Towards Using Source Code Repositories to Identify Software Supply Chain Attacks

Duc-Ly Vu
ducly.vu@unitn.it
University of Trento, IT

Ivan Pashchenko
ivan.pashchenko@unitn.it
University of Trento, IT

Fabio Massacci
fabio.massacci@unitn.it
University of Trento, IT

Henrik Plate
henrik.plate@sap.com
SAP Security Research, FR

Antonino Sabetta
antonino.sabetta@sap.com
SAP Security Research, FR

ABSTRACT

Increasing popularity of third-party package repositories, like NPM, PyPI, or RubyGems, makes them an attractive target for software supply chain attacks. By injecting malicious code into legitimate packages, attackers were known to gain more than 100 000 downloads of compromised packages. Current approaches for identifying malicious payloads are resource demanding. Therefore, they might not be applicable for the on-the-fly detection of suspicious artifacts being uploaded to the package repository. In this respect, we propose to use source code repositories (e.g., those in Github) for detecting injections into the distributed artifacts of a package. Our preliminary evaluation demonstrates that the proposed approach captures known attacks when malicious code was injected into PyPI packages. The analysis of the 2666 software artifacts (from all versions of the top ten most downloaded Python packages in PyPI) suggests that the technique is suitable for lightweight analysis of real-world packages.

KEYWORDS

Software Supply Chain Attacks; Lightweight Analysis; Code injection; package repositories; source code repositories

ACM Reference Format:
Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. 2020. Towards Using Source Code Repositories to Identify Software Supply Chain Attacks. In *Submitted to Conf. on Comp. and Comm. Sec. - CCS'20*. ACM, New York, NY, USA, 3 pages. https://doi.org/tbc

1 INTRODUCTION

Software supply chain attacks occur when an attacker hijacks the complex software development chain to insert malicious code [3]. Among the attack vectors, attackers primarily target language based ecosystems such as NPM for Javascript or PyPI for Python by either stealing package owner credentials to subvert the package releases (e.g., attack on `ssh-decorate` package[1]) or publishing their own packages with names similar to popular ones (i.e., combosquatting and typosquatting attacks on PyPI packages [11]). The examples of the supply chain attacks were discovered several years ago. Since then, the number of cases has been steadily increasing [3, Figure 1].

Given the popularity of packages coming from language based ecosystems, the number of victims affected by software supply chain attacks is significant. For example, in July 2019, three malicious Ruby packages discovered by *MalOSS* [1] exceeded 100 000 downloads and were assigned the official CVEs. In February 2020, two accounts uploaded over 700 typosquatting malicious packages to the RubyGems repository and achieved more than 100 000 overall downloads [6].

The situation becomes even more dramatic as the ongoing and successful attacks have passed unnoticed: 20% of the malicious packages persisted in the NPM, PyPI, and RubyGems ecosystems for over 400 days [1]. This problem likely happens due to the lack of an efficient mechanism for checking malicious code injections in FOSS packages uploaded to the package repositories at a high pace (400 and 100 new packages are uploaded to npm and PyPI, respectively every day [10]). Current malware detection techniques in language based ecosystems, on the other hand, are resource demanding [1], require prior knowledge of previously benign releases [8], or unable to process packages that have a limited number of published releases [2].

Our approach is motivated by an intuition behind the reproducible builds[2]: *it is suspicious if the code in the source code repository differs from the code in the artifacts distributed in the package repository.* In this respect, we propose an approach to detect code injected into software packages by comparing their distributed artifacts (e.g., those in PyPI) with the source code repository (e.g., those in Github). The proposed approach can be used to detect injected code in typosquatting and hijacked packages.

To illustrate our approach, consider a typosquatting Python package `jeIlyfish` discovered by Lutoma [5], which was persistent in PyPI for nearly a year until its detection on

CCS'20, 5 May 2020, For Review Purpose Only

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/tbc

[1] https://www.bleepingcomputer.com/news/security/backdoored-python-library-caught-stealing-ssh-credentials/
[2] https://reproducible-builds.org

December 1, 2019. `jeIlyfish` mimicked the popular package `jellyfish`[3] (the first L is an I) to steal SSH and GPG keys[4]. Our technique processes the suspected `jeIlyfish` artifacts to identify the corresponding source code repository[5]. Then we compare the file hashes and contents extracted from the artifacts with those obtained from the source code repository. Our tool detects two injected files: `setup.py`, and `_jellyfish.py` and reports several lines in `_jellyfish.py` to contain suspicious API calls for decoding and executing the malicious code.

Overall, our tool requires 12 seconds (on a laptop with 4 CPU cores and 8 GB RAM) for processing the source code repository of the package `jellyfish` consisting of 364 commits, 530 individual files, and 28104 unique lines of code. The tool takes only 0.04 seconds for scanning the suspected artifact. Hence, our approach is fast to be integrated into the package distribution pipeline for detecting suspicious injections introduced by packages being uploaded to the package repositories.

