ATTACK SURFACE MANAGEMENT VIA SBOM

Abstract

Large amounts of data come from a variety of sources during the development of software, including collaborators, open data providers, and commercial data suppliers. It becomes increasingly difficult to manage data usage and to keep a clear track of where data in a system has originated and makes onward contributions as the complexity of such data ecosystems grows. This article introduces the conceptual framework of a SBOM model for data traceability and accountability while the software is in a production state, giving the ability to make changes when there arises a zero day vulnerability for a component used in the product with increased accountability or entirely preventing similar attacks with software scans in a practical and easy model.

Introduction

Supply chain attack, also called a value-chain or third-party attack, is a type of cyber attack that targets organizations by focusing on weaker links in their supply chain. The supply chain is the network of all the individuals, organizations, resources, activities and technology involved in the creation and sale of a product. And the attack surface here includes the network which gets exposed as a result of the attack.

As open source is gaining popularity, developers make use of existing off-the-shelf softwares instead of implementing their own and this comes with a necessary contract: an implicit trust that the software will perform as desired and comes with (little to) no vulnerabilities. Because of this, it has become a common practice for developers to inherently trust and use open source software without validation. This can often lead to software being used in a branch of code and then forgotten about, creating the perfect attack window for people who want to exploit vulnerabilities in the dependencies code. Attacks such as Solar winds, and the Log4J attack are two great examples of what can go wrong with supply chain attacks:

The Breach of Solar winds's Orion platform is one of the most widespread and sophisticated hacking campaigns ever conducted against the private sector and the Government. Solar wind's computing networks were breached somewhere in the beginning of September 2019. A Trojanized code was injected into one of the software updates of the solarwinds orion software which provided the threat actor access with a backdoor to all those customers who updated with the malicious update. Since the infected software (Orion) was widely used to monitor network activity on federal and corporate systems, This incident allowed the threat actor to breach infected agency's and company's information systems. This supply chain attack is estimated to have compromised 18,000 customers including Government agencies and other vendors like Microsoft and Intel. In another example: Log4j Java library's role is to log information that helps with the debugging process when errors occur. The vulnerability allows unauthenticated remote code execution. Log4 Shell (CVE 2021-44228) means that attackers can remotely run whatever code they want and gain access to all data on the affected machine. It also allows them to delete or encrypt all files on the affected machine and store them for a ransom demand. This potentially makes it a target for anything that uses a vulnerable version of Log4j to log user-controllable data. With these relatively recent and extremely impactful vulnerabilities we set forth to advocate for a solution to prevent and minimize damage caused by these kinds of attacks.

Background:

Software bill of materials, or SBOMs, are a proposed idea to tackle this problem. They are a tabulated list of all the dependencies used in the making of a software. This solution understands the common practice of trusting other software but assures that no software in the supply chain is forgotten about, this leads to developers being able to compare their SBOMs against known vulnerable third-party software lists and secure their software against supply chain attacks, but this solution is not a perfect one.

In a deeper view, SBOMs are a nested inventory, a list of ingredients that make up software components. An SBOM also lists the licenses that govern those components, the versions of the components used in the codebase, and their patch status, which allows security teams to quickly identify any associated security or license risks all the way down to the version of the third-party software. The lack of visibility into software dependencies has been a major factor behind the push for SBOMs, particularly in the wake of solar wind attacks. Would having an SBOM prevent such attacks from happening? NO. Using an SBOM can help quickly identify and reduce the time to address the vulnerabilities but **is not a panacea for securing the supply chain.**

Threat Model:

Issues that arise when using an SBOM are few but important ones, the two we focus on being: It is difficult to guarantee that the SBOM is completely up to date if the SBOM is, for example, a simple .txt file. It would need developers to know exactly what dependencies are used in the code and to manually write it down. Having to keep an SBOM entirely up to date can create many problems and if not kept up to date correctly does not rectify any issues in supply chain attacks. The other issue being that while SBOMs can protect your software from attacks that have happened against others, it does nothing to protect you if you are the first target of an attack. Even if you do know about all the dependencies in your software, if one is vulnerable you can still be targeted for merely using that dependency.

