jamk

Software bills of materials in production environments

Michael Herman

Bachelor's thesis
February 2024
Information and Communication Technology

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

Herman, Michael

Software bills of materials in production environments

Jyväskylä: Jamk University of Applied Sciences, February 2024, 65 pages

Degree Programme in Information and Communication Technology. Bachelor's thesis.

Permission for open-access publication: Yes

Language of publication: English

Abstract

The practice of using open-source components across different software ecosystems is widespread. This makes is easier for attackers to exploit vulnerabilities without crafting specific attacks. A question then emerges about how to identify and manage hidden dependencies embedded in larger applications. Legacy systems further complicate these issues. One proposed solution is known as a Software Bill of Materials (SBOM). SBOMs are nested inventories of software packages and dependencies. Advocates argue SBOMs promote greater transparency in the software supply chain.

This is desirable because transparency facilitates more efficient and monitoring practices. The implication of this claim is that SBOM-based platforms are more efficient at accurately identifying and remediating vulnerabilities than methods such as active scanning. The study was commissioned by JYVSECTEC to test the feasibility of using SBOM-based infrastructure management in their internal networks. Most of the literature on the subject focuses on the development side. There is surprisingly little literature which examines the relevance of SBOMs from the DevSecOps perspective. This contribution fills that gap.

Three questions organized around the themes of transparency, accuracy and efficiency were formulated to examine the claims above. An additional question concerning the potential use of SBOM-based infrastructure management platforms in offensive contexts such as red team exercises was also asked because of its relevance for JYVSECTEC. The core aim was to test the hypothesis that SBOM-based platforms enhance vulnerability management and security across DevSecOps workflows. An environment was required in which the hypothesis could be tested. Dependency-Track is a popular SBOM-based infrastructure management tool. It was selected due to its maturity relative to the alternatives. Testing occurred in two phases. The first involved setting up a test environment which established Dependency-Track's baseline capabilities. The second phase involved testing these capabilities in a production environment.

The analysis of the production environment identified a range of problems associated with SBOM-based infrastructure management. Accuracy and depth of information was found to be inconsistent. This calls into question claims that SBOMs enhance supply chain transparency. Accuracy problems required manual intervention because automated solutions were in many cases not possible. It is unclear if the platform provides any efficiency gains over existing methods of infrastructure management. The findings suggest SBOMs exhibit limited applicability for DevSecOps.

Keywords

Cybersecurity, SBOM, software bill of materials, automation, virtualization, DevSecOps, DevOps

Contents

Acronyms **5**
1 Introduction **6**
1.1 Problem statement **6**
1.1.1 Background 6
1.1.2 Implications 6
1.1.3 Solutions 7
1.2 Client **8**
1.3 Questions **8**
1.4 Hypothesis **9**
1.5 Methods **10**
1.6 Structure **10**
2 Challenges **12**
2.1 Background **12**
2.2 Concepts **13**
2.2.1 Dependency 13
2.2.2 Vulnerability 14
2.2.3 Components 14
2.2.4 Compliance 15
2.3 Discussion **16**
2.3.1 Active scanning 16
2.3.2 Software Bill of Materials 16
2.3.3 Similarities 17
2.3.4 Differences 17
2.3.5 Temporality 18
2.3.6 Transparency 19
2.3.7 Requirements 20
2.3.8 Summary 21
3 Framework **23**
3.1 Research design **23**
3.1.1 Approach 23
3.1.2 Environment 23
3.2 Variables **24**
3.3 Baseline **25**
3.4 Data **25**

3.4.1 Sources 25
3.4.2 Targets 26
3.5 Comparative analysis 27
3.6 Limitations 27
3.7 Summary 28
4 Methods 29
4.1 Background 29
4.2 Pipelines 30
4.2.1 Image scraping 30
4.2.2 Distro pipeline 31
4.2.3 Container pipeline 31
4.3 Dependency-Track 32
4.3.1 Overview 32
4.3.2 Core Capabilities 33
4.3.3 Known vulnerability analysis 33
4.3.4 Component identification 34
4.3.5 Component analysis 35
4.3.6 Impact analysis 36
4.3.7 Analysis states 37
4.3.8 Policies 38
4.3.9 Alerts 38
4.4 Streamlining 40
4.4.1 Overview 40
4.4.2 get-outdated.py 40
4.4.3 generate-vul-report.py 41
4.4.4 pocsploit.sh 42
4.5 Summary 43
5 Analysis 44
5.1 Overview 44
5.2 Production environment 44
5.2.1 Depth of data 44
5.2.2 Transparency 46
5.2.3 Vulnerability management 46
5.2.4 Component management 47
5.2.5 Identifiers 49
5.2.6 Alerts 50

5.3 Additional observations 51
5.4 Summary 51
6 Conclusion 53
6.1 Findings 53
6.1.1 Overview 53
6.1.2 The good 53
6.1.3 The bad 53
6.1.4 The ugly 54
6.2 Results 55
6.2.1 Questions 55
6.2.2 Hypothesis 55
6.3 Limitations 56
6.4 Future development 56
7 Sources 57
Appendices 65
Appendix 1. Bolt workflow 65

Figures

Figure 1. Environment setup phases 30
Figure 2. Dashboard view 33
Figure 3. Component search 35
Figure 4. Components overview 36
Figure 5. Component analysis results 37
Figure 6. Teams alerts 39
Figure 7. get-outdated output 41
Figure 8. generate-vuln-report.py output 41
Figure 9. pocsploit output 42
Figure 10. SBOM data structure 45
Figure 11. Project view 46
Figure 12. Vulnerable components 48
Figure 13. First-order Nginx dependencies without sub-components 49
Figure 14. CVE-2023-7028 CPE mismatch in Dependency-Track 50
Figure 15. CVE-2023-7028 NVD CPE entry 50

Tables

Table 1. Dependency types 13
Table 2. Variables 24
Table 3. Data sources 25
Table 4. Analysis types 26
Table 5. Automation tools 32
Table 6. Dependency-Track vulnerability intelligence sources 34
Table 7. Analysis states 37
Table 8. Research outcomes 55

Acronyms

API	Application Programming Interface
CISA	Cybersecurity and Infrastructure Agency
CI/CD	Continuous Integration / Continuous Development
CVE	Common Vulnerabilities and Exposures
CPE	Common Platform Enumeration
CVSS	Common Vulnerability Scoring System
CWE	Critical Weakness Enumeration
EPSS	Exploit Prediction Scoring System
JSON	JavaScript Object Notation
NTIA	National Telecommunications and Information Administration
NVD	National Vulnerability Database
PoC	Proof of Concept
pURL	Package URL
SBOM	Software Bill of Materials
SWID	Software Identification
UI	User Interface
VLE	Virtual Learning Environment

1 Introduction

1.1 Problem statement

1.1.1 Background

Recent years have seen a shift from proprietary to open-source solutions in the software development cycle (Perez, 2023; Synopsys, 2022). Open-source software components have come to be viewed as broadly more cost-effective and of a higher quality (Haff, 2020). Their generic nature and widespread utility has resulted in their inclusion in an extensive range of areas. These include operating systems, security tools, CI/CD pipelines and others. A consequence of this trend has been an increase in the number of software components containing exploitable vulnerabilities (Statista, 2024).

The ubiquity of open-source components poses a considerable challenge for network security. The practice of using the same open-source components across different software ecosystems without modifications is widespread (Wirth, 2022). This makes it easy for attackers to exploit vulnerabilities without having to craft specific attacks for each software. A question then emerges concerning how to identify hidden dependencies embedded in larger applications. A related problem is the difficulty associated with managing the vulnerabilities and assessing the security-risks associated with these components. Regulatory and licensing compliance is also rendered more complicated (Ajienka et al., 2017; Carmody 2021).

1.1.2 Implications

This development poses a multitude of challenges across a range of IT infrastructure management domains. Security vulnerabilities, software dependencies and the need for efficient monitoring and maintenance stand out as critical areas of concern (Barclay, et al., 2019). The task of identifying security vulnerabilities within expansive software codebases is intrinsically complex. A rapidly evolving threat landscape exacerbates this problem. Timely patch management can be particularly burdensome within large and intricate systems (Eggers et al., 2022).

This raises the specter of dangerous vulnerabilities potentially remaining unaddressed. The rise of software supply-chain attacks introduces a new dimension of risk. Attackers are more than

capable of exploiting vulnerabilities in third-party and open-source components. Mitigating unknown vulnerabilities before they are weaponized by malicious actors is therefore necessary (Martínez et al., 2021). Legacy systems compound the issue with a lack of support and updates which could otherwise address existing vulnerabilities (Nguyen et al., 2023).

A complex web of third-party libraries and components underpin all modern applications. This complexity hinders the tracking and management of dependencies. It can also lead to compatibility problems between different versions. Managing the lifecycle of dependencies becomes increasingly difficult as ecosystems evolve. This is particularly vexing when dealing with outdated or abandoned components (Chichon, 2021). The vast network of transitive dependencies compounds these challenges. The ripple-effect of complexity makes vulnerability and dependency management a daunting task.

1.1.3 Solutions

Efficient monitoring and maintenance are essential for sustaining a robust IT environment. But these come with their own set of obstacles. Comprehensive monitoring and maintenance practices are essential as IT infrastructures scale and diversify (Eggers, 2022). Managing the plethora of devices, servers, applications and services demands real-time monitoring. This is vital if anomalies and security breaches are to be identified promptly. Sophisticated tools and processes are required. A long-term understanding of the software supply chain is also necessary. This comes with its own set of challenges.

Striking a balance between resource allocation for monitoring and maintenance against other operational needs is another challenge. The role of automation is pivotal in streamlining maintenance efforts (Stoddard et al., 2023). This comes with an important caveat: its implementation must be unerringly meticulous to ensure accuracy and reliability. Adhering to regulatory compliance requirements adds an additional layer of complexity (Carmody et al., 2021). This too involves continuous monitoring, meticulous documentation and comprehensive reporting.

Navigating these multifaceted challenges requires a concerted effort. Technical solutions, well-defined processes and active monitoring, defense and remediation strategies are necessary. The emergence of tools such as the *Software Bill of Materials* (*SBOM*) has the potential to alleviate at least some of these challenges by providing enhanced visibility into software components and

dependencies (Arora et al., 2022, 2022a; Eggers et al. 2022). This can facilitate more effective vulnerability management and offer support for efficient monitoring and maintenance practices. The benefit is an increase in the overall resilience of IT infrastructure in the face of ever-evolving challenges. The objective of this thesis is to explore these themes by testing the viability of SBOMs as an infrastructure management tool.

1.2 Client

JYVSECTEC is a part of the Jyväskylä University of Applied Sciences. It operates under the auspices of the IT Institute. It also engages in commercial activities outside the university environment. The core of these activities is in the field of cyber-security education. This comes primarily in the provision of cyber-security exercises. The exercises target cyber-security professionals working in the Finnish public sector. Exercises take place in environments built by JYVSECTEC according to criteria specified by the client.

The scope of JYVSECTEC's remit adds an interesting dimension to the themes explored in the literature. The thesis conceptualizes software dependencies and vulnerabilities according to two diametrically opposing perspectives. The first concerns problems traditionally associated with managing large IT ecosystems. These are discussed at length in Section 2 and lie at the heart of theoretical debates on the subject.

The second perspective is of direct relevance to JYVSECTEC's business model. A key claim promoted by advovates of the SBOM model concerns its ability to introduce transparance to hitherto opaque software supply chains. The challenge is not in understanding the problem, however. It is in crafting real-world solutions. The thesis examines if SBOM-based infrastructure management strategies can be implemented in a production environment as a practical tool for DevSecOps. It also explores if they can be used in an offensive capacity in red-teaming and penetration-testing scenarios.

1.3 Questions

The above perspectives require a focused exploration of the efficacy of SBOM-based management systems. Research questions are designed to uncover meaningful insights into the management of

security vulnerabilities and software dependencies. Three questions are associated with this task. A fourth question explores the potential for SBOMs in an offensive context.

1. Do SBOMs increase transparency in software supply-chains?
2. Do SBOM-based management platforms perform more efficient identification and remediation of vulnerabilities than active scanning?
3. Do SBOM-based management platforms improve the efficiency and accuracy of workflows in vulnerability identification?
4. Can SBOMs be leveraged for use in red team exercises?

These questions have direct bearing on the practical and theoretical contributions of the thesis. They are organized around three interconnected concepts: *transparency*, *efficiency* and *accuracy*. They are interconnected because a transparent network can by definition be managed accurately. Vulnerabilities can be managed efficiently if they are identified accurately. The opposite is also true: Non-transparent networks imply management processes will be inaccurate and inefficient.

An additional proposition is that SBOM-based platforms cannot only be used to resolve the abovementioned issues; they can also be used to *leverage* them. This duality is evident in the proposed use-case. Penetration testing and red teaming require advanced and up-to-date knowledge of the threat landscape. This information is of course available widely online. But it is neither easy nor convenient to access in a targeted manner. The research speaks to the challenges associated with integrating this informational smorgasbord into a typical offensive-security workflow.

1.4 Hypothesis

The research is hypothesis driven. The hypothesis is that *SBOM-based platforms enhance vulnerability management and security across DevSecOps workflows*. It has been expressly formulated to serve as the distilled essence of the project. It is designed to be relevant across the spectrum of proposed research questions. It is left intentionally broad to afford the research the necessary flexibility to explore the topic as conceptualized. It is readily justifiable and provides the space to explore the full range of potential results. This enhances the validity and comprehensiveness of the research.

The hypothesis makes three assumptions:

- SBOM-based infrastructure management is more effective in identifying and managing vulnerabilities than default management tools
- SBOMs contribute to more effective dependency management
- SBOM-based management platforms such as Dependency-Track outperform traditional methods of vulnerability and dependency management

1.5 Methods

The hypothesis is tested using Docker images and virtual machines hosted in JAMK's Virtual Learning Environment (VLE). Data extracted from these resources have been converted into SBOM-formatted files. These files were then sent to an SBOM-based infrastructure management platform called *Dependency-Track.* It is here where the hypothesis and its core assumptions are analyzed.

