Securing the Supply Chain in a Company

Casaba | Slater Weinstock

What is a software supply chain?

- It is anything that goes into your code or anything that affects your code from development to production
- What all is included in the software supply chain?
  - Code
  - Binaries
  - Open-source software (from repositories, package managers, etc)
  - Internal packages
  - Build scripts
  - Packaging scripts
  - The infrastructure the software runs on
- Also:
  - Who wrote the software, who reviewed it, software licensing, supported versions, when it was contributed
  - The software that checks for known security issues

Where to attack?

- Two main areas where the supply chain can be attacked
  - Components under the company’s control
  - External components not under the company’s control
- Types of attacks:
  - Dependency confusion targeting open-source components or internal packages
  - Typosquatting
  - Developer accidents
  - Compromising an employee’s accounts or otherwise injecting code into the company’s private repository
  - Stealing code-signing certificates
  - Attacking the build environment
  - Ransomware

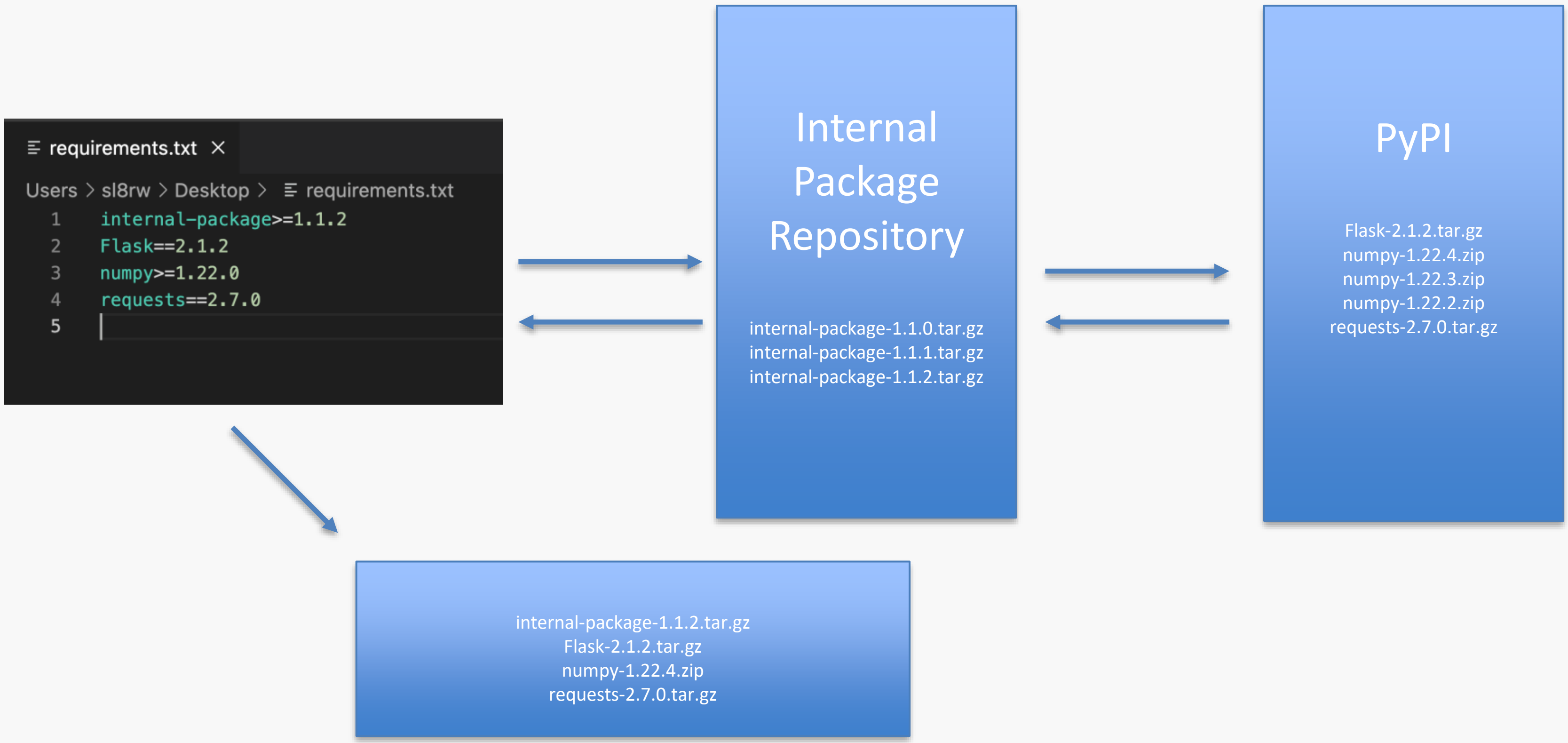
Recent newsworthy attacks

- REvil
  - Happened against Kaseya, a managed service provider
  - Used an authentication bypass in the web UI of VSA, and then used SQL injection to upload the payload and deploy the malicious update
  - Due to the nature of the product, ~1500 downstream businesses were impacted by this
- Trojan Source
  - Hiding malicious code with Unicode characters resulting in it getting missed during a routine code review
  - Early return example