There are more issues that come with using SBOMs that can range from the integrity of the actual SBOM to qualities that you expect an ideal SBOM to uphold, but we decided to focus on the two primary issues discussed above. For the sake of completeness we still outline a more thorough list of issues we found when using SBOMs:

The following are some qualities of what is expected in An Ideal SBOM:

- Supply chain integrity and Adaptability
- Compliance and Buyer power
-

Existing problem in SBOM Models :

- Difficult to maintain and Lack of Data (as discussed above)
- First attack front protection (as discussed above)
- False alarms and time Intensiveness
- Integrity of the SBOM itself

Design:

In our model we implement the SBOM creation in the CI/CD pipeline allowing for fluid software development cycles. This new SBOM model works in two steps, dynamically forms the SBOM on any change to the software repository, keeping track of software and software version, and then performs a software scan on the dependencies compiled in the newly-formed SBOM to assure safe third-party software. The first part solves the issue of having to maintain the SBOM manually. Having it be formed dynamically assures that no dependencies are forgotten about, allowing for any small piece of code to be able to be known about in the event of a known fault in a third-party software. The second part, the software scan, solves the second problem of no protection if you are the first front of

attack. The scan allows for developers to know before changing the main repository of a software if a dependency they added is known to have faults or if the scan has found a fault, this would allow the developers to fix their code and remove that vulnerability before even allowing anyone to take advantage of it. This SBOM model also takes updates into account. When a dependency is updated, a scan on that dependency would be run, if a new issue is found it is notified to the developers. We didn't want to automatically revert to old versions because some libraries do not provide backward compatibility and automatically reverting could break the software in other parts. To accommodate situations like this, we have left the decision for upgrades to the developers.

Implementation:

We make use of the Jenkins pipeline for our implementation. The reason for going with Jenkins is that it is a free open source tools predominantly used in many companies and has greater flexibility in providing integration with many languages like NodeJS, Java, Python., etc.

Currently there are a couple of formats in which SBOM is generated namely CycloneDX, and SPDX. The industry is still yet to standardize a particular format. In our implementation we are making use of the CycloneDX format.

Steps:

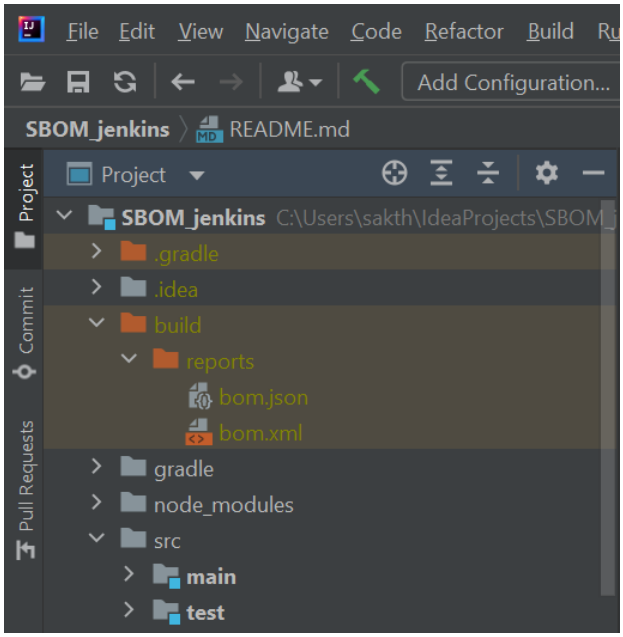
1. Install Jenkins locally for building the pipeline and include the necessary plugins that would be needed. For our implementation, we required NodeJs and Go plugin
2. Created a sample SpringBoot project which includes a couple of packages with known vulnerabilities.
3. We have to create a Jenkinsfile which would have the script for building the pipeline in Jenkins. Jenkins will identify that the Jenkinsfile is a part of the project and will make use of the script inside it to build the pipeline.
4. Created a webhook for the git project, which holds the Spring Boot application and the Jenkinsfile, so that Jenkins will run the pipeline every time there happens a commit to the repository.
5. Since the SBOM creation is a part of the CI/CD pipeline, every time the developer(s) makes a commit the code will be scanned to see if there is a vulnerability. If so it would be noticed earlier than waiting till the product is out for deployment. This way of incrementally checking for vulnerabilities will reduce last minute surprises during production and deployment.