This process requires a combination of custom scripts and ready-made tools. The platform is used to evaluate the extent to which SBOM-based infrastructure management platforms can mitigate the challenges associated with vulnerability and component management challenges associated with networked production environments. This phase of the research aims to establish the baseline procedures required to implement the system in an existing environment. These are then compared to the performance of Dependency Track in one of JYVSECTEC's production environments.

1.6 Structure

Section 2 outlines theoretical challenges associated with infrastructure management in production environments. The aims of this discussion are twofold. The first is to better understand the tasks associated with vulnerability, dependency and component management. The second is to examine the relevance of Software Bills of Material in the context of these challenges. Section 3 outlines the research framework and justifies the approach.

The core work of the thesis is presented in Sections 4 and 5. Section 4 describes the methods used and processes involved in creating a test environment. These are broken down into three core phases: the SBOM-to-Dependency-Track pipeline, platform functionality and customization. The

purpose of the test environment was to develop an SBOM to Dependency-Track pipeline and get a sense of Dependency-Track's baseline vulnerability management capabilities. Section 5 applies the tools developed in Section 4 to a production environment comprised of a subset of JYVSECTEC servers. The section analyzes Dependency-Track's performance with a view towards to gaining an understanding of the utility of SBOM-based management for DevSecOps in a production environment. Section 6 answers the research questions and concludes the project.

2 Challenges

2.1 Background

The thesis emerged from ongoing discourses evaluating the efficacy of SBOM systems relative to existing infrastructure management tools and techniques (Arora et al., 2022; Eggers et al., 2022; Stoddard et al., 2023). Existing literature on the subject typically focuses on the applicability of SBOMs to DevOps. These discussions typically focus on the theoretical benefits of infrastructure management via SBOM.

Literature which systematically tests the performance of SBOM-based infrastructure management is surprisingly thin. There are few if any assessments of SBOM performance in an applied setting. Those which do are typically more interested in the relevance of SBOM for DevOps. See for example Arora et al. (2022), Nguyen (2023) or Eggers et al. (2022). The implications of this focus are addressed explicitly in the analysis section of this work. No empirical studies on the subject from the applied DevSecOps perspective appear to exist at the time of writing (January 2024). The thesis contributes to the literature by filling this gap.

The work begins by examining managerial challenges typically associated with maintaining networked environments from the applied DevSecOps perspective. It specifically refers to challenges relating to the *active scanning* approach to infrastructure management. This is compared to infrastructure management using SBOM-based techniques. The active scanning approach is discussed because it is used widely in the industry.

A clear understanding of the core concepts is necessary before these techniques are discussed. Section 2.2 begins by defining relevant concepts with this consideration in mind. The section defines *what* specifically is being managed. Section 2.3 discusses the management techniques under examination. It outlines how these concepts are understood from an active scanning and then an SBOM perspective. The core differences and similarities between the two approaches are also discussed.

This is followed by a presentation of key SBOM features. These are contrasted with infrastructure management via active scanning. This part of the discussion is broken down into three core elements. The first two refer to areas where differences between the two approaches are the starkest. The first concerns temporality. Here the long-term approach used in SBOM-based

approaches is contrasted with the short-term nature of active scanning. The second of these concerns supply-chain transparency. This is a core demand in contemporary IT infrastructure management and is conceptualized quite differently depending on the approach. The third element discusses the general requirements necessary to implement SBOMs into infrastructure management workflows.

2.2 Concepts

2.2.1 Dependency

A *dependency* refers to a relationship between different software components in which one component relies on or is connected to the other (Cox et al., 2015). Software dependencies determine the interactions and connections between various elements within a system or application. Dependencies occur at every stage of the software supply-chain. Each has their own significance for software development and management. The discussion primarily focuses on examples concerning software management. Table 1 presents the basic dependencies under consideration and their descriptions (Sangal et al., 2005).

Table 1. Dependency types

Dependency	Description
Library	Library dependencies exist when an application or component uses functions or features provided by external libraries. They are essential for reusability and modularity but can introduce risks if not managed properly.
Runtime	Runtime dependencies refer to the external components or services a software application requires to function correctly during execution. Applications frequently depend on specific database or web server environments to run correctly.
Package	Code is often packaged and distributed as reusable components or packages. Package dependencies occur when one software package relies on another to provide certain functionality.
Module/code-level	Dependencies exist between different modules or code units within the application codebases. This can include a module calling functions in another module. This produces what is known as a code-level dependency.

Dependencies affect a wide range of network functions. Understanding and managing the totality of these dependencies is a critical administrative task (Burns, 2018). Dependencies include version functionality, security, reliability and scalability. Compatibility between components and the overall system is necessary in order to prevent conflicts and unexpected behavior.

Regularly updating and patching dependencies is therefore essential for maintaining a secure software ecosystem. Incorrect or missing dependencies can lead to runtime errors or application failures. Properly managing dependencies contributes to the reliability and stability of the software (Duboc et al., 2007). Properly managing dependencies allows for better scalability. The capacity to easily integrate new components or scale existing ones is an absolute requirement in any modern network.

2.2.2 Vulnerability

A *vulnerability* is defined as a weakness or flaw in a system, network, or software application (Krsul, 1998). An attacker can exploit this to compromise the system or its data. Vulnerabilities represent potential points of entry for unauthorized access, attacks or other security breaches. They can exist at various levels within an IT infrastructure. These range from software and networks to operating systems and configuration settings. Vulnerabilities can also emerge from human actions (Safianu et al., 2016).

Software vulnerabilities originate from programming errors, insecure configurations or outdated / unpatched software (Sibal et al., 2017). *Network vulnerabilities* stem from problems in the configuration or design of network components, routers, switches and firewalls. *Operating system vulnerabilities* refer to flaws in the design or implementation of an operating system which may allow attackers to exploit vulnerabilities and gain unauthorized control over the system (Li et al., 2017).

Configuration vulnerabilities refer to insecure or suboptimal configurations of systems, applications or devices. *Human actions* refer to poor password management, social engineering or insufficient employee training on security best practices. All can potentially be exploited to gain unauthorized access or compromise system integrity (Krsul, 1998; Li et al., 2017; Sibal et al., 2017).

2.2.3 Components

Components are the primary building blocks of IT infrastructure. They serve distinct roles in the overall functionality and operation of systems (Laan, 2017). There are two general categories of components. These are *software* components and *network* components. Software components

refer to the individual pieces of code, applications or programs collectively forming the software infrastructure of a system (Paik & Park, 2005). Challenges associated with software component management are of direct relevance to the questions posed in this thesis. Software components include *applications*, *libraries*, *frameworks* and *modules*/*classes* (Crnkovic et al., 2010).

Applications refer to end-user software which performs specific tasks or functionalities. Examples include word processors and web browsers. Libraries refer to reusable code modules which allow developers to incorporate pre-existing functions into applications. A framework is a comprehensive set of tools libraries and conventions which provide a foundation upon which software applications can be built. Modules and classes are smaller units of code within a program which provide specific functionalities intended to enhance code maintainability (Crnovic et al., 2010).

Network components refer to the physical and virtual elements of which network infrastructure is comprised. These enable the communication and exchange of data between devices within the network. Common network components include routers, switches, firewalls, servers and gateways (Perlman, 2000). Both component types should be effectively managed to ensure a secure, reliable and efficient IT environment (Rebaiaia & Kadi, 2013). The challenges associated with physical network infrastructure management are not directly related to the task at hand. They are noted here only for the sake of completeness.

2.2.4 Compliance

Compliance refers to *external* and *internal* regulatory obligations for network infrastructure. External compliance obligations include legal requirements as well as industry-specific guidelines. Internal compliance concerns policies governing information security, data protection and overall operational practices (Jayawickrama, 2006). Additional areas affected by compliance demands include security standards, internal policies and procedures and network security (Bicaku et al, 2018). Of most direct relevance to this discussion is the issue of software license compliance. These need to be tracked carefully to ensure they are used appropriately (Soomro & Hesson, 2012).

Effective monitoring of software licensing agreements is notoriously difficult in large organizations (Ghosh et al., 2019). A core claim made by advocates of SBOM-based infrastructure management concerns their ability to enhance the transparency of compliance efforts. It is achieved by

generating detailed inventories of software components and their dependencies. This in turn aids in the tracking, managing and securing of software assets (Holsing & Yen, 1999). A more detailed analysis of these dynamics is presented in Section 2.3.

Compliance is critical to maintaining a secure and ethical operational environment. It requires continuous monitoring and periodic assessments. Policies and practices should keep up with regulatory developments or evolve as the IT landscape changes. Non-compliance may result in legal consequences, financial penalties or reputational damage (Conway & Conway, 2009).

2.3 Discussion

2.3.1 Active scanning

Active scanning refers to the systematic and intentional examination of computer systems, networks and software applications. This type of activity can be performed by both benign and malicious actors (Popisil et al., 2021). This can be contrasted with *passive scanning.* Passive scanning involves the monitoring and network traffic without interacting with the systems in question (Allman et al., 2007). Active scanning instead involves sending specific requests or probes for the purpose of vulnerability identification. The principal aim is to assess the security posture of software installed on a network (Edgar et al., 2019). This is typically achieved using automated tools.

Such tools can simulate attack scenarios to identify vulnerabilities in operating systems, applications and configurations (Coffey et al., 2018). They actively seek out vulnerabilities by sending requests, queries or attempting to exploit potential weaknesses in the software. Scanning processes includes tasks such as checking for outdated software versions, misconfigurations, default settings and known vulnerabilities in the software stack (Vacca, 2007). This empowers administrators to identify and prioritize vulnerabilities based on their severity and potential impact.

2.3.2 Software Bill of Materials

Software Bill of Materials-based infrastructure management tools are positioned as an alternative to active scanning. SBOMs provide a structured and detailed inventory of the software components, libraries, frameworks, and dependencies used in the development and composition of a specific software application or system (Martin, 2020). They provide an organized breakdown

of the building blocks of any given software. This includes information such as component names, versions, licenses and their interdependencies. It is not unlike a traditional bill of materials seen in manufacturing industries (Arora et al., 2022a).

The primary purpose of an SBOM is to enhance transparency, traceability and security in software development and management processes (NTIA, 2021). SBOMs provide insights into the origins of each software component in a given network. They can be used to identify potential security vulnerabilities, assess licensing compliance and manage supply-chain risks. SBOMs are of relevance in industries where software development practices require close alignment with security, compliance and risk management practices (Bauer et al., 2020).

2.3.3 Similarities

Active network scanning and Software Bill of Materials (SBOM)-based management share a broadly similar focus. The overarching aim is to gain insights into IT infrastructure components. In both cases this involves gathering and cataloging detailed information about system elements. Active scanning achieves this through automated probes and analysis of network components. SBOM-based management achieves this by generating detailed inventories of software components in the network. These inventories catalogue software components, subcomponents and their dependencies (Jones & Tate, 2023).

Both methods identify and mitigate potential security risks and provide visibility to IT infrastructure. Efficient resource allocation is shared objective. Both aim to improve efficiency and facilitate informed resource-allocation decisions within the IT infrastructure. Both prioritize security. Active network scanning aids in the early detection of vulnerabilities and potential security risks. SBOM inventories achieve this through the identification and subsequent management of known vulnerabilities within software components (Stoddard et al., 2023).

2.3.4 Differences

The similarity of goals nevertheless translate to starkly different methodologies and focuses. This extends to the nature of information these approaches provide. Active scanning focuses on the current state of a network with a view towards gaining insights into current operational status and

potential security risks. It is dynamic and immediate in nature. This responsiveness to changes in network environments makes it well-suited for real-time monitoring (Barnett & Irwin, 2008).

SBOM inventories instead provide a static *view* of the software supply-chain. The aim is to highlight the relationships between software components within the IT infrastructure. SBOMs emphasize documentation and transparency (Stoddard et al., 2023). They provide a comprehensive overview of software building-blocks, versions and associated vulnerabilities. The approach is more focused on understanding the long-term implications of software usage (Xi et al., 2023). The temporal element is perhaps the clearest departure in approach. Long-term inventory management is widely considered to have advantages over real-time scanning (Carmody et al., 2021; Martin, 2020; Stoddard et al., 2023). These are discussed in Section 2.3.5 below.

2.3.5 Temporality

The long-term view of inventory management differs in scope and execution to active scanning. Long-term views of inventory management are said to be more consistent over time. This is therefore referred to as a *static* representation of auditing, compliance and planning (Zahan et al., 2023). Information recorded in static inventories remain relatively unchanged over time. This has notable benefits for compliance, auditing and documentation purposes. The fluctuations inherent in active scanning make it more difficult to maintain accurate records of the network state.

An historic perspective of the software supply-chain is attractive from a compliance perspective because it makes long-term planning easier (Bicaku et al., 2018). The static nature of SBOMs provide a detailed and unchanging snapshot of the software supply-chain. This is particularly important in industries where compliance obligations require a consistent and auditable record of component usage (Arora, 2022a).

Static inventory management allows for proactive risk management. Static information provided by SBOMs can be leveraged to proactively assess and address potential risks based on known vulnerabilities. Active scanning will only discover vulnerabilities as they emerge. This type of vulnerability management is viewed as strategic instead of reactive.

Static inventory management also supports efficient software asset management. SBOMs offer a clear and organized overview of software usage. This aids in optimizing licensing, managing

software updates. It empowers informed decision-making based on a stable and comprehensive inventory (Arora, 2022a; Stoddard et al., 2023).

2.3.6 Transparency

The U.S Department of Commerce (2021) conceptualizes software as hierarchical trees. Such trees are comprised of unique components and subcomponents. The challenge of monitoring software usage has only intensified as these ecosystems continue to grow in complexity (Al-Badareen et al., 2010). Maintaining transparency of software origins is far from easy. Transparency of usage necessitates transparency of origin. Transparency of origin is therefore mechanism to ensure software aligns with its claimed attributes. It ensures software is used legitimately (Barclay et al., 2019).