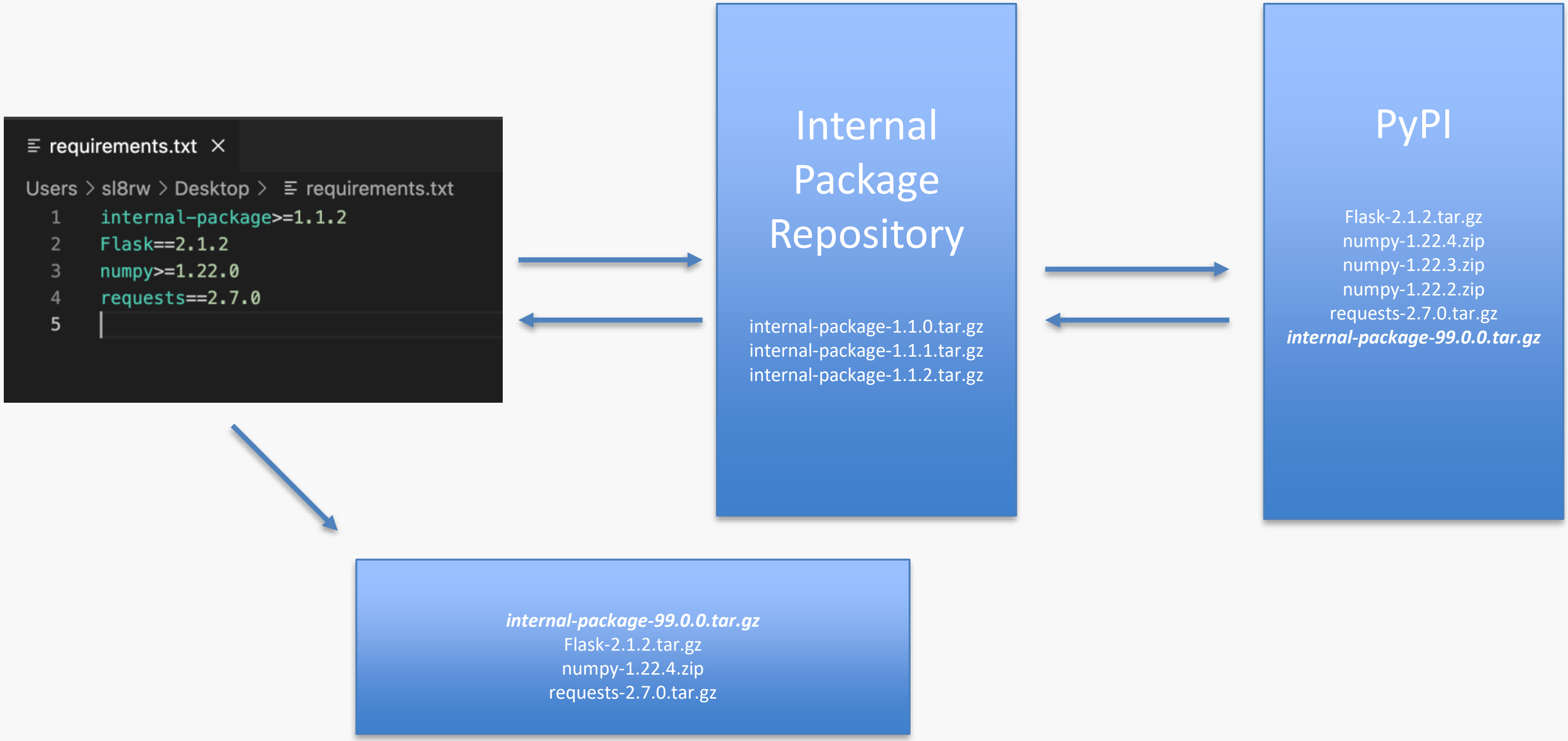
Recent newsworthy attacks

- SolarWinds
  - Targeted Orion, an IT performance monitoring system
    - This software had privileged access and was widely deployed
  - SolarWinds network was infiltrated, build system was targeted, and code injection vectors were tested
  - Code was eventually signed and deployed to the 30K+ customers
- XcodeSpy
  - Trojan Xcode project based on TabBarInteraction, a legitimate Xcode project which adds animation to the tab bar for an iOS application
  - Runs malicious code that downloads the EggShell backdoor from a remote server
  - Acts as a first step in a supply chain attack
- ua-parser-js
  - Took over developer's account (https://github.com/faisalman/ua-parser-js/issues/536#issuecomment-949742904) with the probable help of an email bomb
  - After installation, the script detects the OS and downloads the respective binary, which then downloads a cryptominer to mine cryptocurrency

