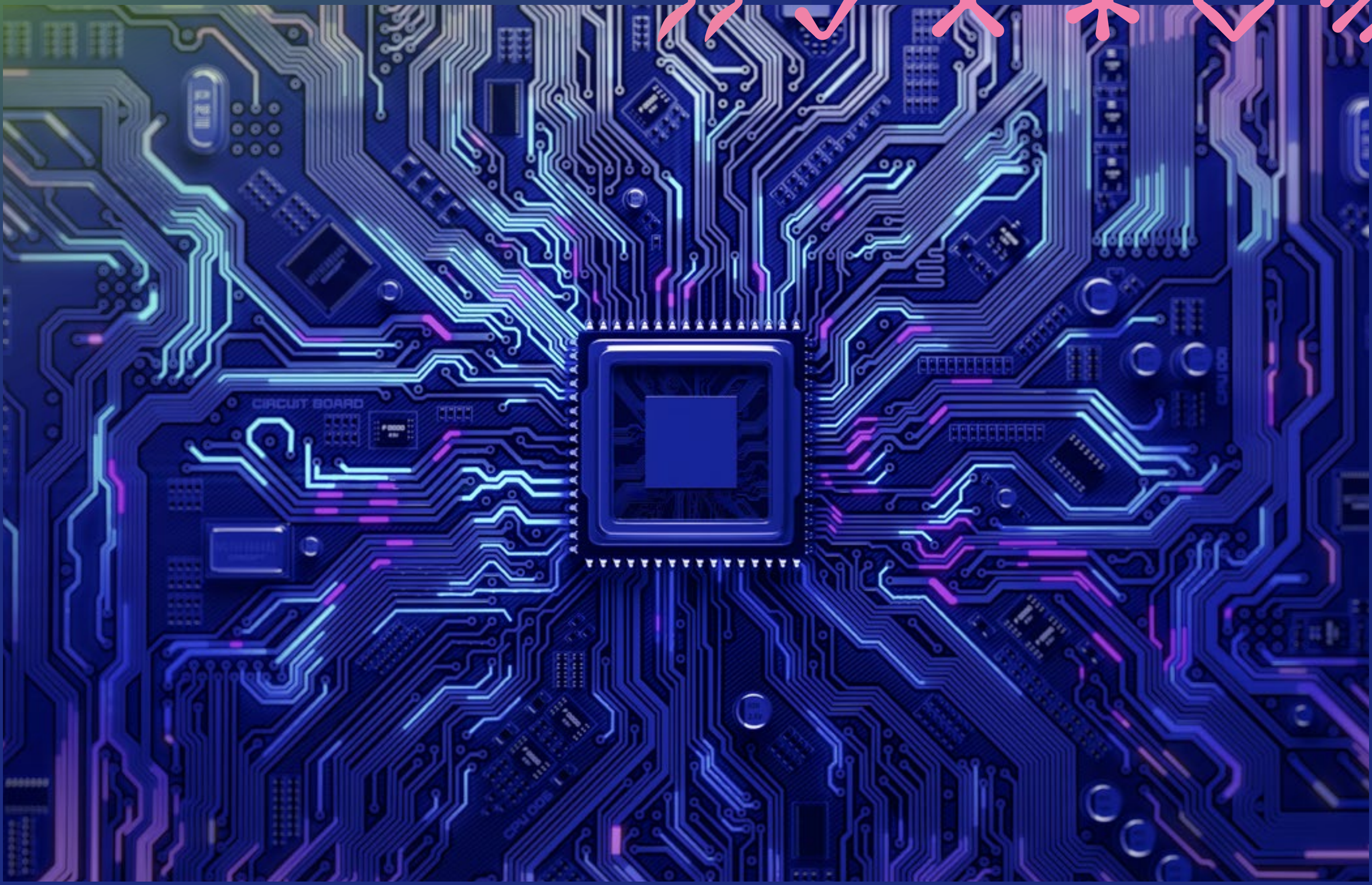
Checkmarx
The world runs on code. We secure it.

WHITEPAPER

An Introduction to Open Source Supply Chain Attacks

Table of Contents



Introduction

When we think about *Supply Chain* attacks, we usually think of third-party suppliers. Traditionally, these are organizations that supply products or services to their customers on the customers' behalf. Under some sort of contract or agreement with their customers, third-party suppliers typically have access to many different types of sensitive data about their customers, their customers' employees, and other client-specific data. This access to private information makes third-party suppliers a prime target for cyberattacks.

However, in this past year, we have seen a surge in supply chain attacks that are targeting a different supply chain—**the open source software supply chain**. Instead of pursuing third-party vendors, their systems, and platforms, attackers are starting to shift their focus, taking direct aim at developers by targeting the developers' best friend: open source software.

