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

```
Users > si8rw > Desktop > requirements.txt
1  internal-package>=1.1.2
2  Flask==2.1.2
3  numpy>=1.22.0
4  requests==2.7.0
```

- **Internal Package Repository**
  - internal-package-1.1.0.tar.gz
  - internal-package-1.1.1.tar.gz
  - internal-package-1.1.2.tar.gz

- **PyPI**
  - Flask-2.1.2.tar.gz
  - numpy-1.22.4.zip
  - numpy-1.22.3.zip
  - numpy-1.22.2.zip
  - requests-2.7.0.tar.gz

- **Internal Package Repository**
  - internal-package-1.1.2.tar.gz
  - Flask-2.1.2.tar.gz
  - numpy-1.22.4.zip
  - requests-2.7.0.tar.gz
An Example – Dependency Confusion

```
<table>
<thead>
<tr>
<th>requirements.txt</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Users &gt; sl8rw &gt; Desktop &gt; requirements.txt</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>internal-package&gt;=1.1.2</td>
</tr>
<tr>
<td>2</td>
<td>Flask==2.1.2</td>
</tr>
<tr>
<td>3</td>
<td>numpy&gt;=1.22.0</td>
</tr>
<tr>
<td>4</td>
<td>requests==2.7.0</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
```

**Internal Package Repository**
- internal-package-1.1.0.tar.gz
- internal-package-1.1.1.tar.gz
- internal-package-1.1.2.tar.gz

**PyPI**
- Flask-2.1.2.tar.gz
- numpy-1.22.4.zip
- numpy-1.22.3.zip
- numpy-1.22.2.zip
- requests-2.7.0.tar.gz
- internal-package-99.0.0.tar.gz
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-js-infra](https://www.npmjs.com/package/yelp-js-infra)
- [https://www.npmjs.com/package/yelp_sitrep](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
  – requeusts
  – request
• This can lead to arbitrary code execution among other issues
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:

```bash
(test) $ rm -rf /tmp/flask*  
(test) $ cd /tmp/requirements.txt  
(test) $ pip install -r requirements.txt  
```

Hash mismatch for Flask version:

```bash
(test) $ rm -rf /tmp/flask*  
(test) $ cd /tmp/requirements.txt  
(test) $ pip install -r requirements.txt  
```

Hash match for Flask version:

```bash
(test) $ rm -rf /tmp/flask*  
(test) $ cd /tmp/requirements.txt  
(test) $ pip install -r requirements.txt  
```

Hash mismatch for Flask version:
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](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 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
• Lack of SLSA
• 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
• 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
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

Slide 4:
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://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

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


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/
References

Slide 16:

Slide 17:
https://www.synopsys.com/blogs/software-security/secure-sdlc/

Slide 18:

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