Alice in (Software Supply) Chains: Risk Identification and Evaluation

Giacomo Benedetti(✉), Luca Verderame, and Alessio Merlo

DIBRIS - University of Genoa, Genoa, Italy
{giacomo.benedetti,luca.verderame,alessio.merlo}@dibris.unige.it

Abstract. The fast pace of modern development paradigms like DevOps boosted the complexity of development pipelines. In particular, developers rely on many external assets and third-party software to build the final product and match the demanding requirements in terms of release cycles and functionalities. However, such a choice impacts all the elements of the development pipeline composing the so-called *Software Supply Chain* (SSC), degrading its maintainability and security. From a security standpoint, successful attacks can go unnoticed and impact many targets that use the affected software before being resolved. Unfortunately, traditional security assessment methodologies might detect the symptoms (e.g., the piece of vulnerable code) but not the cause, i.e., the attack vector and the affected asset of the SSC, failing to mitigate the risk of new attack campaigns.

In this paper, we propose *Sunset*, a methodology with a two-fold objective. First, it allows the automatic reconnaissance of the SSC assets and dependencies to alleviate the burden of monitoring the composition of the SSC. Then, it computes a risk profile, identifying the SSC risk sources and how they can impact the final software to support the identification of the weakest points of the SSC and activate the necessary organizational and technical countermeasures to prevent future SSC attack campaigns.

Keywords: Software supply chain · Software supply chain security · Risk identification · Software security

1 Introduction

The DevOps paradigm has tightly integrated development, delivery, and operations, into the development process, facilitating and speeding up the continuous release of software components [10]. Such a paradigm drove the tight integration of heterogeneous components such as software artifacts (e.g., third-party libraries and binaries), assets (e.g., software repositories and package managers), and personnel that contribute to a software product or that have the opportunity to modify its content (e.g., developers and maintainers). Those elements compose the so-called Software Supply Chain (SSC) [1].

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
A. Vallecillo et al. (Eds.): QUATIC 2022, CCIS 1621, pp. 281–295, 2022.
https://doi.org/10.1007/978-3-031-14179-9_19

In the last years, however, SSC has become hard to maintain and understand, as it needs to be adapted to cope with the evolution of the underlying software and systems, including technological changes (e.g., changes in architectures, operating systems, or library upgrades) [44]. For instance, some parts of the SSC become unnecessary during the evolution of the software and can be removed, or some others (e.g., testing environments) become obsolete and should be upgraded/replaced.

In addition, from a security standpoint, the SSC offers an appealing entry point for attackers that aim to target the final software and its consumers, as witnessed by recent security reports [2,27]. In particular, a successful attack in the software supply chain might go unnoticed for a long period, impacting a large number of companies that use the affected supplier's software. The CodeCov attack [20], for example, exploited a configuration vulnerability in the Docker files of the CodeCov code coverage tool that allowed external attackers to access the source code stored in the repositories of 23000 customers. The difficulty of maintaining and evaluating the security posture of a SSC drove the attention of both the industrial and research community. For this reason, different approaches emerged in recent years. These approaches can be mainly divided into two groups. The first set of methodologies and tools focuses on detecting vulnerabilities in the software code directly in the DevOps pipelines. Notable examples include snyk [31] and slscan [29]. Other solutions, instead, focus on the integrity of software and its dependencies, such as Google SLSA [32], MITRE D3FEND [18], and ReproducibleBuilds [26]. However, we argue that the proposed solutions allow the mitigation of security vulnerabilities but fail to identify their root causes and, thus, to prevent future attack campaigns. Supporting that, the ENISA reports that 66% of attacks targeting the Software Supply Chain come from unknown sources [11]. In order to fill such a gap, this paper presents *Sunset* (Software Supply Chain Risk Identification), a methodology that supports the maintenance and the risk evaluation of Software Supply Chains. First, the methodology allows for the automatic reconnaissance of all the elements of a Software Supply Chain and their dependencies. Then, it supports the identification of all the security risks of the SSC and its components by generating a risk profile that details where threats originate, which path they follow to reach the final software, and their severity.

Structure of the Paper. The rest of the paper is organized as follows: Section 2 introduces the software supply chain along with its vulnerabilities and attacks. Section 3 details the *Sunset* methodology and its architecture. Section 4 discusses the current state-of-the-art for software supply chain security, while Sect. 5 concludes the paper and points out some future extensions of this work.

2 Software Supply Chain

The Software Supply Chain (SSC) contains all the elements, called *assets*, that contribute to the development of a software artifact, namely the final product. SSC assets include structural elements (e.g., code repositories and development

servers), software components (e.g., libraries and executables), and organizational entities (e.g., developers and software maintainers). Inside the SSC, we can distinguish between *supplier assets* and *customer assets*. The former contains all assets not explicitly created or defined for the final product, e.g., an external library or a package manager. The latter comprises assets the final software will interface with once deployed in the production environment.