The main reason that open source is a viable target is due to the pervasiveness of its use in today's codebases. For organizations who thrive on the software they develop, much of that software is actually open source. In fact, several studies show that open source adoption continues to rise. One study shows that 99% of codebases contain at least some open source, with a growing trend of nearly 445 open-source components per codebase. Another study shows that 35% of commercial applications reviewed have at least some open source components. When analyzing code that was developed for internal use, the percentage grew to 75%.

Certainly, there is a growing concern among organizations who rely on their custom software, used to primarily run their own business operations, and serve their clients online. There are a host of reasons why open source usage makes sense, but most of them revolve around time-to-market demands and saving time and money by not reinventing what has already been proven to be effective. Regardless of the reasoning, open source consumption will continue to increase, resulting in increased risk as well.

This white paper is designed for those looking to learn more about the open source software risks they are likely being exposed to. Simply put, if you are not aware of the risks, then you will not be able to manage them effectively.

Open Source Challenges

The increased reliance on open source creates many challenges. First is the need to be *up to date* with the latest and greatest versions. Otherwise, open source libraries quickly become outdated, and even more problematic, updating them can often lead to unwanted functionality or defects. Second, the lack of visibility into an organization's open source usage sets them up for a number of vulnerability issues and growing risks, which could result in disaster. For these reasons, malicious actors are increasingly focusing their attention on open source libraries as the delivery mechanisms to distribute their attacks.

In this white paper, we will look at some of the most popular ways attackers are manipulating open source packages and repositories to take advantage of unsuspecting developers, and consequently, their organizations. These tactics are often very subtle and can present themselves as the simple misspelling of a package name, or a "new" version of a package that gets pulled in a codebase as part of an automated software build process. Other types of attacks prey on abandoned repositories that can be redirected to malicious code, or brand new repositories that are masquerading as trusted entities.

One of the worst outcomes from open source supply chain attacks starts with infecting a developer's computer. This can be particularly damaging due to its relatively high-level privileges and access to the organizations intellectual property (IP). In fact, there are many examples of attacks that targeted developers' computers, using them as catalysts for other attacks like ransomware, gaining remote access, credential stealing, etc.

So, what are the hallmarks of successful supply chain attacks? They are generally:

> Infect/affect a well defended "link" in the chain by targeting a trusted, but less defended one.

> Multiplying impact by infecting one "link" in the chain, that will end up affecting multiple links in the chain.

List of Open Source Supply Chain Attacks

In the next sections, we'll discuss many different open source supply chain attacks and the TTPs in use. Plus, we'll provide a few recommendations to avoid falling victim, and, if you do, how to lessen their potential impact.

Confusion-based Attacks (aka Dependency Confusion)

Throughout this section, we'll highlight a number of different types of supply chain attacks. All of these are a type of a *confusion-based* attack. These attacks make use of a wide variety of strategies ranging from attacks that prey on typos that developers make while searching for package names, to vulnerabilities in package manager configurations and operations.

Internal vs. External Repository Confusion

Many organizations use an internal repository for hosting open source libraries with the goal of making sure that all packages are coming from a *known and safe* location. In an ideal world, this would prevent any kind of confusion. However, in many organizations, package managers download the dependencies from internal private repositories, but they also allow the use of public repositories in certain use cases.

In these types of scenarios, package managers are allowed to retrieve libraries from both private and public repositories based on availability, but not based on *specific logic* to retrieve a very *specific package* from a very *specific source*. In some cases, this logic can be configured to select *external first*, which is often the reason some of these attacks are so successful. This creates an attack vector for malicious actors and it's one of the biggest reasons that attackers can create confusion.

The risks only get higher when attackers can direct these types of dependency confusion attacks towards *internal* packages. In these scenarios, attackers can strategically place malicious code in an external package (named the exact same way as the internal package) in a public repository in hopes that the package managers are configured in such a way that the malicious code is unknowingly downloaded into the organization's application.

Protecting Your Organization from Internal-External Repository Confusion Attacks

Some organizations have taken the initiative and registered the packages names in the relevant package manager. For example, company "aaa" wrote a python package for internal usage using the name "aaa-sdk". If someone would maliciously register a public python package under that name, they might find themselves infecting the internal infrastructure of company "aaa". On the other hand, if "aaa" registered a placeholder package under that name, they would defend themselves against dependency confusion attacks without needing to do any changes in all software where "aaa-sdk" is used.

Dependencies with Higher Version Numbers

Malicious actors can artificially raise the version number of a package to try to trick package managers into believing that a newer version of a package exists. For attackers who want to infect as many targets as possible, they will typically focus this attack on a very commonly used package in the hopes that it will be pulled down unknowingly. However, an even more targeted version of this attack can be focused on specific organizations.