A central claim is that SBOMs contribute to transparency in the software supply-chain through the extensive documentation of the origins and relationships between software components (NTIA, 2021; Martin, 2021; Carmody et al., 2021). This is of critical importance given the increasing reliance on third-party software in networked environments and the widespread practice of component recycling. Extremities of component depth in the software stack will inevitably have deleterious effects on the transparency of supply chains (Kloeg et al., 2024).

Transparent supply-chains ensures the security and integrity of software. Potential security risks and vulnerabilities within IT infrastructure can be more easily identified by tracing the origins and components of software. The logic is that informed decision-making should be contingent upon a clear understanding of software origins and makeup.

A clearly documented software supply-chain gives rise to a proactive and efficient risk-management stance (NTIA, 2019; Stoddard et al., 2023). This is essential for demonstrating compliance with data protection and security standards. This typically involves compliance with legal mandates to maintain transparent and auditable records. Such requirements are notably common in industries such as government, healthcare and finance (Martin, 2020). Transparent supply-chains also contribute to the resilience of an organization's IT infrastructure (Carmody et al., 2021).

A clear understanding of these dynamics facilitates a swift and targeted response to security incidents. Supply-chain transparency therefore promotes more efficient incident responses. The

security practices of software suppliers can be more effectively assessed if these relationships are clear. Trust building through transparency incentivizes adherence to best practices and security standards among all stakeholders (Kloeg et al., 2024).

2.3.7 Requirements

An enhanced understanding of the software supply-chain is of particular benefit when seeking to understand known and emerging vulnerabilities and security risks. SBOMs can be considered a foundational layer in infrastructure management systems. They are not solutions in and of themselves. They are a generalized means to improve infrastructure monitoring capabilities. The United States Department of Commerce (2021) identifies minimum elements necessary for SBOM functionality pursuant to the Executive Office of the President's (2021) Executive Order 14028. The order specifically calls for the United States to improve its cybersecurity posture by expanding its vulnerability detection capabilities.

The United States Department of Commerce (2021) organizes these requirements into the following categories:

- Data Fields
- Automation support
- Practices and processes

Defining and standardizing *data fields* ensures consistency in the information captured within SBOMs. Data fields outline the essential details about software components (Arora et al., 2022a). This includes names, versions, origins, dependencies and other relevant attributes. Consistent and standardized data fields facilitate interoperability and understanding across different SBOM management systems. It ensures key information is uniformly captured. Standardizing data fields is attractive because it reinforces the usefulness of SBOMs across a diverse range of stakeholders in the software supply-chain (Xia et al., 2023).

Automation support emphasizes the use of technology to streamline the generation, updating and utilization of SBOMs. Automation reduces manual errors, enhances efficiency and enables real-time visibility into the software supply-chain (Zahan et al., 2023). This is crucial for handling the complexity and scale of modern network environments. It facilitates a dynamic SBOM generation

and update process. This in turn makes it possible to keep pace with the rapid changes inherent to software component update cycles (Stalnaker et al., 2023).

Practices and processes define the methodologies and workflows involved in the creation, maintenance and exchange of SBOMs. This element establishes best practices to all for SBOM integration into software development and supply-chain management processes (Arora 2022a). Well-defined practices and processes ensure SBOMs are not just static inventories. They are rendered an integral part of the software lifecycle.

Effective integration of SBOMs into the secure development life cycle demands specific practices and processes. These should focus on functional mechanics (United States Department of Commerce, 2021). Principles and processes provide guidance on when and how to generate SBOMs, how to update them and how to share them effectively within the supply-chain (Kloeg et al., 2024). The constantly evolving nature of software supply chains necessitates flexibility and adaptability.

2.3.8 Summary

SBOMs provide insights into the technical attributes, versions and inherited vulnerabilities of components (Martin, 2020). They monitor component vulnerabilities, end-of-life status and software dependencies. The transparency they provide expedites timely delivery of code updates or patches (NTIA, 2021). They help trim unnecessary code by offering comprehensive knowledge of relevant open-source components (Arora, 2022). They can reliably inform suppliers about component licensing obligations and assist in identifying blacklisted elements. The settled nature of SBOM-generated inventories makes them well-suited to compliance, auditing and documentation requirements. Active scanning is simply not designed to provide an overall picture of the environment beyond the here-and-now.

SBOMs are capable of verifying software component origins, compliance with policies and vulnerability scanning via component analysis (Arora 2022a). They also provide supplementary data such as component end-of-life information and supplier claims verification. They empower independent mitigations and informed risk-based decisions during software operation (Xia et al., 2023). This can enhance organization, planning, and cost-effectiveness. Active scanning is beneficial for immediate vulnerability and threat detection.

Open-source software is widely used at the organizational level. Difficulties emerge when discerning what third-party subcomponents are being utilized. SBOM tools identify these intricate and often unseen software dependencies. They also identify license compliance issues and provide vulnerability assessments (Ghandhi et al., 2018). SBOM proponents argue they minimize unforeseen work and code redundancy. Comprehensive subcomponent details facilitate the protection of intellectual property (CISA, 2023).

3 Framework

3.1 Research design

3.1.1 Approach

This section describes the research approach. Its purpose is to delineate the methodological scaffolding and to organize the core tasks. A careful methodological approach enhances the scientific validity of the research and ensure its objectives are met in a logical and cohesive manner (Deb et al., 2019). The core objective is to determine the viability of SBOM-based infrastructure management in a production level environment. Methods have been selected to support this objective.

The research is *descriptive*, *applied* and *quantitative*. This approach is used for three reasons: 1) Descriptive research is useful for comparative case-studies. 2) Applied research is appropriate when an immediate problem being faced needs to be solved (Klukken et al., 1997). This can be contrasted against a *fundamental* research approach. Fundamental research is concerned with generalizations and theory formulation (Deb et al., 2018). 3) Quantitative methods are useful in deductive research in which the variables are justified by the hypothesis being tested (Cresswell, 2002). A quantitative approach is necessary given the nature of the systems under examination. It allows for a thorough investigation into and comparison between dependency and component management in the test environment and JYVSECTEC's production environment.

3.1.2 Environment

Test environments aid in test planning, management, measurement, testing failures and test development and test execution. They provide a basis against which real-world performance can be compared (Eikelmann & Richardson, 1996). Tests environments are useful because they provide controlled and reproducible conditions for evaluation. This enables the systems under examination to perform under specific scenarios. They provide insights into system behavior, performance and reliability under simulated conditions (Deb et al., 2019).

The test environment is designed to be broadly comparable to what system administrators might expect in real-world scenarios. The environment was built with diversity of dependencies and components in mind. Three critical elements were considered when developing the test

environment. These are broadly relevant across a large swathe of software engineering research (Chen & Sun, 2006). They also have direct bearing on the operational logic of this case.

Well-designed test environments can aid in the efficient identification of vulnerabilities and potential security risks. Subjecting software or systems to simulated network conditions can reveal exploitable weaknesses in real-world environments. This approach has the added advantage of fortifying system security measures before deployment (Shahmehri et al., 2012).

The design of parallel and distributed systems is in large part determined by scalability. Test environments are a means to gain insights into this capability. They enable the simulation of various levels and types of network traffic. This ensures the dynamics of system scaling under different loads are well understood. Measurable insights into the nature of system capabilities can be gained. These can concern the relationship between increased user-demand or data-volume and the point at which network performance is adversely affected (Chen & Sun, 2006).

Test environments are also useful when assessing interoperability. This involves mimicking a wide range of network configurations. The approach ensures compatibility with different devices, platforms and network setups. Insights into how well the given system functions in a heterogeneous environment can then be generated (Rezaei et al., 2014).

3.2 Variables

Identifying suitable variables is critical in all forms of quantitative research (Borrego et al., 2009). Well-chosen variables enhance precision. They are central to understanding causal relationships. They enhance relevance and ensure replicability of results (Robinson, 2016). Clearly identified variables enhance the study focus. They aid understanding of cause-and-effect dynamics and their applicability to specific objectives. They also aid in the potential for replication by other researchers (Robinson, 2010; Case & Light, 2013). Table 2 presents the independent and dependent variables identified the project. They have been defined with the above in mind.

Table 2. Variables

Independent Variables	Dependent Variables
Dependency-Track	Vulnerability tracking metrics
Existing pipeline management tools (Bolt, Distro2sbom, container2sbom.sh)	Dependency management metrics
Custom scripts	Component analysis metrics

3.3 Baseline

Initial assessments of both the built and production environments establish the baseline for vulnerabilities, dependencies and components. This serves as the reference point for the initial state of the system. This allows for the identification and tracking of changes over time. Baselines serve as the benchmark against which alterations, updates and modifications can be measured (Chen et al., 2018). This makes it easier to detect unexpected deviations.

Dependencies and components in networked environments are interconnected. Changes in one can impact others. Establishing a baseline for dependencies and components provides a comprehensive view of system architecture. The potential repercussions of modifications, upgrades or additions to the system can be better understood if appropriate baselines are in place (Santos et al., 2012). They also provide a known starting point for identifying the source of potential issues.

Baselines are useful for configuration management. They are also necessary for technical documentation. Documenting desired and functional configurations is critical to systems administration more generally (van Kervel, 2011). Proper documentation ensures consistency across environments. Appropriate baselines facilitate the rapid restoration of systems to a known and stable state in the event of failures (Arnott & Pervan, 2012).

3.4 Data

3.4.1 Sources

The principal data encompass *vulnerabilities*, *dependencies* and *components*. These data are critical to understanding the effectiveness of SBOM-based management in both the built and production environments. It is therefore necessary to gain a clear sense of how these concepts are understood within the context of this research. Table 3 presents concise definitions for the principal data sources under examination (Cox et al., 2015).

Table 3. Data sources

Sources	Description
Vulnerabilities	Exploitable flaws in network configurations and software
Dependencies	The relationship between software and network components
Components	The status of network components

The principal focus concerns *vulnerability*, *dependency*, *component* and *compliance* analysis. A precise understanding of the analysis types under consideration is also necessary. The scope of these analyses encompasses the core administrative tasks performed in networked environments (Barrett et al., 2004). They form the basis of active scanning methods and reflect the necessary elements for SBOM-based infrastructure management.

Their universality means a comparison of the effectiveness of SBOMs and active scanning is possible. Clear definitions of these concepts are central to ensuring the reliability and scientific validity of conclusions (Roberts, 2016). They also enhance the replicability of the research. Analysis types are defined in Table 4 below.

Table 4. Analysis types

Analysis type	Description
Vulnerability	Assesses systems to discover and remediate potential weaknesses. Ensures security threats are dealt with and sensitive data is protected (Foreman, 2019).
Dependency	Assesses how changes and updates to network or software potentially affect the functionality of dependent components. Supports system stability and reliability by minimizing the risk of unexpected issues arising from changes in the environment (Santos et al., 2012).
Component	Evaluates the integrity, compatibility and performance of software and hardware network elements. Enables informed decision-making concerning the inclusion, exclusion of replacement of components. Promotes system reliability and robustness (Syafizal et al., 2020).
Compliance	Tracks systems to ensure compliance with regulatory obligations across networked environments (Trim & Lee, 2016).

3.4.2 Targets

The approach in these forms of analysis differs considerably. An important distinction between active scanning and SBOM-based dependency management is the nature of the analysis performed and the data targeted. Active scanning typically involves probing network traffic data to identify devices, services and potential vulnerabilities (Lipner, 2010). Network scanning provides insights into real-time network behavior and potential security threats. Active scanning also necessitates the collection of system and device configuration data. This in turn generates real-time vulnerability and dependency data which can be used in threat mitigation. Understanding how software configurations affect the environment is therefore a critical part of this process (Elbadawi & Yu, 2011).

SBOM-based analysis has similar aims. The methods used and the specific analysis performed differ significantly, however. SBOMs-based infrastructure management platforms are capable of ingesting detailed metadata about the software and components used in the networked environment. The target in this case is therefore the software stack (Arora et al., 2022). Software version information, dependencies and licensing details are included.

This facilitates a granular understanding of the relationships between network components given long-term view of the software supply-chain inherent to SBOM-based infrastructure management. Administrators are thusly empowered to track changes and updates to the software stack over time. SBOMs also include licensing information. This is critical in tracking regulatory compliance (Camp & Andalibi, 2021).

3.5 Comparative analysis

The comparative analysis between the test environment and the production environment focuses on the identified metrics. The research is descriptive, applied and quantitative. The approach aims to provide a robust foundation for comparing the efficacy of SBOM-based infrastructure management in the given environments. SBOM-based management is positioned as a potential solution to the challenges associated with managing vulnerabilities and dependencies. The comparison with active scanning methods serves to evaluate its practical efficacy.

A comparative analysis provides measurable insights into the performance of SBOM-based infrastructure management in diverse scenarios. The variables identified in the study further strengthen the rationale for the comparative approach. The basis of the comparison is the independent and dependent variables identified in section 3.2. Variables have been selected to focus the overall approach and uncover causal relationships (Höfer & Tichy, 2007). These elements collectively support a thorough investigation into and comparison between SBOM-based infrastructure management and active scanning methods in multifarious environments.

3.6 Limitations

It is necessary to also recognize potential constraints and challenges which may impact the validity and generalizability of findings (Ross & Zaidi, 2019). Limitations in this case concern the controlled nature of the test environment and potential variations in real-world production environments.

Addressing limitations transparently enhances the credibility of the research and guides future studies in refining and expanding the scope of analysis (Pullin & Knight, 2009).

One notable limitation could emerge from the complexity of real-world environments. The degree to which the test environment reflects the intricacies of production environments is a clear limiting factor. Variability in network configurations, software stacks and operational practices across the two environments could introduce factors which may be challenging to control. Caution should therefore be exercised caution when generalizing results to a broader context (Ross & Zaidi, 2019).

