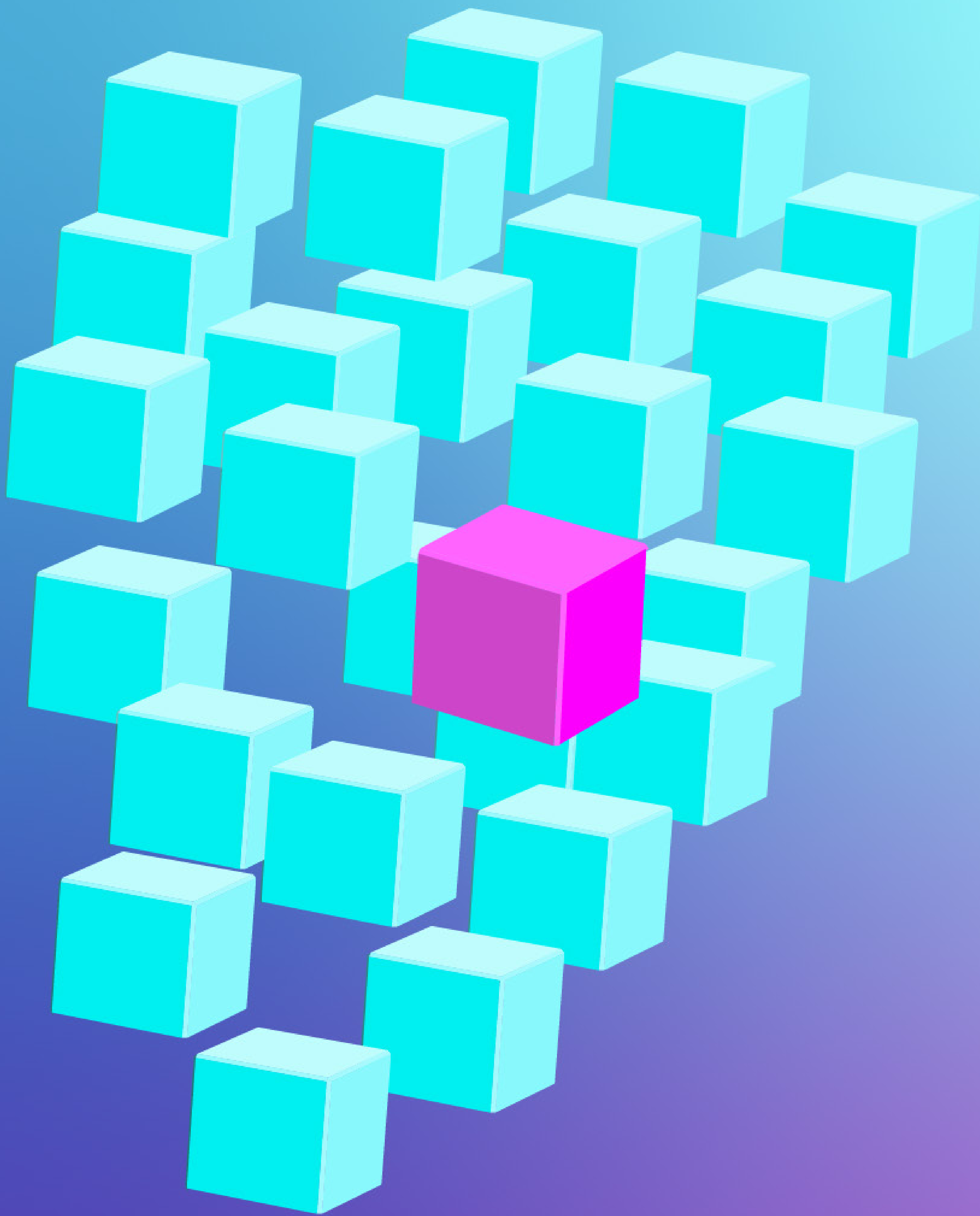


Mitigate Dependency Confusion Risks

activestate

Executive Summary

Package managers have dramatically lowered the overhead of code reuse, leading to modern software's heavy reliance on third-party dependencies. Knowing this, bad actors exploit the trust that organizations have established in code reuse infrastructure, targeting programming language package managers, open source public repositories and binary artifact repositories.