An Example – Dependency Confusion



An Example – Dependency Confusion



But you need the internal package names…

- You can get these by scraping GitHub
- Searching the JavaScript on the website of interest for calls such as:
  - require()
  - Search for paths that are prepended with the company's name
- Stack Overflow
- Guessing

Has this happened?

- Yes, a proof-of-concept was demonstrated in early 2021
- https://www.npmjs.com/package/yelp-bunsen-logger-js
- https://www.npmjs.com/package/yelp-js-infra
- https://www.npmjs.com/package/yelp_sitrep

This seems easy… Why?

- A lot of times it's due to assumptions or confusion
- For example, in Python:
  - Developers using --extra-index-url, which does the same type of check as previously described, instead of --index-url.
  - Using Artifactory, which is used in many corporations, mixes internal and public libraries into its own type of library. Therefore, it shares the same type of vulnerability

Other derivatives - Typosquatting

- Typosquatting, which has been known about since 2016
- Let's say I want to install requests, I do: pip install requests
- As the name typosquatting implies, an attacker takes advantage of common typos, such as:
  - requets
  - reqeusts
  - requsts
- This can lead to arbitrary code execution among other issues

CrateDepression - Rust

- This attack targeted rust_decimal by uploading a malicious crate called rustdecimal
- Once the machine is infected, the environment variable GITLAB_CI is searched for
- The Poseidon payload (written in Go) is downloaded to the targeted Linux or macOS platform. Once downloaded, the binary is set as an executable and on macOS the quarantine bit is removed
- C2 communication is set up

How do you mitigate these types of attacks?

- To mitigate against dependency confusion in Python:
  - Avoid using --extra-index-url
  - Use version pinning
  - If possible, use version hashing (pip install --hash)

Hash match for Flask version:

```
requirements.txt
Users > sl8rw > Desktop > requirements.txt
Flask==2.1.2 \
    --hash=sha256:315ded2ddf8a6281567edb27393010fe3406188bafbfe65a3339d5787d89e477
numpy==1.22.4 \
    --hash=sha256:425b390e4619f58d8526b3dcf656dde069133ae5c240229821f01b5f44ea07af
requests==2.7.0 \
    --hash=sha256:398a3db6d61899d25fd4a06c6ca12051b0ce171d705decd7ed5511517b4bb93d
```

```
(test)  sl8rw@Slaters-MBP  ~/Desktop  pip install -r requirements.txt
Collecting Flask==2.1.2
  Using cached Flask-2.1.2.tar.gz (631 kB)
  Preparing metadata (setup.py) ... done
```

Hash mismatch for Flask version:

```
Users > sl8rw > Desktop > requirements.txt
Flask==2.1.0 \
    --hash=sha256:315ded2ddf8a6281567edb27393010fe3406188bafbfe65a3339d5787d89e477
numpy==1.22.4 \
    --hash=sha256:425b390e4619f58d8526b3dcf656dde069133ae5c240229821f01b5f44ea07af
requests==2.7.0 \
    --hash=sha256:398a3db6d61899d25fd4a06c6ca12051b0ce171d705decd7ed5511517b4bb93d
```

```
(test)  sl8rw@Slaters-MBP  ~/Desktop  pip install -r requirements.txt
Collecting Flask==2.1.0
  Downloading Flask-2.1.0-py3-none-any.whl (95 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 95.2/95.2 kB 1.3 MB/s eta 0:00:00
ERROR: THESE PACKAGES DO NOT MATCH THE HASHES FROM THE REQUIREMENTS FILE. If you have updated the package versions, please update the hashes. Otherwise, examine the package contents c
arefully; someone may have tampered with them.
    Flask==2.1.0 from https://files.pythonhosted.org/packages/ef/31/3ca16c38b92da4850301e426105ba2b747c480d7fe2715ff703360d3880f/Flask-2.1.0-py3-none-any.whl#sha256=e4c69910f6a096cc57
e4ee45b7ba9afafdcad4cc571db6eb97d5bd01b95422ea (from -r requirements.txt (line 1)):
        Expected sha256 315ded2ddf8a6281567edb27393010fe3406188bafbfe65a3339d5787d89e477
             Got        e4c69910f6a096cc57e4ee45b7ba9afafdcad4cc571db6eb97d5bd01b95422ea

(test)  ✘ sl8rw@Slaters-MBP  ~/Desktop  _
```