In a typical DevOps scenario, depicted in Fig. 1, the SSC contributes to the pre-release phase, where the organization selects (plan), implements (code), packs (build), and tests (test) all the elements composing the final software.

Fig. 1. The DevOps workflow.

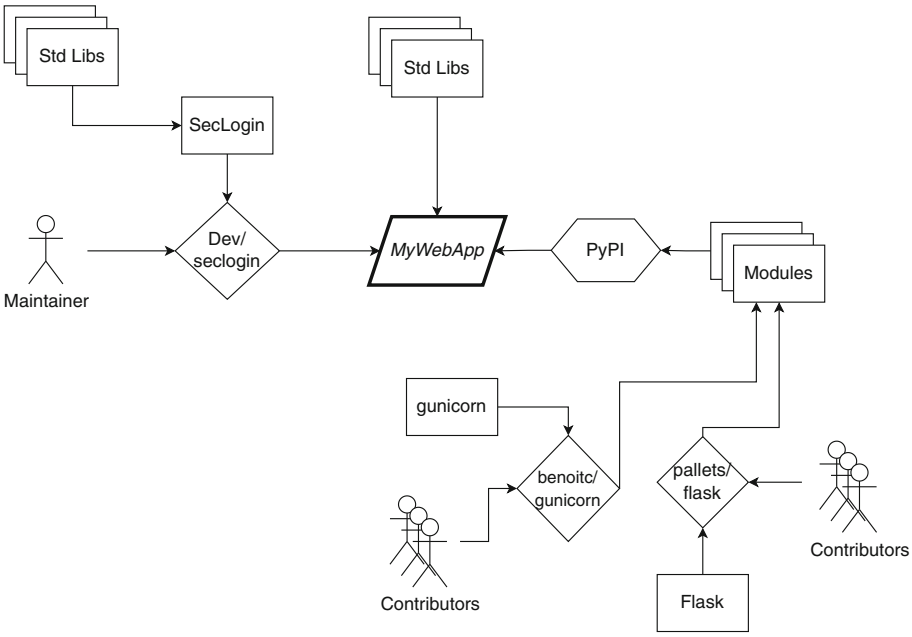

Fig. 2. *MyWebApp* software supply chain.

Figure 2 depicts an example of supplier assets composing the SSC of a Python web app called *MyWebApp*. *MyWebApp* uses a set of standard python libraries (e.g., os and glob) and two external modules (i.e., Flask and Gunicorn) imported

using the PyPI package manager. Both modules are hosted on public GitHub repositories, where maintainers and contributors provide regular updates and new functionalities. Also, *MyWebApp* manually imports another library, called SecLogin. SecLogin is developed and hosted on a GitLab repository by a single maintainer and relies on standard python libraries as well.

2.1 Software Supply Chain Vulnerabilities and Attacks

An *SSC vulnerability* is defined as a security vulnerability affecting an asset that could evolve into an attack once exploited. Such vulnerability may happen during the different stages of software development. In particular, Common Weakness Enumeration [40], i.e., CWE, highlights that 91% of security weaknesses are introduced during design (462 CWEs) and implementation (724 CWEs).

One of the most significant advantages of attack campaigns targeting software supply chains is that their impacts are not limited to the final software. They can also harm assets belonging to more than one SSC and the affected software customers. As a result, this form of attack is more likely to go unnoticed and deliver a higher payout to the attacker [43].

Supporting the importance of the security evaluation of software supply chains, the MITRE ATT&CK framework [39], identified the *supply chain compromise* as an initial access tactic.

Let's consider the SSC example in Fig. 2. If the maintainer introduces a Static Application Security Testing (SAST) tool, during the build phase of the final software, to evaluate *MyWebApp*, the analysis may identify some security vulnerabilities and map them to known CWEs. In our example, the code analyzer detects CWE-20 (Improper Input Validation) [34], CWE-89 (Improper Neutralization of Special Elements used in an SQL Command) [37], and CWE-798 (Use of Hard-coded Credentials) [36].

Thanks to the SAST security report, the developers of *MyWebApp* can patch the source code. However, the developers do not have information regarding the SSC asset that caused the vulnerability or the used attack vectors.

This lack of information is caused by the closed range analysis provided by vulnerability assessment techniques, which aim to just alert developers of vulnerability. Then developers cannot effectively patch the vulnerability by taking actions considering the smallest possible piece of asset which originates the threat.