The most popular class of new cyberattacks involve dependency confusion, which is an exploit that takes advantage of the fact that software is often built using a mix of both internal and external dependencies. Software development processes that don't implement safeguards can become "confused" into installing a compromised external dependency instead of the organization's internal version. The result can be compromised commercial software that is propagated downstream to an ISV's customers where it is deployed as a trusted resource.

Best practices like namespacing, checksum and URL verification, dependency vendoring and using hermetically-sealed build systems can all help mitigate dependency confusion risks. Alternatively, instead of building environments from individual dependencies, consider creating them from secure, prebuilt runtime environments.

Introduction

Part of the beauty of modern software creation is that it's easy to include third-party dependencies to help build and subsequently deploy an application thanks to the use of open source language package managers. Need a Node package? Just use npm to download and install it on demand. Want a Python module? Pip will include it in your software environment in seconds.

The ease and simplicity with which third-party dependencies enable robust capabilities to be added has led to the ubiquity of employing them in everything from small, homegrown business-specific apps, to commercial, enterprise-class software. But this popularity is a double-edged sword, as it can also attract the attention of malicious authors who are always looking for new attack vectors.

When third-party dependencies are combined with internally-developed dependencies, the stage is set for systems to potentially become "confused" over which dependency to use. In this case, failure to verify the source of a package can expose your organization to dependency confusion attacks in which a package manager may inadvertently download a compromised third-party dependency from an external URL rather than the local, internally-developed dependency.

According to the PortSwigger Top 10 Web Hacking Techniques of 2021, dependency confusion ranked number one due to its:

1. **Ubiquity** - 89% of organizations rely on open source software, and 98% of applications incorporate open source dependencies in their codebase.
2. **Extensibility** - the attack method is trivial to extend to any open source programming language that relies on a package manager.
3. **Effectiveness** - dependency confusion is proven to be effective at targeting some of the largest, high-profile enterprises.

While modern software development practices make it nearly impossible to avoid reliance on third-party dependencies, there are best practices and tools that can help guard against software supply chain attacks like dependency confusion. For example, one of the most effective anti-dependency confusion resources that can help avoid the stigma of being the subject of a security breach headline is using pre-built runtime environments. More on this technique later.

What Is Dependency Confusion?

Dependency confusion is a software supply chain attack that substitutes malicious third-party code for a legitimate internal dependency. There are various approaches to creating this kind of substitution, including:

- **Namespacing** – by uploading a malicious package to a public repository that is named similar to a trusted, internally-used package, systems that omit a namespace/URL check may mistakenly pull in the malicious package.
- **DNS Spoofing** – by using a technique like DNS spoofing, systems can be directed to pull dependencies from malicious repositories while displaying what looks like legitimate, internal URLs/paths.
- **Scripting** – by modifying build/install scripts or CI/CD configurations, systems can be tricked into downloading dependencies from a malicious source rather than a local repository.