The selected variables might not encompass all relevant factors influencing the comparison. Additional variables or contextual nuances not explicitly considered in the study can also impact research outcomes (Hutton & Williamson, 2000). This limitation may introduce a degree of incompleteness in the analysis. The scope and boundaries of the study are clearly delineated with a view towards addressing this concern.

3.7 Summary

A structured methodological framework aligns with the stated research objectives and questions. It provides a systematic means to test the core elements of the hypothesis (McKenzie, 2004). The structured approach enhances the scientific validity of the study and strengthens conclusions (Benz & Newman, 2008). Methods have been chosen expressly with these considerations in mind. They have been identified as providing the most appropriate means to evaluate the effectiveness of SBOM-based platforms in managing vulnerabilities and dependencies in IT workflows.

Methods have been selected with explicit consideration of the research goals. They are tailored to provide the most appropriate means for evaluating the phenomena under investigation (Williams, 2007). Aligning methods closely with the research objectives ensures the study provides a focused and relevant examination of the topic under discussion. The approach is designed to enhance the credibility of the findings. It demonstrates a clear link between methods and the overarching research goals (Borrego et al., 2009).

4 Methods

4.1 Background

The research objective calls for a test environment to assess the viability of SBOMs and SBOM-based infrastructure management platforms. An analysis of the Dependency-Track platform and its dependency and vulnerability management capabilities has been performed with this in mind. A virtual environment comprising of approximately 1600 container images and a handful of virtual-machines has been used for this purpose. The focus on container images emerged out of necessity. It was simply the most efficient means to create SBOMs at scale given the limited number of virtual-machines immediately available.

A host of SBOM-based platforms are available. Dependency-Track was found to be the most appropriate platform with which to test the hypothesis and establish baseline capabilities and behaviors. There are good reasons for this. Its main focus is vulnerability management. It is a mature project under continuous development by OWASP for a decade (Dependency Track, n.d.). The official documentation is detailed and useful in most cases. It has full backend API support and integrates with a wide range of tools. The platform is fully open-source and maintained actively. Vulnerability data comes from a diverse range of sources. A wide range of repositories and environments are supported for component identification. These include PyPi, Maven, NuGet and NPM. A range of features can be configured in the UI to suite individual needs.

SBOMs can be generated in three different formats: *CycloneDX*, *SWID* and *SPDX*. CycloneDX has been selected because it is the most widely supported of the three. It was also developed by OWASP (OWASP, n.d.). This makes it a natural fit to be used in conjunction with Dependency-Track. Tools using the CycloneDX category typically focus on SBOM ingestion, vulnerability analysis and vulnerability management.

The work is broken down into three phases. The first phase outlines the SBOM-to-Dependency-Track pipeline. It documents the process of scraping docker images and extracting system information from local machines. It then describes how this information is converted into SBOM formatted files and the automation processes involved in sending this data to Dependency-Track via the API. The second phase describes the baseline capabilities of Dependency-Track. These are broken down into three broad categories: vulnerability scanning, component monitoring and

compliance analysis. The third phase presents three scripts developed to streamline management tasks in the abovementioned categories and reduce the need to interact with the UI. The Dependency-Track API is again central in this task. Figure 1 breaks down the general workflow of these phases.

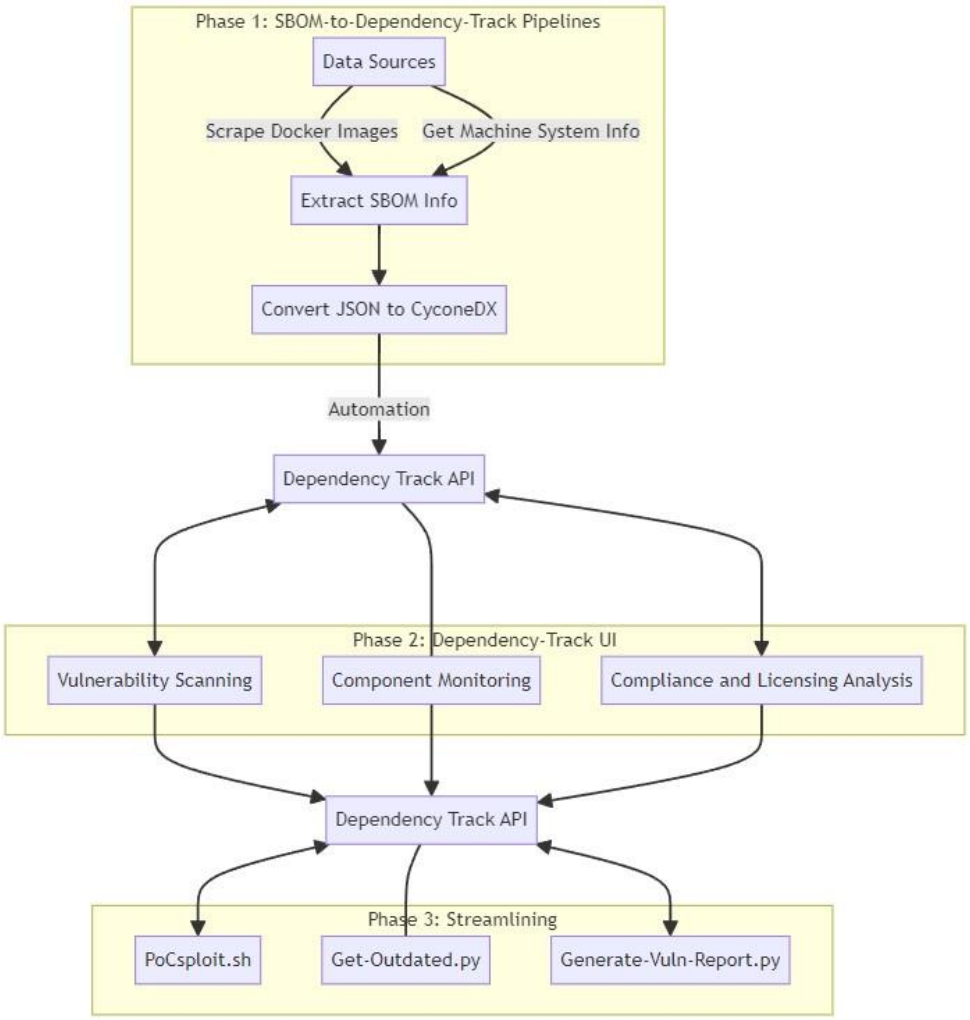

Figure 1. Environment setup phases

4.2 Pipelines

4.2.1 Image scraping

The most technically interesting aspect of this phase of the project was the development of a scraper to automate the container-harvesting process. Container images have been sourced from Docker Hub. Scraping is performed with a script called scrape-dockerimages.py. The script uses *Selenium* and *BeautifulSoup* to navigate Docker Hub search pages, extract image links and retrieve pull commands. The extracted information is then saved to a text file.

The process is not without its pratfalls and idiosyncrasies. Class names in Docker Hub are liable to change over time. Confirming class names is therefore a necessary part of the process. Another challenge is the rate limiting imposed on image pulls using the docker pull command. Non-subscribers are restricted to one hundred pulls in a six-hour period. Rate-limitations therefore also have to be considered. A convenient workaround is to instead pull images with *Grype*. The lack of rate limitations imposed on this tool results in a far more efficient scraping process. It is also used to generate SBOM files from the scraped container data in the next phase of the process. This makes for an elegantly symmetric solution.

4.2.2 Distro pipeline

Dependency-Track has full API functionality but does not include a tool for SBOM generation. SBOM data for Docker images and locally scanned virtual-machines have to be generated in separate processes. Two separate scripts are used for this purpose. A script from the Python Package Index repository called *Distro2sbom* is used to convert virtual-machine system information into CycloneDX formatted SBOMs. The package is incorporated into a bash script (somewhat confusingly called *distro2sbom.sh*) to automate key elements of the process.

The distro2sbom.sh script automates key elements of the file management process before virtual-machine data is converted into SBOMs by the Distro2sbom Python package. These relate principally to naming conventions (based on hostname, distribution name and kernel version) and the directory location (*~/.dep_data*). The script then installs the Distro2sbom package using *pip*. Disto2sbom then extracts the necessary data from an *inventory.yaml* file containing basic system information and converts them to CycloneDX-formatted SBOM. These are then sent to Dependency Track via the API.

4.2.3 Container pipeline

A separate bash script has been written to convert container image data into SBOMs. The script is called container2sbom.sh to maintain consistency in naming conventions. The script begins by installing Grype if it is not present on the system. An output directory is created for storing CycloneDX JSON files. The script retrieves container names and uses Grype to scan each container for vulnerabilities. The results are then saved in the CycloneDX format. The script integrates with Dependency Track by extracting the project UUID, constructing project names based on containers

and hosts and sending POST requests to the SBOM endpoint to create or update projects with vulnerability information.

Grype is officially marketed as an open-source vulnerability scanning tool (Anchore, 2022). Scans with Grype are performed in this case because the results are stored in CycloneDX formatted JSON files. These files can then be ingested by Dependency-Track for further analysis without further conversion. This is an important detail because Dependency-Track does not integrate with external vulnerability scanners.

Bolt is an orchestration tool which can be used for a range of automation tasks. (Puppet, 2024). It is used here to automate the execution of the scripts to generate container and virtual-machine SBOMs. A visual representation the orchestration workflow is provided in Appendix 1. The tools used in these processes are presented in Table 5.

Table 5. Automation tools

Tool	Purpose
scrape-dockerimages.py	Docker repository scraper
distro2sbom.sh	SBOM generator for virtual-machines
container2sbom.sh	Container SBOM generator
Grype	SBOM scanning tool for container images and filesystems
Dependency-Track API	Endpoint communications
Bolt	Pipeline orchestration

4.3 Dependency-Track

4.3.1 Overview

Dependency-Track utilizes SBOMs to increase transparency and traceability throughout the software supply-chain. The platform can consume SBOMs generated by a range of tools and platforms. Dependency-Track analyzes this data to identify and assess potential security vulnerabilities associated with the components under examination. The platform tracks the lifecycle of components, assess their risk profiles and enforce security policies based on the information provided in the SBOM (Dependency-Track, 2023). This is claimed to promote informed decision-making as to whether components should be included in or excluded from a given environment. Figure 2 presents the initial state of the environment after all SBOMs have been scanned and ingested.

Figure 2. Dashboard view

4.3.2 Core Capabilities

The platform has the following capabilities (Dependency-Track, 2023):

- Tracks all systems and applications with SBOMs
- Components defined in SBOMs are analyzed for known vulnerabilities using multiple sources of vulnerability intelligence
- Displays all identified vulnerabilities and vulnerable components for every SBOM analyzed
- Identifies all systems and applications with a specific component or are affected by a specific vulnerability
- Support for the Exploit Prediction Scoring System (EPSS)
- Evaluates the portfolio of systems and applications against user-configurable security, operational and license policies

4.3.3 Known vulnerability analysis

Dependency-Track conducts known vulnerability analysis by leveraging a comprehensive database of vulnerabilities. It utilizes SBOMs to identify and assess the risk associated with software

components. The platform integrates with multiple sources of vulnerability intelligence to identify components with known vulnerabilities. Table 6 (Dependency-Track, 2023) displays the sources of vulnerability data utilized by the platform:

Table 6. Dependency-Track vulnerability intelligence sources

Analyser	Description
Internal	An internal analyzer Identifies vulnerable components from an internal directory of vulnerable software
National Vulnerability Database (NVD)	A U.S. government repository of standards-based vulnerability management data represented using the Security Content Automation Protocol (SCAP)
OSS Index	A service provided by Sonatype which identifies vulnerabilities in third-party components
Vuln DB	A commercial service which identifies vulnerabilities in third-party components

Analysis involves checking the version information of each component against known vulnerability entries. Dependency-Track provides information about the specific vulnerability if a match is found. Severity, potential impact and available remediation advice are included. This facilitates the efficient identification of components and their security status. Accurate and efficient vulnerability analysis hinges on the effectiveness of these processes.

4.3.4 Component identification

Dependency-Track handles component identification by creating a detailed inventory of software components within a project. SBOMs provide a standardized and structured representation of the components, their versions and their dependencies. The platform extracts and processes SBOMs after they are ingested. A detailed picture of the relationships between components is provided. This includes component names and versions. The most useful way to view this information is via the search function of the *Components* view. The results from a typical component search are presented in Figure 3 below.

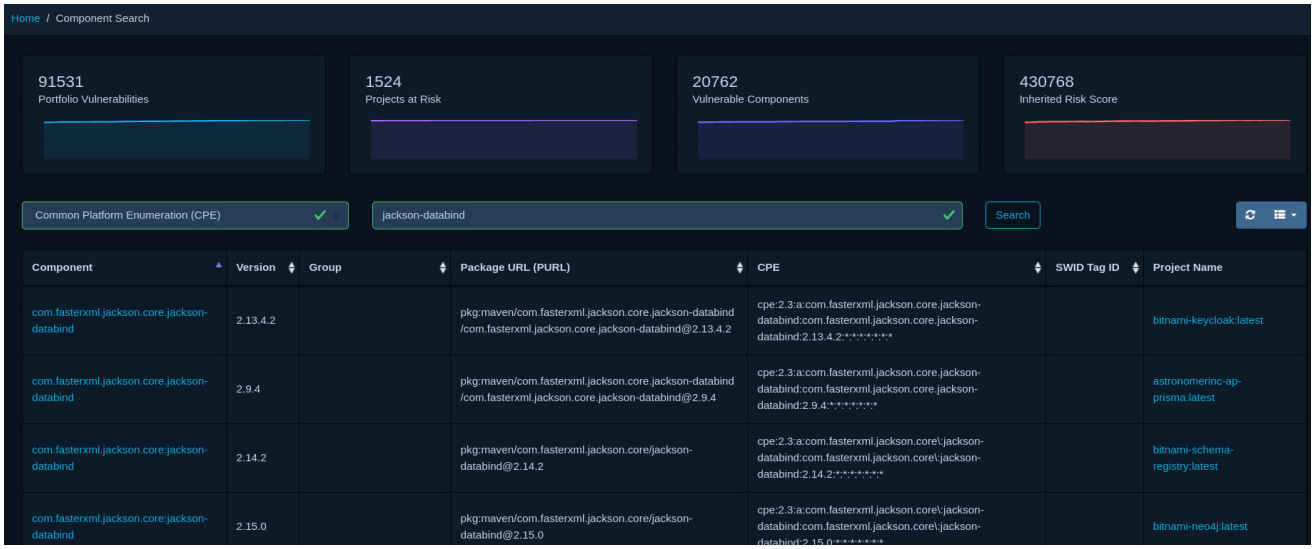

Figure 3. Component search

This data is utilized to establish a reliable inventory which forms the basis for subsequent analyses. Dependency-Track's approach to component identification employs both static analysis and ongoing tracking and monitoring throughout the software development lifecycle. An accurate record of the software components can be maintained even as projects evolve and new components are introduced or existing ones are updated.