Event-Stream - Dependencies of dependencies attack

- A JavaScript library known as Event-Stream is downloaded approximately 2 million times a week
- The original author gave another developer/maintainer access to the repository (Right9ctrl)
- The new developer added a dependency to Flatmap-Stream(which was an injection attack) and bumped the minor version of Event-Stream
- A few days, that developer removed Flatmap-Stream and bumped the major version of Event-Stream
- Millions of users were affected as it’s downloaded so often
- Ultimately, the malware targeted Copay. If wallet is found, it executes and attempts to steal your bitcoin wallet
- https://github.com/dominictarr/event-stream/issues/116

Secure SDLC

- Having clear security requirements defined up front and distributed to developers
- Perform threat modeling
- Establish design requirements and use appropriate cryptographic standards
- Keeping an inventory of all 3rd-party components that are used and performing dependency analysis/SCA
- Using approved tools/versions and using appropriate compiler flags
- Performing static analysis against source code before compiling
- Perform dynamic analysis of the compiled software
- Pen testing

Threat Modeling

- The 5 major steps are:
  - Define the security requirements of the application
  - Diagram the application
  - Identity the potential threats and security boundaries
  - Apply mitigations
  - Validate that the threats have been mitigated
- Having to make a significant change to an application costs a huge amount of money, causes delays, optics, etc

Establish design requirements/Crypto standards

- Many features are susceptible to vulnerabilities just based on complexity
  - Authentication
  - Role-based access controls
  - Logging
  - Cryptographic protocols
- Examples
  - Two-factor authentication
  - Password complexity
  - Collision resistant hashing algorithms

Dependency Analysis/Software Composition Analysis (SCA)

- Identification of vulnerabilities in open-source code used by companies
- Many tools exist including:
  - Snyk.io
  - Synopsys Black Duck
  - npm-audit or yarn audit
  - Manual review of the package manifests
- Each of these tools have various strengths and weaknesses.
- What if these tools identify a huge number (I have seen 20K+ findings in the past)
  - This does not mean the application is vulnerable. It is possible that the application does the not leverage a vulnerable component in a vulnerable way.
  - It is important to ensure that each of the findings though are reviewed to determine if a vulnerability exists.
- A lot of the findings can be fixed by updating to the latest point release. Think back to Event-Stream, if you updated to the major release a few days later, you would no longer be vulnerable.

Tool Use and SAST

- What IDEs are being used? Are appropriate plugins being used?
- In the build pipeline, what compiler/linker flags are being used?
  - For example (GCC): -Werror, -Wall, -Wextra, -fstack-protector, -Wsign-conversion, -Wconversion
- Is the source code free of secrets including in the history?
- Is the developer documentation free of secrets?
- What static analysis tools are being used?
  - For example, Semmle, PVS-Studio, Semgrep

Other areas to look at in the supply chain

- Build reproducibility
- Code review process
- Release process
- Build artifact storage and retention
- Access revocation
- Session longevity
- Automation of the build/scripting

Introducing SLSA:
Supply-chain Levels for Software Artifacts

- Introduced by Google
- SLSA provides an easy framework to measure the security of a supply chain
- Consists of 4 levels where 3 is the most secured – Updated from 0.1 spec
  - Level 0: No real requirements, no SLSA implemented
  - Level 1: Provenance shows how the package was built
  - Level 2: Signed provenance and generated on a hosted build platform
  - Level 3: Build platform itself has been hardened

  - Source Requirements
  - Build Requirements
  - Provenance Requirements
  - Common Requirements

SLSA Build L0

- Lack of SLSA

SLSA Build L1

- Consistent build process
- Build platform generates provenance automatically
  - Provenance describes:
    - What entity built the package
    - What build process was used
    - What the top-level input to the build was
- Provenance is distributed

SLSA Build L2

- Everything in L1
- **Build runs on a hosted build platform**
  - **Provenance is signed and generated on the build platform**
- **Verification of provenance includes validation of authenticity**

casaba

SLSA Build L3

- Everything in L1 and L2
- Build platform must prevent each run from influencing another one
- Prevent any secrets that are used to sign the provenance from being accessible