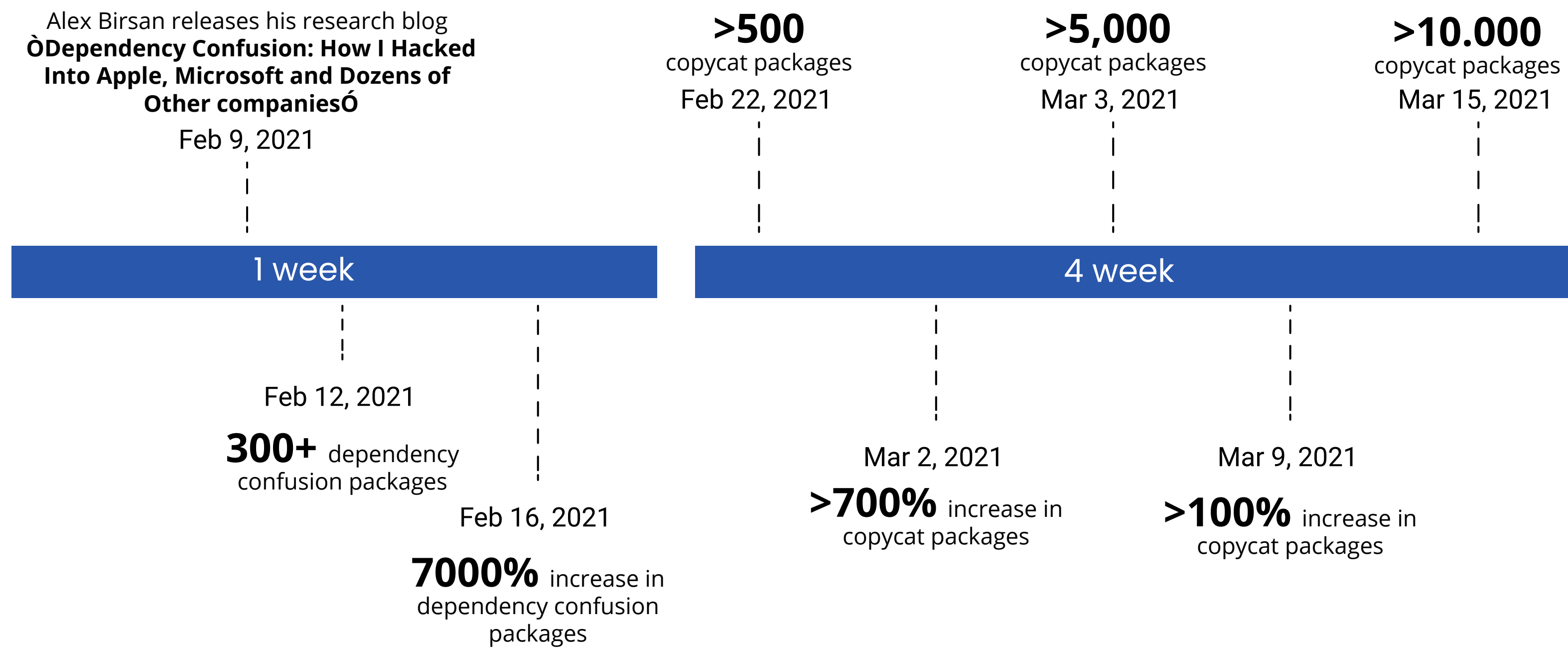
Here's what a typical dependency confusion attack scenario might look like:

1. A bad actor researches the name of a software dependency used internally by an organization to develop their software application.
2. The bad actor creates a similar dependency, embeds malware, names it the same as the internal dependency and sets the version number to be higher than the one discovered through research.
3. The bad actor uploads the compromised dependency to a public repository.
4. The next time the package manager requests the specific dependency, it may pull the compromised dependency from the public repository rather than the local repository (for example, pip will default to installing the dependency with the highest version number).

The compromised dependency is typically a clone of the original (to fulfill all functional requirements for use in an application), along with malicious code designed to exfiltrate data, implant a backdoor in the execution environment, or otherwise implement a security threat.

Since the exploit's discovery by security researcher Alex Birsan, who briefed the affected organizations before publishing his findings in a Medium post on February 9, 2021, the threat of dependency confusion has grown. Thousands of copycats have introduced hundreds of confusing NPM packages and executed numerous real-world confusion attacks.

Dependency Confusion Growth Timeline



The threat of dependency confusion is exacerbated by:

- **Implicit Trust in Public Repositories** - far too many organizations continue to blindly trust open source repositories despite the fact that they provide little in the way of security measures.
- **Detection Difficulty** - there is simply no means of scanning for dependency confusion problems until a compromised dependency has already been included in your software.

While dependency confusion doesn't lead to fundamentally new types of security risks, it does represent a new, never-before-seen vector of attack that security-conscious organizations will need to address.