For instance, the vulnerability assessment tools cannot detect that the credential of one of the contributors of the GitLab repository of the SecLogin module has been compromised using a social engineering attack. Indeed, thanks to this attack vector, attackers can inject malicious code into the module and, thus, the SSC, reaching the final software. This lack of information prevents developers from identifying the compromised repository and adopting a mitigation action (e.g., disconnecting the SecLogin repository) to cope with the risk to the final software.

3 *Sunset*

This section introduces the basics of *Sunset* (Software Supply Chain Risk Identification), a methodology to automatically model the SSC and to identify the risk that assets pose to the final software.

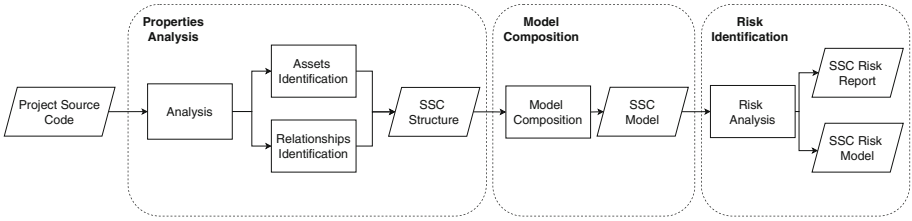

Fig. 3. *Sunset* architecture.

Sunset automatically extracts a model of the Software Supply Chain in terms of assets and dependencies, given only the source code of the final software. Then, it extracts the cybersecurity risk of each asset and computes how it can impact the security of the final software product.

The workflow of the methodology, depicted in Fig. 3, consists of three phases: *(I) The identification* of assets and the *extraction* of their functional properties. *(II) The modeling* of the assets as well as the relationships linking them. *(III) The identification of the risk* of specific assets and the computation of the risk propagation to the final software.

3.1 Property Analysis

Asset Identification. *Sunset* identifies assets based on four distinct categories, namely *software artifacts*, code *holders*, *distribution networks*, and *actors*.

- **Software Artifact.** It represents any kind of software included or developed in the SSC. *Sunset* further discerns between *(i)* compiled software (e.g., binaries and pre-compiled libraries) and *(ii)* source code artifacts.
- **Holder.** This type of asset is responsible for storing and maintaining software artifacts. A Holder can be further categorized in:
  - Local Storage. It represents storage solutions that are not connected with any management system. This category includes, for instance, a local folder containing software artifacts.
  - Version Control System (VCS). A VCS allows managing software artifacts using a management system that supports code control features like versioning and tracking. The VCS kind of holder can be classified in *remote* and *local* depending on its location.

– **Distribution Network.** Also known as package managers, it includes services and systems that allow the categorization, search, and distribution of software artifacts. Typical distribution networks include Maven and PyPI, which support the distribution of Java and Python libraries, respectively. The majority of open and closed source projects take advantage of distribution networks to import software dependencies [13].
– **Actor.** Actors represent humans involved in the software supply chain. An actor can be classified based on its privileges on the asset it is connected to, i.e.:
  • Maintainer. It has full access to the asset who is connected to. Moreover, it is able to set privileges for other actors connected to the same asset.
  • Contributor. It has restricted access to the asset who is connected to.

The identification of assets happens in the first stage of the workflow depicted in Fig. 4. In particular, the methodology analyses the project files and their content to detect the assets composing the software supply chain.

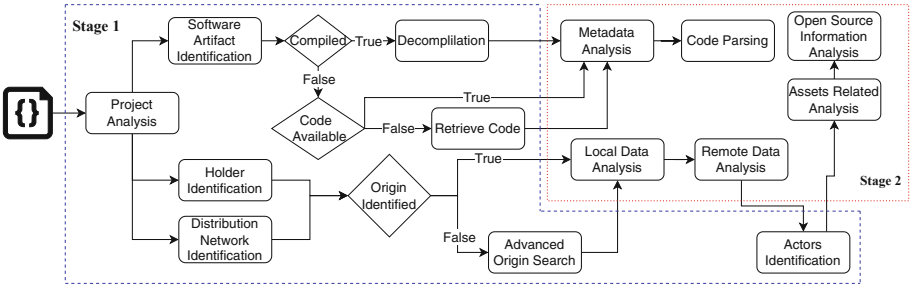

Fig. 4. Property extraction workflow.

Assets belonging to the software artifact category are identified starting from the entry file of the project. The methodology recursively identifies software artifacts via parsing the code files in the project. When *Sunset* is not able to match a software artifact with the parsed source code, it tries to find a corresponding artifact online (e.g., by searching for publicly available implementation of the software artifact). If the methodology identifies multiple artifacts, it selects the most exhaustive implementation (considering lines of code and last update time, if available). Nevertheless, when this event happens, *Sunset* collects the differences between artifact versions as evidence of possible attack vectors (e.g., typosquatting [42], i.e., trick users into downloading a malicious package by squatting the name of a popular package).

