DepScan – Using Metadata to Detect Software Supply Chain Attacks

Prabhav Srinath *psrinath6@gatech.edu*
Georgia Institute of Technology

December 2023

Abstract

Package managers have emerged as indispensable tools in modern software development that help in installing, updating, configuring and uninstalling software packages on the computers, thereby streamlining the process of package installation and management across diverse interpreted languages. The increasing reliance on these tools have made them an attractive target to malicious actors. Over the past few years, there is a growing trend in attacks focused on abusing the behavior of these package managers. By deploying malicious packages in public repositories with uniquely crafted metadata, the package managers can be forced to download these packages unleashing a new wave of attacks called supply chain attacks. The two most prevalent attacks, namely the squatting attacks and confusion attacks are the focus of this project. The project aims to provide a tool to detect such attacks across public repositories. The approach used in this project is to contextualize the analysis of metadata and detect patterns exhibited by these attacks and alert the end users on the same. The lack of a dedicated tool coupled with the results of evaluation show that using context-based analysis is particularly effective in detecting potential attacks with lower false positives. This tool can also be integrated to be used against the package list fed into automation engines such as Jenkins, Chef, etc. to alert on malicious package thereby mitigating the risk of supply chain attacks.

1 Introduction

Package managers have become integral for automating the installation and management of packages across various interpreted languages [9]. However, their widespread usage has made them susceptible to a range of supply chain attacks, including typo/domain squatting, abuse of indexing behavior, and dependency confusion [10].

In 2021, security researcher Alex Birsan [1] uncovered a novel supply chain attack. By scouring the web for private package data from diverse organizations, Birsan successfully published identically named packages across multiple managers like pypi, npm, NuGet, RubyGems, etc. These managers unwittingly downloaded and installed these malicious packages, opening avenues for sophisticated attacks due to the potential for arbitrary/remote code execution upon package installation [1][2] [12].

While organizations have taken steps like securing namespaces and version pinning in public repositories to reduce their attack surface, various vectors persist, such as typo squatting [14], Python’s path precedence [5], and package manager behaviors, evident in the recent PyTorch nightly repository attack [4][15]. An attacker under the guise of ethical research released a package mirroring the nightly repository's name. Leveraging PyPI's path index precedence [4][5], the malevolent package was installed before its takedown [15].

Supply chain attacks exploit not only exposed private repositories but also the idiosyncrasies of package managers to pick up malicious packages. Given the proliferation of such attacks targeting private packages and package manager behaviors across public repositories, it is imperative to develop a ubiquitous detection tool to identify these vulnerabilities across various package managers.

2 Background

This section takes a deep dive into the stakeholders involved, the patterns seen in attacks, the current solutions available, the threat model and an outline of the two attack patterns considered in this project providing a comprehensive overview of the landscape for effective analysis and strategic planning.

2.1 Stakeholders

The primary stakeholders are

1. Developers: They are responsible for developing the packages and publishing these packages to repositories. It should be noted that developers can also be users because these developers may deploy packages which have other packages as dependencies. Developers may also be maintainers if they are actively updating their packages.
2. Registry Managers: They are responsible for hosting packages published by developers and facilitate the communication between developers and users. Essentially, they are responsible for allowing verified developers to publish packages, ensure they are not tampered and are highly available to all users and allow users to download these packages easily.

3. Maintainers: They are responsible for releasing new packages and maintaining existing packages. This could range from patching existing packages to developing streamlines build pipelines for new packages.

4. Users: They make use of the end-user applications that use the packages. They essentially consume the packages. They will access the public repositories and download/use the packages.

This project focuses on mitigating the risks faced by the users of packages downloaded from public repositories.

2.2 Existing Solutions

Along with industry recommended best practices, there exist tools that aim to mitigate the risks posed by publicly available packages. However, these tools pose drawbacks discussed below. These drawbacks are described in greater detail in the evaluation section.

1. ***Snyk.io's snync*** - Apart from analyzing packages against the NVD, the snync tool is used to compare if private packages are present in public repositories and report suspected lingering namespaces.

   Drawback – The tool does not address the concerns of typo squatting, index precedence behavior [5] and path squatting [4]. Furthermore, the solution is implemented and optimized only for npm package manager but does not scale to other managers as mentioned on their Github page [7]. A malicious actor can release a squatted package with malware and the tool will not be able to detect the malware if it does not have a corresponding CVE entry [7]. Additionally, the tool does not address the issue of scoped confusion attacks.

2. ***Confused*** - Confused is an open-source tool that performs checks to identify lingering namespaces. This tool has been augmented to support all public repositories.

   Drawback – The tool does not check for typo-squatting attacks. tool's only metric for dependency confusion attacks is to identify lingering namespaces. However, a malicious actor can introduce a scoped attack as seen in the Pytorch attack [4][15] which will not be detected by the tool. Additionally, its over reliance on comparison of private and public URLs to check for attacks results in a high false positive as mentioned on the GitHub page [6].

3. ***OWASP Dependency Check*** - The is a Software Composition Analysis (SCA) tool that attempts to detect publicly disclosed vulnerabilities contained within a project's dependencies. It does this by determining if there is a

Common Platform Enumeration (CPE) identifier for a given dependency. If found, it will generate a report linking to the associated CVE entries.

Drawback – This tool is similar to snync tool with an objective of tagging packages with CVEs. It does not possess the ability to evaluate package for supply chain attacks. [8]

4. ***Npm-Audit*** – NPM-Audit uses a known advisory database to alert the user of potential malicious packages or packages with malicious code (based on CVEs).

   Drawback – While the tool is the default for NPM to alert against malicious actors, the tool alert's but proceeds to install the package. Furthermore, the tool references an advisory database of known vulnerabilities and cannot check if the package is vulnerable to the dependency confusion attack by validating package metadata. [11]

5. ***DustiLock*** - DustiLock recursively inspects package directories for dependency confusion attacks in pypi and npm. This tool builds on the Confused tool to detect scoped confusion attacks.

   Drawback – While this tool it is better than trivial namespace resolution, the tool faces similar challenges as Confused. With the URL being the primary source of truth, the tool suffers from a high degree of false positives. [13]

6. ***Code Analysis Tools*** – These tools are able to decompile the package and identify code-based vulnerabilities or pattern of malicious code. These tools can be used in conjunction with the tool described in this project.

   Drawback - While static analysis tools can identify and tag threats with high accuracy, these tools general require an understanding of the coding language, the structure of the package and the package must be downloaded before analysis increasing scope for malware infections on download.

2.3 Threat Model

Before arriving at the solution, it is prudent to understand the threat landscape and the prevalent issues that need to be addressed in the solution. The objective is to enumerate vulnerabilities and explore attack vectors from the views of the package manager and the maintainers/repositories. This will assist in elucidating the attack surfaces and drive design decisions for the scanning tool.

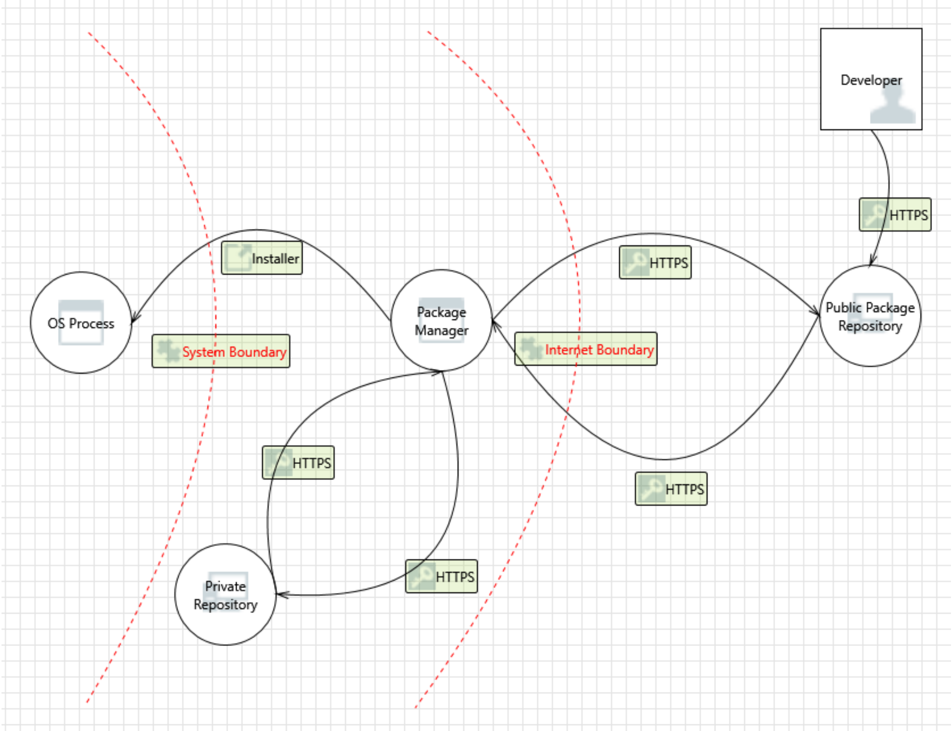
1. **Assets**

The assets under the purview of the objectives here are the packages and the end user systems. These are the assets targeted by adversaries.