4.3.5 Component analysis

Dependency-Track performs outdated component analysis by comparing the versions of software components listed in the Software Bill of Materials (SBOM) against the latest available versions. This identifies components which are no longer maintained or have known security vulnerabilities due to outdated versions. The UI is somewhat confusing on this front. The easiest way to view this information is not via the *Components* tab as one would expect. It can instead be found by clicking on the Components tab of the *Projects* view. The *Components* tab is shown in Figure 4.

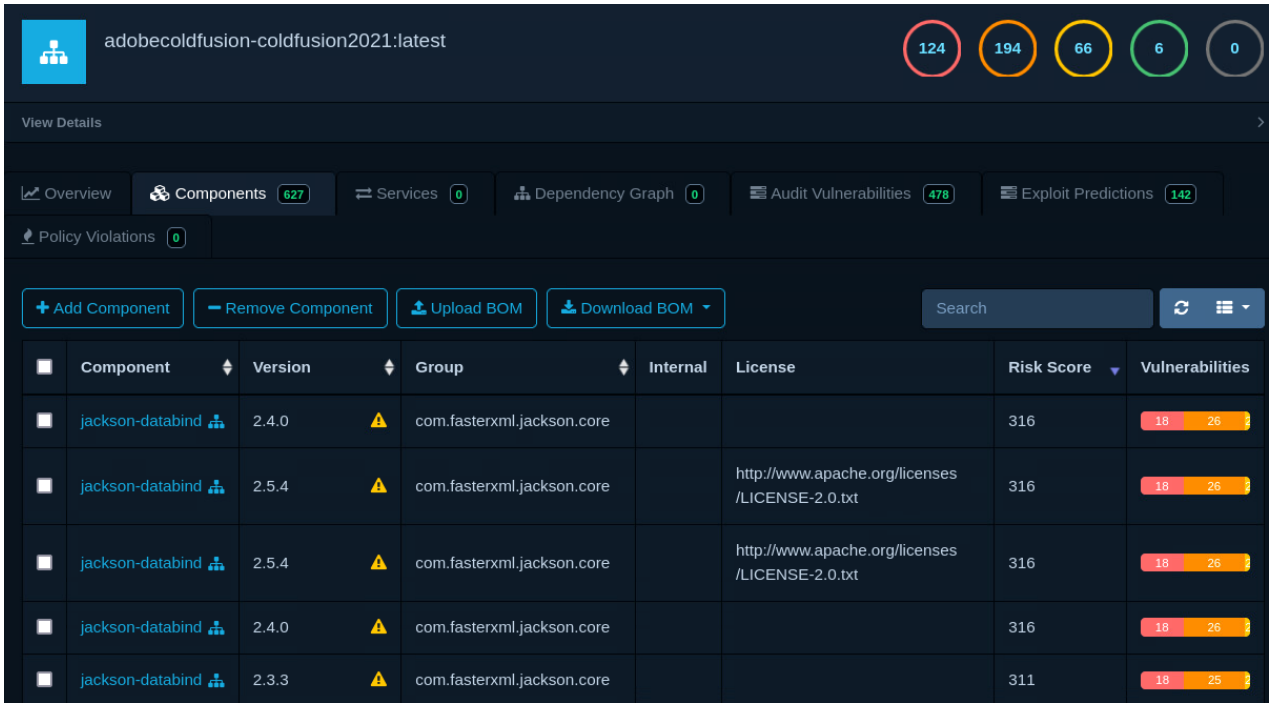

Figure 4. Components overview

Dependency-Track integrates with a range of vulnerability databases and version tracking mechanisms to assess whether a component version is up to date. The platform checks the version information of each component against the latest versions available in its database when the project SBOM is initially submitted to Dependency-Track. An alert is raised about the potential risks associated with using a particular version if the component is found to be outdated.

4.3.6 Impact analysis

Dependency-Track evaluates the potential impact of vulnerabilities in a given environment. It can help identify all affected projects across the environment. It is possible to look up vulnerability information in the platform if it is published in a data source with Dependency-Track integration (NVD, OSS Index, VulnDB and so on). Impact analysis aims to answer two critical questions:

- What is affected?
- Where am I affected?

The platform contains a full mirror for each of the vulnerability data sources it supports. Captured public information about the vulnerability includes the description, affected versions, Critical

Weakness Enumeration (CWE) and severity. A list of affected projects is also provided. This list is dynamically generated based on internally generated data. Performing a search on a component will reveal a list of vulnerabilities if the component name and version are known. The search output lists all projects with a dependency on the component. See Figure 5.

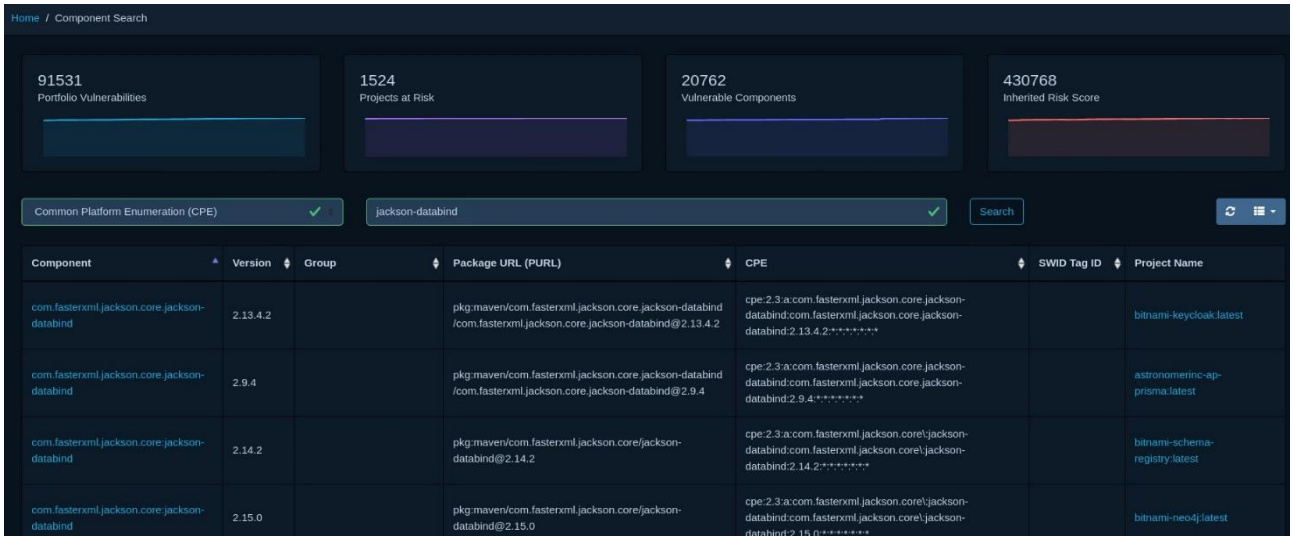

Figure 5. Component analysis results

4.3.7 Analysis states

An analysis decision can be made for each finding when triaging results. The supported states are presented in Table 7 (Dependency-Track, 2023):

Table 7. Analysis states

State	Description
EXPLOITABLE	The finding is exploitable (or likely exploitable)
IN_TRIAGE	An investigation is in progress to determine if the finding is accurate and affects the project or component
IN_TRIAGE	The finding was identified through faulty logic or data (i.e., misidentified or incorrect vulnerability intelligence)
NOT_SET	Analysis of the finding has not commenced

Individual findings can be suppressed regardless of analysis state. Suppressed findings will have a positive impact on metrics whereas unsuppressed findings will not. Suppressed findings may:

- Decrease the number of *portfolio vulnerability* metrics
- Decrease the number of *vulnerable project* metrics
- Decrease the number of *vulnerable component* metrics
- Decrease the number of vulnerabilities in specific components
- Decrease the *inherited risk* score

Unsuppressed findings will continue to have an impact on all metrics. A finding remains in the suppressed state unless a user removes the suppression from the finding. It is considered best practice to suppress findings with an analysis state of *NOT_AFFECTED* or *FALSE_POSITIVE*. The inherited risk and corresponding metrics can then be taken into consideration when drawing analytical conclusions. Vulnerabilities associated with suppressed findings are assumed resolved.

4.3.8 Policies

Dependency-Track provides a robust policy management system enabling users to create, configure and enforce policies at both portfolio and project levels. The three main types are license, security, and operational violations. License violation policies specify SPDX license IDs or groups. There is considerable flexibility in how custom license policies are defined. Security violations involve setting vulnerability severity conditions which trigger a violation when the severity matches the policy.

Organizational violations enable the creation of lists for allowable or prohibited components based on attributes like coordinates, package URL, CPE (Common Platform Identifier), SWID (Software Identification) Tag or hash. Vendor-supplied SBOMs can be assessed per the OWASP Software Component Verification Standard by creating a new project for procured software. A comprehensive component analysis is provided upon upload. The analysis covers component inventory, vulnerabilities, outdated component status and policy violations (Dependency-Track, 2023). This functionality was not tested extensively in the test environment because it is peripheral to the main objectives of the research.

4.3.9 Alerts

Dependency-Track's alerting capabilities were specifically earmarked for testing by the JYVSECTEC. In this case the platform offers a flexible notification framework to alert users or systems about newly discovered vulnerabilities and known vulnerable components added to projects. Users

receive portfolio notifications on objects like vulnerabilities and audit decisions. Notifications for system-level informational and error conditions are also available. Notification publishers enable users to define notification structure (MIME type, template) and delivery methods (publisher class). Integration with platforms like Teams and Slack are supported. Webhook and console notifications are also offered.

Notifications cover new vulnerabilities, vulnerable dependencies, project audit changes and global audit changes. Users with SYSTEM_CONFIGURATION permission can use the administrative page to create notifications. The process involves creating an alert with details such as name, scope, notification level and publisher. Alerts can be configured by selecting notification groups and specifying destinations after creation. Portfolio notifications are broadcast by default. This behavior can be adjusted by optionally limiting the affected projects. Alerts are viewable in the UI. A warning and informational alert have been configured for demonstration purposes. These are sent via Webhook to a Teams channel. Figure 6 presents alerts created in the Alert section.

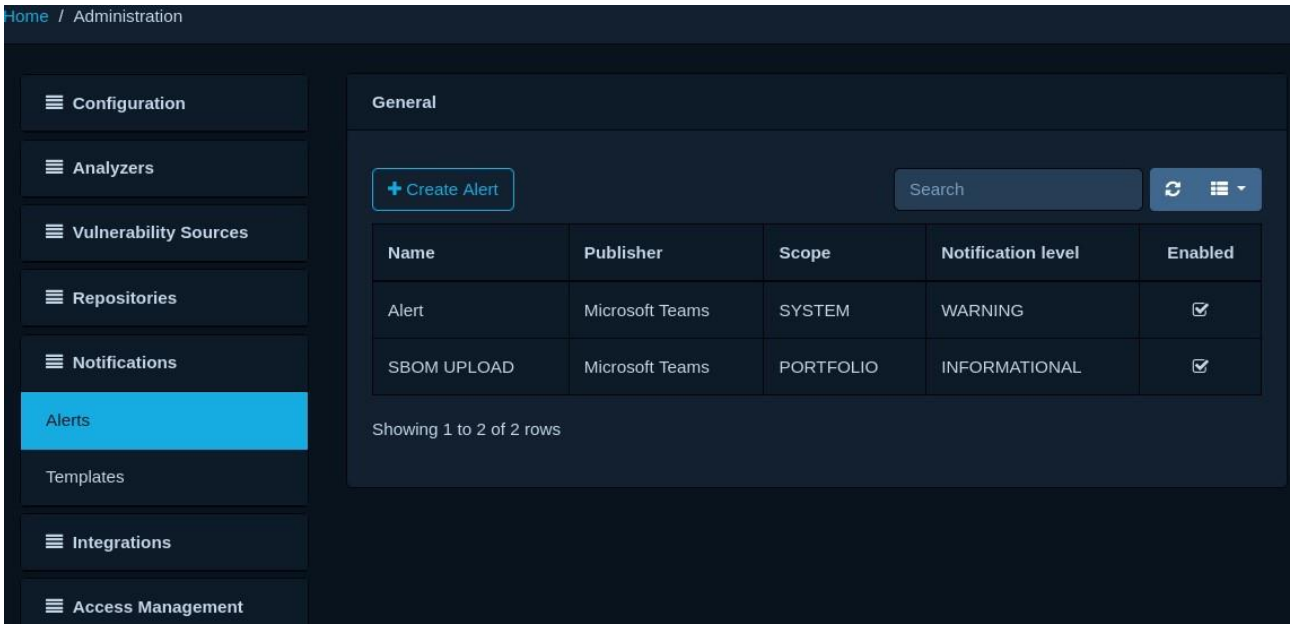

Figure 6. Teams alerts

4.4 Streamlining

4.4.1 Overview

Three scripts have been written to streamline infrastructure management and reduce reliance on the occasionally byzantine corridors of the Dependency-Track UI. The scripts leverage the Dependency-Track API to provide targeted solutions to specific research-related use-cases. Their general purpose is to improve efficiency and reduce the complexity inherent to software version management. One is designed to identify outdated software components at scale. Two are designed for vulnerability management and identifying exploitable vulnerabilities within diverse projects. The scripts are as follows:

- *get-outdated.py*
- *generate-vuln-report.py*
- *pocsploit.sh*

Each script considers the overarching questions guiding the research. The *get-outdated.py* script provides a structured approach to identify and remediate outdated components. The *generate-vuln-report.py* script automates vulnerability tracking. pocsploit aims to leverage the Dependency Track API in red team exercises via a streamlined process of vulnerability identification and exploit-sourcing.