Instead of targeting open source packages, this specific type of attack can even be used to target *internally developed packages!* But how would an attacker know the name of an internally developed package? Easy. When making changes to open source packages, some organizations contribute the changes back to the community.

Attackers often write code that can crawl through public repositories like GitHub, and look for package manager dependency lists, which often contain information about the private libraries that were used internally by the organization. An example of these types of files might be the package.json file in NPM, or the requirements.txt file in Python.

Previously, we briefly discussed how in many organizations, package managers download dependencies from internal private repositories. But they also allow the use of public repositories in certain use cases.

In these scenarios, we have a mix of both private and public package dependencies. The internal dependencies are hosted privately (for example in Nexus or Artifactory), while the public dependencies are hosted from the internet. For an example, see *Figure 1*.

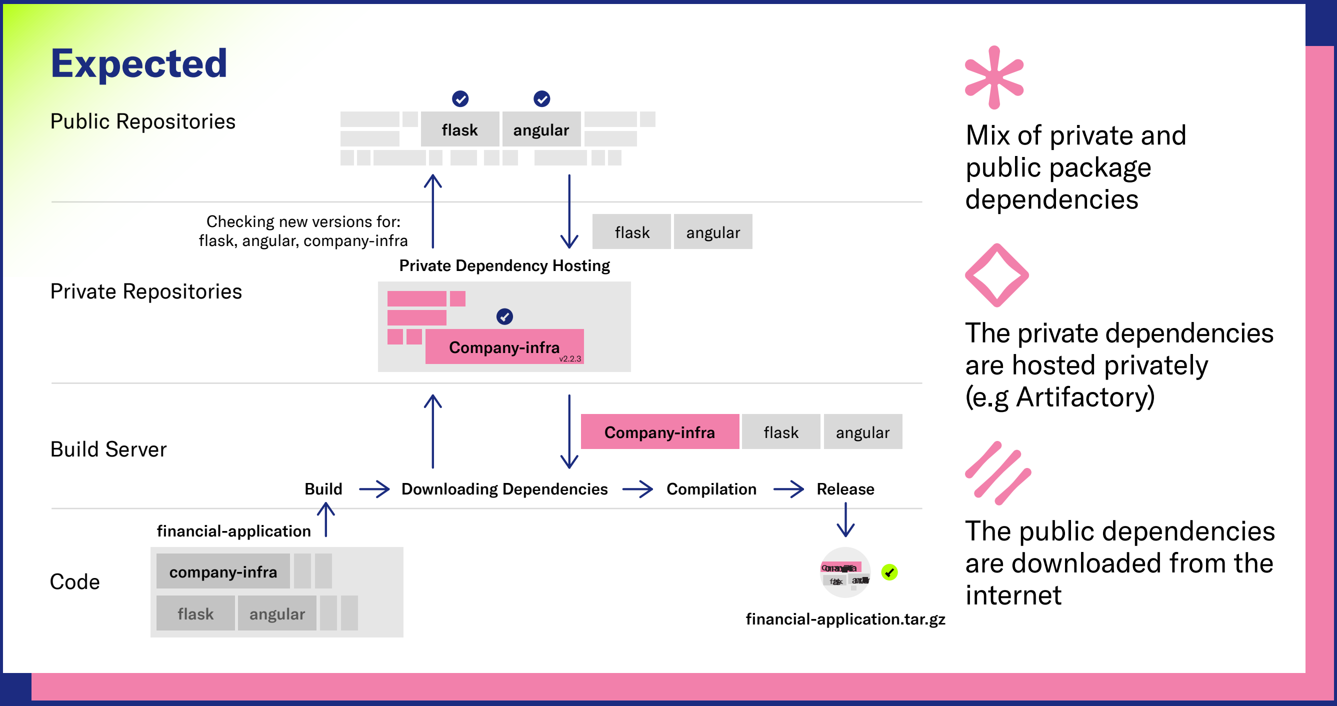

Expected
Public Repositories
flask
angular
Checking new versions for: flask, angular, company-infra
flask
angular
Private Dependency Hosting
Private Repositories
Company-infra
v2.2.3
Build Server
Company-infra
flask
angular
Build
Downloading Dependencies
Compilation
Release
financial-application
Code
company-infra
flask
angular
financial-application.tar.gz
Mix of private and public package dependencies
The private dependencies are hosted privately (e.g Artifactory)
The public dependencies are downloaded from the internet

Figure 1. Mix of private and public package dependencies

During the build process, the expected outcome is that the appropriate packages will be pulled from their respective locations. In Figure 1, “flask” and “angular” are both retrieved from public repositories, while “company-infra” is retrieved from an internal private repository. Now let’s take a look at a compromised scenario where an attacker publishes a malicious code package.