The methodology identifies holders through the analysis of project files. VCSs use the local file system to deal with code versioning (e.g., indexing files and configuration files), then *Sunset* detects this class of holder through these fingerprint files. In the case there is no local fingerprint, the methodology takes advantage of software artifacts source code to explore available public repositories in order to

couple software artifacts to a VCS. When the methodology fails to associate software artifacts to a VCS, it creates a local storage holder to contain the software artifacts.

Programming languages of software artifacts defines the distribution networks involved in the SSC, i.e., a Python software artifact rely on PyPI, while a rust program on Cargo. *Sunset* identifies also private distribution networks searching for specific configurations in software artifact source code or configuration files.

The methodology identifies the actors during the analysis of holders and distribution networks. Indeed, actors interact with both of these categories of assets. An actor interfaces with an asset through a virtual identity that differs from its physical person. The methodology refers to this virtual identity to model the actor asset.

Properties Extraction. The methodology extracts a different set of properties for each category of assets. The properties of interest are divided in two categories, namely:

- *Structural properties*, which are oriented to the understanding of the structural composition of the asset. These properties provide information about the structure and the quality of the asset in terms of usage and involvement in the software supply chain.
- *Security properties*, which concern the security posture of the asset. They provide information regarding possible flaws and entry points.

Each category contains different groups of properties, as detailed in Table 1.

Sunset extracts properties from assets during their identification. Depending on the asset category, the methodology extracts the proper groups of properties. The single properties of each group receive a quantitative evaluation, depending on their characteristics and the availability of plugins to support the extraction, e.g., a static code analyzer for security properties of software artifacts.

Similarly to the identification process, Sunset adopts a strategy to extract properties based on the asset category. Stage 2 of the workflow (Fig. 4) depicts the corresponding extraction workflow.

For software artifact assets, properties extraction consists of analyzing the metadata, the source code (if available), and the result of SAST tools. In the case of a compiled software artifact, instead, *Sunset* analyses the decompiled code relying on state-of-the-art decompilation tools [17,23,25].

Assets belonging to the Holder and Distribution Network categories are analyzed by considering *(i)* the metadata located in the project (e.g., indexing files, mirror files), *(ii)* the remote information (e.g., remote branches, pull requests, versioning information), and *(iii)* the existence of known security issues on publicly available vulnerability databases.

For the actor assets, the methodology takes advantage of the information provided by the assets from which the actor has been obtained (e.g., contribution to the repository). The information gathered through the asset where the

Table 1. Properties categories and groups divided per asset category.

	Structural	Security
Software artifact	Conditional statements	Buffers validation
	Functions	Input sanitization
	Required user interactions	Insecure patterns
	Read and write operations	
Holder	Commits	Security policies
	Pull requests	Community standards
	Issues	Known security issues
	Workflows	
Distribution network	Mirrors	Known security issues
	Packages required	
Actor	Homepage	OSINT results
	Overall contributions	Known malicious actions
	Public repositories	
	Forks	

actor contributes is integrated with the analysis of open-source information [7]. Different actors can be connected to a single physical person. Analyzing the links bounding them to a physical person allows for a better understanding of their involvement in the software supply chain. Thanks to this understanding, it would be possible to capture information on potential threat actors. The identification of these links involves the use of state-of-the-art tools for the analysis of personas [21,30].

3.2 Model Composition

The model composition phase (Fig. 3) allows for the building of a structured representation of the software supply chain.

The generation of the model organizes the assets and their inter-dependencies using a direct graph structure, where nodes represent assets and edges detail their relations.

The final software is the central node of the model. This particular node has only entering edges. The edges entering the node connect the final software to the subgraphs containing the assets identified during the Asset Identification phase. Figure 5 depicts the set of possible relationships between two assets.

Figure 6 depicts the model generated by the analysis and modeling phase of the SSC of *MyWebApp* of Fig. 2.

The SSC model also supports the graphical plotting and manipulation using state-of-the-art tools, e.g., [16]. The visual representation allows a high-level overview of the SSC that can support maintenance tasks and preliminary security

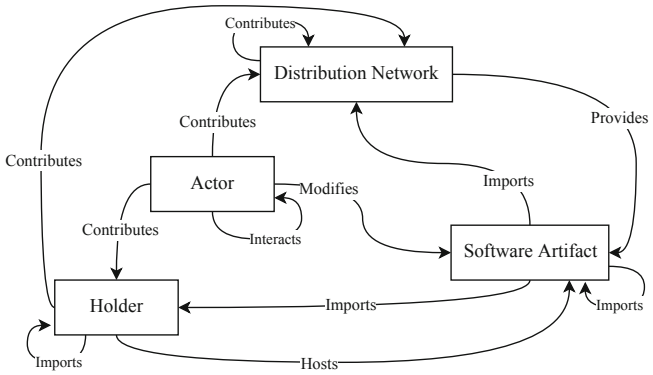

Fig. 5. Possible relationships among SSC asset categories.

assessments. For instance, the centrality of nodes [4] can be used to understand which nodes have more influence w.r.t. the final software.

