Microsoft

3 Ways to Mitigate Risk When Using Private Package Feeds

Secure Your Hybrid Software Supply Chain

Latest version located at: https://aka.ms/pkg-sec-wp

Contents

Introduction

Software today has become an assembly of components from a wide range of sources. Individual packages may be developed in-house, acquired from third-parties, or downloaded from free and public sources. The security risks of these sources are straightforward to understand in isolation. But, when used in concert, new interactions arise that can compromise even a conscientious organization.

Many organizations use public package feeds — such as Maven Central, npm, NuGet Gallery, and the Python Package Index (PyPI) — to take advantage of the open ecosystems they offer. As a result, organizations fail to treat these uncurated feeds as a potential source of malware. The truth is that projects that consume packages from multiple public and private feeds may be exposed to supply chain vulnerabilities unique to these hybrid configurations.

When private packages are not private enough

Over the last decade, there has been a dramatic increase in the scope, quality, and availability of free, open-source software. Even closed-source applications commonly depend on freely shared libraries, tools, and operating systems. A critical factor in this has been the development of ecosystems around package indexes that allow anyone to publish, such as Maven Central, npm, NuGet Gallery, and Python Package Index.

A public, open-package index allows anyone to share their code without proving their identity. This ease of publishing has resulted in large, active ecosystems of packages. These ecosystems can include various elements, from generic building blocks to niche, domain-specific algorithms, along with powerful tools for discovering, acquiring, and composing these packages.

Organizations (or "clients") use private feeds as package index mirrors or to distribute internal packages to protect against upstream compromises, such as package hijacking and typo-squatting. Package management tools typically allow specifying multiple sources from which to download components, making it easy to consume from public and private indexes.

This hybrid configuration can enable new ways for attackers to enter your systems through otherwise secure infrastructure due to how package management tools resolve names across multiple sources.

In past whitepapers, such as the Microsoft Digital Defense Report, we discussed some potential attacks through open-package indexes that are often mitigated using private feeds. This paper will cover the unique risks of hybrid configurations — particularly those using both public and private feeds — and review available mitigations for developers using .NET, Python, JavaScript, and Java.

> “Open-source projects have an average of 180 package dependencies.”
>
> —GitHub State of the Octoverse Report, 2019

> “The pervasiveness has attracted the attention of attackers, who in recent years have increasingly turned their focus to the open-source software supply chain.”
>
> —Microsoft Digital Defense Report, September 2020

What are the risks of a hybrid configuration?

One common hybrid configuration that clients use is storing internal packages on a private feed but allowing the retrieval of dependencies from a public feed. This ensures that the latest package releases are automatically adopted when referenced from a package that does not need to be updated. Internal developers publish their packages to this private feed, and consumers check both private and public feeds for the best available versions of the required packages. This configuration presents a supply chain risk: the substitution attack.

A substitution attack happens when an attacker discovers that a client is using a private package that's not present on the public feed. After the attacker uploads a higher version of the private package to the feed, the client downloads it automatically because it has the same file name. Services that merge package feeds also allow this substitution if packages from public sources may override those from private sources. A related risk with a similar impact can emerge if the package publisher's credentials have been compromised.

Most clients will automatically install the attacker's package from the public feed, which presents an attack opportunity.

What does an attempted attack look like?

At its simplest, an attack involves someone installing a package that is different from what the client expected. This is most concerning in automated builds where the installation messages are often not reviewed on success. Most substitution attacks will successfully install and fail later in the build unless the attacker has cloned the package's functionality.

Unexplained build or test failures should be treated as a potential warning of a package substitution attack. Some public package feeds allow publishers to remove or delist packages, which could cause failing builds to start succeeding again, making it difficult to detect.

Importantly, even though the build failed, the attacker likely achieved remote code execution. Any secrets or access tokens may have been exfiltrated or misused. Reducing the use of secrets in automated builds can help reduce the impact, for example, by separating build and publish steps. Using hosted build VMs rather than on-premises machines further can limit the exposure.

Even though your build failed, the attacker still achieved remote code execution.

