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

2 Background

3 Solution

3.1 Core Idea

3.2 Feature Selection

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

3.3.2 Modules

3.3.3 Report Structure

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

4.2 Scanning Speed

4.3 Error Rate

4.4 Performance v/s existing tools

5 Usage and Deployment

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/.