Towards Using Source Code Repositories to Identify Software Supply Chain Attacks

Duc-Ly Vu[1], Ivan Pashchenko[1], Fabio Massacci[1,2], Henrik Plate[3], Antonino Sabetta[3]
[1]University of Trento (IT), [2]Vrije Universiteit Amsterdam (NL), [3]SAP Security Research (FR)

Cyber Security for Europe — NECS — SPARTA

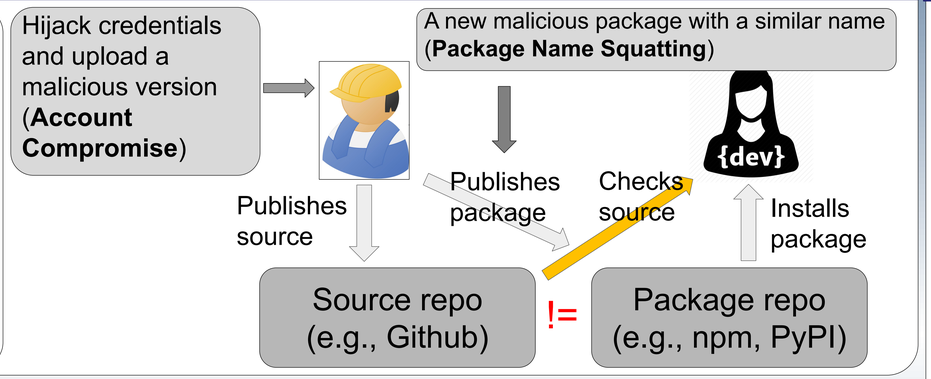
Third-party package repositories (e.g., npm, pypi) are an attractive target for software supply chain attacks

Software supply chain attacks: source repo != package repo

Definition: *Software supply chain attacks occur when an attacker hijacks the complex software development chain to insert malicious code*[1]

Observation: Distributed artifacts in the **package repository** do **not** necessarily **correspond** to the **source repository** due to **benign** (developer's carelessness) or **evil reasons** (malicious code injections)

[1]*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-researchreports/ report/breaking-trust-shades-of-crisis-across-aninsecure-software-supply-chain/.*



Identification of Code Injections

Our approach is motivated by an intuition behind the reproducible builds [3]: **it is suspicious** if the **code in the source code repository differs** from the **code in the artifacts distributed in the package repository**.

1. For each package, identify the source code repository by mining metadata properties (e.g., homepage)
2. Clone the repository and extract all the commits in the master branch. For each commit, check out involved file, calculate the file hash, and collect the file content. The file hashes and contents are stored into a database
3. Download and extract all artifacts of the package. For each extracted file, we calculate the hash and collect the file content.
4. Compare the file hashes and contents from step (3) with those extracted from step (2). This comparison results in files (and their lines) whose hashes differ from the source code repository
5. For the unknown lines, check the presence of API calls (e.g., *urlopen*) and imports (e.g., *base64* using regular expressions

1. https://reproducible-builds.org/

Dataset

- We used the malware dataset collected by Ohm et al.[2]
  - 23 packages
  - 34 malicious artifacts
- The **top ten** most downloaded packages in PyPI.
  - *urllib3*, *six*, *botocore*, *requests*, *python-dateutil*, *certifi*, *s3transfer*, *idna*, *chardet*, *pip*.
  - 2587 artifacts

[2]*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..*

Contact information

E-mail: ducly.vu@unitn.it
Skype: vuly16
Web-site: lyvu.me

Preliminary Findings

- Malicious artifacts (34)
  - The ***setup.py*** files are the **most common file being injected** (22 artifacts)
  - One artifact injects code into the ***__init__.py* file,** three attacks inject code into the **functional modules** (e.g., *_common.py* of the package *python3-dateutil*)
  - The **median number of different files** in distributed artifacts is **2**
  - Attackers can build a **new malicious package** (e.g., 20 new files in *openvc-1.0.0*)
  - Most **common imported libraries** are ***urllib3*** (591 occurrences), ***socket*** (13 occurrences), ***base64*** (12 occurrences)
- Distributed artifacts of top most downloaded packages (2587)
  - **97%** of the artifacts feature **no difference** from their source code repository.
  - Some artifacts contain
    - https proxy issues fixes
    - compatibility and encoding fixes
    - version declaration fixes
    - changes in test files.

Future work

Automatic Detection Of Malicious Code Changes

We consider extracting the following features:

- File-level analysis.
  - number of new added files
  - number of modified files
- Code-level analysis
  - presence of sensitive APIs (e.g., urlopen)
  - presence of new imports