2 BACKGROUND

Several studies (e.g., [10, 11]) based on the edit distance (e.g., Levenshtein distance) between package names to flag suspicious packages. Although the approaches are fast, manual analysis of the several generated candidates confirmed that the packages are benign [11] and exist due to coincidence, because they have similar functionality or one package was derived from the other (e.g., a fork) for legitimate reasons [10].

Taylor et al. [9] used six package name patterns (e.g., repeated characters) and the number of package downloads to identify typosquatting candidates of a given package in PyPI. Although this approach provides early warnings about a potential typosquatting package, the lack of essential features (e.g., source code similarity of suspected packages) might cause the approach to generate a high number of false alerts.

Duan et al. [1] proposed *MalOSS*, an extensible framework for detecting malicious packages in language based ecosystems. The framework combines metadata, static, and dynamic analysis to extract various features of packages. Although *MalOSS* can effectively detect malicious packages, the proposed system is computationally expensive, and only process packages of a specific environment (e.g., Linux Ubuntu).

Garrett et al. [2] proposed an anomaly detection based approach using features extracted from packages' metadata and source code to detect suspicious updates. Although the proposed technique can effectively label the known malicious updates, and reduce the review effort by 89 percent, only 31% of the packages in their dataset, which are frequently updated and have at least two versions, can be processed.

Ohm et al. [8] proposed *Buildwatch*, a framework for dynamic analysis of software and its third-party dependencies. The authors observed a high number of activities related to files (e.g., files written operations) in malicous verisons compared to the benign versions that were previously released of the analyzed packages. Although the approach provides insight into malware behaviors, it requires knowing of previously published benign releases that are difficult to obtain [1].

In summary, current approaches are limited to checking package names for providing early warnings for squatting attacks and are prone to false alerts. On the other hand, the malicious code detection techniques are resource demanding and might not be approach might not be suitable for real-time processing of packages uploaded to the package repositories.

3 IDENTIFICATION OF CODE INJECTIONS

Our approach compares distributed artifacts in package repositories (e.g., PyPI) and the source code repository (e.g., Github) to detect the injected code by the following steps:

(1) For each package, we identify the source code repository by mining metadata properties (e.g., homepage)
(2) We clone the repository and extract all the commits. For each commit, we check out each involved file, calculate the file hash, and collect the file content. The file hashes and contents are stored into a database
(3) We download each artifact of the package from the package repository, decompress it into files. For each file, we calculate the hash and collect the file content.
(4) Then we compare the file hashes and contents from step (3) with those extracted from step (4). This comparison results in files (and their lines) whose hashes are not recorded (differ from) in the source code repository
(5) For the unknown lines, we check the presence of API calls (e.g., *urlopen*) and imports (e.g., *import os*) using a regular expression.

During the packaging process, the packaging tools (e.g., `setuptools`[6] in Python) create new (benign) metadata files (e.g., `METADATA`, `WHEEL`), these files are specified in PEP 427[7]. Hence, we exclude such files from our analysis and focus on the differences in code files (e.g., .py, .js, .rb).

4 PRELIMINARY FINDINGS

We evaluate the proposed approach on the two datasets of known malicious and top ten most downloaded packages.

Known malicious examples. We used the malware dataset collected by Ohm et al. [7]. The dataset contains 34 malicious artifacts of 23 packages that were collected for both real and research purpose attacks between November 2015 and November 2019. Several packages have multiple distributed artifacts (e.g., `python3-dateutil` has ten malicious versions).
Findings: All the 34 malicious artifacts contained lines of code that were not present in the source code repositories; some of these lines are suspicious API calls or library imports. More specifically, we observed the following patterns:

[3] https://pypi.org/project/jellyfish/
[4] https://www.zdnet.com/article/two-malicious-python-libraries-removed-from-pypi/
[5] in this case, it was the same as the mimicked package `jellyfish` https://github.com/jamesturk/jellyfish
[6] https://pypi.org/project/setuptools/
[7] https://www.python.org/dev/peps/pep-0427/

- `setup.py` files are the most common file being injected; 22 artifacts whose the malicious code in this file;
- attackers can insert malicious code into different files in a distributed artifact of a package: one artifact injects code into the `__init__.py` file, three artifacts inject code into the functional modules (e.g., `_common.py` of `python3-dateutil`). While the median number of different files is 2, attackers can build a new malicious package (e.g., 20 new files in `openvc-1.0.0`);
- among the most common imported libraries, we noticed such libraries as *urllib3* (591 occurrences), *socket* (13 occurrences), *base64* (12 occurrences). These imports suggest that malicious imports focus on opening URLs, establishing connections, and encoding/decoding data.

The top ten most downloaded packages in PyPI. We obtained the top most downloaded PyPI packages[8] from the Kemenade database [4] released in August 2020. The packages contain 2666 distributed artifacts across all releases.