In this scenario, the attacker publishes a package using the *same name as the internally hosted package* and he gives it a new, *higher version number*. This package is published to a *public* repository.

Now during a build process, when the build is downloading dependencies, it still pulls down “flash” and “angular” from the public repository. However, this time, the process recognizes a *new* version of “company-infra”, which is a version that is *higher than the one stored internally* on the private repository. If your package manager is not configured properly, this can be a disastrous scenario, since the malicious package will be pulled from the public repository, instead of grabbing the safe package internally. See *Figure 2* for an example.

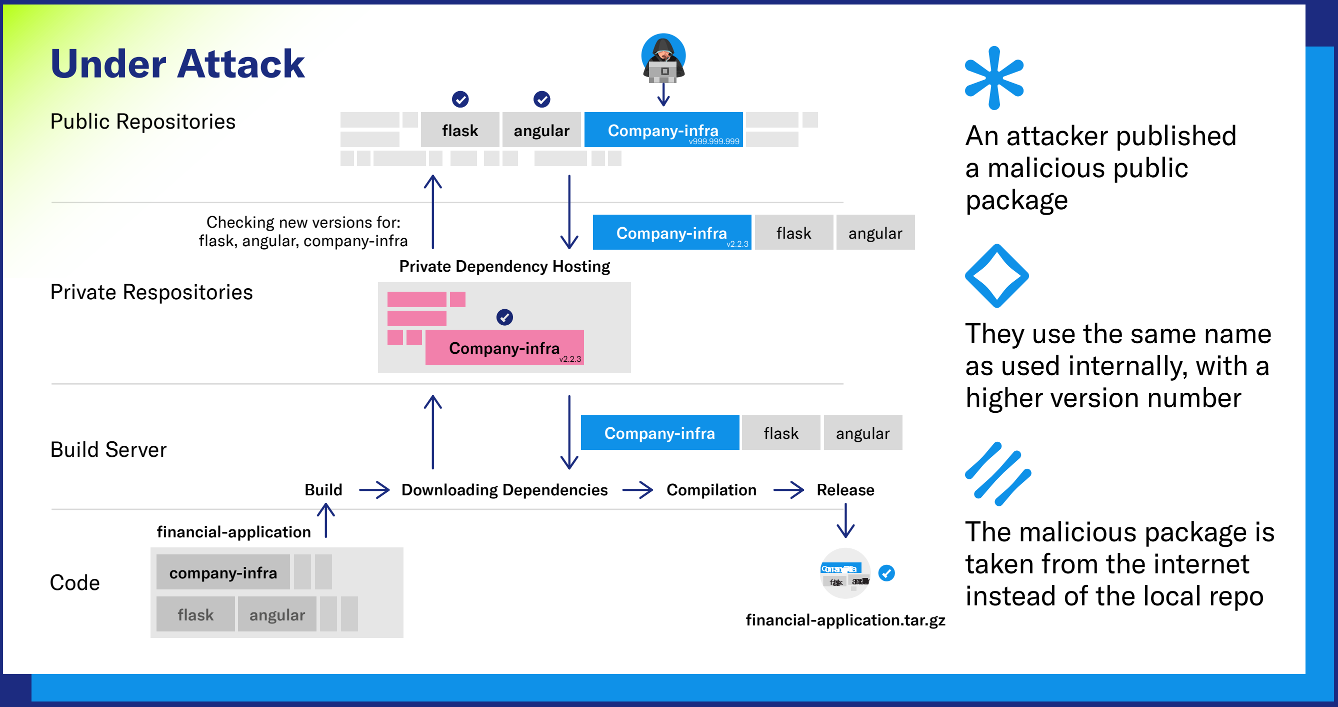

Under Attack
Public Repositories
flask
angular
Company-infra
v999.999.999
Checking new versions for: flask, angular, company-infra
Company-infra
v2.2.3
flask
angular
Private Dependency Hosting
Private Respositories
Company-infra
v2.2.3
Build Server
Company-infra
flask
angular
Build
Downloading Dependencies
Compilation
Release
financial-application
Code
company-infra
flask
angular
financial-application.tar.gz
An attacker published a malicious public package
They use the same name as used internally, with a higher version number
The malicious package is taken from the internet instead of the local repo

Figure 2. Example of higher version number attack

Protecting Your Organization from Inflated (Higher) Version Numbers

We suggest mitigating this risk in a similar way to the dependency confusion attack above. This risk can also be mitigated by registering a placeholder package on the relevant package manager with a low version number. In our effort to help organizations prevent these types of attacks, and to help identify dependencies in your project(s) that might be vulnerable to this kind of attack, Checkmarx has released this **solution**.