2. Scope

The scope defined as part of the threat model is focused on the interaction between developers, managers, and maintainers. The scope is restricted in accordance with objectives to the data flow (model below) between the managers and maintainers.

3. Model

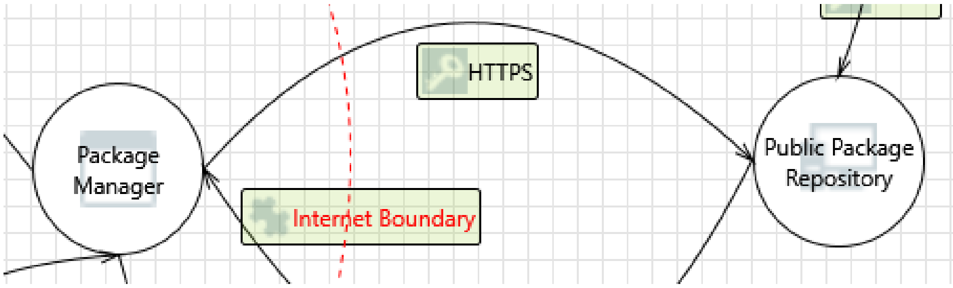


The data flow in the above diagram represents the interaction between end user ⇌ managers ⇌ developers. The two trust boundaries impacting data integrity in the flow are System (User-Kernel space) Boundary and the Internet Boundary.

The threat model is based on the various interactions in the system. The interaction between the manager and the repository is secure as it is managed on an internal network by the organization. The manager communicates with an OS process through either an IPC or a fork-exec'd subprocess and can contribute to data flow past the system boundary. All other communications are HTTPS communications.

4. Analysis

Interaction-1[Manager → Package Repository]

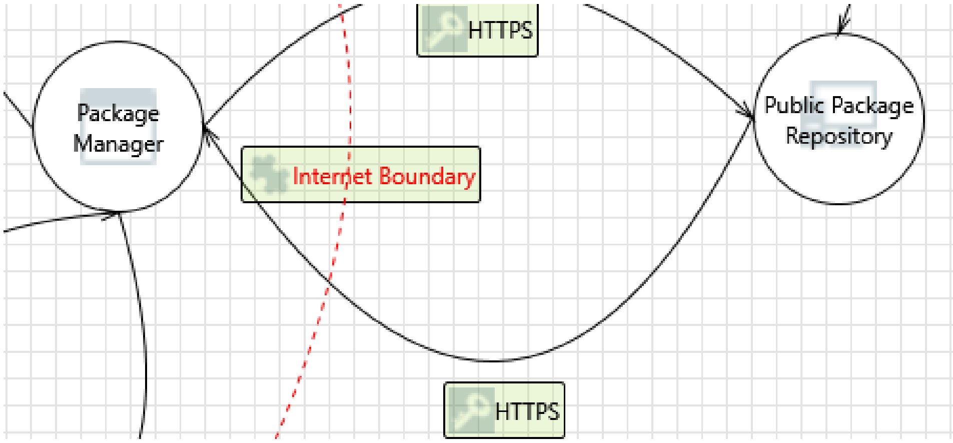


a. Elevation Using Impersonation [Priority: High]
Category: Elevation of Privilege
Description: Public Package Repository may be able to impersonate the context of Package Manager to gain additional privilege

b. Public Package Repository May be Subject to Elevation of Privilege Using Remote Code Execution [Priority: High]
Category: Elevation of Privilege
Description: Package Manager may be able to remotely execute code for Public Package Repository.

c. Potential Data Repudiation by Public Package Repository [Priority: Low]
Category: Repudiation
Description: Public Package Repository claims that it did not receive data from a source outside the trust boundary.
Mitigation: Consider using logging or auditing to record the source, time, and summary of the received data. (Out of scope for the project)

Interaction-2 [Package Repository → Manager]



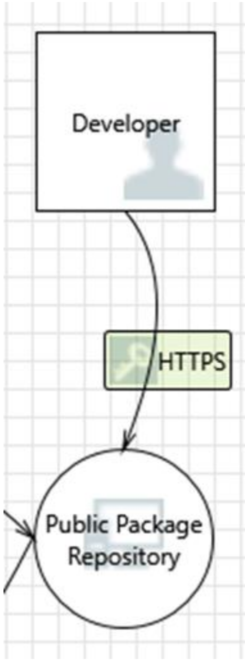
a. Package Manager May be Subject to Elevation of Privilege Using Remote Code Execution [Priority: High]
Category: Elevation of Privilege
Description: Public Package Repository may be able to remotely execute code for Package Manager.

b. Elevation Using Impersonation [Priority: High]
Category: Elevation of Privilege
Description: Package Manager may be able to impersonate the context of Public Package Repository to gain additional privilege.

c. Public Package Repository Process Memory Tampered [Priority: Medium]
Category: Tampering
Description: If Public Package Repository is given access to memory, such as shared memory or pointers, or is given the ability to control what Package Manager executes (for example, passing back a function pointer.), then Public Package Repository can tamper with Package Manager.
Mitigation: Consider if the function could work with less access to memory, such as passing data rather than pointers. Copy in data provided, and then validate it.

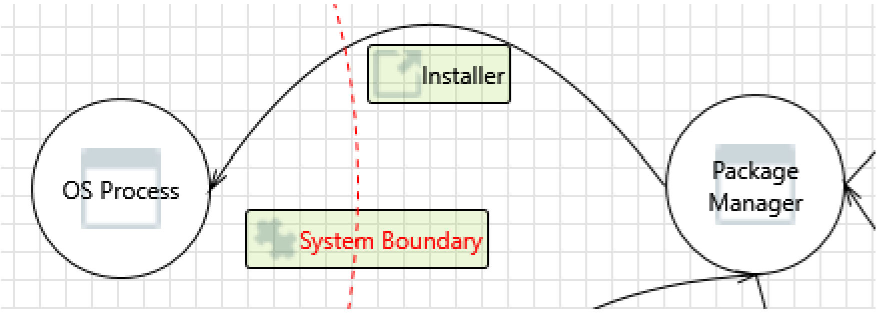
d. Spoofing the Public Package Repository Process [Priority: Medium]
Category: Spoofing
Description: Public Package Repository may be spoofed by an attacker, and this may lead to unauthorized access to Package Manager.
Mitigation: Consider using a standard authentication mechanism to identify the source process.

Interaction-3 [Developer → Package Repository]



a. Spoofing the Developer External Entity [Priority: Low]
   Category: Spoofing
   Description: Developer may be spoofed by an attacker, and this may lead to unauthorized access to Public Package Repository.
   Mitigation: Consider using a standard authentication mechanism to identify the external entity.

Interaction-4 [Package Manager → Installer]



a. Elevation by Changing the Execution Flow in OS Process [Priority: High]
Category: Elevation of Privilege
Description: An attacker may pass data into OS Process to change the flow of program execution within OS Process to the attacker's choosing.

b. OS Process May be Subject to Elevation of Privilege Using Remote Code Execution [Priority: High]
Category: Elevation of Privilege
Description: Package Manager may be able to remotely execute code for OS Process.

c. Data flow Installer can potentially change system parameters and lead the malicious installs [Priority: High]
Category: Tampering
Description: Package Manager may be able install code from the public repository as the OS process causing an infection of the system with adversarial code.

