Using Software Bill of Materials Metadata to Validate the Integrity of Third-Party Software Components

CS6727 Project Report
December 12, 2021

Jeffrey B. Otterson
School of Cybersecurity and Privacy
Georgia Institute of Technology
Atlanta, Georgia, USA
otterson@gatech.edu

***Abstract*— Cybercriminals are getting smarter in attacking software. They are now introducing deliberate vulnerabilities and even malicious code into third-party software that vendors then ship. This has been described as a "software supply-chain attack." Software vendors need to identify solutions to detect and prevent these supply-chain attacks and to prevent malicious software from entering their release process.**

Software applications can have many dependencies on third-party components, often open source. These third-party components provide functionality such as logging, data access, web endpoints and other APIs, and can be used in places where they may operate with elevated permissions. Consequently, these third-party components represent a very real supply-chain risk to software application vendors and producers of enterprise software applications.

The approach described in this paper is based on the idea that there is enough metadata available in "Software Package Data Exchange" (SPDX) software bill of materials (SBOM) files to perform integrity validation of third-party components and help prevent supply-chain attacks.

Using a sha256 hash, file type, and concept of an "ideal" SBOM, this approach proved that unexpected changes to third-party dependencies can be detected at build-time and also at delivery time. The ideal SBOM approach requires that the third-party dependencies be declared before the build.

This paper describes the approach, methodology, and tools. The results indicate that this is a viable and useful approach, it is lightweight and easy to use.

Keywords—cybersecurity, open-source, software bill of materials, SBOM, SPDX, supply chain, third-party

I. INTRODUCTION

Modern software packages often include many open-source and other third-party software components. One enterprise application surveyed included 115 open-source software components. That application is over 85% third-party code. Software vendors often include many third-party components when they build their products – but how are these added into the product, and when? How does the vendor know that the component shipped in their package is authentic and what they meant to distribute?

Recent high profile "supply chain" attacks on vendors such as SolarWinds [1] and Kaseya [2] have shown the power and reach of these attacks. It has been estimated that up to 18,000 Solar Winds customers were compromised by that attack, and about 1,500 of Kaseya's customers were affected by the

Kaseya supply chain attack. [3] Supply chain attacks continue to grow, as the attackers move "upstream" and try to attack common components used by many developers. [4] [5]

Every added third-party component adds risk to the application. Software vulnerabilities in third-party software represent a significant risk and poor inventory control of the third-party software adds to that risk. Because there are so many components used, the management of dependencies becomes burdensome and sometimes shortcuts are taken, which may result in the improper, damaged, deliberately defective and/or vulnerable components being included either by accident or malicious intent. This can be called "configuration drift."

Considering the vast amount of third-party, often open-source software that is included as components of modern software packages, how can a package builder ensure that the components delivered are in fact the desired components? How can a package builder detect and act on accidental or deliberate change to the third-party components in their package?

My solution uses metadata in Software Bill of Materials [6] (SBOM) files to catalog third-party dependencies and validate file integrity by comparing cryptographic hashes for the components with an ideal, prebuilt SBOM. The tools can verify the content and integrity of software package build outputs as well as software packages installed in customer environments.

SBOM has recently gained attention since Section 4, part e of Executive Order 14028 [7] includes language that may allow NIST to require a SBOM be available for software packages. NIST responded by updating the Secure Software Development Framework (SP 800-218) [8] to include new "recommended practices" including "PS.3.2" and "PW.4.5" that require SBOM be shared and to "confirm the integrity of software components through digital signatures or other mechanisms." This project demonstrates a solution that uses SBOM metadata to meet those recommended practices.

This paper will show how the methodology and tools have detected "configuration drift" and file corruption, or (even better) a lack of drift and lack of corruption, which improves confidence in the overall security of a software application release.

The rest of the paper will discuss what research was done, the approach to the project, the methodology to use the project, how the work was tested, and results of evaluation of the project.

II. METHODS

A. Research

1) Background

There are several existing approaches that allow for file integrity monitoring; however, none currently support integrity verification across many phases of the product lifecycle. Tools such as Gradle [9] can provide dependency integrity verification but only during the build phase. The Linux package manager "rpm" [10] can verify software package integrity, but only in the run-time environment. There are tools that can generate manifests of software packages or a software bill of materials. However, these tools run after the build phase completes, so they can catalog what has been built, but there is no mechanism to verify that build output includes the intended third-party components, and that the components are authentic.

2) Selection of Software Bill of Materials

To verify integrity of the components of a software package release, I needed to have a manifest that enumerates and describes the components. A software bill of materials (SBOM) is a software analogy to a bill of materials used when assembling a product. Just as a bill of materials for a physical object should include a detailed list of all the individual components, a SBOM should itemize and describe all the individual components that are included with a software package. SBOM is not a new idea but its further development and use has been encouraged and accelerated by Executive Order 14028 "Improving the Nation's Cybersecurity." [7] This was sufficient impetus to use SBOM technology for this project.

Since the inception of this project, NIST has produced a draft of "Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities" [8] and a new draft of "Cybersecurity Supply Chain Risk Management Practices for Systems and Organizations (2nd Draft)" [11] —both of which suggest the use of SBOM data to mitigate supply chain and other

software vulnerabilities. This is encouraging and affirms that this project has taken the proper direction.