Typosquatting

In the context of open source package management, typosquatting attacks are caused by attackers creating and publishing malicious packages using names that are very close to legitimate ones. There are several types of typosquatting techniques, ranging from combosquatting, omission, repetition, and transposition, which we'll cover next. These variations of typosquatting are all designed to sneak a package past unsuspecting, or often very hurried developer eyes, by taking advantage of developers' minor typing errors. It allows attackers to unsuspectingly introduce malicious packages into development environments and then spread into organizations at large.

Typosquatting attacks have a very wide-ranging impact, simply because the attack preys on an inadvertent spelling mistake, which is something every developer has likely done many times. Add to this the fact that package names are becoming more eclectic with their naming conventions, and it can quickly become challenging to understand what is real, and what is being passed off as real, but is actually malicious.

Combosquatting

Combosquatting is a technique that often tries to impersonate legitimate open source packages by adding (or often appending) common words, terms, or letters to the authentic package name. Combosquatting is considered to be the most common of the typosquatting examples.

Here are a couple of examples of Combosquatting:

> dflow **-** dflow-node > xpath-converter **-** xpath-converter.js > lodash **-** lodashs

In the examples above, "node" and "js" are both frequently used terminology for developers who work with JavaScript, making these types of attacks very difficult to protect against by relying on human awareness alone.

Transposition

The focus for this type of confusion attack is when attackers switch the position of two adjacent letters, which is often completely overlooked, due to the way the human eye and brain interact. Often the most common errors in typing occur when users press keys in the wrong order, which most of us know happens quite often. This fact, combined with common spelling mistakes, makes transposition errors very hard to detect.

One of the most well-known examples of transposition typosquatting is the now-removed package “electorn,” which was created as a spinoff from the “electron” package by simply switching the order of a few letters as follows:

> electron **-** electorn   > middleware-js **-** middelware-js

Omission

By definition, omission is centered around a person or thing (in this case a letter, word, or phrase) being left out or excluded intentionally. The most common omissions are usually a single letter or character, like a hyphen. This type of attack is often targeted towards words that have repeating letters in them.

The NPM registry **updated their naming rules** recently after a user on Twitter informed the NPM community that a package with a very similar name to the popular “cross-env” package was sending environment variables from its installation context out to npm.hacktast.net. See an example below:

> cross-env **-** crossenv

In total, NPM found around 38 different packages that were attributed to some form of omission-based typosquatting.

Repetition

Repetition takes advantage of words that sound like they may have reoccurring letters, or are often misspelled by lots of people. A common example is the JQuery package, which is often mistyped with repeated letters.

> jquery **-** jquerry

Protecting Your Organization from Typosquatting Attacks

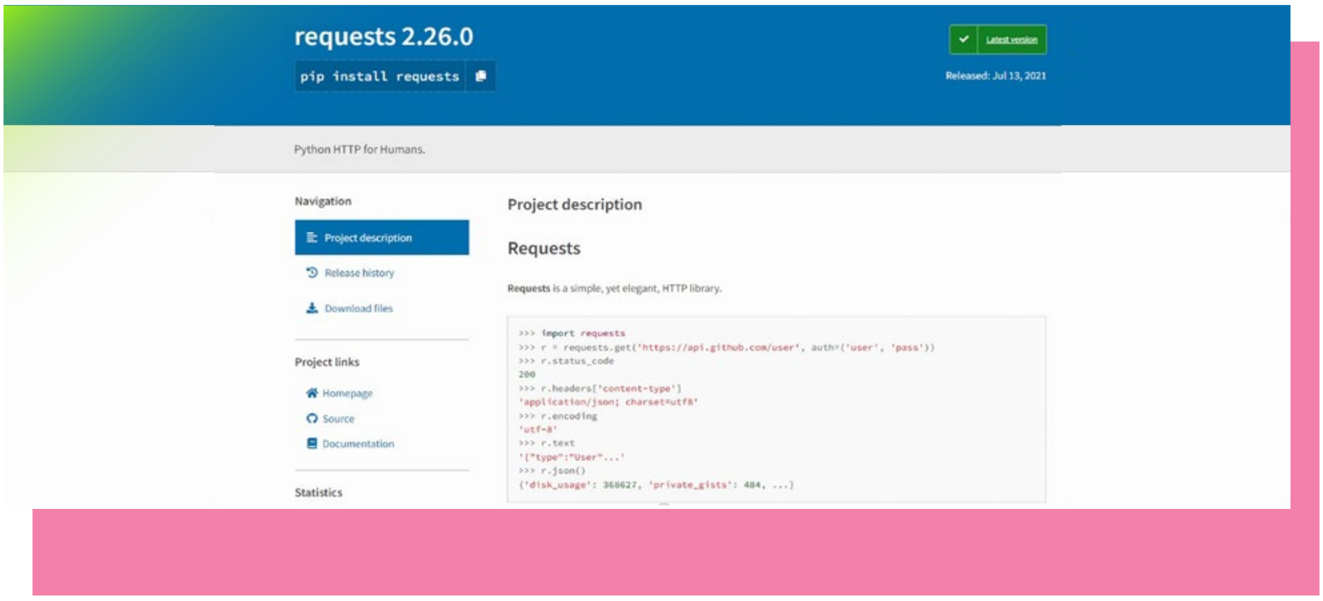
These types of attacks are difficult to protect against by vigilance alone, since they prey on the one consistent truth that all developers eventually face—distractions. Developers are constantly given more responsibilities and shorter deadlines due to organizations demanding newer features, faster. These types of working conditions force developers to work both *faster and more efficiently*, which is often difficult to do.