2.4 Attack Analysis

Now that threat landscape has been evaluated, the following section outlines the attacks considered for the project and deep dives into the patterns exhibited by these attacks.

1. **Typo-squatting attacks**

   Typo-squatting in public repositories involves creating malicious or misleadingly similar package or file names to popular or legitimate ones within these repositories. Developers or users might unknowingly mistype a package or file name when trying to install or download a legitimate resource, leading them to inadvertently download a malicious version.

   A deep dive [18] into this attack shows that attackers typical exploit human errors and strategic injections to achieve success in typo-squatting campaigns. These campaigns target popular packages and transform the package names in the ways described below [18],

   a. Given a word of length n, there are n ways to insert a character.
   b. Given a word of length n with r repeated characters, there are n-r ways to reduce characters.
   c. Given a word of length n with r repeated characters, there are n-r ways to reduce characters.

With a large number of transformations, attackers relied of phases of injections to delivery squatting campaigns [18]. These campaigns were strategically released based on distance errors [19][20]. Levenshtein distance [16] is a metric that demonstrates how far apart two strings are. More about the Levenshtein distance is discussed in the evaluation section but packages with a mixture of low and high Levenshtein distance names ensured that registering only 75% of all possibilities will ensure a successful typo-squatting campaign [18]

2. **Dependency confusion attacks**
   Dependency confusion is a sophisticated attack that targets the dependency management systems used by developers/end-users [1]. This attack exploits the way some package managers resolve dependencies. If the internal package isn't explicitly specified or locked to a specific version in the project's configuration, the package manager might prioritize the external (malicious) version due to its higher version number, assuming it's the latest or most updated version.

   In a dependency confusion attack, a malicious actor uploads a package with the same name as one a developer uses internally within their organization. This uploaded package might have a higher version number than the internal one. When the developer's build system looks for this package, it defaults to the public repository and unknowingly downloads the malicious version instead of the internal, legitimate one [3].

   Dependency confusion attacks rely on two characteristics based on previous campaigns [3][21].
   1. Registering a package with same metadata as a target package but with higher version.
   2. Registering a package with scope higher than the target package.

   By squatting in peculiar methods as described above, these attacks essentially force package managers (using path precedence and version precedence) [2][5] to successfully download malicious packages on victim systems. One of the more recent attacks [4] exploited the path precedence of pip in PyPI to upload malware that was downloaded over 5000 times over 3 days [4].

3 Solution

3.1 Core Idea

The core idea is motivated by the research of Ruia et al. [9] and Ladissa et al. [10]. These papers explored trends in repository/registry abuse through metadata, static, and dynamic analysis, enumerating various threats and evolving trends for maintainers/repositories to implement relevant security controls. However, this doesn't assist developers and end users. Nevertheless, these papers demonstrated a key point: the use of metadata in detecting supply chain attacks. All packages in public repositories must adhere to basic standards, including attaching metadata for each package published to these repositories, even those deployed with malicious intent. Therefore, metadata serves as a good indicator to identify such attacks.

Consequently, the project builds on this research to mitigate existing risks for end users. Employing metadata analysis provides two specific advantages. First, it is lightweight compared to static/dynamic analysis. Static and dynamic analysis tools require significant domain expertise to operate and comprehend the output. They also necessitate considerable user interaction, making integration into CI/CD tools that use package managers impossible. Metadata analysis tools offer a lightweight alternative promising adequate security analysis for the mentioned attacks, as previously explained. Secondly, since every package on a public repository has metadata, this data can be parsed and interpreted to detect abnormal patterns.

Additionally, the project aims to contextualize metadata analysis, helping to identify relevant bounds for analysis and prevent false positives. For instance, if a package was released with a version of 35.0 and grows from this number, testing against versions 0-34 and flagging them as missed/erroneous versions constitutes false positives because the bound encompasses the inclusive range of 35.0 to the current version. Similarly, bounds can be set for multiple security parameters, and the analysis can be carried out within these bounds. Furthermore, if a package is present on the local machine and is being updated, the analysis essentially boils down to matching the security features of the package on the local system with the package being downloaded from the repository. If the package is being downloaded for the first time, the analysis can fall back on public popularity metrics, comparing features such as download rate, package age, and valid URL certificates.

3.2 Feature Selection

To facilitate contextual metadata analysis, feature identification and classification are paramount. The two challenges to address are: 1. Ensuring the uniqueness of the selected metadata feature per package, and 2. Ensuring the feature's presence across repositories. Following the analysis of metadata in multiple repositories (npm, pypi and ruby gems), the following features were chosen for metadata analysis:

1. Package Name – Describes the unique name of each package while remaining consistent across multiple versions of a given package.
2. Author Name – Describes the author's name. While authors may change, repositories introduce author lists that encompass all authors, allowing tracking across packages for potential changes.

3. Author Email – Describes the author's email, useful for author validation and tracking across packages for potential changes.

4. Maintainer List – Describes all maintainers responsible for deploying new packages and patching existing ones.

5. Package Versions – Describes the current package version and lists all previous versions released for the package. These versions are strictly increase across newer package versions, aiding in tracking the package's historical releases.

6. Package Download URL – Provides the package's homepage URL, which remains consistent across versions. Furthermore, the certificates for these URLs can be validated against blacklists to identify malicious URLs.

7. Package Summary – Offers a single-line description of the package's function, aiding in identifying differences in packages.

8. Package Dependencies – Represents a strictly increasing list of dependencies required for the current package to function. Significant changes in the dependency list can indicate malicious actions.

9. Package License – Remains fixed across multiple package versions, and any change in the license can indicate potential malicious actions.

3.3 Implementation

This section describes the implementation of the project. It outlines the high-level design of the tool, the structure of the evaluation report, the classes, modules and structures used, and the rationale for selecting features for test.

3.3.1 Design

The project follows the pattern described in the figure below.

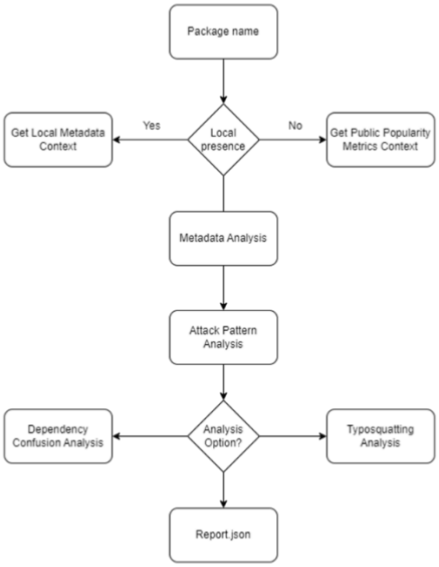


Figure 1

The tool uses a two-stage analysis approach. Initially, the tool parses package name from the packages file, then identifies if the package to be downloaded has a local counterpart. In this phase, the tool sets the context for further analysis. It either sets up the metadata context from local package metadata or if a local package is not present, the tool fetches features mentioned in the previous section from trusted public repositories.

Subsequently, two ubiquitous metadata objects are created. These are dictionaries with the features instantiated as dictionary key, value pairs. The primary intention of developing repository agnostic metadata structures is to allow for cross platform analysis. These structures are passed into the two analysis modules which perform relevant tests described in subsequent sections and generate an analysis report. The analysis is carried out in two stages, i.e., the metadata analysis and attack pattern analysis. The result of the first stage is fed into the second. Additionally, the first stage acts as a fast failure module where any issues with the metadata will lead to termination of analysis with the logging of relevant errors in the report. The tool is implemented as a two staged multi-module analyzer. Figure 2 outlines the design of the tool.