4.4.2 get-outdated.py

The *get-outdated.py* script is a simple software management tool. Its primary purpose is to track software versions across various projects using the Dependency-Track API. It works by initiating a detailed list of projects and compiling a list of the components of which they are comprised. This includes names, versions and package URLS (pURLs). The script then checks for the latest available version of each package based on its pURL. It compares the existing project version to its latest version. This automated process is designed to aid the software update process. It is a quick and dirty means to view version information in the environment. Example output is shown in Figure 7 below.

```
┌──(kali㉿kali-vle)-[~/jyvsectec/sbom/setup-scripts/usage]
└─$ python3 get-outdated.py
Project: accurics-terrascan:latest
Package: cloud.google.com/go
Installed version: v0.65.0
Latest version: v0.110.7
----------------------------------------
Project: accurics-terrascan:latest
Package: cloud.google.com/go/storage
Installed version: v1.10.0
Latest version: v1.32.0
----------------------------------------
Project: accurics-terrascan:latest
Package: github.com/Azure/go-autorest/autorest
Installed version: v0.11.18
Latest version: v0.11.29
----------------------------------------
Project: accurics-terrascan:latest
Package: github.com/Azure/go-autorest/autorest/adal
Installed version: v0.9.13
Latest version: v0.9.23
----------------------------------------
```

Figure 7. get-outdated output

4.4.3 generate-vul-report.py

The generate-vuln-report.py script automates vulnerability tracking across multiple projects. It begins by retrieving a list of projects and their associated vulnerability data via the Dependency-Track API. Vulnerabilities are filtered based on CVSS (Common Vulnerability Scoring System) and EPSS (Environmental Public Security Score) data.

The tool then compiles project details, vulnerability IDs and component versions into a structured list based on EPSS values. The resulting information is then outputted to a text file. The script facilitates a rapid assessment of potential security issues in a given environment across diverse projects. An output example is presented in Figure 8.

NAME	VERSION	UUID	VULN-ID	CVSS	EPSS	COMPONENT-NAME	COMPONENT-VERSION
atlassian-pipelines-agent:latest	1.0.1t-1+deb8u5	5e5bad65-6719-45dd-81c2-692b93d72fab	CVE-2014-0160	7.5	0.97588	openssl	1.0.1t-1+deb8u5
atlassian-stash:latest	1.0.1t-1+deb8u2	a8531426-82a6-463a-93e7-1d57753d6a1d	CVE-2014-0160	7.5	0.97588	openssl	1.0.1t-1+deb8u2
bitnami-debian-base-buildpack:latest	1.0.1t-1+deb8u6	9f8d660d-5a44-41d3-b263-ffe0881d4c4f	CVE-2014-0160	7.5	0.97588	openssl	1.0.1t-1+deb8u6
circleci-builder-base:latest	1.0.1f-1ubuntu2.19	3ad90207-c6f7-45eb-8f0f-d16a964e8875	CVE-2014-0160	7.5	0.97588	openssl	1.0.1f-1ubuntu2.19
circleci-cassandra:3.10	1.0.1t-1+deb8u6	66de1b3e-a13f-49c3-8e9e-54c8d461c836	CVE-2014-0160	7.5	0.97588	openssl	1.0.1t-1+deb8u6
circleci-java:latest	1.0.1t-1+deb8u6	bb5977ec-3edd-48c8-b8a8-63ca02604a25	CVE-2014-0160	7.5	0.97588	openssl	1.0.1t-1+deb8u6
continuumio-conda-concourse-ci:latest	1.0.1e-57.el6	5019d5c9-cac4-4798-84a3-d8eb19e1640e	CVE-2014-0160	7.5	0.97588	openssl	1.0.1e-57.el6
cockroachdb-postgres-test:latest	1.0.1k-3+deb8u4	6acce233-5c80-4678-aa6a-cfb75dc5eeca	CVE-2014-0160	7.5	0.97588	openssl	1.0.1k-3+deb8u4
docker-example-voting-app-worker:latest	1.0.1t-1+deb8u5	6cc1a0f6-145f-4dd2-81d4-a24caad939dc	CVE-2014-0160	7.5	0.97588	openssl	1.0.1t-1+deb8u5
docker-whalesay:latest	1.0.1f-1ubuntu2.11	9ea22e90-1983-4b9c-a0b2-0a25b03cfc10	CVE-2014-0160	7.5	0.97588	openssl	1.0.1f-1ubuntu2.11
mirantis-rethinkcli:v2.2.0-ni	1.0.1t-1+deb8u7	2c822f6c-8b6c-4584-9dda-275c7df38595	CVE-2014-0160	7.5	0.97588	openssl	1.0.1t-1+deb8u7
onlyoffice-mailserver:latest	1.0.1e-58.el6_10	98483bc0-288f-458e-b401-36e28f0fcd41	CVE-2014-0160	7.5	0.97588	openssl	1.0.1e-58.el6_10

Figure 8. generate-vuln-report.py output

4.4.4 pocsploit.sh

The *pocsploit.sh* script fetches information about known exploited vulnerabilities and the projects they affect. The script uses the Dependency-Track API to uncover exploitable vulnerabilities in the environment. This is achieved in two phases. The script first gathers project data from the Cybersecurity and Infrastructure Security Agency (CISA's) Known Exploited Vulnerabilities (KEV) database via the Dependency Track API. Vulnerabilities are linked to publicly available Common Vulnerabilities and Exposures (CVEs) in the second phase.

CVEs are then checked against exploits available from two sources. The first is Exploit-DB. Exploits from this source are available locally using the *searchsploit* tool natively available in Kali Linux. The second is a GitHub repository housing an extensive archive of Proof of Concept (PoC) exploits dating back to 1999. The repository is updated regularly.

Finding exploitable vulnerabilities in a red-team scenario is frequently tedious and time-consuming. pocsploit streamlines this process. It has the added advantage of being directly connected to the environment. This raises the possibility for red teams to find more unique vulnerabilities to exploit. There is clear potential to produce a more dynamic in-game environment. Example output is presented in Figure 9.

```
┌──(kali㉿kali-vle)-[~/jyvsectec/sbom/setup-scripts/usage]
└─$ ./pocsploit.sh | grep mirantis -B 1 -A 6
+-------------------------------------------------------------------------------------------+
| mirantis-ucp-openstack-ccm:3.5.3                                                          |
+-------------------------------------------------------------------------------------------+
| CVE                | CVE-2019-15752                                                       |
| UUID               | 73bca520-77e7-4bad-931a-30f04284b446                                 |
| Exploit-DB         | https://www.exploit-db.com/exploits/48388                            |
| PoC                | Not available                                                        |
+-------------------------------------------------------------------------------------------+
--
+-------------------------------------------------------------------------------------------+
| mirantis-salt:saltstack-ubuntu-bionic-salt-2018.3                                         |
+-------------------------------------------------------------------------------------------+
| CVE                | CVE-2020-11652                                                       |
| UUID               | 5fc83cd6-d491-461c-a954-f1fa4289413e                                 |
| Exploit-DB         | https://www.exploit-db.com/exploits/48421                            |
| PoC                | https://github.com/fanjq99/CVE-2020-11652                            |
+-------------------------------------------------------------------------------------------+
+-------------------------------------------------------------------------------------------+
| mirantis-salt:saltstack-ubuntu-bionic-salt-2018.3                                         |
+-------------------------------------------------------------------------------------------+
| CVE                | CVE-2020-11651                                                       |
| UUID               | 5fc83cd6-d491-461c-a954-f1fa4289413e                                 |
| Exploit-DB         | https://www.exploit-db.com/exploits/48421                            |
| PoC                | https://github.com/chef-cft/salt-vulnerabilities                     |
+-------------------------------------------------------------------------------------------+
```

Figure 9. pocsploit output

4.5 Summary

Section 4 presented the work of setting up the test environment and establishing its baseline capabilities. The aim was to examine the viability of SBOM-based management platforms for DevSecOps. The platform selected to test this proposition is Dependency-Track. It was chosen for its mature vulnerability management focus, extensive documentation, open-source nature and API support. The work took place in three phases: developing the SBOM-to-Dependency-Track pipeline, establishing Dependency-Track's capabilities and developing custom scripts to streamline vulnerability management.

Dependency-Track's core capabilities encompass tracking, uploading SBOMs, vulnerability analysis, and policy enforcement. The platform handles known vulnerabilities, component identification. It offers robust policy management, notification systems and impact analysis. These tools seem to work well. Hidden dependencies and their associated vulnerabilities appear more visible. Its baseline capabilities and behaviors align broadly with expectations. The platform is not without its quirks, however. These are discussed in Section 5 along with more substantive problems encountered during this process.

One of the project's unique elements is the development of three scripts leveraging Dependency-Track API. These scripts are designed to facilitate software version tracking, automated vulnerability reporting and exploitable vulnerability identification without navigating the UI. There is a solid use-case for red teaming and penetration testing scenarios. But the usefulness of these tools ultimately depends on how SBOMs in general and Dependency-Track in particular stand up to scrutiny in the production environment. This task is undertaken in Section 5.

5 Analysis

5.1 Overview

Section 5 compares the baseline Dependency-Track capabilities presented in Section 4 to that of the production environment. The core aim is to gain a better understanding of Dependency-Track's operational capacities and limitations. SBOMs were generated for this purpose from a subset of fewer than ten production servers as a test-case. These were then ingested by Dependency-Track using the methods outlined in Section 4.

The environment was small-scale by design. The reasoning behind this decision was simple: large-scale testing would only be performed if further investigation were required. The analysis is broken down into five core areas. These are *depth of data*, *transparency*, *vulnerability management*, *component management* and *alerts*. The main analysis section is followed by a brief presentation of some additional observations. The section ends with a summary of the analysis.

5.2 Production environment

5.2.1 Depth of data

Data in Dependency-Track appears at a glance to be comprehensive. But closer examination reveals this to not be the case. A notable challenge relates to the complexity of data being processed into SBOMs. The problem can be illustrated with two simple examples: the Linux kernel and systemd. The kernel consists of around twenty-seven million lines of code (Larabel, 2020). systemd is a critical service configuration daemon used across a wide range of Linux distributions. It and its subcomponents consist of approximately twelve million lines of code (Larabel, 2020a).

The idea of maximalism lies at the heart of the SBOM project. A maximalist approach to vulnerability management implies every line of code should monitored and accounted for. This appears to be what is observed in the environment at a glance. CycloneDX formatted JSONs can quickly become larger than a megabyte of text. But even this provides insufficient granularity relative to the scale of these codebases.

The answer lies in the level of detail SBOMs claim to provide. These are *licenses*, *modules*, *patch levels* and *backports* (Graham, 2019). The detail provided by SBOMs in the production

environment hovers around metadata at the licensing level. This is useful from an open-source perspective for DevOps because dependencies frequently have to be installed manually. But it has limited utility from a DevSecOps perspective. A typical CycloneDX-formatted SBOM JSON is presented in Figure 10 below.

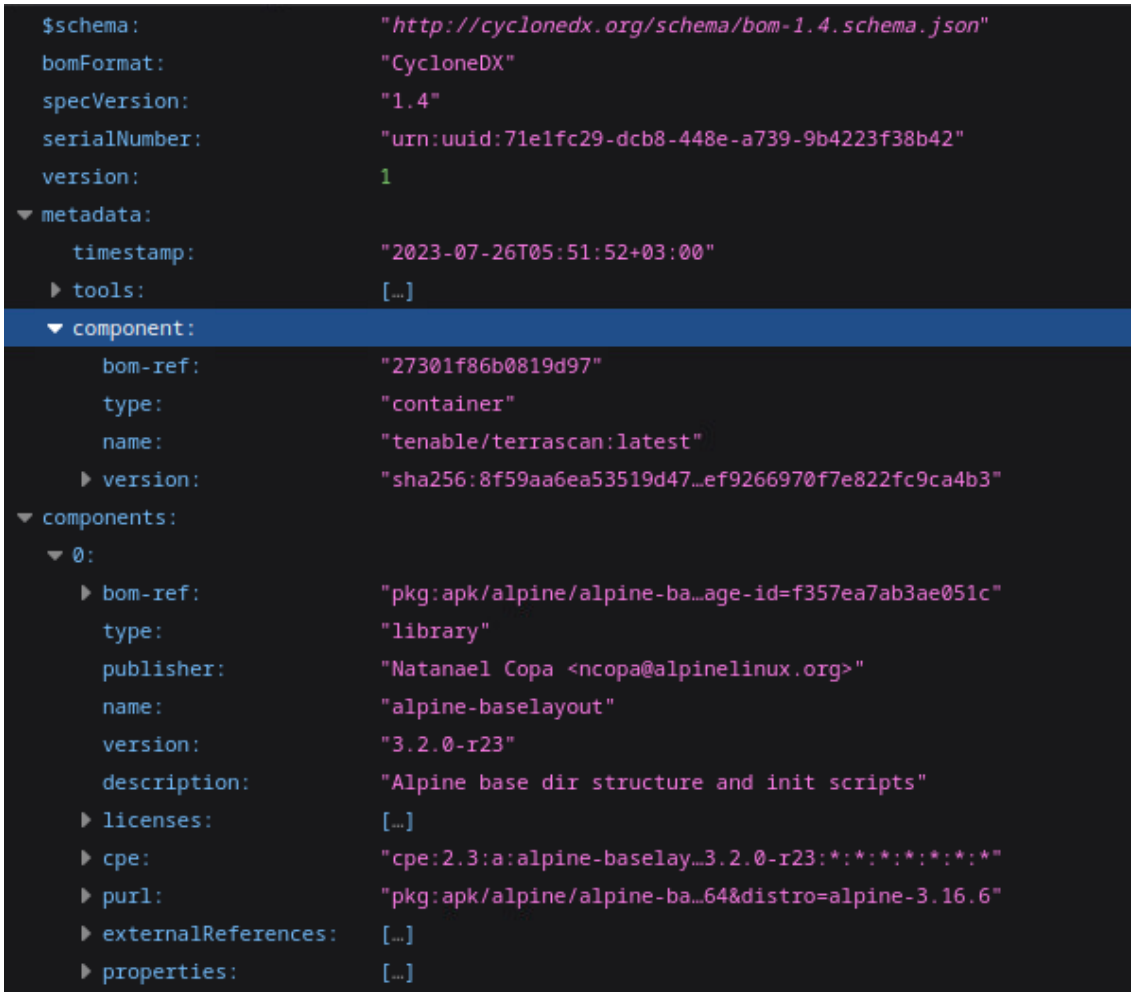

Figure 10. SBOM data structure

Transparency from the DevSecOps standpoint is contingent upon a full and granular accounting of the entire codebase. This is simply not feasible from a resourcing standpoint. The obvious corollary is that a full and granular understanding of the attack-surface is equally unfeasible. A lack of depth has important implications for transparency claims and the effectiveness of SBOM-based vulnerability management more generally.