3) Selection of a SBOM Format

Executive Order 14028 required the NTIA to produce a document that detailed the "minimum elements for a SBOM." [12] The NTIA document describes the minimum required data for a SBOM to be effective. It specifies seven elements as a required minimum. The NTIA elements document also recommends several additional elements, two of which are needed for this scheme to succeed. These include a cryptographic hash of the component and license information for the component.

There are two well-known SBOM formats in use: one is the Open Web Application Security Project's (OWASP) "CycloneDX" [13] format, the other is the Linux Foundation's "SPDX" [14] format. I read and compared specifications and library support for CycloneDX and SPDX. I chose to implement using the SPDX format, but I believe the same approach would work with CycloneDX format by using storing the additional metadata (notable file path) using the generic "properties" name/value extension [15].

One of the attractive features of the SPDX format was the 'tools-python' [16] library. This library allows easy creation and maintenance of SPDX format SBOMs. The SPDX standard supports one or more cryptographic hashes for any file in the SBOM and requires that a SHA1 hash be present for each file. SHA1 is widely considered "broken" and "unsuitable" since 2017 [17]; therefore, it did not make good sense to develop a new cybersecurity application using a known weak cryptographic hash. Written specification notwithstanding, the 'tools-python' library did not support any cryptographic hashes other than SHA1, so I had to make improvements to the library [18] to support a strong cryptographic hash: sha256. I have submitted a git "pull request" [19] to get these changes merged with the mainline 'tools-python' library.

The selection of SPDX format was later confirmed as the proper choice when the ISO chose to recognize SPDX as an international standard. [20]

4) Selection of Implementation Language

I chose to implement this project using Python. The SPDX project has API support for Go, Java, and Python. I selected Python because it has broad support on the most popular operating systems and often is pre-installed. Python scripts are easily integrated into existing DevOps chains. Also, I felt that I would be able to finish the implementation more quickly in Python.

As described above, the 'tools-python' library was not completely compliant with the SPDX 2.2 specification. There was no support for any "checksum" other than SHA1, and only one file type value was supported by the library. This required me to modify the library to deliver the functionality I needed.

All the Python scripts used by this approach were written for the Python 3.9 environment. At this time, the changes to the tools-python library have not yet been merged to the "upstream" library, so the patched library from GitHub is needed.

B. Approach / Methodology / Design

1) The Approach

The solution uses a pre-built "ideal" SBOM that identifies all the approved, permitted third-party software identified as well as all other expected build outputs. The "ideal" SBOM is generated by merging build SBOM data with a SBOM containing a "curated" list of third-party components. The curated list of third-party components must be created by hand since each third-party component must be individually vetted.

New application builds are verified by comparing them to the "ideal" SBOM. The build comparison process also produces a "release" SBOM that can be shipped with the software package and can be used to validate the integrity of all the binary executables and libraries in the run-time environment.

The initial approach used an "ideal" SBOM but did not use the "third-party" SBOM file. This made maintenance of the ideal SBOM labor-intensive. Separating the third-party components into their own file made maintenance of both the third-party SBOM and the ideal SBOM much simpler.

2) How it Works

The first step to validate the integrity of the third-party components of an application is to identify and catalog them. In this approach, this catalog stored as a SBOM—the "third-party" SBOM. This "third-party" SBOM file contains filename, license information, and a sha256 hash value for every third-party component

file. The creation and maintenance of third-party SBOM files is discussed below.

Next, an "ideal" SBOM file for the application package is created. The "ideal" SBOM is a contract for what the delivered application package should look like: what files in what locations and particularly, the sha256 cryptographic hash and license data for each third-party component that are included in the application package.

The "ideal" SBOM is created by merging a "bootstrap" SBOM with the "third-party" SBOM. The "bootstrap" SBOM is created by scanning the application package and contains filenames and other metadata for every file found in the package.

After an application package build completes, the build is verified by comparing a new "build" SBOM, made from the build with the "ideal" SBOM. This comparison validates that no files were added or removed from the build (used to detect "drift") and that the sha256 hash for every third-party component matches the ideal SBOM. Detected "drift" or hash mismatches will cause an error to be reported and can block automatic DevOps toolchains. The comparison also produces a "product build" SBOM, which can be shipped with the application package.

The application package can be verified in the run-time environment using the "product build" SBOM. This process verifies the sha256 hash for all identified binary executables and library files in the package, not just the third-party components. The ability to validate the integrity of all the identified binaries in the package was a serendipitous outcome.

SBOM files used by these tools have their integrity validated by digital signature. All the SBOM files created by these tools can apply a digital signature to the SBOM files and all the tools that read SBOM files can validate the integrity of the SBOM file by testing the digital signatures. The digital signature uses an RSA keypair created by ssh-keygen [21].

3) SBOM Metadata

The solution uses the SBOM metadata. The following fields are used:

- File Name – used for testing file presence
- File Type – the file type "APPLICATION" is used to select files for hash matching
- File Checksum – the SHA256 value is used to confirm file integrity
- License Concluded – the file license information is used to identify third-party components
- Document Comment – The SPDX document comment field is used to store the digital signature for the SBOM file.

Note that the SHA256 checksum is the only metadata item that would not be present by default in a SPDX SBOM document. The digital signature for the SBOM, stored as document comment metadata, is not required in order to use this solution, but it does allow the SBOM integrity to be validated.