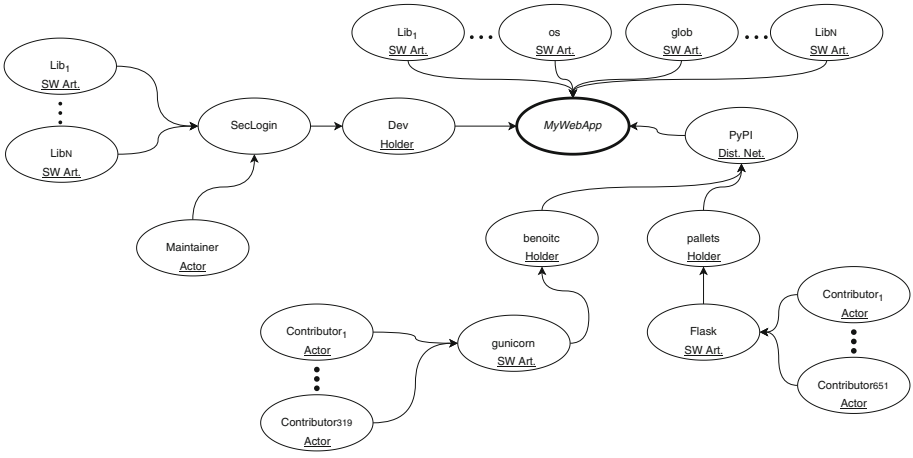

Fig. 6. Example of model representation.

3.3 Risk Identification

The *Sunset* methodology takes advantage of SSC model obtained in the model composition phase to carry on risk identification. Risk identification concerns searching and analyzing risk sources in the software supply chain. This phase considers the risk generated on the single assets and its propagation through the software supply chain. Figure 7 depicts the workflow for the risk identification phase.

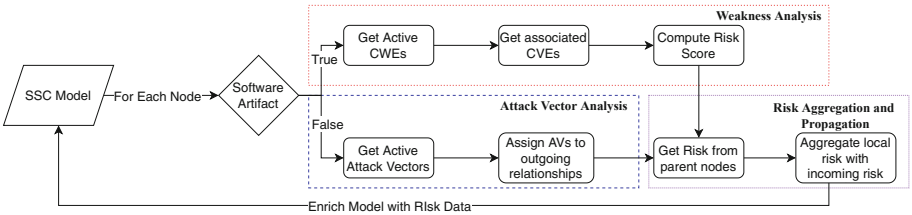

Fig. 7. Risk analysis module workflow.

Sunset explores the model graph with a breadth-first search algorithm [6]. The exploration starts from the border of the graph to represent how the outer assets' risk impacts inner nodes and reaches the final software.

The methodology defines two types of analysis depending on the category of the asset:

- Weakness analysis, for software artifacts.
- Attack Vector analysis, for holders, distribution networks, and actors.

Sunset relies on a Knowledge Base for the evaluation of weaknesses and attack vectors. The knowledge base maps assets' structural and security properties with either a CWE or an attack vector (AV). *Sunset* considers all CWEs linked to the pre-release phase, grouped in the CWE View 699 [38] and the set of attack vectors listed by ENISA [11].

In detail, the knowledge base contains a list of first-order logic statements evaluating structural properties (P), security properties (S), and the presence of specific attack vectors (AV). Each property can be compared with a threshold value (T), joint or disjoint with other properties or predicates, and evaluated in its presence or absence ($\neg$ operator).

For example, the first expression in Listing 1.1 states that attack vector AV_x is enabled when both properties P_1 and P_2 are greater than their respective critical values C_1 and C_2 and when the attack vector AV_y is active. The second statement in Listing 1.1 details how CWE_z is enabled either when properties P_1 and P_2 are greater than critical values C_1 and C_2 or when the attack vector AV_x is active. More detailed examples are presented in Listings 1.2 and 1.4.

1. $\mathrm{AV}_x \implies (P_1 > C_1) \wedge (P_2 > C_2) \wedge \mathrm{AV}_y$

2. $\mathrm{CWE}_z \implies (P_1 > C_1) \wedge (P_2 > C_2) \vee \mathrm{AV}_x$

Listing 1.1. Mapping rules of *Sunset* knowledge base.