As a result, developers need a completely reliable and consistent way of identifying which open source packages are being used within their applications. A traditional Software Composition Analysis (SCA) tool can assist with this requirement; however, not all of them are capable of looking for packages with typosquatting-like attributes.

For many SCA tools, if a package is misnamed by any of the strategies outlined above, the tool will most likely not recognize the package at all, and often completely ignore it. More advanced tools might list the package but will not be able to match it to a known package repository, so there will be little to no additional information.

The most advanced and comprehensive SCA solutions will perform additional analysis to determine if the package is pretending to be something it is not. They perform this analysis by looking at pre-existing typosquatting examples, then applying the same principals to other package names. When found, these SCA solution will reflect the package as malicious giving the developers a chance to remediate *before* deployment, which is the ideal way to effectively manage risk.

One practice that might help avoid this kind of mistake is to always copy the installation command from the package manager web page. Most of these web pages provides easy to copy commands to be used in your CLI. Looking at the authenticity of this web page can reduce the number of mistakes, although it is not foolproof. See *Figure 3*.



```
pip install requests
```

Requests is a simple, yet elegant, HTTP library.

```
>>> import requests
>>> r = requests.get('https://api.github.com/user', auth=('user', 'pass'))
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf8'
>>> r.encoding
'utf-8'
>>> r.text
'{"type":"User"...'
>>> r.json()
{'disk_usage': 368627, 'private_gists': 484, ...}
```

Figure 3. Python HTTP for Humans: requests 2.26.0

Repository Jacking (aka ChainJacking)

Repository Jacking (also known as ChainJacking) refers to a lesser-known type of attack that can occur when the current owner of a repository changes their username, brought about by a specific feature that exists in GitHub called "**Repository Redirects**" This happens when an existing and trusted repository URL starts to redirect to a different repository.

Redirects can happen in three different ways:

1. When a user *renames their repository.*
2. When a user/organization *renames their account.*
3. When someone *transfers a repository to another user* or organization.

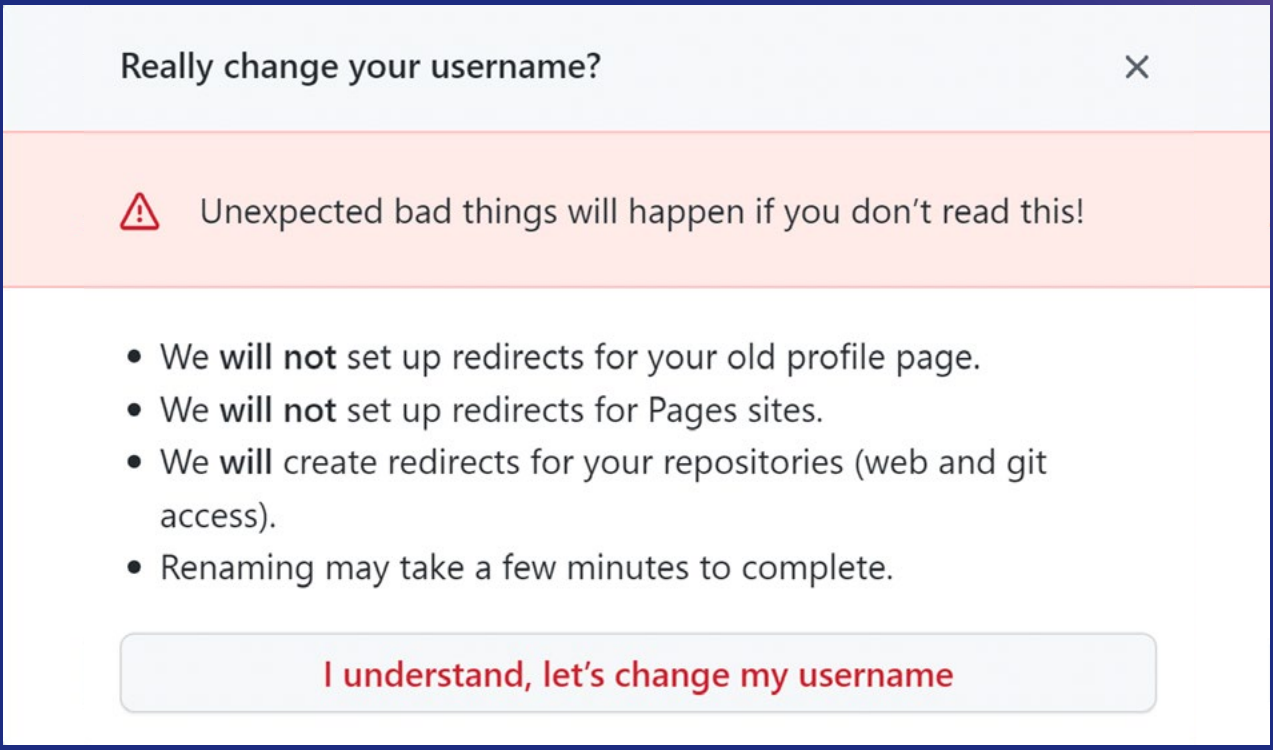
In theory, this is a helpful feature that most embrace. However, the redirects open the door for attackers to masquerade their code as known entities. All they need to do is re-register the abandoned GitLab account name, provide code to a same-named repository, and then any project that depended on that repository would start pulling code from the newly re-registered repository. Repository Jacking can be especially challenging with the Go programming language.