5.2.2 Transparency

Advocates of SBOM-based infrastructure management typically draw attention to their capacity to clarify the software supply-chain. This is seen as a vital step in enhancing security measures and ensuring infrastructural robustness. The specific nature of this claim is discussed in Section 2.3.6. Observations made in the environment confirm the problems raised in the previous section and outline specific scenarios in which they appear. Components and their dependencies are clearly visible. It is indeed possible to pinpoint occurrences of components across a range of projects. But beyond these basic elements some cracks quickly emerge in the façade.

The most obvious of these is the prevalence of false positives reported upon ingestion. The tool's ability to distinguish active vulnerabilities from background noise appears quite limited. An up-to-date generic server in the production environment running a simple web-based application reported forty-six critical and 180 high vulnerabilities. Most of this was due to versioning of the RHEL-derivative Linux distro used in the environment. The project view is presented in Figure 11.

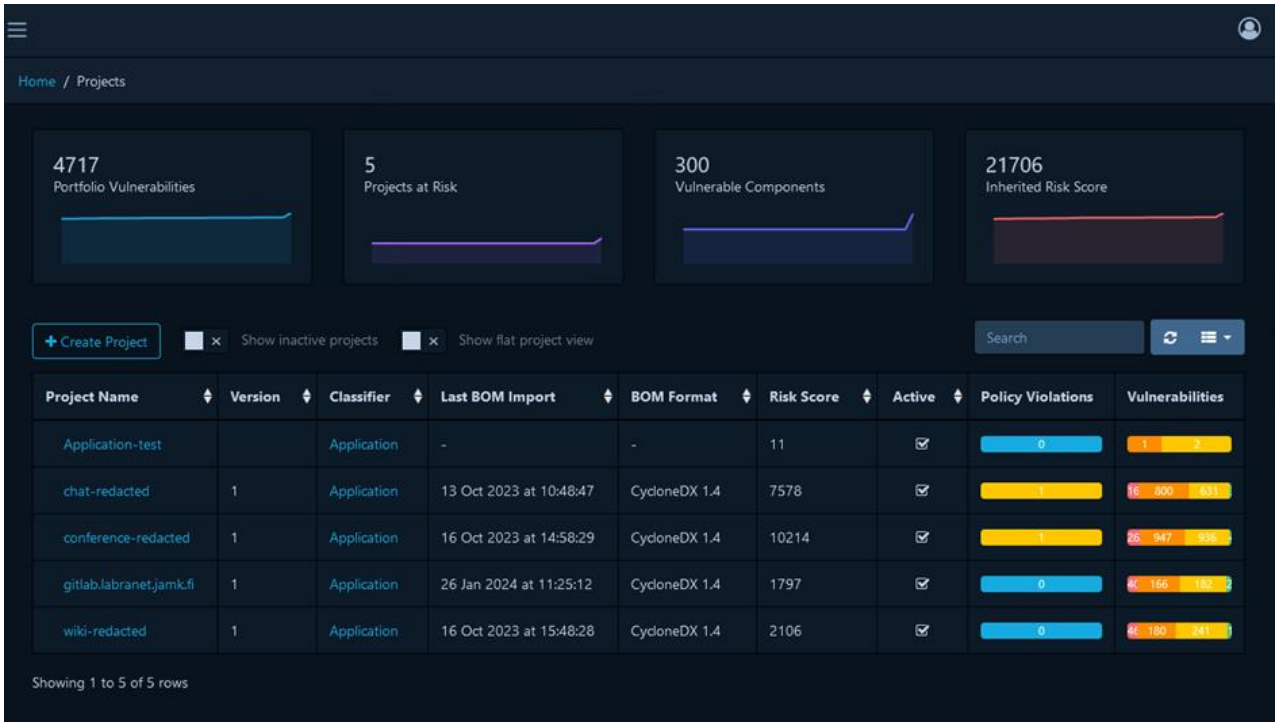

Figure 11. Project view

5.2.3 Vulnerability management

The problem described above is connected to the practice of *backporting.* Backporting refers to the practice of taking updates or fixes from a recent version of a software package and applying it

to an older version (Red Hat, n.d.). This is typically done to address a specific need or issue in an older software version without having to upgrade to the latest release. This is beneficial because module owners can declare a subset of versions they will support and for how long. It is these long-term support (LTS) versions which SBOMs track.

The practice is not strictly connected to usage patterns. This means software components can be in use long after they are officially supported. It is a problem commonly associated IoT devices. These can lag behind current versions well beyond manufacturer supported LTS (Akiyama et al., 2023). Vendors typically opt to patch only the most recent versions when a vulnerability is discovered. Older versions will be susceptible to any emerging vulnerabilities. Users of outdated software frequently have to fix bugs themselves as a result.

Backported security-fixes are effectively the black hole of SBOM-based vulnerability management. They are entirely separate from official development channels. They are difficult to track because backported fixes can occur without a corresponding change in version number. This makes it virtually impossible for Dependency-Track to distinguish actual vulnerabilities from those which have already been patched. This further negates the transparency argument. Backports can just as easily be injected with whatever code the maintainer wants. This creating further possibilities for a supply-chain attack.

5.2.4 Component management

Dependency-Track struggles to accurately assess the security impact of a range of commonly-used software modules. This is because they are categorized as embedded subcomponents of components instead of standalone products used in conjunction with other software. It becomes difficult to identify their significance when these kinds of modules are not clearly visible in the dependency hierarchy. Dependency-Track's approach to libraries is emblematic of this problem.

Lower-level system libraries are typically unrelated to top-level application services. One such example in the environment is the C library glibc. The library is commonly used on Linux systems. It has nothing to do with the upper-level applications providing the servers. This creates a false narrative around the nature of vulnerabilities in the component and produces considerable operational noise. Distorted perceptions of risk lead to inefficient prioritization and resource allocation. Figure 12 shows how these are dealt with in the environment.

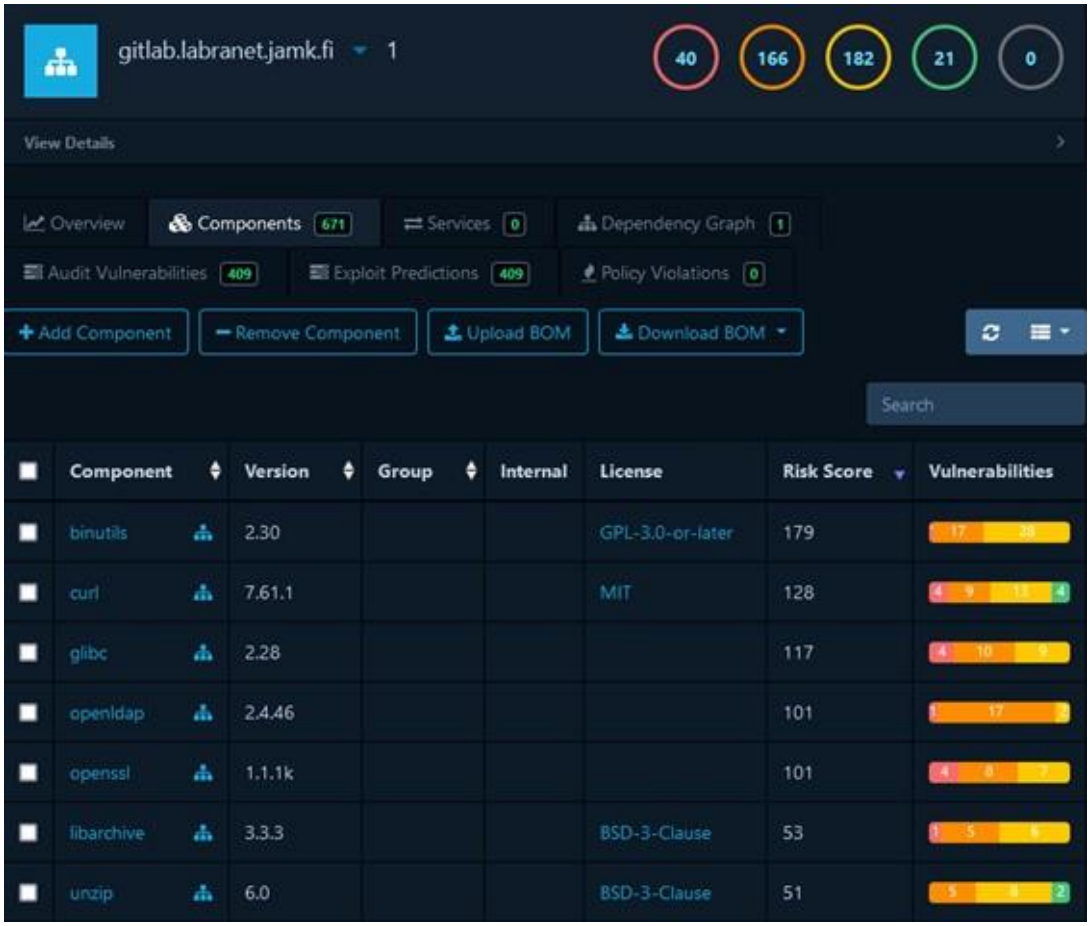

Figure 12. Vulnerable components

The nested SBOM structure makes it harder to detect and accurately identify the number of components using a given library. Tracking capabilities at the module and sub-component levels is a somewhat hit-or-miss affair. Even widely-used components can be hard to identify as standalone entities. They instead appear as parts of larger structures. This makes component vulnerabilities harder to track even though they may be critical dependencies for a large number of applications in the network (Graham, 2019). Components can furthermore be statically built into software or dynamically loaded. For example, OpenSSL exists as a system-wide library but any software can ship with a differing version built in. Relying on versions reported by the system is not a critical necessity.

Figure 13 presents first-level dependencies for the Nginx HTTP server. But these dependencies do not include the enormous and interconnected subsystem of components necessary for it to function. Efficient sub-component management hinges upon highly granular tracking capabilities.

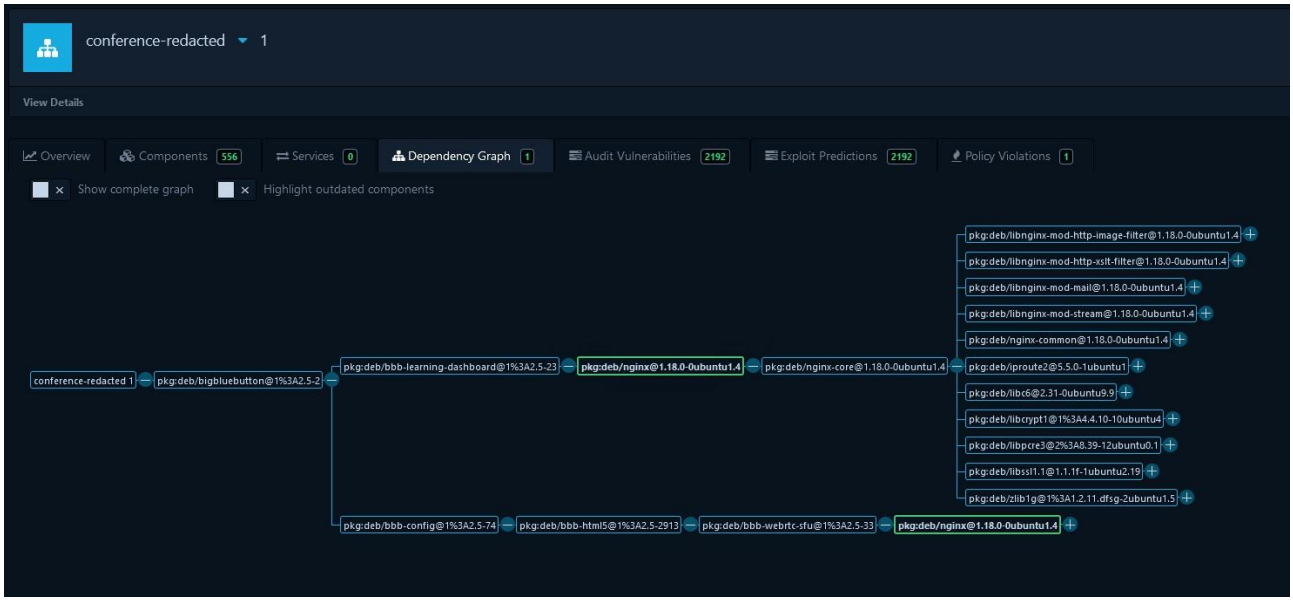

Figure 13. First-order Nginx dependencies without sub-components

5.2.5 Identifiers

Another problem identified in the environment concerns mismatches and delays to Common Platform Enumeration (CPE) identifiers. Discrepancies with upstream vulnerability identifiers can be attributed to several factors. Dependency-Track relies on a wide range of databases, repositories and feeds to compile vulnerability information (Dependency-Track, 2023). Differences in naming conventions, versioning or even the interpretation of the vulnerability are likely explanations for these inconsistencies. Vulnerability sources in Dependency-Track are not guaranteed to align cleanly with those of the NVD for this reason. Synchronization and standardization of data is likely to be an ongoing challenge.