When all package installs are proxied through an internal feed, even if the malicious package is removed from upstream it will remain cached, making it easier to detect and analyze. Internal feeds that remember

each package version's original source will likely show one or more private versions of the package, with the most recent version(s) being cached from a public source.

In the following pages, we review the available mitigations for these risks. Each approach can mitigate the issue independently, although the varying impact on development teams may make one preferable over the alternatives.

Mitigation strategies

1. Reference one private feed, not multiple

Most package manager clients will query all package feeds listed in the local configuration without regard for order or priorities. npm is a notable exception as it requires controlled scopes, which we discuss in the second mitigation strategy.

For package managers who *do not* prioritize feeds, we recommend configuring the client to reference a single private feed. This may require pushing public packages to your private feed manually or configuring the private feed to pull them automatically.

Protect against broken updates and targeted attacks

Configuring the package manager to use only a single source isolates you from unexpected public feed changes. This includes broken updates and removals, as well as targeted substitution attacks. Azure Artifacts offers upstream sources to combine multiple feeds with suitable prioritization. If you cannot combine feeds into a single source, you will need to ensure that each package name referenced is controlled by trustworthy publishers across every feed.

Using a single feed is strongly recommended for the Python Package Index, NuGet Gallery, and Maven. To ensure your projects are following this recommendation:

- **For Python:** Use the index-url option in pip's configuration file or command line to specify the feed, overriding the default. Avoid the extra-index-url option, which is additive and may lead to having multiple indexes.

- **For NuGet Gallery**: Ensure your nuget.config packageSources section starts with a <clear /> entry to remove any inherited configuration, and use a single <add /> entry for your private feed.

- **For Maven:** Configure a single mirror that is <mirrorOf>*</mirrorOf> to direct all requests through a single repository. Alternatively, the default public repository can be overridden to disable the <releases> setting.

- **For Gradle:** There are no default repositories, making it easy to only specify your private feed.

Even with this configuration, if your feed allows public packages to override private packages, a substitution attack may still be possible. You should either ensure your feed is configured to disallow this, claim your private packages' names on the public index, or use another mitigation.

2. Protect your packages using controlled scopes

Some package managers support controlled scopes, namespaces, or prefixes. The details vary by ecosystem, but the purpose is to protect a range of package names. These can be used with packages that you control to protect against an attacker publicly claiming a name that you use privately, or to provide confidence that your own team released public packages.

Here is our recommendation for using controlled scopes as your mitigation strategy:

- **For npm:** Using a scope prefix in combination with registry configuration allows you to specify the source for each package. Because only a single registry will be searched, this protects against substitution attacks through the public registry. These options can be configured for each project or an entire machine using an npmrc file. Similar options exist for Yarn through the .yarnrc.yml file.

- **For NuGet Gallery:** An ID prefix can be registered by publishers to restrict uploads to the public gallery. Packages under a registered prefix can only be uploaded by approved accounts, which also protects against public substitution attacks. This reservation can be done whether you intend to publish your packages to NuGet.org or not. Using a registered ID prefix for private packages helps ensure that an attacker cannot claim any of your names.

Note that both approaches may require changing the names of packages and updating any code that uses them.

3. Utilize client-side verification features

Beyond the protection offered by carefully managing sources, package managers provide additional client-side verification features to protect against supply chain attacks. These include options such as version pinning and integrity verification.

Version pinning is recommended as the baseline mitigation and is supported by most clients. Specifying precise versions for packages and transitive dependencies, rather than an open range ("3.5.4" rather than ">=3.5" or "3.5.*"), will mitigate forced upgrade or downgrade attacks. However, they will not prevent a compromised index from serving an alternate package and claiming it to be the same version.

Specify precise versions for packages and dependencies to mitigate forced upgrade or downgrade attacks.

Integrity verification: While most indexes use HTTPS and their own integrity mechanisms, you can add additional protection against package substitution attacks with local integrity verification. Integrity verification ensures that a downloaded package is identical to the first time it was downloaded and will abort if any inconsistencies are detected.

Here's how to mitigate risk using these features:

- **For npm:** Installing from a package.json file automatically updates the package-lock.json file with versions and file hashes of packages. When package-lock.json is included with your project, running the "npm ci" command will replicate the install using version pinning and integrity checking.