Weakness Analysis. The weakness analysis leverages the CWE database, the CVSS scoring system [12], and the properties extracted for the software artifacts. *Sunset* considers all CWEs linked to the pre-release phase, grouped in the CWE View 699 [38].

During the analysis of a software artifact asset, the methodology identifies active CWEs. A CWE is active on an asset if it is included (or can be derived) from a security property of the asset or from an attack vector. By verifying asset properties and the attack vectors inherited from the connected edges, *Sunset* provides the list of active CWEs of the asset.

For example, the presence of conditional statements (P_{cond}) but the lack of variable sanitization (S_{san}) triggers the rule on CWE-478 [35] (lack of default condition in switch statements) detailed in Listing 1.2.

$$\text{CWE-478} \implies P_{\texttt{cond}} \wedge \neg S_{\texttt{san}}$$

Listing 1.2. KB rule for CWE-478.

After the evaluation of active CWEs, *Sunset* proceeds with the definition of the overall risk associated with the asset. For the computation, the methodology retrieves all the Common Vulnerabilities Exposures (CVE) [33] grouped w.r.t. a given CWE, i.e., the vulnerabilities linked to each CWE that are available on public databases. For each CVE, *Sunset* extracts the corresponding CVSS score [12] and uses it to compute the CVSS risk value of the asset.

In detail, we define G_x as the group of active CWEs on an asset X **(1)**; for each CWE in G_x, the methodology gathers the corresponding list of CVSS vectors (S_i), one for each CVE. Each S_i contains L different metrics M **(2)**. The risk score of the CWE j (R_{CWE_j}) is a new vector where each metric K_i is the mean value of all the same metrics of each CVSS score contained in CWE j **(3)**. The overall score R_x of the asset is the max value among the set of R_{CWE} **(4)**.

(1) $G_x = \{\text{CWE}_1, ..., \text{CWE}_T\}$
(2) $\text{CWE}_j = \{\text{S}_1, ..., \text{S}_N\}$ where $S_i = \{M_{i1}, ..., M_{iL}\}$
(3) $R_{\text{CWE}_j} = \{K_1, ..., K_L\}$ where $K_i = \frac{M_{1i}+...+M_{Ni}}{N}$
(4) $R_x = \max\{R_{\text{CWE}_1}, ..., R_{\text{CWE}_T}\}$

Listing 1.3. Equations for computing the risk score of an asset.

Attack Vector Analysis. Attack vector analysis follows the same concept as weaknesses analysis. In detail, *Sunset* identifies a set of active AVs insisting on an asset X if and only if the functional and security properties allow their presence. For the evaluation, the methodology exploits the rules defined in the knowledge base.

For example, the asset X is susceptible to manipulation attacks (AV_{man}) if the asset X inherits the attack vector social engineering (AV_{se}) from a linked asset and the security property weak password (S_{wp}) is below the threshold T_l, according to the rule in Listing 1.4.

$$\mathrm{AV}_{\mathrm{man}} \implies \mathrm{AV}_{\mathrm{se}} \wedge (S_{\mathrm{wp}} < T_l)$$

Listing 1.4. KB rule for the manipulation attack vector.

Risk Aggregation and Propagation. Attack vectors and risk evaluations propagate in the SSC model according to the interconnections among the different assets. Weaknesses and attack vectors flow from the boundary of the SSC model toward the final software. In this sense, a relationship between two assets has two goals:

I. Carry attack vectors useful for the weakness analysis.
II. Transport the risk value obtained on connected assets.

Hence, the total risk on an asset consists of aggregating the risk value generated on the asset with the risk values inherited from inbound relationships.

In detail, *Sunset* starts the exploration of the SSC model from the border of the final software SSC using a breadth-first search algorithm.

For each inbound connection, the methodology adds the risk score and the attack vectors inherited from the parent node. Then it iterates the process for each node until it finds the final software. On each step, *Sunset* updates the weakness analysis and the AV analysis to match the new conditions.

Looking back at example in Fig. 2, the sources of risk of the *MyWebApp* are the relationships incoming from the *Dev/seclogin* holder, the *Std Libs* software artifacts, and the *PyPI* distribution network. Suppose that the methodology reports a social engineering attack vector on the maintainer of *Dev/seclogin*. In that case, such AV will be propagated on the holder and, consequently, in the software artifact of the module and the final software.

4 Related Work

Both the industrial and scientific communities proposed several solutions to increase the security of software [9]. Most of the activities focused mainly on vulnerability analysis and software integrity.