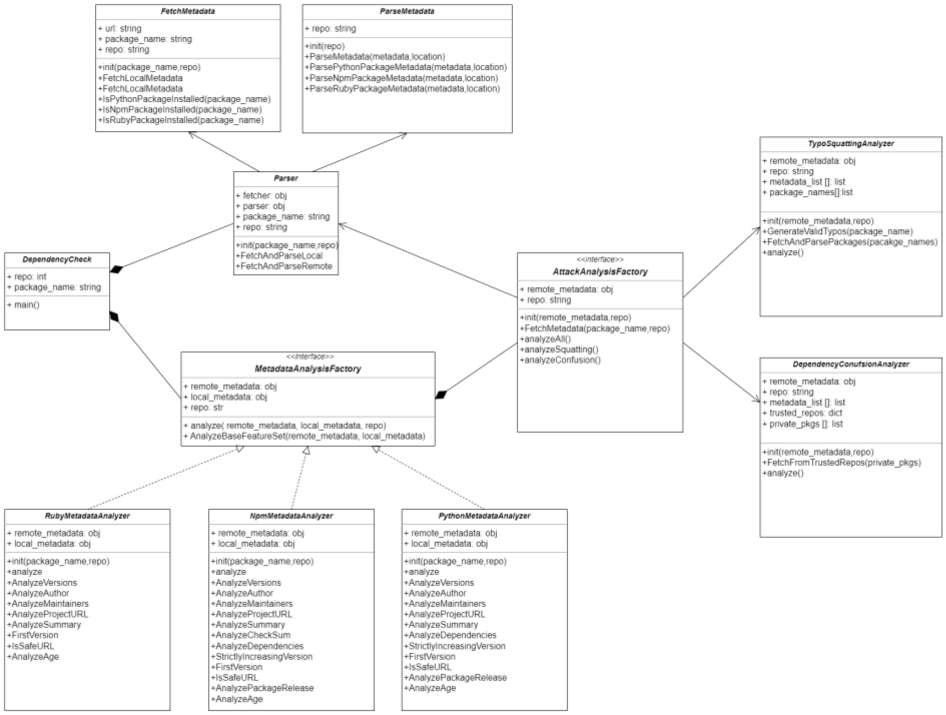


Figure 2

3.3.2 Modules

There are three major modules in the project: The Parser, Metadata Analyzer and Attack Pattern Analyzer. The former is responsible for preparing the repository agnostic metadata object while the latter two are responsible for carrying out the analysis.

3.3.2.1 Parser Module

This utility module serves two primary objectives: building the metadata object, and analyzing and setting the context. When initialized, it gets the metadata name, repository and it is responsible for building the agnostic metadata object. To build the metadata object, it contains methods and functions that parse package metadata for each repository.

Additionally, the module also verifies for the presence of the package on the local system and if not fetches the package metadata from trusted public repositories. The functions FetchLocalMetadata and FetchRemoteMetadata are implemented to achieve this task. These functions generate a local_metadata and remote_metadata objects with the major features represented as member of these objects as mentioned in the previous section.

3.3.2.2 Metadata Analyzer Module

This module is responsible for analyzing the remote_metadata object with local_metadata as context. Based on the presence of local_metadata, the module performs the tests mentioned in the subsequent section to evaluate if the metadata features are present and are constrained to valid bounds. This is implemented as an abstract factory called the MetadataAnalysisFactory. When initialized, it receives the local and remote metadata objects along with the relevant repository and it starts up the repository relevant analyzer. It gets an output as a JSON dictionary which it sends to the primary module to write to a file.

3.3.2.3 Attack Pattern Analyzer Module

This module is responsible for analyzing remote_metadata object without the context of the local_metadata. This module builds on the analysis of previous module and dives into testing specific metadata features to detect squatting or confusion attacks. This module is implemented as an abstract factory called AttackAnalysisFactory which initializes the various attack analysis modules and stores the result in a dictionary. Currently this factory class instantiates two modules,

1. TypoSquattingAnalyzer
   This module receives the remote_metadata and the package name. It follows the butter fingers protocol [25] to generate the typos at an edit radius of 1 and validates each of these potential candidate packages against the remote_metadata.

2. DependencyConfusionAnalyzer
   This module receives the remote_metadata and recursively searches well known public repositories for a similar package. If found, it tests the metadata of the candidate package with the remote_metadata and stores the result in a JSON object.

3.3.3 Report Structure

The evaluation report follows the JSON structure with each package being the key and the value is a dictionary of the result of the various analysis modules. Each of these modules has three lists, ALERT list, WARN list and FATAL list. The results of metadata tests based on severity are placed in one of these lists. The results per module follow the *TestName:Message* pattern. For instance, missing package name will generate the following error, *"ALERT: [AnalyzePackageName:Missing feature package name]"*. The intuition of building a nested JSON object is that these objects can be easily parsed.

Therefore, the result can be consumed by downstream tools in an automated setup to make decisions on either downloading or dropping the package.

3.3.4 Structure of Tests

The tests are divided across the MetadataAnalysis and AttackAnalysis Modules. Each of these modules decides on a set of tests based on the context value this section describes the various tests carried out by these modules and the justification for including these tests. Furthermore, considering base feature set of 9 features, if the number of messages in either of the three lists is greater than half the number of features, the message is upgraded. For instance, if there are 5 WARNs, these are moved to ALERT. Similarly, if there are 5 ALERTs for a single package, these alerts are upgraded to FATAL indicating to the user that there is an attack with a high degree of confidence.

1. MetadataAnalysis:
    a. Base Feature Set
    This function tests for the presence of all the relevant metadata features. If any feature is missing, this test raises a WARN message indicating that subsequent tests consuming this feature will not run.

    Justification
    This is a best practice to ensure that the tests fast fail in case the required features are not present. However, the message is a WARN at this stage as there could be a possibility that the feature is absent in all releases of the package and that must be validated in the subsequent tests.

    b. Local Context
    These tests will compare the metadata metrics of the local instance of the package and the remote package.
        i. Version Comparison: This test validates that the remote package instance has a version greater than or equal to the version of the local package instance. If greater, the test validates that the local package versions list is a subset of the remote package versions list. It raises an ALERT if there is a partial mismatch and a FATAL message if there is a complete mismatch between the version lists.

        Justification: If the package being downloaded has a local counterpart, it is likely a minor or major update. In this case the version must be great than the existing package. Additionally, the list of released versions of the remote package must contain the entire list of released versions of local package. While the version is controlled by the package developer and can be tampered with, the version list is injected by the repository and is a good indicator of tampering. This in accordance with the vector for confusion described in [10].

ii. Author Comparison: This test validates the author details for both package instances. There are three fields to be validated. The author_name, author_email and authors. If the author_name match fails, the tries to find the author of the local instance in the authors list of the remote instance. If found, the test generates a WARN message indicating the author has changed. If not found, the test generates and ALERT message indicating to the user that the package is developed by different developers.

Justification: An authors list is an append only log and any changes to authors will reflect as an entry into the authors. In context of the local package, if the author list is entirely different, this is an indication that the package is malicious. Additionally, [10] touched upon coercion and ownership transfers of packages. This test also tracks such ownership transfers. This is not a conclusive sign of malicious activity.

iii. Maintainer Comparison: This test validates that the local package's maintainers list and maintainers_email list are equal to or a subset of the remote package's lists. If neither of these test's pass, an ALERT message is generated.

Justification: Similar to the versions list, the maintainers list is a structure maintained by the repositories and can not be altered. Hence, if the list is different, it is indicative of a different package [10].

iv. Project URL Comparison: This test compares the URL of the homepage listed in both packages. If they match, the certificates of these URLs are also compared to ensure they are valid and point to the same domain. If these tests fail, a WARN message is generated.

Justification: Attacks such as in [4] take advantage of the path precedence of package managers to deploy scoped out packages that are downloaded. In such cases, the URL will not be the same. Packages from the same author/group are always released from the same github/webpages and a change in this can indicate a malicious package. However, the change in domain is not a definitive indication of a malicious actor, therefore generating a WARN message. This test in conjunction with other tests provides a definitive answer.

v. Summary Comparison: This test compares the summary of the local package with the summary of the remote package. If they are different, a WARN/ALERT message is generated.