References

Slide 2:
https://github.blog/2020-09-02-secure-your-software-supply-chain-and-protect-against-supply-chain-threats-github-blog/

Slide 3:
https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610
https://docs.microsoft.com/en-us/microsoft-365/security/intelligence/supply-chain-malware?view=o365-worldwide

Slide 4:
https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/diving-deeper-into-the-kaseya-vsa-attack-revil-returns-and-other-hackers-are-riding-their-coattails/
https://trojansource.codes/trojan-source.pdf

Slide 5:
https://sysdig.com/blog/software-supply-chain-security/
https://www.techtarget.com/whatis/feature/SolarWinds-hack-explained-Everything-you-need-to-know
https://www.crowdstrike.com/blog/sunspot-malware-technical-analysis/
https://www.securemac.com/news/xcodespy-mac-malware-targets-developers
https://www.bleepingcomputer.com/news/security/new-xcodespy-malware-targets-ios-devs-in-supply-chain-attack/
https://www.truesec.com/hub/blog/uaparser-js-npm-package-supply-chain-attack-impact-and-response
https://blog.aquasec.com/npm-library-supply-chain-attack

Slide 6:
https://medium.com/ochrona/preventing-dependency-confusion-attacks-in-python-fa6058ac972f

Slide 7:
https://medium.com/ochrona/preventing-dependency-confusion-attacks-in-python-fa6058ac972f

Slide 8:
https://redhuntlabs.com/blog/top-organizations-on-github-vulnerable-to-dependency-confusion-attack.html
https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610

References

Slide 9:
https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610

Slide 10:
https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610
https://azure.microsoft.com/mediahandler/files/resourcefiles/3-ways-to-mitigate-risk-using-private-package-feeds/3%20Ways%20to%20Mitigate%20Risk%20When%20Using%20Private%20Package%20Feeds%20-%20v1.0.pdf

Slide 11:
https://incolumitas.com/2016/06/08/typosquatting-package-managers/

Slide 12:
https://www.sentinelone.com/labs/cratedepression-rust-supply-chain-attack-infects-cloud-ci-pipelines-with-go-malware/

Slide 13:
https://medium.com/ochrona/preventing-dependency-confusion-attacks-in-python-fa6058ac972f

Slide 14:
https://github.com/dominictarr/event-stream/issues/116
https://5stars217.github.io/2021-05-03-metadata-analysis-flatmap/
https://snyk.io/blog/a-post-mortem-of-the-malicious-event-stream-backdoor/

Slide 15:
https://www.synopsys.com/blogs/software-security/secure-sdlc/
https://snyk.io/series/open-source-security/software-composition-analysis-sca/
https://www.microsoft.com/en-us/securityengineering/sdl
https://www.microsoft.com/en-us/securityengineering/sdl/practices#practice2

References

Slide 16:
https://www.microsoft.com/en-us/securityengineering/sdl/practices#practice2
https://www.microsoft.com/en-us/securityengineering/sdl/threatmodeling

Slide 17:
https://www.synopsys.com/blogs/software-security/secure-sdlc/
https://snyk.io/series/open-source-security/software-composition-analysis-sca/
https://www.microsoft.com/en-us/securityengineering/sdl
https://www.microsoft.com/en-us/securityengineering/sdl/practices#practice2

Slide 18:
https://snyk.io/series/open-source-security/software-composition-analysis-sca/

Slide 19:
https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html
https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html
https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

Slide 20:
https://github.com/slsa-framework/slsa
https://security.googleblog.com/2021/06/introducing-slsa-end-to-end-framework.html

Slide 21:
https://kalilinuxtutorials.com/slsa/
https://github.com/slsa-framework/slsa/blob/main/docs/_spec/v0.1/requirements.md

References

Slide 22:
https://slsa.dev/spec/v1.0/levels
https://github.com/slsa-framework/slsa/blob/main/docs/_spec/v0.1/requirements.md

Slide 23:
https://slsa.dev/spec/v1.0/levels
https://github.com/slsa-framework/slsa/blob/main/docs/_spec/v0.1/requirements.md

Slide 24:
https://slsa.dev/spec/v1.0/levels
https://github.com/slsa-framework/slsa/blob/main/docs/_spec/v0.1/requirements.md

Slide 25:
https://slsa.dev/spec/v1.0/levels
https://github.com/slsa-framework/slsa/blob/main/docs/_spec/v0.1/requirements.md

Slide 26:
https://slsa.dev/spec/v1.0/levels
https://github.com/slsa-framework/slsa/blob/main/docs/_spec/v0.1/requirements.md

Thank You

Slater Weinstock
slater@casaba.com