Slipping through the cracks between the designs of GitHub and Go Package Manager could allow an attacker to take control of popular Go packages, poison them, and even infect developers' and users' computers.

Go build tools provide an easy way for developers to download and use open-source libraries in their projects. Compared to other languages such as Python and Rust, Go doesn't use a central repository to download libraries from. Instead, the Go tooling pulls code packages straight from version control systems such as GitHub. This fact is likely to increase risk.

GitHub is the largest source-code repository on the internet, hosting the majority of Go packages. One feature GitHub provides allows users to change usernames, abandoning their old ones, and leaving them available for others to claim. When changing the username, the old name becomes available for anyone to re-register them. Until the old name has been registered again, GitHub redirects any old URLs to the new username URL. It is up to the previous owner to change all of their old URLs.

The change of username process is quick and simple. As shown below, a warning lets you know that all traffic for the old repository's URL will be redirected to the new one. See *Figure 4*.



Really change your username?

Unexpected bad things will happen if you don't read this!

- We **will not** set up redirects for your old profile page.
- We **will not** set up redirects for Pages sites.
- We **will** create redirects for your repositories (web and git access).
- Renaming may take a few minutes to complete.

Figure 4. Username change dialog on GitHub

What GitHub fails to mention is an important implication that it does list in its documentation as follows:

> ***"After changing your username, your old username becomes available for anyone else to claim..."***

Since the owner of the open-source project has no control over other projects that use their package, this can be misused. An attacker can easily claim the abandoned username and start serving up malicious code to anyone who downloads the package, relying on the credibility gained by its former owner. Doing so in a popular Go package repository could result in a chain reaction that substantially widens this code distribution, and infects large numbers of downstream products and subsequent organizations. In fact, Checkmarx has published a **blog** that provides additional details on this issue.

Direct ChainJacking

To better understand the concept of Direct ChainJacking, let's step through an example as follows:

A developer named Annastacia opens a GitHub account under the username "Annastacia." She then publishes a useful Go package in a repository under the nam e "useful." Anyone who wants to use this package either downloads or installs it via the command:
> "go get github.com/Annastacia/useful"

Or imports the package into their code via: **> "import github.com/Annastacia/useful"**

This action will add an entry to the "go.mod" file, allowing the tooling provided by Go to update the package when new versions are released.

For example, imagine that some time goes by and thanks to its usefulness, the package becomes rather popular. For some reason, Annastacia decides she wants a shorter name for her repository, and with just a few clicks, she changes her GitHub username to "Anna".

Subsequently, two things will happen:

1. The username "Annastacia" is now available to be registered by anyone else.
2. All requests for "github.com/Annastacia/useful" are now redirected to "github.com/Anna/useful".

See *Figure 5* for GitHub username change example.