   Justification: The summary is static across packages and a change might be an indication of an attack, thereby generating a WARN message. However, if the summary contains keywords such as squatting, confusion, attack as in [15], this is a definite indication that the package is malicious which is when an ALERT is raised.

vi. License Comparison: Compares the license for the local and remote packages and if they are different raises a WARN.

   Justification: The license for a project is acquired for the lifetime of the project and generally move from more permissive to less permissive [26]. If the license is downgraded from less permissive to more permissive, additional features must be checked to upgrade the WARN. If not, a WARN is generated.

vii. Dependency Comparison: Compares the dependency list for the local and remote packages and raises a WARN/ALERT if there is a mismatch.

   Justification: The dependency list is a comprehensive list of packages required for the package to operate. If the local package's dependency list is a subset its remote counterpart, this could mean additional dependencies were added thereby generating a WARN. If there is a complete disconnect between the two lists, this is indicated registry abuse in [9] and confusion/similarity attacks as described in [10].

viii. Repository specific metrics: This tests for feature fields specific to repositories in local and remote packages and generates an ALERT on mismatch.

   Justification: The specific metrics include keywords (npm) and checksum (gems) and these values do not change across package versions. If there exists disparity between these values, it is a strong indication of issues with the package.

c. Public Context

These tests run after the base feature set tests and solely depend on trusted public repositories.

i. Analyze Popularity Metrics: This test fetches the download metrics from the URL that hosts the package and validates this metric against a threshold. On failure, it generates an ALERT message.

Justification: Analyzing the popular 100 packages listed in npm and pypi, a download threshold for popular packages was established. If there are valid metrics for the package and the download rate (per week and per month) are lower than the thresholds, this is an indication that the package is not popular and could be malicious. Additionally, if the package's download metrics are absent in the trusted repository, this is indicative that the URL of the package must be verified and an ALERT message is generated.

ii. Analyze Missing Versions: This test validates the versions list of the remote package to ensure that all versions are present and there is no sudden jump of versions. If this test fails, it generates an ALERT.

Justification: The packages start from version 0 or higher depending on the repositories but are linearly increasing [9] across releases. A sudden jump of major versions can indicate a malicious package indicating a specific attack such as similarity or confusion attacks [10].

iii. Analyze Strictly Increasing Versions: This test is to ensure that versions in the version list are strictly increasing and a lower version package is not released after its higher counterpart. In such a case, it raises an ALERT message.

Justification: In [9], one of the attacks involve vulnerable publishes where an attacker releases a higher version first and then a lower version. In such cases the append only versions list will not be sorted indicating an attack.

iv. Analyze Package URL: This tests if the URL is valid and has a domain certification which has not been blacklisted. If so, it raises an ALERT message.

Justification: An attack in NPM repository [24] embedded values in metadata that were later picked up by the malicious hook code. In such cases, the URL can be a malicious URL that the code is trying to reach out to and validating the URL and its domain certificate against the internet blacklist will help identify attacks.

v. Analyze Package Age: These tests augments versions check in section (iii) to ensure that the package versions released all have increasing ages. A failure of this tests results in a FATAL message.

Justification: The versions list if altered can pass test (iii). However, the release timestamp is a value maintained by the repository and cannot be tampered with. Therefore, an unsorted age list represents a clear mismatch and potential attack.

2. Attack Analysis
This module specifically tests metadata patterns exhibited by the two attacks. This module assumes that since the program has reached this module, there are no errors in the metadata as it did not fast fail at the previous modules.

a. Typo-squatting Analysis
This module specifically tests for typo-squatting attacks. The features used for these tests are extracted from the analysis in Section 2.5. Furthermore, the analyzer first sets up the candidate names based on the Levenshtein threshold [16] and the butter fingers [25] generator. Post generation of the candidate package names, the metadata, if exists is fetched and run through the analysis.

i. Author Comparison: This tests to see if multiple packages share the same author and raises a WARN message on failure.

Justification: The same author might release multiple packages (such as a regular version and a beta/nightly version) and maintain multiple packages. Additionally, based on industry practices, the author might also squat the namespace for packages within the typo radius. If the author is different however, the package might be malicious or an intersection in the typo radius. Therefore, a WARN is generated.

ii. Popularity Comparison: This tests to see if the package is the most popular package in it’s typo radius. This is done by comparing the download rate (per week and month) against all the candidate packages. On failure, it raises an ALERT.

   Justification: A legitimate package is often well known and has a high download rate compared to its counterparts in the typo radius.

iii. Age Comparison: This tests to see if the package is the oldest package in it’s typo radius. This is done by comparing the age of the package against the candidate packages. On failure, it raises an ALERT.

   Justification: If the package is not the oldest package released, then it could indicate an attacker is trying to squat a popular package with this package. This is a good indicator to thwart attacks such as the pytorch attack [4].

b. Dependency Confusion Analysis
This module tests for dependency confusion attack [2][3]. This module performs recursive scanning of trusted public repositories to look for packages with same name. If no public counterpart is found, it generates a WARN stating that the namespace is susceptible to squatting. Else, it compares the security features of the candidate packages. Since packages can be hosted on multiple repositories, these tests prevent false positives. Additionally, majority WARNs are upgraded to ALERTs as in the metadata analysis module.

i. Same Version: This tests if the package from the trusted source has the same version as the remote package and generates a WARN on failure.

   Justification: Dependency confusion relies on similar package structure but higher versioning. Therefore, a mismatch of version between the two packages can abuse the precedence behavior of package managers forcing download of the malicious package [9].

ii. Same Summary: This tests to see a change in summary between the remote package and the package from the trusted source. Additionally, it checks for specific keywords (“attack”, “squatting”, “malware”) and generates a WARN on failure.

Justification: The NPM attack [24] and the Pytorch attack [4] had keywords (“attack”, “squatting”) in their summaries indicating they were squatting attacks but existing tools missed parsing and matching summaries. Therefore, it is a good practice to parse and detect such trivial attacks.

iii. Same Author: This tests to see if the author details between the remote package and the package from the trusted source are the same. If not, it generates an alert.

Justification: If the packages have a different author but other features are similar, then it is a high indication that the package is part of the dependency confusion attack.

iv. Same Project URL: This tests to see if the URL details between the remote package and the package from the trusted source are the same. If not, it generates an alert.

Justification: If a package with the same name is hosted on multiple URLs, it is a clear indication of a confusion attack.

v. Same Dependencies: This tests the dependencies of the remote package with the package from the trusted source and generates an alert on failure.

Justification: Similar validation to previous tests, if a package exhibits similar features but distinct dependencies, then the package is a candidate for dependency confusion attack.

4 Evaluation

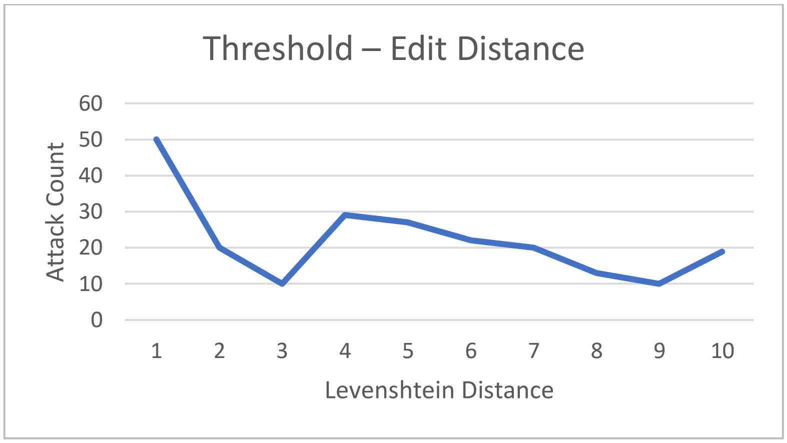
This section describes the evaluation process for the effectiveness of each of the tests that are performed by the modules using an input package. It also describes what results were found for these tests. After the results are described, a discussion on the interpretation of these results is also given.

4.1 Thresholding

1. Levenshtein Distance

This method measures the "edit distance" between two-character sequences. For example, 'cat' and 'bat' have an edit distance of one while 'moon' and 'spoon' have an edit distance of 2. The Levenshtein distance is critical in understanding the frequency of misspellings at each edit distance.