- **For NuGet Gallery:** A packages.lock.json file can be enabled for your project, which will be automatically created on "nuget restore." When the file exists and is included with your project, it will be used by "nuget restore --locked-mode" to validate that the packages have not changed using version pinning and integrity checking.

- **For Python:** Pip's hash-checking mode ensures that the downloaded file matches a known SHA256 hash stored in your project. Any attempted package substitution attack must compromise both server and client.

Generating the hashes currently requires an additional tool such as pip-compile.

- **For Maven:** Dependencies are checked for modifications since the original upload but cannot be automatically verified against prior installs.

- **For Gradle:** Gradle dependency verification for packages downloaded from Maven Central can be enabled by following this documentation.

Tool comparison

Here is an overview of our recommendations for each of the following public package feeds:

	How to handle multiple indexes	**Our recommended configuration**	**Best practice**
Gradle	Gradle only uses explicitly listed repositories but will select any with the best available version.	Specify a single private Maven repository and enable upstreams on the private repository.	Enable checksum verification and signature verification in your Gradle configuration.
Maven Central	Multiple repository URLs can be specified in user or project profiles. These are queried in order, though the order is not obvious from any one configuration file.	Specify a single mirror for all repositories, to ensure your private repository takes priority. Enable upstreams on the private repository.	Maven currently has no integrated features for further checks. Switching to Gradle is recommended for checksum and signature checks.

npm	Registries are linked to a package name scope, making npm safe for properly scoped packages**.**	Either use scopes for all private packages or override the default registry with your private registry and enable upstreams.	Also include package-lock.json with your sources and use "npm ci" to install matching packages without performing any upgrades.
NuGet Gallery	Multiple package sources specified by user or project. Latest version from any source will be installed.	Clear all packageSource settings in project configuration or nuget.config, add only your private gallery, and enable upstreams.	Prefer the "nuget restore --locked-mode" command and include a generated packages.lock.json with your project.
Pip	One default package index and multiple extra index URLs. Latest version from any index will be installed.	Use pip's index-url setting to specify your private index and enable upstreams. Avoid extra-index-url.	Use "pip-compile" to generate locked file with hashes and enable hash-checking mode when installing.
Yarn	Registries are linked to a package name scope, making Yarn safe for properly scoped packages.	Either use scopes for all private packages or override the default registry with your private registry and enable upstreams.	Also include yarn.lock with your sources and use "yarn install –immutable --immutable-cache --check-cache" to ensure matching packages are present.

Mitigating using Azure Artifacts

Azure Artifacts is part of the Azure DevOps suite, available in the online service and the on-premises Azure DevOps Server. It is a fully integrated package management system with interfaces for the tools used in .NET, Python, Node.js, and Java ecosystems.

Azure Artifacts provides upstream sources to automatically merge packages from public feeds into your private feed for all ecosystems. Configuring upstream sources is the recommended way to enable the single source mitigation described earlier in this whitepaper.

Since February 2021, the hosted Azure Artifacts service has enabled additional protections to prevent public packages from unexpectedly replacing or merging with private packages. This mitigates substitution attacks when using a single Azure Artifacts feed that upstreams to other feeds, including public feeds.

For on-premises deployments, consult the Azure DevOps Feature Timeline.

To mitigate a substitution attack, Azure Artifacts automatically prevents public versions from replacing or merging with private packages.

Summary

While essential to modern development, public package feeds add risk to your supply-chain security that can be mitigated by using private feeds. However, improper configuration can expose your automated builds to unexpected substitution attacks, leading to remote code execution and compromise.

Ensure all your private packages are unavailable on the public feeds or accessed through a single feed that safely merges public and private views. Package manager configurations should avoid listing multiple feeds, as most tools do not prevent external sources from overriding internal ones. Where possible, enabling hash, checksum, or signature verification further protects against substitution attacks.

For more information, here are some resources that can help you get started:

- Azure Artifacts guidance on upstream sources
- NuGet guidance on signature verification
- Gradle guidance on dependency verification
- Pip guidance on hash verification
- Npm ci command

Version: 1.0

Publish Date: February 9, 2021