4) Testing

Testing was performed using two application packages. (These packages are proprietary commercial software, so their names will not be disclosed in this paper.) The first package—"Package One"—is a 275 mb collection that contains 3,420 files, of which 148 are third-party components. "Package Two" is a 51 mb collection that contains 7,292 unique files, of which 14 are third-party components.

Testing was performed over a period of six weeks, from October 10 to November 20, 2021. Build outputs were validated twice weekly. "Drift" was detected in "product one" on four occasions. Three of these were due to third-party component upgrades, the fourth was caused by the addition of a new conversion script to the product. In each of these four cases, the cause of the drift was determined to be legitimate and the "ideal" SBOM was updated. No malicious changes to third-party components were detected by the tools with this real-world testing; however, this was the hoped-for, expected result.

Testing showed that file "drift"—the appearance of new or disappearance of existing files—was successfully detected by the tools. Updated third-party components were detected using the drift detection mechanism. Since no malicious modifications of the third-party components were expected to occur, deliberate modifications were made to some of the third-party component libraries and these

modifications were detected by the hash mismatches. Both build-time and run-time tests for corrupted third-party components were performed successfully. The deliberate modifications that were detected included:

- Renaming a prior version of the component
- Extending the size of the component by adding two bytes to the end of the file
- Truncating the component file

Testing also revealed a prior version of a third-party component was not removed from the build environment and was improperly added to a subsequent build. This was also revealed with the drift-detection mechanism.

The approach is intended to be lightweight, fast, and easy to use. Testing showed that for these two packages, the worst-case impact on automated build processes was under four seconds. The SBOM files used by the process are not prohibitively large and provide significant value after-the-fact.

Table 1 shows typical elapsed time and file sizes for the two application packages.

TABLE I. TYPICAL EXECUTION TIME AND FILE SIZES FOR TESTED APPLICATION PACKAGES

Application Package	Number of files	Size, MB	Create SBOM, seconds	Validate Build, seconds	Size of "ideal" SBOM, bytes
Package One	3420	275	2.7	1.1	1379633
Package Two	7292	51	1.8	1.8	2234475

5) The Tools

The tools are all built with the philosophy that tools should be simple and do one thing well. All the tools can be run from the command line and all but 'edit-sbom.py' can be used in scripts, for automated access during the software build process. The tools are available in a GitHub repository [22].

The entire toolset consists of about 2,000 lines of Python.

a) Description of the Tools

'create_sbom.py' is a tool that can create a SPDX SBOM file from files on disk or files in "zip" archives, either with or without full path information. This tool is used to produce SBOM data at several stages in the process. The initial "third-party," "bootstrap," and "package-build" SBOMs are created with this tool.

'edit-sbom.py' is a tool that allows editing individual file entries in a SBOM. This is useful for setting license information for third-party components. This tool is a Python "text user interface" program, it runs in a terminal window.

'merge-and-test.py' is a tool used at build time to validate the third-party components at build time. It will also notify of any new or deleted files in the build output, "drift", which allows new third-party components to be detected. 'merge-and-test.py' uses an "ideal" SBOM as input and produces "build" SBOM data as output.

'merge-by-sha256.py' is used to merge third-party component data with "bootstrap" SBOM data in order to create the "ideal" SBOM.

'validate-sbom.py' uses the "build" SBOM to validate the integrity of both third-party components and identified components of a software package installed in a run-time environment. (Validation of "application" components, in addition to the third-party components, was a valuable extra benefit that this solution provides.)

6) Additional Python Modules

'signature_utilities.py' contains helper functions for digital signatures. This module contains functions to create and verify digital signatures, and helper functions to read the RSA keys created by ssh-keygen.

'spdx_utilities.py' contains functions and data that are used to assign file types to files, to read and write tag/value format SPDX SBOM files, to create new SPDX documents, and to apply or fetch a digital signature in a SPDX document.

'validation_utilities.py' contains functions that are useful for enumerating files in a tree-structure and calculating cryptographic hashes for disk files.

a) Digital Signature Implementation

The SPDX standard does not support digital signatures in the SBOM files but digital signature for SBOM was a required design goal. It is critical that the authenticity and integrity of the SBOM files can be proven, else the metadata in the SBOM files is worthless. The tools implement a digital signature scheme with a method that is compatible with the

SPDX standard. The digital signature is stored as part of the SPDX document comment. The entire SPDX document (except the comment) is serialized and then used to create or validate the digital signature.

The digital signature algorithm uses an RSA key pair for signing and validating. The private key is used to sign and the public key is used to validate. The keys are stored in OpenSSH format and can be created with the ssh-keygen utility. This approach was selected as the most lightweight, most users will already have OpenSSH installed and can readily create a key pair.

Though ssh keys were used for testing digital signatures, the tooling also supports the use of RSA public keys in X.509 certificates. This allows SSL certificates issued by a certificate authority to be used to validate the digital signature on a SBOM file. The associated RSA private key must be used to sign these SBOMs in this case. The traceability of a X-509 certificate that was issued by a "real" certificate authority may be more palatable for some users. This is currently supported by the *read_public_key_from_x509_cert()* and *read_rsa_private_key()* functions in *signature_utilities.py*.