Methodology:
This method extends the typo squatting analysis [18] to evaluate 50 typo-squatting campaigns [17] [19] [20] across 3 repositories. All of these campaigns delivered attacks in two phases and relied on multiple edit distances. Utilized the data from [22] to identify the typo squatting for the top packages across the PyPI repository and used the APIs for download metrics in NPM and Ruby. Plotting a graph with the x axis as the Levenshtein distance and the y axis being the number of attacks,



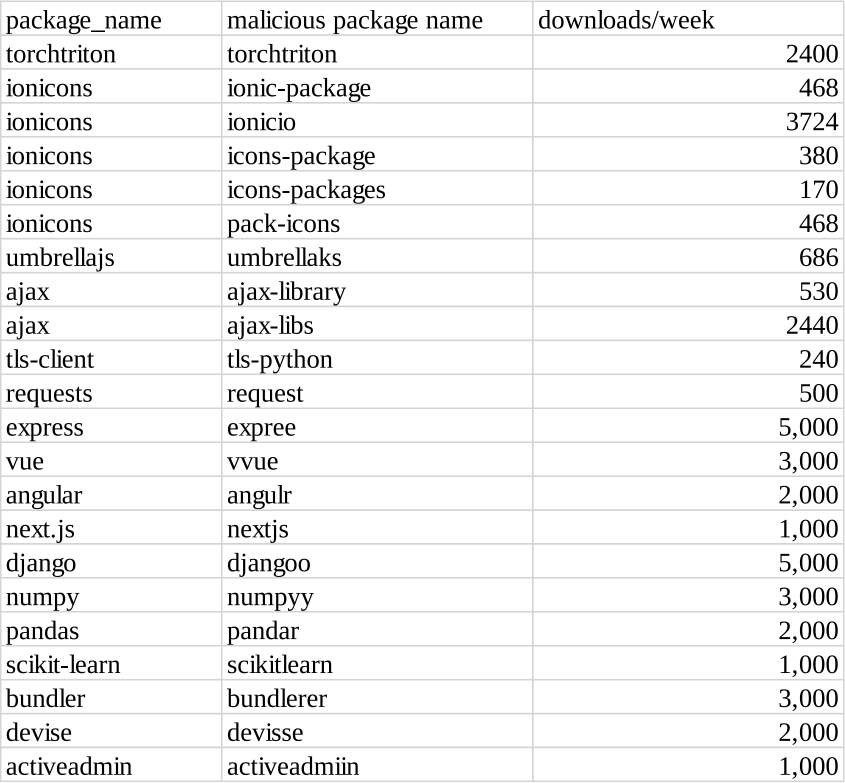
This data is in corroboration with [18] which demonstrated that campaigns often rely on more than one Levenshtein distance in a single attack. However, it is straightforward to note that the Levenshtein distance of 1 attracted the greatest number of attacks.

Result:
The Levenshtein distance of 1 was chosen for this project. This follows intuition that misspellings are typically off by a single character and these candidates provide the campaigns the greatest chance of success.

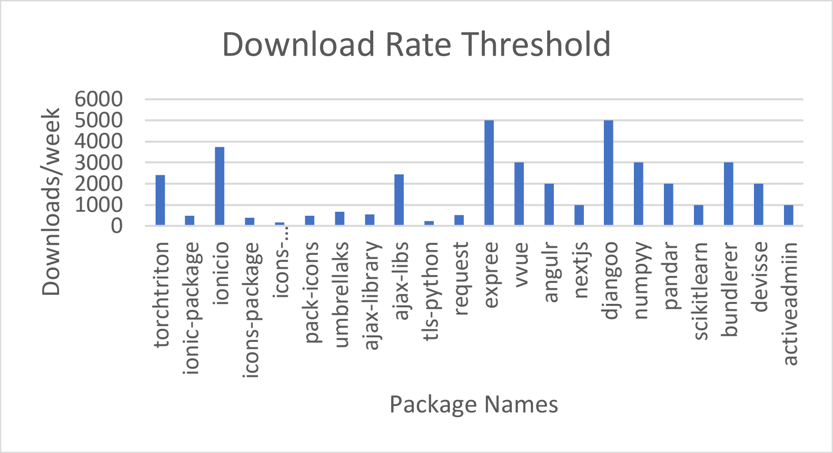
2. Download Rate:
The typo-squatting and dependency confusion modules rely on identifying malicious packages based on download rate threshold.

Methodology:

The top 25 most successful campaigns were identified and the download metrics were fetched. These campaigns were a mix of typo-squatting and confusion attacks across the three repositories. The download rate for these packages was fetched through the wayback machine [24]. The table below shows the attack on the package name and the graph shows the aggregated downloads/week for these malicious packages.



package_name	malicious package name	downloads/week
torchtriton	torchtriton	2400
ionicons	ionic-package	468
ionicons	ionicio	3724
ionicons	icons-package	380
ionicons	icons-packages	170
ionicons	pack-icons	468
umbrellajs	umbrellaks	686
ajax	ajax-library	530
ajax	ajax-libs	2440
tls-client	tls-python	240
requests	request	500
express	expree	5,000
vue	vvue	3,000
angular	angulr	2,000
next.js	nextjs	1,000
django	djangoo	5,000
numpy	numpyy	3,000
pandas	pandar	2,000
scikit-learn	scikitlearn	1,000
bundler	bundlerer	3,000
devise	devisse	2,000
activeadmin	activeadmiin	1,000

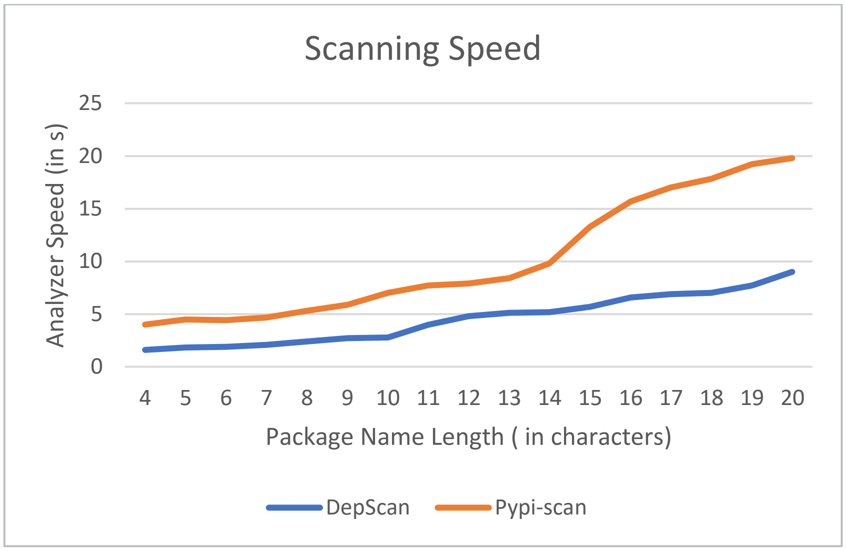


Result: A download threshold of 5000 was set below which packages are classified as malicious.

4.2 Scanning Speed

Methodology:

The scanning speed for the tool is essentially described as the sum of the scanning speed of the two different modules. The metadata analysis module and the dependency confusion modules run independent of size of the user input so they can be amortized to a O(n) time complexity. The only module that is dependent on the size of the user input is the typo-squatting analysis module. Since the typo-squatting module generates typos within the Levenshtein distance for the package name, a change in the package name can impose a performance cost. This is denoted as O(n^3) as there are three transformations each of which are n and n+r in size. The tool was timed for packages with name of varying lengths using python's time library and time was recorded as below. In a similar fashion



Result:

- The average scanning speed for the tool is about 5 seconds.
- The metadata analysis and dependency confusion take roughly 2.3 seconds independent of package size
- The typo-squatting analyzer takes about 1.8 seconds for a package name with 4 characters and depicts performance degradation with increasing in package name length.
- Pypi-Scan which is the prevalent typo-squatting analyzer performed slower than DepScan. This is due to the fact that Pypi-scan calculates all possible candidates (even implausible) adding an exponential performance penalty with growth in size of package name. DepScan only focuses on likely candidates by utilizing the characters, one hop way from the actual character. For example, for the value 'g', pypi-scan considers all 26 alphabets for the 3 operations while DepScan considers the character set ['t', 'r', 'f', 'v', 'b', 'n', 'h', 'y'] which form the one hop neighbor for the character 'g'.
- DepScan was run against snync and Confused to detect confusion attacks. DepScan performed about 1.3s slower than Dustilock and was on par with snync tool. The slowing down is due to the fact that DepScan performs metadata analysis in addition to the recursive namespace crawl. The metadata analysis helps reduce the false positives.