(Continued implementation on next page)

6. Snippet of the Jenkinsfile:

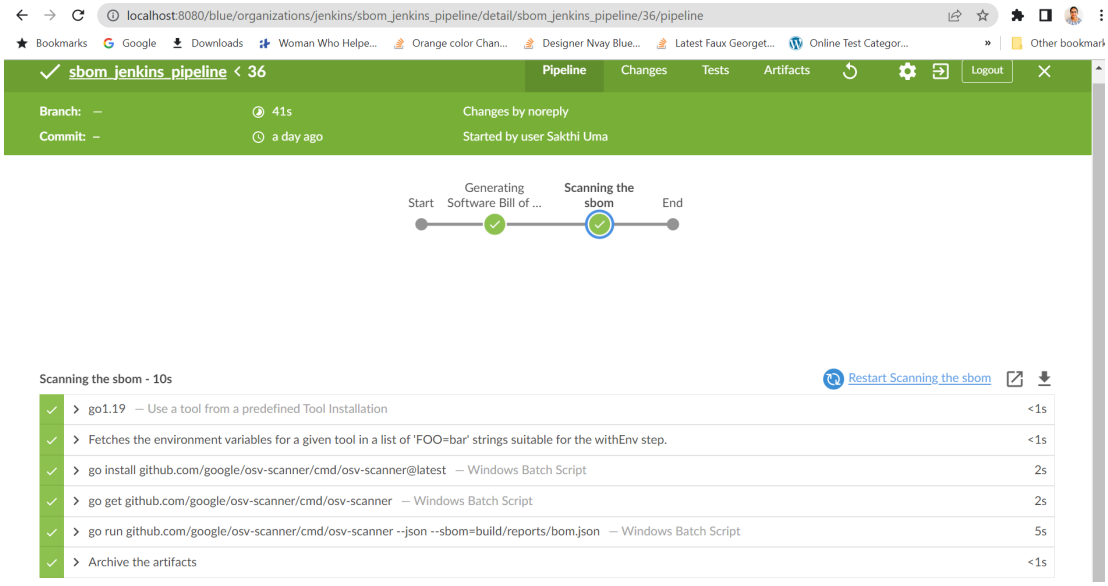
```
pipeline {
    agent any
    tools { go 'go1.19'}

    stages {
        stage ('Generating Software Bill of Materials') {
            steps {
                //Building the dependencies to generate SBoM
                bat './gradlew cyclonedxBom'
            }
        }
        stage ('Scanning the sbom') {
            steps {
                bat 'go install github.com/google/osv-scanner/cmd/osv-scanner@latest'
                bat 'go get github.com/google/osv-scanner/cmd/osv-scanner'
                bat 'go run github.com/google/osv-scanner/cmd/osv-scanner --json --sbom=build/reports/bom.json'
            }
        }
    }
    post {
            always {
                archiveArtifacts artifacts: 'build/reports/bom.json', onlyIfSuccessful: true
            }
        }
}
```

7. Here, we made use of a gradle plugin for including the CycloneDX as a part of the application. However, there is also cyclonedx generation that is provided as a part of npm which could also be used. (There were minor issues while setting up the NodeJs plugin as a part of the pipeline hence made use of this gradle plugin which also has the similar functionality)
8. The generated bom file are stored in the build/reports/ by default.



9. Also this will be stored as a part of the pipeline artifact after successful completion of the pipeline build (lines 20-25)
10. The created SBOM is then scanned using a OSV scanner, an open source vulnerability database that holds the information for vulnerabilities across many open source projects and Languages.
11. Lines 12-17 pulls the git project and runs the scan on the generated SBOM.