7) Using the Tools

a) Creating the "third-party" SBOM

A SBOM file is created for all the "approved" third-party components in the package by using the 'create-sbom.py' script, pointed at the repository of approved third-party components. Note that the "approval" process requires human activity, it is up to people to decide what is "approved." Then, this third-party components SBOM is edited with the 'edit-sbom.py' script to insert license information for the third-party components. The presence and type of license is later used to target third-party components in the build outputs. Figure 1 shows this entire process. Figure 2 demonstrates the use of the 'create-sbom.py' script to create a third-party SBOM and Figure 3 demonstrates the 'edit-sbom.py' script.

b) Creating the "ideal" SBOM

Next, a "bootstrap" SBOM is created for the software application by using the 'create-sbom.py' script. This bootstrap SBOM is merged with the third-party SBOM using the 'merge-by-sha256.py' script, which combines the license data, file types, and expected hashes to produce the "ideal" SBOM for the application. The "ideal" SBOM can be edited for maintenance using the 'edit-sbom.py' script. Figure 4 shows this process, and Figure 5 demonstrates the creation of the bootstrap SBOM and merging with third-party data to produce the "ideal" SBOM.

c) Testing the Release and Creating the "release" SBOM

When a build is made, the 'create-sbom.py' script is used to create a SBOM for the build outputs. The 'merge-and-test.py' script is used to compare the build output SBOM to the ideal SBOM. Missing and added files in the build output are identified, but more importantly, any third-party components with mismatched hashed are identified. This process also creates the 'release' SBOM with license and other metadata merged from the ideal SBOM. Figure 6 shows this process and Figure 7 shows a demonstration.

d) Validate Installed Release

Finally, the release SBOM can be used to validate the integrity of the installed package in the runtime environment using the 'validate-sbom.py' script. This is shown in Figure 8.

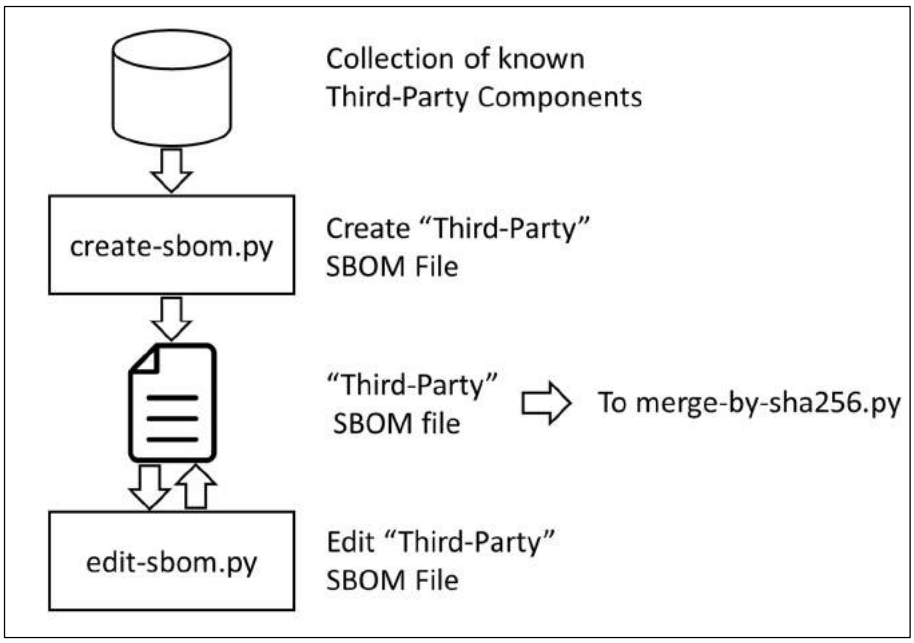

Fig. 1. Create and Edit Third-Party SBOM

```
[jeff.otterson@otterson3 sbom-validator]$ ./create-sbom.py --flat --sbom-file third-party-20211203.spdx --
package-path ../lib --private-key test-key-for-sbom-validator
2021-12-03 10:48:38,708 INFO Enumerating files at ../lib...
2021-12-03 10:48:38,708 INFO Directory enumeration found 151 files
2021-12-03 10:48:39,392 INFO Writing file third-party-20211203.spdx
2021-12-03 10:48:39,394 INFO Reading file third-party-20211203.spdx
[jeff.otterson@otterson3 sbom-validator]$
```