Tools like the OWASP Dependency-Check [24], snyk [31], slscan [29], and shhgit [8] provide the developer with detailed vulnerability reports of vulnerability patterns and insecure dependencies. Also, the scientific community provided several solutions to detect and mitigate software vulnerabilities, such as [14,19].

Another field of activities was devoted to ensuring the integrity assurance of open-source software. Such works aim to prevent unauthorized modifications/-tampering of the software during the development pipeline.

Two of the most recent industry proposal are SLSA [32] and Reproducible builds [26]. Supply chain Levels for Software Artifacts (SLSA) is an end-to-end framework to guarantee the integrity of dependencies all along the development process. Through SLSA certifications, developers obtain information on

the integrity assurance a given artifact can offer. However, the certification process requires an extended interaction with the developers and hardly copes with the level of automation needed for the DevOps paradigm.

Reproducible builds [26], instead, is a collection of software development processes that aims to standardize the build and compilation process in terms of configurations and requisites. This approach enables maintainers to detect if an attacker has compromised the building process by comparing the assets generated during the compilation process.

Hence, Reproducible Builds focuses on the integrity compromise happening in the build step. Weaknesses inserted in the software by mistake or intentionally are then considered a trusted part of the build. On the same approach, the authors of LastPyMile [41] proposed a methodology to detect the differences between build artifacts of software packages and the respective source code repository.

The aforementioned approaches might detect vulnerable dependencies and insecure code and contribute to software packages' integrity. Still, they hardly cope with the root cause of the problem, the attack vector, and the affected asset of the SSC, failing to mitigate the risk of new attack campaigns. Such lack of control is particularly disruptive in complex software supply chains containing thousands of assets, thereby limiting the benefits of adopting VA and integrity solutions. *Sunset* is one of the first attempts to mitigate such pain for developers and SSC maintainers.

5 Conclusion and Future Work

In this paper, we introduced *Sunset*, a new methodology to model software supply chains and evaluate their security risk and the detail of the single asset. *Sunset* is not intended to substitute traditional VA and PT procedures or risk management activities. The methodology, instead, aims to alleviate the burden of maintaining a secure and updated SSC by providing a means to *(i)* evaluate the risk of each asset and how it will influence the security posture of the final software and *(ii)* identifies the sources of risk to prioritize mitigation activities. Also, the evaluation of *Sunset* can be performed offline without impacting the performance of the development process.

In future works, we plan to extend the methodology to cope with the current limitations. First, we will enrich the type of assets and properties that can be modeled with *Sunset* to support complex scenarios. Then, we will provide an open-source prototype implementation of the methodology to test its applicability and efficacy in tracing security vulnerabilities, and sources of risk in real-world scenarios like mobile [3,22], CPS [15] and IoT [5], whose development pipeline and threat model are well-known to our research group. The implementation will exploit both state-of-the-art tools (e.g., slscan [29] and COSMO [28] to extract security properties) and ad-hoc heuristics (e.g., a module for detecting GitHub software artifacts).

References

1. Alberts, C.J., Dorofee, A.J., Creel, R., Ellison, R.J., Woody, C.: A systemic approach for assessing software supply-chain risk. In: 2011 44th Hawaii International Conference on System Sciences, Kauai, HI, pp. 1–8, January 2011. https://doi.org/10.1109/HICSS.2011.36
2. Argon: 2021 software supply chain security report (2021). https://info.aquasec.com/argon-supply-chain-attacks-study
3. Armando, A., Costa, G., Merlo, A., Verderame, L.: Enabling BYOD through secure meta-market. In: Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks, WiSec 2014, pp. 219–230. Association for Computing Machinery, New York (2014). https://doi.org/10.1145/2627393.2627410
4. Barabási, A.L.: Network Science. Cambridge University Press (2016). http://networksciencebook.com/
5. Caputo, D., Verderame, L., Ranieri, A., Merlo, A., Caviglione, L.: Fine-hearing google home: why silence will not protect your privacy. J. Wireless Mob. Netw. Ubiquit. Comput. Dependable Appl., 35–53 (2020). https://doi.org/10.22667/JOWUA.2020.03.31.035
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press, Cambridge (2017)
7. Cumming, A.: Open Source Intelligence (OSINT): Issues for Congress (2007)
8. Darkport Technologies Limited: shhgit. https://github.com/eth0izzle/shhgit
9. Dowd, M., McDonald, J., Schuh, J.: The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities. Pearson Education (2006)
10. Ebert, C., Gallardo, G., Hernantes, J., Serrano, N.: DevOps. IEEE Softw. (3), 94–100 (2016). https://doi.org/10.1109/MS.2016.68
11. European Union Agency for Cybersecurity: ENISA Threat Landscape for Supply Chain Attacks. Publications Office, LU (2021). https://data.europa.eu/doi/10.2824/168593
12. FIRST.ORG Inc.: CVSS. https://www.first.org/cvss/
13. Flynn, C.: PyPI Stats. https://pypistats.org/packages/__all__
14. Ghaffarian, S.M., Shahriari, H.R.: Software vulnerability analysis and discovery using machine-learning and data-mining techniques: a survey. ACM Comput. Surv. (4) (2017). https://doi.org/10.1145/3092566
15. Gobbo, N., Merlo, A., Migliardi, M.: A denial of service attack to GSM networks via attach procedure. In: Cuzzocrea, A., Kittl, C., Simos, D.E., Weippl, E., Xu, L. (eds.) CD-ARES 2013. LNCS, vol. 8128, pp. 361–376. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40588-4_25
16. Graphviz Authors: Graphviz. https://graphviz.org/
17. Hex-Rays: Ida Decompiler. https://hex-rays.com/decompiler/
18. Kaloroumakis, P.E., Smith, M.J.: Toward a knowledge graph of cybersecurity countermeasures (2021)
19. Liu, B., Shi, L., Cai, Z., Li, M.: Software vulnerability discovery techniques: a survey. In: 2012 Fourth International Conference on Multimedia Information Networking and Security, pp. 152–156 (2012). https://doi.org/10.1109/MINES.2012.202
20. Jackson, M.: Codecov supply chain breach - explained step by step. https://blog.gitguardian.com/codecov-supply-chain-breach/
21. Maltego Technologies: Maltego. https://www.maltego.com/