12. The pipeline:

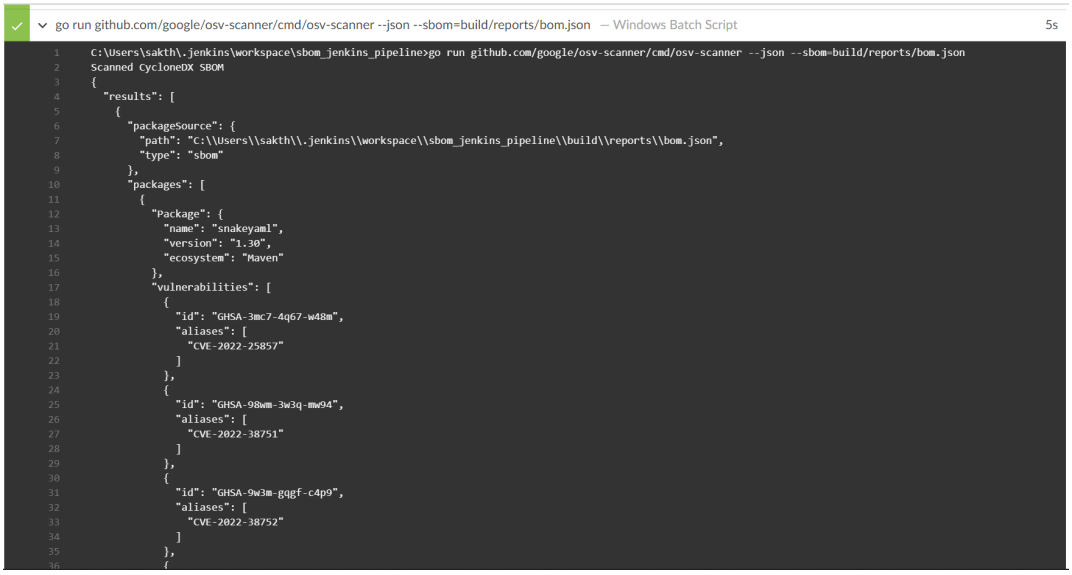


All of this code can be found at: https://github.com/sakthiuma/SBOM_jenkins

Evaluation:

There are two parts that come with evaluating our model, the first being the actual SBOM tabulation and the second being the software scan. We created an example project in the above GitHub repository and ran our software against it. The resulting SBOM file was created and it correctly found and labeled all the dependencies. A brief section of the CycloneDX file can be seen below:

To test the second part of our SBOM model we knowingly installed vulnerable libraries into the project before the SBOM creation and scan. We expected the result to be a list of all of the unsafe dependencies we installed.
After running the model we were able to get the correct output, a JSON file of any package that was either known to have a vulnerability or where a vulnerability was found. Below is a brief example of our output:



As an added test for our software we added Log4j into our GitHub repository knowing the security flaws and ran our software again. This time we received a newly compiled SBOM and a newly compiled scan that included the expected vulnerabilities including importantly Log4j.
Since we received the expected output of our two tests we believe that our solution is effective and returns all expected information of the SBOM and the Scan.

Related Solutions:

While compiling this paper we came across a company called Rezillion that advertises their product that introduces a dynamic SBOM, similar to the one proposed in this paper that also utilizes software scans. In the following paragraph we have decided to highlight the key differences between the one they provide and our proposed model, but as their product is extremely new (past quarter) we have limited information on which we can make comparisons. This is also the reason why we were unable to find any information about this product during our research phase and only came across it later on during the writing of this paper.

The product sold by Rezillion advertises a simple solution to the CI/CD SBOM flaws, including a dynamic SBOM created at runtime along with scans. The two biggest deltas to ours are as follows: Their SBOM and scans might be run at a different time to ours, later in the production pipeline, which we think is inefficient and flawed but we have no way of supporting that with the information available to use therefore we will not comment on it further. The other difference, and the more important one, being that when scans are found they implement automatic fixes for vulnerabilities. We see this as creating bigger issues, while we have no data on what these fixes are exactly, on their website they describe it as “ automatically mitigates exploitable vulnerabilities”. We can only assume that it would either completely stop the app currently running with the vulnerability or use a different software but either way, it would have a predictable countersafe. We believe that this can create bigger issues as mitigating the vulnerabilities allows for a completely predictable path that the CI/CD pipeline will take that can then be taken advantage of by attackers.