Fig. 2. Creating a Third-Party SBOM

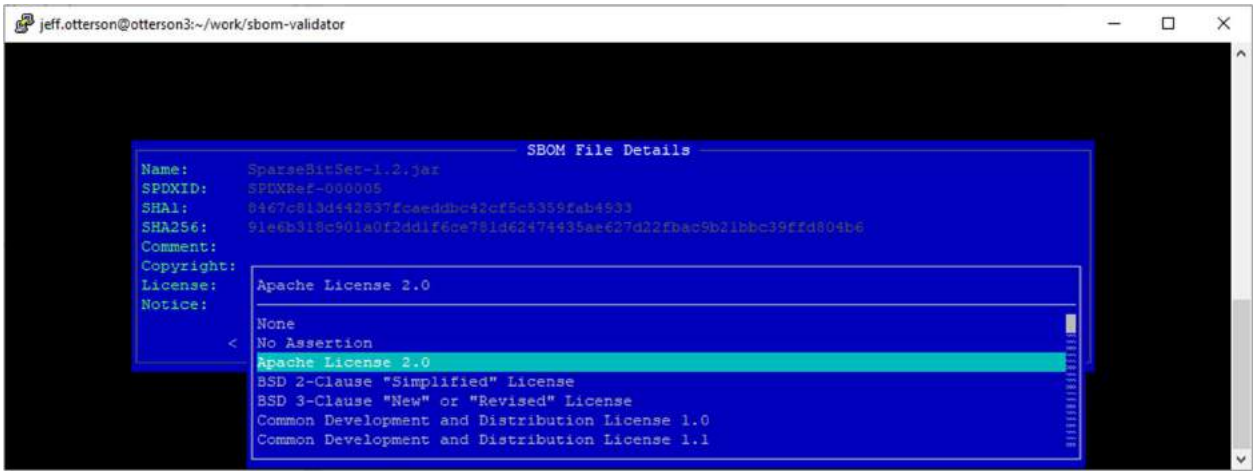

Fig. 3. Editing a Third-Party SBOM with the Text UI

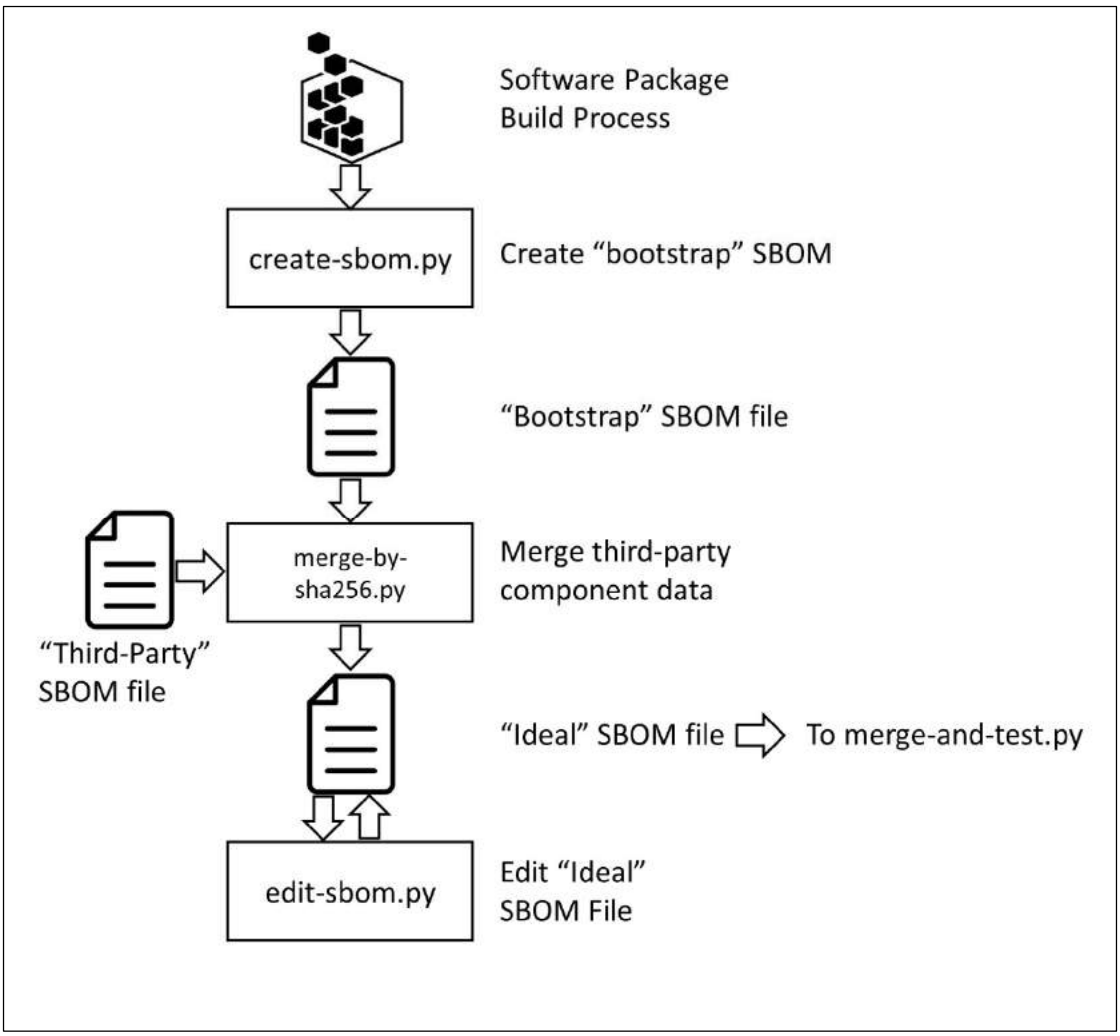

Fig. 4. Create/Edit the "Ideal" SBOM

```
[jeff.otterson@otterson3 sbom-validator]$ ./create-sbom.py --sbom-file cm-20211130.spdx --package-zip
../(redacted)-cm-(redacted).zip --private-key test-key-for-sbom-validator
2021-12-03 11:23:23,113 INFO Enumerating files in ../(redacted)-cm-(redacted).zip...
2021-12-03 11:23:23,125 INFO Zipfile contains 3523 files
2021-12-03 11:23:26,055 INFO Writing file cm-20211130.spdx
2021-12-03 11:23:26,094 INFO Reading file cm-20211130.spdx
[jeff.otterson@otterson3 sbom-validator]$ ./merge-by-sha256.py --info --source-sbom cm-20211130.spdx --
merge-sbom third-party-20211104.spdx --result-sbom cm-ideal-20211130.spdx --private-key test-key-for-sbom-
validator --public-key test-key-for-sbom-validator.pub
2021-12-03 11:23:32,118 WARNING File names do not match but sha does: log4j-12-api.jar should be log4j-12-
api-2.13.3.jar
2021-12-03 11:23:32,118 WARNING File names do not match but sha does: log4j-api.jar should be log4j-api-
2.13.3.jar