4.3 Error Rate

This section evaluates the tests that are a part of the metadata analysis module. The evaluation is performed on two data sets.

One data set comprises of 100 packages with top 50 packages from python and top 25 packages from npm and ruby gems repositories. This data set is the benign data set as the packages were manually verified to be non-malicious.

The second data set comprised of 100 manually generated packages with metadata fields left blank/filled with dummy values. Of these, 10 packages were known malicious which tried to squat benign packages.

1. Author Comparison: This test alerted for over 20 packages in the benign list out of which 18 were python packages. The primary reason was that the repository does not parse the author and author_email fields properly. Most of the instances suffered where author name was in the author email field triggering an alert for these field. This test had a 20% false positive rate but this was a result of incorrect values supplied by the repository maintainers. However, in the second data-set the tool flagged 15 packages for an alert out of which 10 were malicious. The false positive rate is much lower than the true positive rate showing that this is an effective comparison metric.

2. Maintainer Comparison: Similar to Author values, maintainer values for python and ruby packages are incorrect causing 29 WARN messages and 13 ALERTS on the first sample. For the second sample, the tool flagged all 10 packages but also flagged 3 packages which had null values for the maintainer email field. This was a WARN which was upgraded to an ALERT.

3. Project URL Comparison: This test flagged 18 packages because they had a. no project URLs and b. The URL values changes between the versions. This demonstrates a 18% false positive rate and not an effective independent security feature. However, with the dataset 2 which triggered context-based analysis, the test performed better by flagging 11 packages boasting a higher true positive rate.

4. Summary Comparison: For the first dataset with top 100 packages, the test flagged 33 packages and for the second data set it flagged 11 packages. The first data set had multiple packages with the malicious keyword set causing an alert. For example, ‘requests 2’ is a popular python package. However, a package requests3 had ‘name squatting’ in its summary and since these two packages were served by completely different groups, it generated an ALERT. However, this was a false positive as the domain was squatted to prevent attacks. With 33% false positive, summary alone is not effective security feature.

5. License Comparison: This test alerted for 9 packages in set 1 and 24 packages in set 2 and all of these alerts were because the License field did not have a value. Additionally, for data set 2, this test reported exactly 10 malicious packages thereby having no false positive rate. However, changes to the License must be updated in the tool beforehand to ensure that it generates a WARN and not an ALERT.

6. Dependency Comparison: This test flagged 19 packages in set 1 and 10 packages in set 2. A false positive rate of 19%, this feature and a true positive of 100%, this field is effective when used with other metrics to upgrade/downgrade the ALERT generated by this test through the majority principle.

7. Popularity Metrics: This test alone flagged 80% of all the malicious packages in set 2 and no packages were flagged in set 1. With a 0% false positive rate, this field is the driver field to turn a WARN into an ALERT and is an effective security field.

8. Missing Versions: This test flagged over 53 packages in set 1 and 47 packages in set 2. This was because specific versions were skipped. However, these were intended by design. With over 53% false positive rate, this test is not helpful in identifying attacks.

9. Strictly Increasing Versions: This test showed 0% false positive. However, it also failed to flag 3 out of the 10 packages in set 2 as malicious. Since it looks for a monotonically increasing list of versions, on its own, the feature is not indicative of attacks.

10. Package Age: This test boasted a 0% false positive rate. It could also identify 9 of the 10 packages as malicious thereby making the true positive rate 90%. This test in conjunction with test 9 will prove to be an effective in detection of malicious packages.

4.4 Performance v/s existing tools

This section analyzes the efficacy of the tests in the Attack Pattern Analysis module and compares the performance of DepScan with existing tools.

Methodology:
The evaluation was carried out on 20 packages that were deployed in a local environment. Out of these 18 packages mimicked confusion attacks and two packages were legitimate packages. The public and private repositories were simulated by deploying two servers through python's http server and the

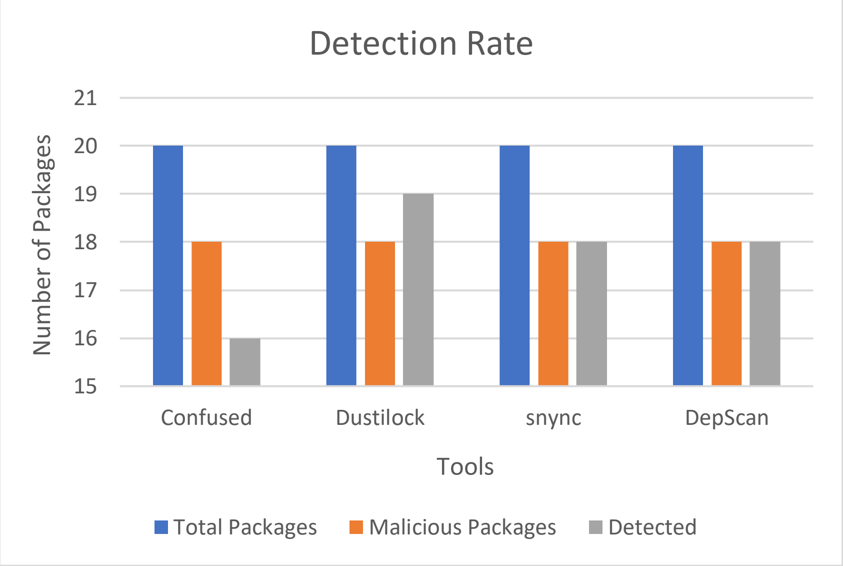
malicious packages were present on the public repository. packages were present in scoped format.

For this project, the legitimate package was hosted at
http://www.privaterepo.com/scope1/scope2/scope3/scope4/package

and the malicious packages were deployed as
http://www.publicrepo.com/scope1/scope2/scope3/scope4/package
http://www.publicrepo.com/scope1/scope2/scope3/package
http://www.publicrepo.com/scope1/scope2/package
http://www.publicrepo.com/scope1/package

These packages all had the same versions. The subsequent packages had varying versioning numbers. The intention was to deploy 10 packages as groups of 5 where one group mimics the scoping attack and the other groups mimics the version confusion attack. For the remaining packages, a minority set of the security features described in the design were altered to test if the tool can identify a confusion attack by upgrading multiple WARN labels to an ALERT label.

Additionally, the public popularity metrics such as downloads/week and age were set up to ensure that the two benign packages were downloaded more than the 18 malicious packages and also were deployed onto the server one day before the malicious packages. This setup was used to test DepScan, Confused, Dustilock, and snync.



Result:

- Since Confused does not perform recursive path traversal, it was unable to identify the fully scoped packages. Therefore, it was able to detect 16 out of the 18 malicious packages
- Dustilock registered on the other end of the results spectrum by flagging the non-malicious package also as malicious. This was because give a package it flagged every other package sharing its name across scoped paths as malicious.
- DepScan identified accurately all the malicious packages. This builds on the Dustilock solution but also evaluates metadata to reduce false positives. In the case of DepScan, public popularity metrics evaluation conducted at the metadata analysis phase prevent the benign package as being classified as malicious.
- DepScan performed at par with snync. However, snync is restricted to and hence optimized for analysis of the NPM repository while DepScan builds ubiquitous metadata objects and can analyze all repositories.

Methodology:
The second test was to sandbox known malicious attacks. Since the project was centered around metadata analysis, only the package's metadata was required to recreate the environment. This metadata was fetched through the wayback machine [23]. The packages were deployed to the same setup described in methodology one. 5 such packages were deployed along with their 5 legitimate counter parts.