For the sake of clarity, this example highlights the flaw: Imagining an application set to run, there has been a recent update to a dependency of said software that has included a flaw, their software creates an SBOM and a software scan is set running and it finds that flaw. The software mitigates it and uses a different software dependency or product. In the future, attackers would be able to see the app using the different software, but that software app is now not accounted for in the SBOM, attackers could target that dependency and try to find flaws with it. This example works for any bypass and mitigation their SBOM might utilize as any new mitigation would be a new target for attackers. Our solution would entirely solve this example by very simply regressing to a past version of that dependency that was, very importantly, known to be clear from past scans of the SBOM or depend on the developers which, while less practical, resolves any issues that might arise from a predictable bypass of the software. From the information at our disposal we believe that this response would not be as flawed as the one, as it allows for much less predictability and more security, from the Rezillion software and why we believe that ours is more effective.

Conclusions:

In this paper we highlighted both the flaws in current methods of securing software supply chains and the motivation for removing all vulnerabilities that attackers could target in third-party dependencies. From the recent Log4j attacks to the Solar Winds attack we believe that a lot of the damage caused could be remedied by using a simple and practical SBOM model that we described in this paper. We laid out the steps to implement this framework in a Jenkins pipeline and demonstrated that the results were correct and expected. The answer we demonstrate is of course still not perfectly secure, and we propose future research to look into possible verification and integrity of the SBOM itself while integrating it securely and practically into our solution. The possible benefits from this solution were shown, but are not limited to the results, in the evaluation section and we believe it can be highly beneficial to anyone working on complicated software with many dependencies. Especially with the relatively recent NTIA guidelines of declaring the SBOM as a necessary step in the software development cycle this model can become highly useful in securing countless projects.

References:

https://www.rezilion.com/wp-content/uploads/2022/07/R3.0_SB_SupplyChainSec_0722_R3.pdf
https://www.rezilion.com/solutions/software-supply-chain-security/
https://www.microfocus.com/en-us/what-is/sast
https://jfrog.com/knowledge-base/why-and-how-to-perform-open-source-security-scans/
https://www.techtarget.com/searchsecurity/definition/supply-chain-attack
https://www.prplbx.com/resources/blog/log4j/
https://www.fortinet.com/resources/cyberglossary/supply-chain-attacks
https://redmondmag.com/articles/2021/12/16/log4j-attack-methods-explained-by-crowdstrike.aspx
https://www.synopsys.com/blogs/software-security/software-bill-of-materials-bom/
https://www.ntia.gov/files/ntia/publications/ntia_sbom_use_cases_roles_benefits-nov2019.pdf
https://www.protocol.com/enterprise/biden-sbom-open-source-software
https://www.enterprisenetworkingplanet.com/security/software-bill-of-materials-sbom-pros-cons/
https://www.simplilearn.com/tutorials/cryptography-tutorial/all-about-solarwinds-attack
https://www.businessinsider.com/solarwinds-hack-explained-government-agencies-cyber-security-2020-12
https://cybellum.com/blog/ntia-minimum-elements-for-a-software-bill-of-materials-sbom-a-guide/
https://www.techtarget.com/searchsecurity/post/The-benefits-and-challenges-of-SBOMs
https://www.ntia.gov/files/ntia/publications/ntia_sbom_healthcare_poc_report_2019_1001.pdf
https://healthitsecurity.com/features/2021s-top-healthcare-cybersecurity-threats-whats-coming-in-2022
https://healthitsecurity.com/features/using-software-bill-of-materials-sboms-for-medical-device-security
https://arxiv.org/abs/1904.04253
https://www.nature.com/articles/s41746-021-00403-w
https://github.com/google/osv-scanner
https://security.googleblog.com/2022/06/sbom-in-action-finding-vulnerabilities.html

Submitted by

Group 26:

Sakthi Uma Maheswari
sm10539@nyu.edu

Rutuj Anturkar
rsa4873@nyu.edu

Roman Negri
rvn9303@nyu.edu

Nadesh subramanian
ns5407@nyu.edu