2021-12-03 11:23:32,118 WARNING File names do not match but sha does: log4j-core.jar should be log4j-core-
2.13.3.jar
2021-12-03 11:23:32,118 INFO 145 spdx files merged using without error.
2021-12-03 11:23:32,118 INFO 3 spdx files merged with warnings.
2021-12-03 11:23:32,118 INFO Writing result SBOM cm-ideal-20211130.spdx
2021-12-03 11:23:32,196 INFO done.
[jeff.otterson@otterson3 sbom-validator]$
```

Fig. 5. Create bootstrap SBOM and merge with ideal SBOM

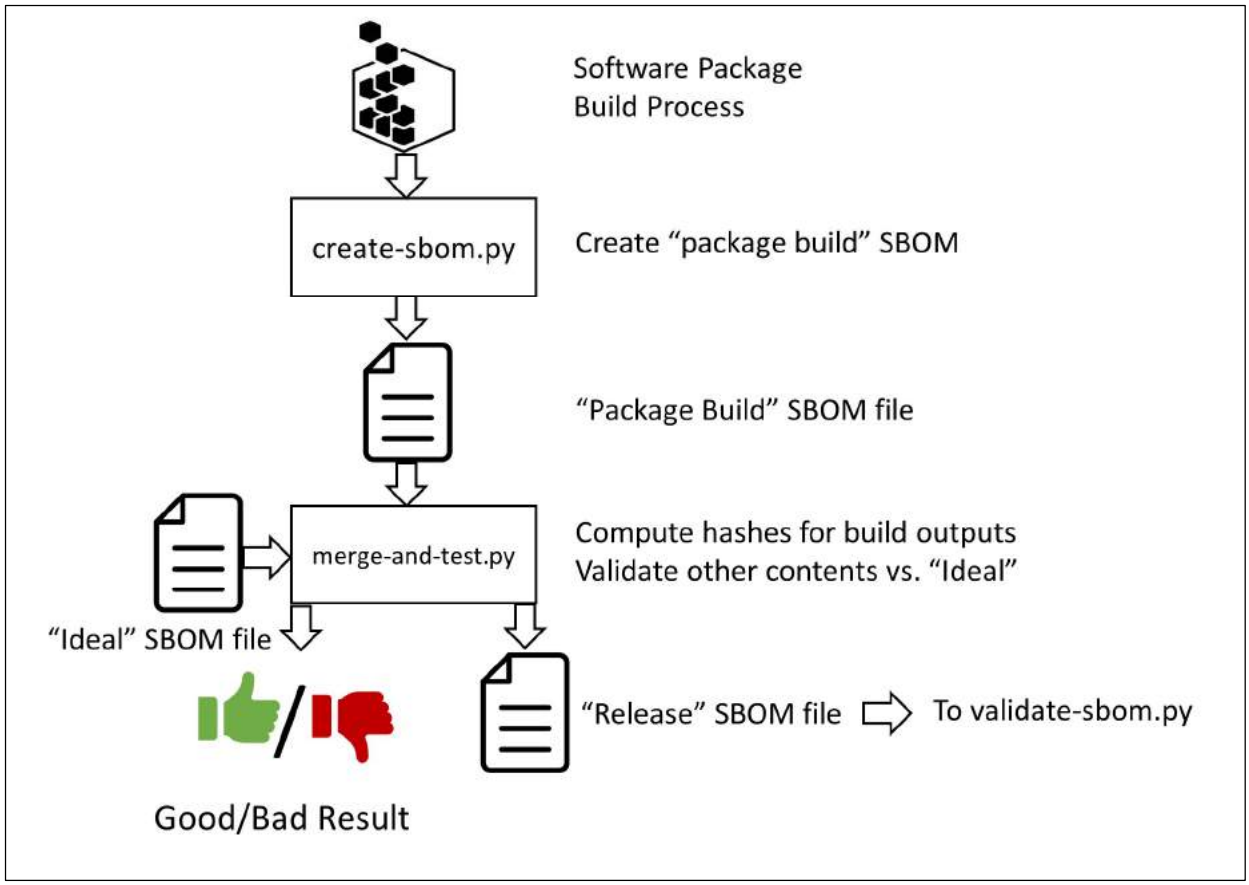

Fig. 6. Testing the Release and creating the “release” SBOM

```
[jeff.otterson@otterson3 sbom-validator]$ ./merge-and-test.py --info --build-sbom cm-20211130.spdx --ideal-
sbom cm-ideal-20211130.spdx --result-sbom=cm-result-20211130.spdx --private-key test-key-for-sbom-validator
--public-key test-key-for-sbom-validator.pub
2021-12-03 11:34:41,584 INFO 148 third-party dependencies had data merged.
2021-12-03 11:34:41,585 INFO No errors or warnings.
2021-12-03 11:34:41,585 INFO Writing result SBOM cm-result-20211130.spdx
2021-12-03 11:34:41,661 INFO done.
[jeff.otterson@otterson3 sbom-validator]$
```

Fig. 7. Validating a release.

