security

## Introduction

Modern web applications are built on a foundation of open-source packages. While this accelerates development and promotes code reuse, it also creates security risks.

Attackers are increasingly targeting open-source dependencies, seeking to have malicious code added to these libraries, instead of directly attacking applications. If they are able to compromise a dependency that is used in many different applications, then one successful attack yields much more reward for the attacker. These types of attacks are often referred to as “software supply-chain attacks.”

Package registries face a constant stream of malicious packages; security researchers report finding [around 100 new supply chain attacks per week](https://socket.dev/blog/risky-biz-podcast-how-socket-goes-beyond-vulnerabilities-to-tackle-modern-supply-chain-attacks) across major ecosystems like npm, PyPI, and Maven. While the majority of these malicious packages have relatively few downloads (often under 50), an attacker only needs to trick one person into installing their package in order to execute an attack.

When a malicious package is installed, an attacker can:

* Exfiltrate data from the production application (stealing environment variables, secrets, etc.)
* Inject cryptocurrency miners
* Execute ransomware
* Inject backdoors into the application
* Install malware on developers’ local machines

## Supply chain attack vectors

There are several different ways that an attacker might get malicious code included in an application.

1. **Typosquatting**: An attacker might register package names similar to popular packages, hoping that a developer will make a typographical error when running npm install. For example, they might register loadash instead of lodash, express-js instead of express, or reactdom instead of react-dom. While npm now prevents the creation of package names very similar to popular packages, reducing the ease of these attacks, they still occur. Additionally, attackers have started executing a variant of this attack called [*slopsquatting*](https://socket.dev/blog/slopsquatting-how-ai-hallucinations-are-fueling-a-new-class-of-supply-chain-attacks), registering non-existent package names hallucinated by AI models to target developers running commands generated by LLMs.
2. **Dependency Confusion**: An attacker might exploit how package managers resolve names between private and public registries. If they notice a company’s *internal* package name (i.e. not registered publicly) in public source code, they may register that package on the public npm registry. Then, when npm next installs the dependencies for a project, it will resolve to the library from the public npm registry instead of the company’s internal registry. A writeup of such an example attack is available [here](https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610).
3. **Account Compromise**: An attacker may gain access to a maintainer’s account to publish malicious versions of a library, e.g. via phishing or [CI credential compromise](https://socket.dev/blog/pypi-on-ultralytics-breach-no-security-flaws-in-pypi-exploited).
4. **Maintainer Burnout**: An attacker may target overburdened maintainers, offering to make legitimate contributions to help maintain a project, then pushing malicious code after gaining trust. This is what happened in the famous [xz-utils attack](https://cyberscoop.com/open-source-security-trust-xz-utils/), where a bad actor spent two years making legitimate contributions, gaining trust and control, before inserting a backdoor. [Open-source maintainers frequently face burnout](https://socket.dev/blog/the-unpaid-backbone-of-open-source), and in extreme cases, even receive harassment (examples [one](https://socket.dev/blog/ldapjs-open-source-project-decommissioned-after-maintainer-receives-abusive-email) and [two](https://socket.dev/blog/maintainer-faces-backlash-over-controversial-decision-to-support-legacy-node)), making open-source projects more vulnerable to this attack vector.

## Defense Strategies

### Use Lockfiles and Precise Versions

Lockfiles are critical for security because they ensure all developers and environments use exactly the same package versions. The npm lockfile records exact package versions and cryptographic hashes for all dependencies used in a project:

{  
 "dependencies": {  
 "express": {  
 "version": "4.18.2",  
 "resolved": "https://registry.npmjs.org/express/-/express-4.18.2.tgz",  
 "integrity": "sha512-9....",  
 "dependencies": {  
 "accepts": "1.3.8",  
 "array-flatten": "1.1.1"  
 }  
 }  
 }  
}

The pinned version will ensure that a production deployment does not automatically start using a new package version before engineers have had a chance to test and screen it. The integrity hash ensures that the dependency does not secretly change, which prevents an attacker from serving different variants of the same package version, e.g. it would be bad if a security auditor received a different set of source code from a production server running the library. This also helps to mitigate dependency confusion attacks.

For applications, lockfiles should *always* be committed to version control to ensure that everyone is using the same dependency source code.

### Minimize Dependencies

Having a large number of dependencies, especially when maintained by many different people, can increase the supply chain attack surface of a project. As such, developers should seek to minimize the number and complexity of dependencies in use, while preferring packages actively maintained by reputable maintainers. This can be difficult; a project might use a small number of packages directly, but if those packages themselves use many dependencies, then in total, the project might depend on a large number of packages transitively.

Tools like [depcheck](https://www.npmjs.com/package/depcheck) may be used to prune unused dependencies from package.json, and tools like [socket-optimize](https://socket.dev/blog/introducing-socket-optimize) help reduce the count of transitive dependencies.

Note that choosing dependencies involves balancing multiple tradeoffs, and it’s important to consider dependency quality and maintenance history besides simply minimizing dependency count. Additionally, using quality dependencies is usually more secure than reinventing the wheel and reimplementing code from scratch. However, as open-source dependencies are increasingly targeted for attack, developers should be mindful of the code coming from sprawling dependency trees.

### Use Dependency Scanning and Monitoring Tools

Node.js provides an npm audit command to check for known vulnerabilities in dependencies. This command can be run in CI scripts to monitor for supply chain risks.

However, npm audit has significant limitations due to the fact that it only checks for packages that have known CVEs. In order to have a CVE, someone must have found a vulnerability in a package and submitted it to the National Vulnerability Database. It may take days or weeks for CVEs to be reported, and typosquat packages generally do not have CVEs because the usage of these packages is so low. Also, npm audit audits packages that have already been installed, and a package with [install scripts](https://socket.dev/alerts/installScripts) may have already compromised the host machine.

Alternatively, tools like [Socket](https://socket.dev/) or [Snyk](https://snyk.io/product/open-source-security-management/) scan dependencies in real time for a wider range of signals, such as changes in maintainership, addition of network APIs to packages that didn’t use them previously, typosquat likelihood, and AI-detected malware alerts. These tools can block malicious packages before they are installed and before CVEs are reported.

## Summary

The software supply chain represents a critical attack surface that traditional security tools often miss. Attackers increasingly target package registries and dependency management systems rather than applications directly. Effective defense requires a combination of:

1. Technical controls (lock files, security tooling)
2. Process controls (dependency reviews, minimizing dependencies)
3. Automated tooling (dependency scanners, behavior monitoring)