Findings: We observed that 2587 artifacts (97%) feature no difference from their source code repository. We explained the identified differences as follows:

- backporting security fixes: e.g., developers introduced a fix for CVE-2019-9740 into `urllib3-1.24.3`;
- fixing of configuration issues: some `urllib3` artifacts contain fixes for https proxy issue[9], several `six` artifacts include compatibility fixes, and some `chardet` artifacts have changes to fix encoding bugs, and `certifi-0.0.6` accommodates version declaration fixes;
- testing of features: e.g., 11 lines of code injected into release `requests-0.6.2` to show odd behavior of `urllib3` *PoolManager*. Similarly, several `six` artifacts featured changes in test files.

The median time to process an individual package artifact does not exceed 0.04 seconds. Processing the source code repositories (steps 1-2 in Section 3) takes 33 seconds median execution time (maximum 464 seconds (~8 min)). Hence, the approach is applicable for real-time checking software artifacts being uploaded to the package repository.

5 AUTOMATIC CLASSIFICATION OF MALICIOUS CHANGES

Although our current implementation detects the injections in distributed artifacts of a package, manual effort is still required to understand the injections' intention. Thus, we plan to develop an automated classifier for distinguishing malicious and benign code injections for future research.

File-level analysis. We could extract various features from detected files (e.g., # new files, # modified files) to distinguish benign and malicious artifacts. For example, suppose an artifact has all files different (newly added) with respect to its source code repository. In that case, the artifact is likely crafted from scratch with different malicious or benign functionality.

Code-level analysis. We could leverage code features (e.g., those proposed by [1, 2]), for detecting malicious injections

(1) presence of APIs that connect a remote URL, download/send content, decode or execute a code fragment;
(2) presence of imports of additional libraries to support malicious activities indicated in the previous item.

6 CONCLUSIONS

In this paper, we propose a lightweight technique based on the comparison between distributed artifacts and the source code repository of a package to detect the presence of injected code in software packages. Our results show that the proposed approach captures the known malicious packages, and reduces the review effort for distributed software releases by 97 percent. The median time for processing an individual artifact is less than a second, and 33 seconds for processing source code, which are fast. Hence, the approach is capable of the on-the-fly detection for injection attacks for packages being uploaded to the package repositories. For future work, we plan to build a classifier to discriminate benign and malicious injections using features from injected files and codes.

ACKNOWLEDGMENTS

This research has been partly funded by the EU under the H2020 Programs H2020-EU.2.1.1-CyberSec4Europe (Grant No. 830929), NeCS: European Network for Cyber Security (Grant No. 675320) and SPARTA project (Grant No. 830892).

REFERENCES

[1] R. Duan, O. Alrawi, R.P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee. 2020. Measuring and preventing supply chain attacks on package managers. *arXiv* (2020).
[2] K Garrett, G. Ferreira, L. Jia, J. Sunshine, and C. Kästner. 2019. Detecting suspicious package updates. In *Proc. of ICSE-NIER'19*.
[3] T. Herr, J. Lee, W. Loomis, and S. Scott. 2020. Breaking Trust: Shades of Crisis Across an Insecure Software Supply Chain. https://www.atlanticcouncil.org/in-depth-research-reports/report/breaking-trust-shades-of-crisis-across-an-insecure-software-supply-chain/.
[4] H.V. Kemenade. 2020. *hugovk/top-pypi-packages: Release 2020.08.* https://doi.org/10.5281/zenodo.3969444
[5] Lutoma. 2019. PSA: There is a fake version of this package on PyPI with malicious code. https://github.com/dateutil/dateutil/issues/984.
[6] Tomislav Maljic. 2020. Mining for malicious Ruby gems Typosquatting barrage on RubyGem software repository users. https://blog.reversinglabs.com/blog/mining-for-malicious-ruby-gems.
[7] M. Ohm, H. Plate, A. Sykosch, and M. Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *Proc. of DIMVA'20*.
[8] M. Ohm, A. Sykosch, and M. Meier. 2020. Towards detection of software supply chain attacks by forensic artifacts. In *Proc. of ARES'2020*.
[9] M. Taylor, R.K. Vaidya, D. Davidson, L. D. Carli, and V. Rastogi. 2020. SpellBound: Defending Against Package Typosquatting. *arXiv* (2020).
[10] R.K. Vaidya, L. D. Carli, D. Davidson, and V. Rastogi. 2019. Security issues in language-based sofware ecosystems. *arXiv* (2019).
[11] D.L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta. 2020. Typosquatting and Combosquatting Attacks on the Python Ecosystem. In *Proc. of EuroS&PW'20*.

[8] Top ten packages: `urllib3`, `six`, `botocore`, `requests`, `python-dateutil`, `certifi`, `s3transfer`, `idna`, `chardet`, `pip`. We skip `docutils` because it is hosted in sourceforge.
[9] https://github.com/urllib3/urllib3/issues/1850