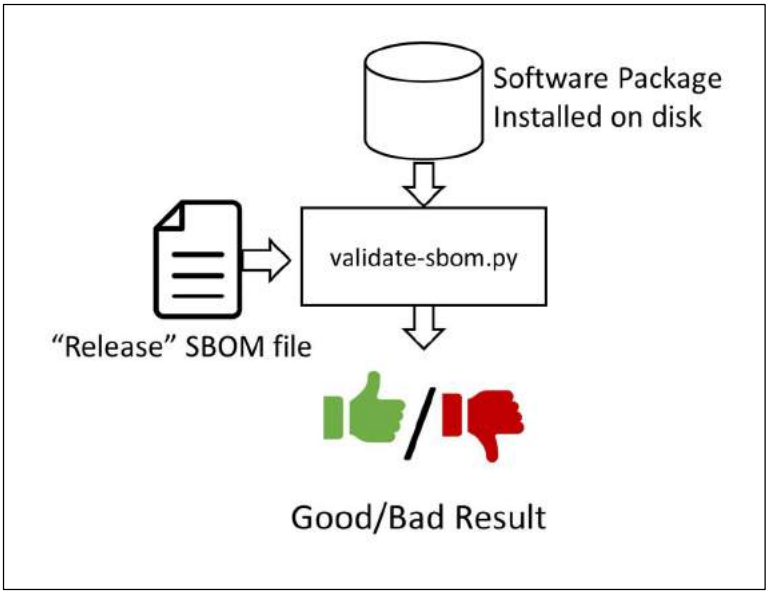

Fig. 8. Validating a Installed Release

```
[jeff.otterson@otterson3 sbom-validator]$ ./validate-sbom.py --info --package-path /opt/(redacted)/cst -
sbom-file cst-result.spdx --public-key test-key-for-sbom-validator.pub
2021-12-03 11:24:01,345 INFO Reading SBOM file cst-result.spdx
2021-12-03 11:24:02,401 INFO SBOM file contains 7291 file entries
2021-12-03 11:24:02,486 INFO Digital signature on SBOM file is good. SBOM file appears authentic.
2021-12-03 11:24:02,487 INFO Enumerating files in /opt/(redacted)/cst...
2021-12-03 11:24:02,498 INFO Directory enumeration found 7291 files
2021-12-03 11:24:02,693 WARNING Checksum mismatch! File .lib/commons-exec-1.0.1.jar sha256 checksum does not
match the SBOM
2021-12-03 11:24:02,700 INFO 1651 file(s) were excluded from checksum matching.
2021-12-03 11:24:02,700 WARNING Package fails integrity testing.
[jeff.otterson@otterson3 sbom-validator]$
```

Fig. 9. Validating an installed software package, with failure.

III. RESULTS

The approach detailed in this paper solves the problem of validating the integrity of the third-party components throughout several phases of the application's life cycle. The approach is unique and useful because it utilizes metadata in SBOM files to perform the validation, and it is becoming increasingly likely that SBOM files will soon be required for application software sold (at least) to the United States government.

The approach is lightweight. Little additional metadata needs to be added to the SBOM files, the scripts execute quickly, and the data is easy to maintain. The approach adds value to application SBOMs by merging license data for the third-party components into the application SBOMs. This allows license detailed compliance documentation to be created automatically from the SBOMs.

A. Findings

Testing this approach over a period of six weeks detected expected "drift" of the package contents, added files were reliably detected, and updated third-party components were found. No malicious or unexpected changes to any third-party components were detected; however, that was the expected result.

IV. CONCLUSION

This project proved that SBOM metadata can be used to validate the integrity of third-party components of a software application across multiple phases of the software lifecycle. This functionality can be used as part of an overall approach to manage software risk throughout the supply chain. Little additional metadata was required to be added to the SBOM files to make this approach work and the approach itself is designed to be minimally invasive to the software development and release process. Testing demonstrated that the impact both in CPU time and disk space consumed was reasonable.

A. Future Opportunities

It would be useful to have additional functionality in the SBOM editing tool 'edit-sbom.py.' In particular, the 'add' functionality needs to be completed. I believe that tools for scripted merge to and delete from SBOM files would be useful for build automation.

The SPDX standard does not currently support digital signatures in the SBOM files. The tools built for this approach implement their own digital signature mechanism. It would be better if the standard supported this.

This paper will be added to the GitHub repository that contains the code. I plan to bring this solution to the attention of the "SPDX Outreach Team." I created this solution to address a need that I had, and I believe this solution can be useful to others, so I am going to publicize it.