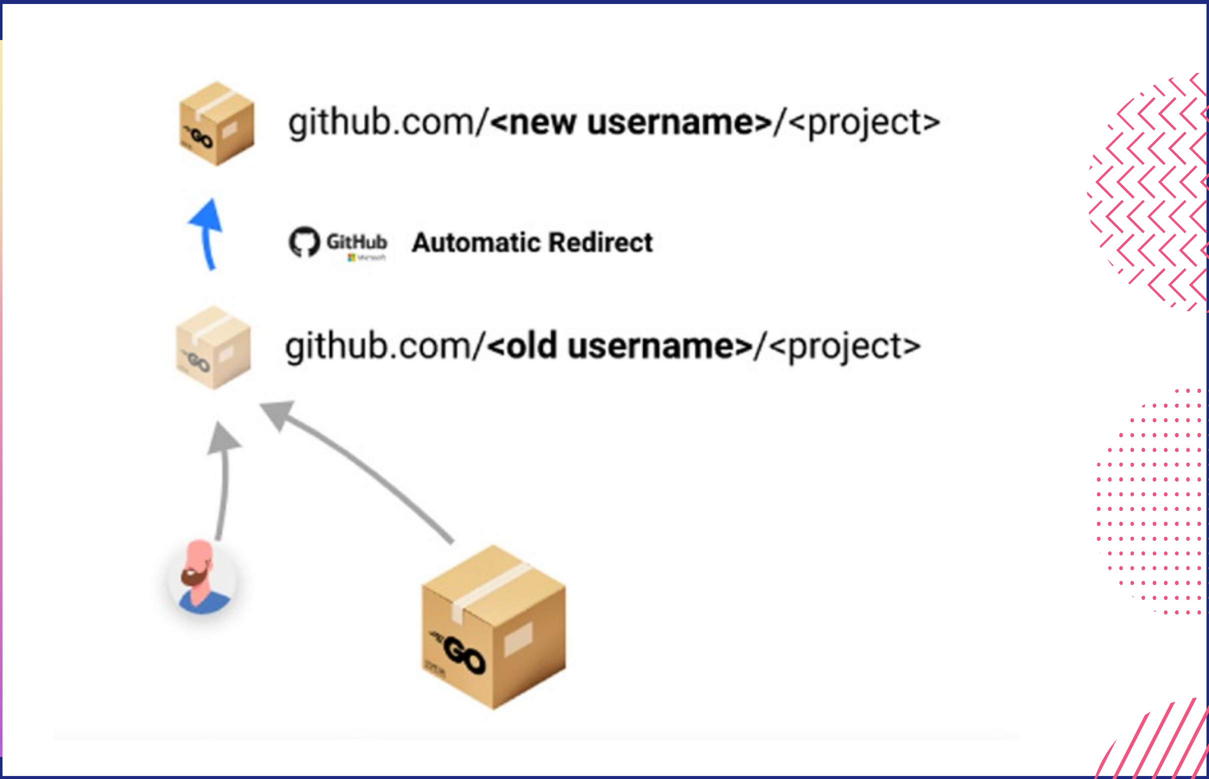

github.com/<new username>/<project>
GitHub Automatic Redirect
github.com/<old username>/<project>

Figure 5. GitHub Username Change Example

All current packages using “github.com/Annastacia/useful” can still use it as before, so nothing breaks and there are no user complaints (as of yet).

If an attacker manages to claim this “Annastacia” username, they can publish their own malicious code under the repository name “useful”.

This action breaks the redirect to “Anna/useful,” and GitHub now serves the attacker’s malicious code from “github.com/Annastacia/useful,” which could compromise anyone using the old URL.

The main concept is rather simple. Now, every new installation of this package can potentially infect the developer’s machine or cause other unwanted outcomes. Even more potentially damaging, any new package or third-party product written in the future that depends on this poisoned package, will also cause infection on the machine it is installed on.

Protecting Your Organization from Repository Jacking

As package owners, the best way to prevent this kind of attack is to avoid renaming your GitHub repository. If you must do so, make sure you retain the old username and leave it as a place holder to prevent malicious actors from grabbing it.

As developers, if you’d like to make sure your own code does not include vulnerable packages, you should use the Checkmarx open-source ChainJacking tool, to find which of your GitHub dependencies is susceptible to a ChainJacking attack.

Conclusion

Although many industry thought leaders indicate the open source supply chain is “at risk,” today’s developers and security teams need to first be equipped with knowledge of supply chain attacks, indicators of compromise, and attackers’ TTPs. This knowledge is invaluable. Beyond knowledge, developers and security teams also need to be equipped with technology (tooling) that can help them identify issues before allowing those issues to become part of their own risk profiles. Proper knowledge and tooling are imperative to safely navigate the open source supply chain.

Final Comments

We have provided a considerable amount of attack scenarios, unwanted outcomes, and suggested mitigation techniques for some of the scenarios. We hope you find this information useful when securing your own open source usage.

Checkmarx

Checkmarx at a Glance

1,675+
Customers in 70 countries

750
Employees in 25 countries

45%
of the Fortune 50 are customers

30+
Languages & frameworks

500k+
KICS downloads in 2021

The world runs on code. We secure it.