Best Practices for Mitigating Dependency Confusion Risks

Preventing dependency confusion exploits is key to securing the software supply chain, but there is no one solution that can mitigate all potential substitution threats. Instead, there are a number of best practices that can be implemented/followed to help manage the risks, including:

- **Utilize Scopes/Namespaces** - some package managers allow for namespaces, IDs or other prefixes, which can be used to ensure that internal dependencies are pulled from private repositories defined with the appropriate prefix/scope.
- **Secure the Build Environment** - create a dedicated, locked-down, secure build environment. This will help mitigate the risk that attackers insert malicious dependency paths in build scripts and CI/CD configurations, or pull in remote transitive dependencies during a build step.
- **Validate Hashes/Checksums** - wherever possible, validate that a dependency's checksums match those documented on official package sources. This can be difficult to automate with changing dependencies/versions, but once a definitive set of dependencies is created, you can take advantage of your package manager's support for lock files and automated hash checking.
- **Vendor Dependencies** - rather than pulling dependencies from private and public repositories on demand every time an environment is built, reduce the risk of dependency confusion by embedding the source code for all dependencies – internal and external – in your code repository. Package managers can then be configured (and verified) to utilize only a single source for all dependencies. While dependency vendoring is an effective approach, be warned that it can also be quite complex.

Implementing a routine to verify URLs and checksum hashes is fairly simple and straightforward, but securing and regularly auditing build environments or vendoring all external dependencies require far more time and resources. In some cases, the cost to mitigate dependency confusion attacks may be disproportionate to the actual risk.

A far more cost-effective solution may be the use of secure, pre-built runtime environments such as those offered by trusted vendors like ActiveState. In this case, all dependencies are built by the vendor and packaged into a runtime suitable for deployment to development, test, production and other environments. Pre-built runtime environments make dependency confusion attacks the responsibility of the vendor, but rather than blindly trusting the vendor, ensure that their build process includes:

- **Scripted Builds** – build scripts that cannot be accessed and modified within the build service, preventing exploits.
- **Ephemeral, Isolated Build Steps** – every step in a build process executes in its own container, which is discarded at the completion of each step. In other words, containers are purpose-built to perform a single function, reducing the potential for compromise.
- **Hermetic Environments** – containers have no internet access, preventing (for example) dynamic packages from including remote resources.

Like dependency vendoring, using a pre-built environment such as that incorporated in the ActiveState Platform restricts the need for URL validation to a single link. And using the ActiveState Platform also means you can avoid all the complexity of dependency vendoring, as well.

Conclusions

Any software built with both internal and external dependencies is susceptible to dependency confusion attacks. While no silver bullet currently exists to eliminate the threat, simply performing link and checksum validation during the execution of a secure and locked-down build process can significantly reduce the risk of compromise.

Security-conscious organizations, however, can realize greater risk mitigation by adopting a dependency management strategy like dependency vendoring, or by leveraging prebuilt runtime environments. These techniques limit the number of URLs to be verified to one and one only.

Additionally, prebuilt runtime environments can offer further benefits beyond security, including:

- Faster container build times since there's no need to wait for dependencies to be individually downloaded and resolved.
- Greater environment consistency by limiting configuration drift through a single, central runtime environment. Branches for development, test and production environments, as well as the entire CI/CD pipeline, can then be programmatically derived from these.
- Simpler runtime deployment, requiring only a single command to install and/or update the target environment.

About ActiveState

ActiveState enables DevOps, InfoSec, and Development teams to improve their security posture while simultaneously increasing productivity and innovation to deliver secure applications faster.

With a single platform that tames open source complexity, teams get a continuously secure software supply chain, unparalleled observability, robust vulnerability management, continuous upgrades, and governance support that enhance collaboration across the organization.

All from the trusted partner that pioneered and continues to lead enterprise adoption and use of open source software.

Start An Enterprise Trial

©2024 ActiveState Software Inc. All rights reserved. ActiveState®, ActivePerl®, ActiveTcl®, ActivePython®, Komodo®, ActiveGo™, ActiveRuby™, ActiveNode™, ActiveLua™, and The Open Source Languages Company™ are all trademarks of ActiveState.

www.activestate.com • Toll-free in NA: 1-866.631.4581 • solutions@activestate.com