ACKNOWLEDGMENTS

I would like to thank Dr. Karl Grindal of Georgia Institute of Technology's School of Cybersecurity and Privacy. During the inception phase of this project, Dr. Grindal pointed out the SBOM requirements that were included in the "Executive Order on Improving the Nation's Cybersecurity" and that lead me to use SBOM as the manifest for this work.

REFERENCES

[1] A. Culafi, "SolarWinds confirms supply chain attack began in 2019," TechTarget Network, 12 January 2021. [Online]. Available: https://www.techtarget.com/searchsecurity/news/252494704/SolarWinds-confirms-supply-chain-attack-began-in-2019. [Accessed 8 October 2021].

[2] L. Whitney, "Kaseya supply chain attack impacts more than 1,000 companies," TechRepublic, 6 July 2021. [Online]. Available: https://www.techrepublic.com/article/kaseya-supply-chain-attack-impacts-more-than-1000-companies/. [Accessed 8 October 2021].

[3] L. Tung, "Kaseya ransomware attack: 1,500 companies affected, company confirms," Ziff-Davis, 6 July 2021. [Online]. Available: https://www.zdnet.com/article/kaseya-ransomware-attack-1500-companies-affected-company-confirms/. [Accessed 8 October 2021].

[4] P. Sawyers, "Next-gen software supply chain attacks up 650% in 2021," VentureBeat, 15 September 2021. [Online]. Available: https://venturebeat.com/2021/09/15/next-gen-software-supply-chain-attacks-up-650-in-2021/. [Accessed 8 October 2021].

[5] P. Wait, "Don't Blindly Trust Software Building Blocks, Report Says," NextGov.com, 10 November 2021. [Online]. Available: https://www.nextgov.com/cybersecurity/2021/11/dont-blindly-trust-software-building-blocks-report-says/186775/. [Accessed 30 November 2021].

[6] "Software bill of materials," 2 August 2021. [Online]. Available: https://en.wikipedia.org/wiki/Software_bill_of_materials.

[7] "Executive Order on Improving the Nation's Cybersecurity," 12 May 2021. [Online]. Available: https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/.

[8] M. Souppaya, K. Scarfone and D. Dodson, "Draft NIST Special Publication 800-218," 30 September 2021. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-218-draft.pdf. [Accessed 21 November 2021].

[9] Gradle developers, "Verifying dependencies," 2021. [Online]. Available: https://docs.gradle.org/current/userguide/dependency_verification.html. [Accessed 23 November 2021].

[10] RedHat, Inc. and others, "Verifying Installed RPM Packages," 23 November 2021. [Online]. Available: https://docs.fedoraproject.org/en-US/Fedora_Draft_Documentation/0.1/html/RPM_Guide/ch04s04.html. [Accessed 23 November 2021].

[11] J. Boyens, A. Smith, N. Bartol, K. Winkler, A. Holbrook and M. Fallon, "Draft (2nd) NIST Special Publication 800-161: Revision 1: Cybersecurity Supply Chain Risk Management Practices for Systems and Organizations," 28 October 2021. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-161r1-draft2.pdf. [Accessed 21 November 2021].

[12] National Telecommunications and Information Administration, "The Minimum Elements For a Software Bill of Materials (SBOM) Pursuant to Executive Order 14028 on Improving the Nation's Cybersecurity," 21 July 2021. [Online]. Available: https://www.ntia.doc.gov/files/ntia/publications/sbom_minimum_elements_report.pdf.

[13] "CycloneDX Software Bill of Materials (SBOM) Standard," 28 August 2021. [Online]. Available: https://cyclonedx.org/.

[14] Linux Foundation, "SPDX specification v2.2.0," 28 August 2021. [Online]. Available: https://spdx.github.io/spdx-spec/.

[15] OWASP Foundation, "CycloneDX v1.3 JSON Reference," 04 May 2021. [Online]. Available: https://cyclonedx.org/docs/1.3/json/. [Accessed 27 November 2021].

[16] P. Ombredanne, P. Tardy and N. Voss, "Python library to to parse, validate and create SPDX documents," 16 September 2021. [Online]. Available: https://github.com/spdx/tools-python. [Accessed 24 September 2021].

[17] M. Stephens, E. Bursztein, P. Karpman, A. Albertini and Y. Markov, "The first collision for full SHA-1," 23 February 2017. [Online]. Available: https://shattered.io/static/shattered.pdf. [Accessed 23 November 2021].

[18] J. B. Otterson, "Python library to to parse, validate and create SPDX documents," 24 September 2021. [Online]. Available: https://github.com/jotterson/tools-python. [Accessed 24 September 2021].

[19] J. B. Otterson, "add support for multiple file checksums, file types #200," 26 November 2021. [Online]. Available: https://github.com/spdx/tools-python/pull/200. [Accessed 21 November 2021].

[20] International Organization for Standardization, "ISO/IEC 5962:2021 Information technology — SPDX® Specification V2.2.1," August 2021. [Online]. Available: https://www.iso.org/standard/81870.html. [Accessed 3 December 2021].

[21] The OpenBSD Project, "ssh-keygen(1) - Linux man page," 14 April 2013. [Online]. Available: https://linux.die.net/man/1/ssh-keygen. [Accessed 27 11 2021].

[22] J. B. Otterson, "sbom-validator," 23 September 2021. [Online]. Available: https://github.com/jotterson/sbom-validator.

APPENDIX: SOURCE CODE