22. Migliardi, M., Merlo, A.: Improving energy efficiency in distributed intrusion detection systems. J. High Speed Netw. **3**, 251–264 (2013). https://doi.org/10.3233/JHS-130476
23. National Security Agency: Ghidra. https://ghidra-sre.org/
24. OWASP Foundation Inc.: OWASP dependency-check. https://owasp.org/www-project-dependency-check/
25. radareorg: Radare2. https://rada.re/
26. ReproducibleBuilds: Reproduciblebuilds. https://reproducible-builds.org/
27. Revenera: The 2022 state of the software supply chain report (2022). https://info.revenera.com/SCA-RPT-OSS-License-Compliance-2022/
28. Romdhana, A., Ceccato, M., Georgiu, G.C., Merlo, A., Tonella, P.: COSMO: code coverage made easier for android. In: 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST), pp. 417–423 (2021). https://doi.org/10.1109/ICST49551.2021.00053
29. ShiftLeftSecurity: SLScan. https://slscan.io/
30. Shodan: Shodan. https://www.shodan.io/
31. Snyk Limited: Snyk open source. https://snyk.io/
32. The Linux Foundation: SLSA. https://slsa.dev/
33. The MITRE Corporation: CVE. https://cve.mitre.org/
34. The MITRE Corporation: CWE-20: improper input validation. https://cwe.mitre.org/data/definitions/20.html
35. The MITRE Corporation: CWE-478: missing default case in switch statement. https://cwe.mitre.org/data/definitions/478.html
36. The MITRE Corporation: CWE-798: use of hard-coded credentials. https://cwe.mitre.org/data/definitions/798.html
37. The MITRE Corporation: CWE-89: improper neutralization of special elements used in an SQL command ('SQL Injection'). https://cwe.mitre.org/data/definitions/89.html
38. The MITRE Corporation: CWE VIEW 699: software development. https://cwe.mitre.org/data/definitions/699.html
39. The MITRE Corporation: MITRE ATT&CK. https://attack.mitre.org/
40. The MITRE Corporation: MITRE Common Weakness Enumeration. https://cwe.mitre.org/
41. Vu, D.L., Massacci, F., Pashchenko, I., Plate, H., Sabetta, A.: LastPyMile: identifying the discrepancy between sources and packages. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens Greece, pp. 780–792, August 2021. https://doi.org/10.1145/3468264.3468592
42. Vu, D.L., Pashchenko, I., Massacci, F., Plate, H., Sabetta, A.: Typosquatting and combosquatting attacks on the python ecosystem. In: 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS PW), pp. 509–514 (2020). https://doi.org/10.1109/EuroSPW51379.2020.00074
43. Yan, D., Niu, Y., Liu, K., Liu, Z., Liu, Z., Bissyandé, T.F.: Estimating the attack surface from residual vulnerabilities in open source software supply chain. In: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), pp. 493–502 (2021). https://doi.org/10.1109/QRS54544.2021.00060
44. Zampetti, F., Geremia, S., Bavota, G., Di Penta, M.: CI/CD pipelines evolution and restructuring: a qualitative and quantitative study. In: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 471–482 (2021). https://doi.org/10.1109/ICSME52107.2021.00048