Risks linked to external dependencies

May 2, 2022

Share

Nowadays, most of the software is based on external components that are created and maintained by external entities. External components are also named "third-party" components and can be, for example, a library. The objective is, most of the time, to delegate specific operations to dedicated components. This facilitates the maintenance of the main application and lets the developers focus on the code providing the business features. The type of operation performed by a component can be, for example, Processing of specific file format, logging, handling of business data formats (e.g., SWIFT) and so on.

The risk associated with the usage of external components

There are three major risks related to the usage of third-party components in software:

1. Risk of the presence of vulnerabilities in a component increasing the attack surface of the software[1].
2. Exposure to the "dependency confusion" attack[2].
3. Presence of malicious code in a component[3].

Vulnerabilities in a component

A component can be affected by a vulnerability. The vulnerability can be public and be the subject of a CVE[4] (it is not always the case). It is important to keep in mind that, a vulnerability in a component does not mandatory imply that the software is at risk. Indeed, perhaps the affected code is not used by the software or not accessible by an attacker. An analysis is always needed to validate the exposure.

There are various proprietary and open-source tools to analyze a project and identify the components with known CVE. For instance, "OWASP Dependency-Check"[5] can be used as a starting point because it allows the developers to **slowly mature in the processes of dealing with vulnerable components**[6].

```
# Scan a java maven project dependencies

$ mvn dependency:copy-dependencies

$ dependency-check –project TEST –scan ./target/dependency –format JSON –out ./dependency-check-report.json

# Cf. https://github.com/righettod/toolbox-pentest-web/blob/master/scripts/generate-report-odc.py

$ python3 generate-report-odc.py dependency-check-report.json

[+] Vulnerabilities:

CVSS v3  Severity    File                 CVE

———  ———-  ————————  —————

6.6  MEDIUM      log4j-api-2.17.0.jar  CVE-2021-44832
```

Exposure to the "dependency confusion" attack

The context of this attack is the following: The external component resolver (e.g., Nexus, Artifactory, NPM, etc.) has a **configuration problem**.

This configuration problem is that it first looks for a component in the defined **external repositories** before consulting the defined **internal repositories**.

Therefore, if an attacker deploys a component that is supposed to be **internal only** on a defined **external repository,** then the resolver will retrieve the version from the **external repository** and use the component deployed by the attacker instead of the expected one from the **internal repository**.

Tools like "confused"[7] or this script[8] allows testing exposure to this attack for some technologies.

In the following example, some dependencies are not existing publicly, as they are issued from internal development. Therefore, if an attacker creates a dependency with that name and publishes it, the component resolver will take the public release instead of the private one first.

```
$ ./confused -l npm package.json

Issues found the following packages are not available in public package repositories:

[!] eyetool

[!] db-middleware
```

A review of the list of dependencies available on the package file should be made to check that all the internal dependencies are listed in the tool results. If an internal dependency is not shown by the tool, it implies that this one is existing publicly and could inform that a "dependency confusion" attack was already performed or pending.

Presence of malicious code in components

This case occurs when one of the components used by the project (or else by one of its components, a component may itself have external dependencies) contains malicious code.

This malicious code can be introduced explicitly by the supplier of the component, or by a third-party entity following the compromise of the component's development chain.

The malicious code can be introduced, for example, only in a specific version of the component, so, a component can be safe (initial version) and, at a time, becomes malicious (for example following a compromise) then, becomes safe again (closing of the security incident).

Such a case is very complex to detect in a fully automated way. Indeed, the fact that a component becomes malicious often requires a manual analysis due to the various existing techniques for hiding and executing code. In addition, the number of external dependencies of a project, even a simple one, is quickly substantial due to the notion of nested dependencies, in fact, it is a tree with several levels of depth.

```
# Count the dependencies of a java project using Spring 2.5.8 with Spring Web + Spring Rest Repositories modules generated with the online tool: https://start.spring.io/

$ mvn -q dependency:copy-dependencies

$ ls target/dependency/ | wc -l

68
```

For the dependencies loaded for a web application, some mitigation actions could be implemented. If the application is using external dependencies hosted on a content delivery network (CDN) for example, the *Subresource Integrity*[9] feature can be used. The implementation of that feature requires only two steps.