Result:

- Snync detected all three npm packages that were malicious but did not detect the python and gems packages
- Confused and Dustilock reported 6 and 7 malicious packages respectively.
- DepScan identified all the 5 malicious packages and did not flag the benign packages. On dumping the logs for the process execution, it was observed that the majority rule was not triggered and although it wrongfully flagged 3 features for a specific package, it remained under a WARN thereby not triggering an ALERT.

5 Usage and Deployment

The tool can be run from the command line as described below, *Python depscan <VALUE> <FILE NAME> <OPT ARGS> <REPORT FILE>*

The VALUE field takes one of the following values

1. GEMS Packages
2. NPM Packages
3. Python Packages

The FILE NAME is essentially the requirements file with the set of package names that are to be downloaded

The OPT ARGS tells the tool which attacks must be tested again.

-a	run all attack analysis checks
-d	run dependency confusion analysis checks
-t	run typo squatting analysis checks

The analyzer output is written to the REPORT FILE. If no file name is provided, the tool defaults to a terminal output.

The user can run the command which will parse each package, run the relevant analysis and generate a report for the user to validate. This report is going to contain either a WARN message, an ALERT message or a FATAL message demonstrating the severity of the issue flagged by the analyzer. The user can then proceed to take action on the respective package by either deleting the package from the packages file or updating the package details.

6 Limitations and Future Work

6.1 Limitations

One of the limitations of the tools is that the tests run independently are not effective. These are only effective in a group of tests considering the context. This essentially means a subset of these tests are effective at detecting the malicious packages. While the tool in general is better at detecting these attacks, the tests could be improved to ensure that multiple focal points exist for concrete independent analysis.

Another limitation identifying in the evaluation section is that the evaluation depends on features which are populated by the repository managers, developers. Due to the absence of standards across repositories, these values show a high degree of variation. This also increases false positives in the tool as the it is dependent on these values and can only operate with the metadata values presented to it by the package.

Furthermore, it is seen that thresholding values are empirical and can change in the future. The current implementation of the tool requires manually tweaking these values. This can be automated by integration of machine learning.

In addition, the ruby gems repository implemented a custom YAML based metadata format which required a dedicated parser. Future changes to the structure of the metadata will require redesigning the parser.

Finally, while the report is JSON and can be consumed by multiple downstream processes to make decisions, the report can be made more human readable. This could be parsing the JSON as an HTML file and displaying report as a webpage.

6.2 Future work

The tool was developed to serve as a lightweight alternative to process intensive analysis tools. While the tool is able to effectively detect the primary attack vectors, the tool can be augmented with static analysis to identify malicious application logic to detect complex attacks. [24]

Thresholding values such as the download rate change with internet traffic and can increase or decrease over time. Static thresholding analysis required period analysis and refreshing of the threshold values. Incorporating a machine learning algorithm that can learn the internet traffic and trends to dynamically adjust the threshold not only across time but also across repositories will ensure the tool maintains a low false positive.

7 Conclusion

This project explored the burgeoning supply chain attacks against public repositories and proposed a tool to identify these attacks. The report also went over the existing solutions and their shortcoming expanding on the importance of context-based metadata analysis to detect these attacks. The tool additionally demonstrated that despite a huge variation in how metadata is handled across repositories (custom YAML for ruby vs json for npm), it is possible to create a ubiquitous metadata object and build a unified analysis platform that can scale to multiple repositories.

Through the course of the project, it was also demonstrated that while some of these tests previously existed, adding metadata context to them will greatly enhance their effectiveness. For instance, contextualizing popularity metrics and age ensured that they had a high degree of true positive and almost no false positives. Additionally, the project introduced a new approach of cumulative metadata analysis. The ability to accumulate multiple lower-level messages (eg. WARN) and generate a higher-level message (eg. ALERT) demonstrates cumulative analysis and ensures that true positives are detect even in testing environments (eg. Dependency confusion in multi scoped URLs).

8 Acknowledgement

I would like to thank Professor Mustaque Ahamad for his constant support and valuable feedback during the course of this project. I would also like to thank my peer Brandon D Moore for his weekly feedbacks and help with designing and implementing this project as well as advice with challenges faced while developing this project. I extend my thanks to my classmates and friends for their advice and feedback through this project.

References

[1] Alex Birsan, "Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies," Medium, Feb. 09, 2021. [Online]. Available: https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610

[2] "A Pentester's Guide to Dependency Confusion Attacks," Cobalt Blog. [Online]. Available: https://www.cobalt.io/blog/a-pentesters-guide-to-dependency-confusion-attacks

[3] "Dependency Confusion: Understanding and Preventing Attacks," FOSSA Blog. [Online]. Available: https://fossa.com/blog/dependency-confusion-understanding-preventing-attacks/

[4] "PyTorch Discloses Malicious Dependency Chain Compromise Over Holidays," BleepingComputer. [Online]. Available: https://www.bleepingcomputer.com/news/security/pytorch-discloses-malicious-dependency-chain-compromise-over-holidays/.

[5] " Path order precedence issue in PyPI," GitHub. [Online]. Available: https://github.com/pypa/pip/issues/5045.

[6] "Confused Tool to detect dependency confusion attack," Visma, GitHub. [Online]. Available: https://github.com/visma-prodsec/confused.

[7] "SNYNC - Snyk Notification Canary," Snyk, GitHub. [Online]. Available: https://github.com/snyk-labs/snync.

[8] "OWASP Dependency-Check," OWASP. [Online]. Available: https://owasp.org/www-project-dependency-check/.

[9] “Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages”, Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee, 2021. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/ndss2021_1B-1_23055_paper.pdf.

[10] “Taxonomy of Attacks on Open-Source Software Supply Chains”, Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais, 2022. [Online]. Available: https://arxiv.org/pdf/2204.04008.pdf

[11] "NPM-Audit Documentation", NPM, [Online]. Available: https://docs.npmjs.com/cli/v9/commands/npm-audit.

[12] "A Road from Dependency Confusion to RCE," SecureLayer7 Blog. [Online]. Available: https://blog.securelayer7.net/a-road-from-dependency-confusion-to-rce/.

[13] "Dustilock," Checkmarx, GitHub. [Online]. Available: https://github.com/Checkmarx/dustilock.

[14] “Clarify pypi server behavior on conflict", GitHub. [Online]. Available: https://github.com/pypi/warehouse/issues/11139.

[15] “Compromised PyTorch-nightly dependency chain”, PyTorch, 2022. [Online]. Available: https://pytorch.org/blog/compromised-nightly-dependency/

[16] "Understanding the Levenshtein Distance Equation for Beginners", https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0

[17] D.-L. Vu, “Typosquatting and combosquatting attacks on the python ecosystem,” 07 2020. https://ieeexplore.ieee.org/document/9229803

[18] "A PyPI Typosquatting Campaign: Post-Mortem," Phylum, [Online]. Available: https://blog.phylum.io/a-pypi-typosquatting-campaign-post-mortem/.

[19] " How programmers can be tricked into running bad code," Help Net Security, June 15, 2016. [Online]. Available: https://www.helpnetsecurity.com/2016/06/15/programmers-running-bad-code/.

[20] " Over 700 Malicious Typosquatted Libraries Found On RubyGems Repository," The Hacker News, [Online]. Available: https://thehackernews.com/2020/04/rubygem-typosquatting-malware.html.

[21] "Dependency Confusion Exploitation," Blaze Information Security, [Online]. Available: https://www.blazeinfosec.com/post/dependency-confusion-exploitation/.

[22] H. van Kemenade. Top PyPI Packages. [Online]. Available: https://hugovk.github.io/top-pypi-packages/

[23] Internet Archive. Wayback Machine. [Online]. Available: https://web.archive.org/

[24] R77 Rootkit Typosquatting - npm Threat Research. ReversingLabs Blog. [Online]. Available: https://www.reversinglabs.com/blog/r77-rootkit-typosquatting-npm-threat-research

[25] A. Yorke, "Butter Fingers," [Online]. Available: https://github.com/alexyorke/butter-fingers.

[26] The Hitchhiker's Guide to Python. "Open Source Licensing," [Online]. Available: https://docs.python-guide.org/writing/license/.