First, the hash of the resource must be computed. That step could be made using a bash command like the following one. The result will be the hash value of the file.

```
$ cat dependency_1.js | openssl dgst -sha384 -binary | openssl enc -base64 -A

OMZSse+JQ3VM1WaqM54ez2Q60OV3BcteFrqEOBMRbjHPVd0t2ztVNRcwhUg3xcKB
```

Then, the hash must be added to the source code of the page by adding the **integrity** tag like shown in the following example:

```
<script src="https://CDN/dependency_1.js&#8221;

integrity="sha384-OMZSse+JQ3VM1WaqM54ez2Q60OV3BcteFrqEOBMRbjHPVd0t2ztVNRcwhUg3xcKB"

crossorigin="anonymous"></script>
```

With that hash defined on the script tag, the browser will load the resource and compute the SHA384 hash of the file and compare it to the value. If the resource doesn't have the same hash, this one will not be used by the browser.

The important step is ensuring that the original resource is the real one. Then, in case of a breach of the CDN or if an attacker can alter the resource, the change will be detected by the browser and the client will stay safe during the browsing (even if some features will be broken to the absence of the elements provided by affected resource).

Software Bill of Materials

To create and manage the inventory of all external components used across all software developed by a company, a "Software Bill of Materials"[10] inventory can be created, by project, and then gathered in a central system.

SBoM (Software Bill of Materials) is a list of all the open-source and external components present in a codebase. A software BoM also lists the licenses that govern those components, the versions of the components used in the codebase, and their patch status.

The tools provided by the "OWASP CycloneDX"[11] project can be used to generate the SBoM inventory for a project, several technologies are supported:

```
# Generate the SBoM inventory JSON file for a java maven project

$ mvn -q org.cyclonedx:cyclonedx-maven-plugin:makeAggregateBom

# List components used

$ cat target/bom.json | jq ".components[].name"

"spring-boot-starter-data-rest"

"spring-data-rest-webmvc"

"spring-data-rest-core"

…
```

Regarding the centralizing of all projects SBoM inventory, the project "OWASP Dependency-Track" can be used[12].

Additional new risk: Components intentional sabotage

Around January 2022, a new risk emerged through the form of "sabotage". Indeed, maintainers of open-source components deployed corrupted versions to protest against the usage of the free and open-source components in commercial/proprietary software[13] [14] [15]. These corrupted versions break the application using the components.

Exposure to such risk will occur during the upgrade of components, for example, following an upgrade of a framework that depends on components affected by a "sabotage" operation.

To detect exposure to such risk, it is possible to add a dedicated continuous integration pipeline[16] compiling and executing the unit test suites of the project against the last versions of all external components used. Most dependency management systems like maven, nuget, npm, pypi and so on support a syntax referring to last the version of a component.

This dedicated continuous integration pipeline is different from the pipeline in charge of validating the security posture of the application. Indeed, for this dedicated pipeline, versions of components **will not be fixed** and the last version of every component will always be used.

Example of syntax for maven for the project descriptor for pipeline detecting corrupted components:

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://http://www.w3.org/2001/XMLSchema-instance&#8221;

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<!– … –>

<dependencies>

<dependency>

<groupId>commons-lang</groupId>

<artifactId>commons-lang</artifactId>

<!– Want a version >= 1.0.0 of the dependency –>

    <!– The version 2.6 will be selected by Maven because –>

    <!– it is the higher one –>

    <!– at the time of creation of this code snippet –>

<version>[1.0.0,)</version>

</dependency>

</dependencies>

<!– … –>

</project>
```

```
# List components used

$ mvn dependency:tree

[INFO] — maven-dependency-plugin:2.8:tree (default-cli) @ sandbox —

[INFO] lu.xlm:sandbox:jar:1.0-SNAPSHOT

[INFO] \- commons-lang:commons-lang:jar:2.6:compile
```

Conclusion

As external dependencies are present in most in-house developments as well as open-source/proprietary software, the ability to list, manage, detect vulnerabilities, and update them becomes a critical point of a Software Development Life Cycle.

Valentin GIANNINI

Dominique RIGHETTO

Uncategorized

Recent Posts

The Necessity of Cyber Crisis Exercises

The Art of Password Spraying: A Comprehensive Analysis

Common client-side vulnerabilities of web applications

Categories

Advice
Consulting
Cyber Blog Post
General
Newsletter
PR
The Cyber Blog Times
Uncategorized

Search

Search …