The timing of updates adds another dimension of uncertainty to the vulnerability management process (Anwar et al., 2021; Ruohonen, 2019). Dependency-Track neither receives updates in real time nor at the same intervals as its data sources. Informational disparities are the inevitable result. The subjective nature in which vulnerability ratings are assigned further complicates the issue (Stenberg, 2023, 2023a). Mismatches of this nature can only be addressed manually on a piecemeal basis. Continuous efforts in data normalization are therefore required. The preponderance of these mismatches imply Dependency-Track will not only report false positives; it may also report false *negatives*. This effectively forecloses the possibility of meaningful automation of risk-assessment processes. The problem of CPE mismatch is illustrated in Figures 14 and 15 (NVD, 2024).

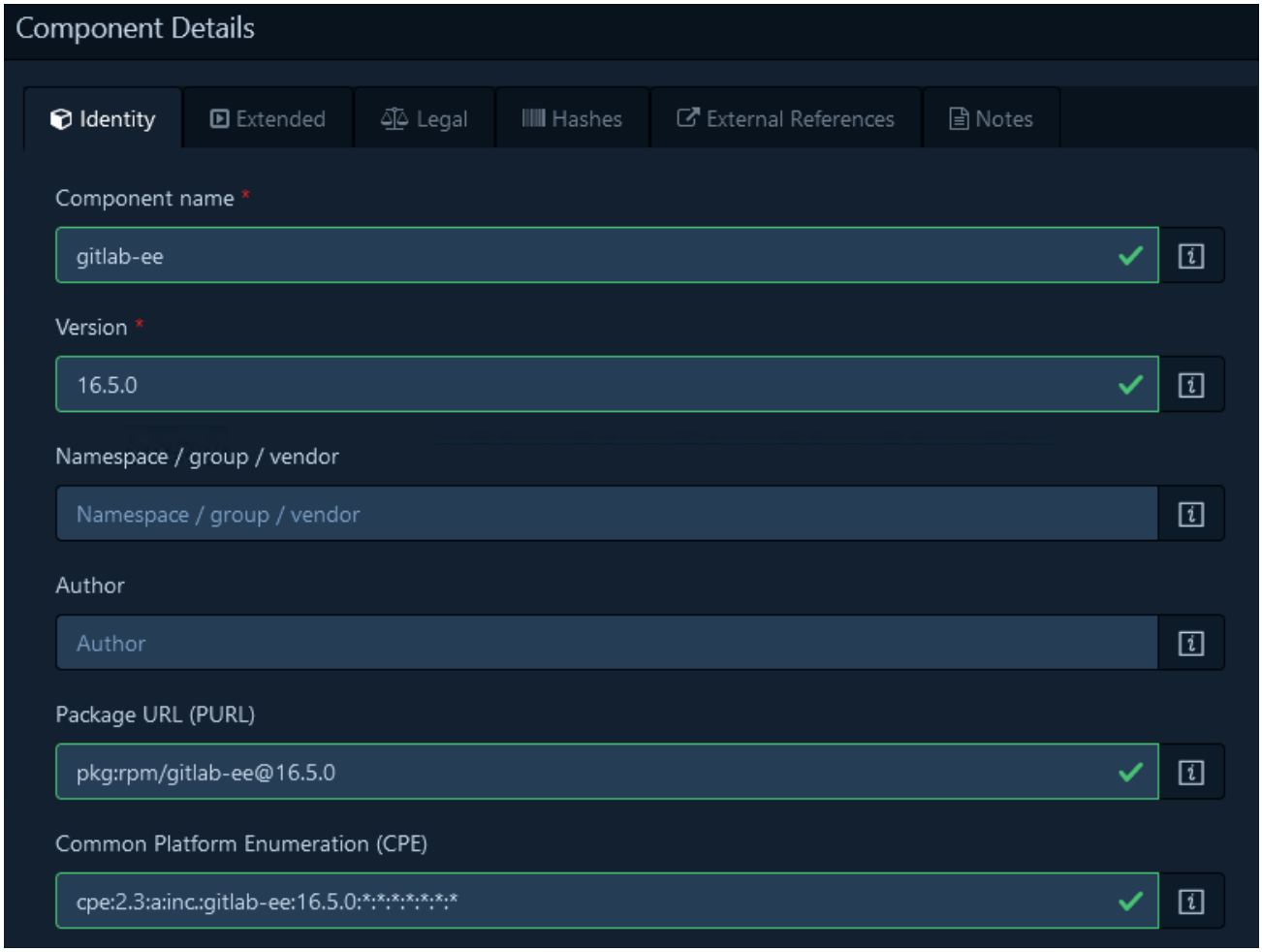

Figure 14. CVE-2023-7028 CPE mismatch in Dependency-Track

cpe:2.3:a:gitlab:gitlab:*:*:*:*:community:*:*:*	From (including)	Up to (excluding)
Show Matching CPE(s) • cpe:2.3:a:gitlab:gitlab:12.2.0:*:*:*:community:*:*:* • cpe:2.3:a:gitlab:gitlab:12.2.1:*:*:*:community:*:*:* • cpe:2.3:a:gitlab:gitlab:12.2.2:*:*:*:community:*:*:* • cpe:2.3:a:gitlab:gitlab:12.2.3:*:*:*:community:*:*:* • cpe:2.3:a:gitlab:gitlab:12.2.4:*:*:*:community:*:*:* • cpe:2.3:a:gitlab:gitlab:12.2.5:*:*:*:community:*:*:* • cpe:2.3:a:gitlab:gitlab:12.2.6:*:*:*:community:*:*:*	12.2.0	16.5.6

Figure 15. CVE-2023-7028 NVD CPE entry

5.2.6 Alerts

The challenges raised above have broad implications for the functioning of Dependency-Track's alert system. Inaccurate reporting leads inevitably to inaccurate alerting. False positives in the system means alerts will be raised which are inapplicable to the specific applications being

monitored. Automated systems in Dependency-Track do not contextualize information particularly well. The lack of contextual understanding produces alerts which do not accurately reflect the actual risk or impact of a given vulnerability.

Lack of integration between different tools and data-sources aggravates the problem. The convergence of these factors makes the alerting system less effective and more prone to generating noise. Alert fatigue is a significant cause for concern because it can potentially result in critical security issues going undetected or misdiagnosed. Alerts in the platform will require ongoing manual adjustments to be of any utility for DevSecOps.

5.3 Additional observations

The issues below also need to be accounted for when considering Dependency-Track as a tool for DevSecOps. These are discussed in more detail in the concluding remarks.

- A possible workaround for false positives could be using policy violation definitions to implement rules for generating alerts based on installed software
- Noise could be reduced by selectively uploading main applications as components to mitigate excessive vulnerability numbers (though this might produce false negatives)
- Limited applicability for offensive use without considerable manual intervention
- Improving the accuracy of CVE scoring mechanisms and CPE identifiers is critical in determining the tool's utility in real-world scenarios
- The platform is a more effective development and compliance tool and has limited utility for systems administration

5.4 Summary

There are significant challenges associated with Dependency-Track from a DevSecOps perspective. The core theme to emerge from this analysis revolves around the volume of inaccurate vulnerability data being reported. This manifests in a range of cross-cutting and mutually reinforcing arenas. Both false negatives and false positives can be expected. CPE mismatch and delayed matches are an additional problem. Such issues require considerable manual intervention. Inaccurate vulnerability reporting also means inaccurate alerting. This affects the utility of the alerting system more generally. The claim that SBOMs promote transparency in the software supply-chain is simply not credible this far down the supply-chain. A use-case exists on the development side. The case is weaker from administrative and DevSecOps perspectives.

The core strength of the platform appears to lie in its policy management and compliance functionality. This makes it possible to weed out unwanted versions. It is a useful one-stop-shop to search for specific CVEs potentially affecting the environment. The platform can be used to check if known unfixed/unmitigated CVEs exist in the environment or has affected a server. This might be complicated by the presence of undocumented backports, however.

6 Conclusion

6.1 Findings

6.1.1 Overview

Dependency-Track cannot be considered a comprehensive solution for managing vulnerabilities in dynamic software ecosystems. The core problem observed in the production environment relates to the platform's ability to accurately track and report vulnerability findings. This casts doubt on its utility for DevSecOps. SBOMs are far from a panacea for robust security practices. They not only fail to bring transparency to the supply chain; in many cases they achieve the opposite.

It is unclear if SBOM-based management platforms offer any practical advantages over infrastructure management via active scanning. But the news is not all bad. Some features in Dependency-Track work well and can potentially be incorporated into management workflows. Whether these capabilities are sufficient to make up for the platforms numerous deficiencies is an open question. To paraphrase Morpheus in the 1999 sci-fi classic, *The Matrix*: How far down does the rabbit-hole go? The answer is: Not nearly deep enough.

6.1.2 The good

Policy management emerged surprisingly as one of Dependency Track's most useful features. Policy management was far from the focus of the study. The platform is also a useful aggregator of vulnerability data from a wide range of sources. Another potential use-case could be to verify the existence of unfixed or unmitigated CVEs on servers. The tool's ability to retrospectively check if a specific CVE had previously affected a server is suitable for historical vulnerability analysis. The caveat is this capability can potentially be hamstrung by unaccounted-for backporting in the network.

6.1.3 The bad

The analysis identified critical drawbacks. Accuracy and depth of reporting are areas of concern. There is a notable lack of granularity in vulnerability data. This affects how Dependency-Track interprets patch levels, backports and the relationships between low-level libraries and top-level applications. A lack of depth has the effect of reducing supply chain transparency. The irony of this

does not go unnoticed: Achieving supply chain transparency is the *raison d'être* of the SBOM project writ large. This raises fundamental questions concerning the overall efficacy of SBOM-based vulnerability management for DevSecOps.

SBOMs were expected to enhance visibility into the software supply chain. The study revealed the opposite. Instances of false positives, false negatives and challenges in distinguishing active vulnerabilities from background noise are recurring themes. Dependency-Track provides neither a comprehensive nor accurate representation of the software components and their dependencies.

Potential workarounds to these issues include cherry-picking services inventories in the system to mitigate the challenges posed by excessive vulnerability numbers. But this is a less-than ideal solution. There is a risk of overlooking critical dependencies even in the best-case scenario. The findings illustrate the inherent difficulties in achieving a comprehensive and granular understanding of software components in production environments.

6.1.4 The ugly

It is fair to say the challenges uncovered in the analysis are significant. But they are not insurmountable. The main obstacle lies in the substantial investment of time and resources required to address these issues effectively. Nearly all potential solutions necessitate continuous manual intervention to yield meaningful and sustainable results. This is the principal constraint imposed on the efficiency and scalability of infrastructure management via SBOM. Whether it is worth the effort is ultimately a question of individual judgment.

The question of upstream identifier inaccuracy is another vexing problem. This is inherent to neither SBOMs in general nor Dependency-Track in particular. It is instead connected to aggregation and categorization methods used by external data sources. Accurate labeling of CVEs will not always be guaranteed.

A connected issue is how organizations like the NVD assess vulnerability. A 9.8 severity rating does not always correspond to the same level of local risk. Severity is in large part connected to organizational methodologies. The disparate nature of data sources further complicates Dependency-Track's aggregation processes. This is a universal problem affecting all infrastructure management systems. It is simply too difficult and costly to collect all relevant data in one place. Fragmentation is the inevitable result.

6.2 Results

6.2.1 Questions

Table 8 presents the research outcomes. There are clear limitations and challenges associated with achieving meaningful levels of supply-chain transparency in the SBOM context. Issues encountered include inaccuracies, false positives and limited capacity to distinguish legitimate vulnerabilities from those with backported fixes. These issues were widespread. All required manual fixes. Expectations of a more efficient identification and remediation process relative to active scanning are not realistic.

Table 8. Research outcomes

	Question	✓	✖
1	*Do SBOMs increase transparency in the software supply-chain?*		✖
2	*Do SBOM-based systems perform more efficient identification and remediation of vulnerabilities than active scanning?*		✖
3	*Do SBOM management platforms improve the efficiency and accuracy of workflows in vulnerability identification?*		✖
4	*Can SBOMs be leveraged for use in red team exercises?*		✖

The environment is flexibly configurable but it is unclear how much monitoring of the constitutive platform elements would be required over the long term. These problems achieve the opposite effect: The extent of manual intervention required implies an accuracy deficit in automated processes. The inevitable consequence is an overall *reduction* in the efficiency of vulnerability identification workflows. These concerns can perhaps be addressed by rethinking the way in which SBOMs are ingested by and accessed in Dependency-Track. But the platform in its current state has limited relevance for DevSecOps. These findings suggest SBOMs will have limited utility for red teams in cyber exercise scenarios.

6.2.2 Hypothesis

The study hypothesized that *SBOM-based platforms enhance vulnerability management and security across DevSecOps workflows.* This hypothesis cannot be confirmed. Certain features indeed show potential for streamlining vulnerability management. But efficiency gains in these areas are unlikely to compensate for the considerable manual work necessary to manage infrastructure using these methods.

6.3 Limitations

Variability in network configurations, software stacks and operational practices across the two environments could introduce factors which are challenging to control. This is worth considering because of the specific nature of the environment being tested. The study was intentionally limited in scope. This means additional variables or contextual nuances not explicitly considered may also impact research outcomes. Caution should therefore be exercised when generalizing the results to a broader context.

6.4 Future development

The Software Bill of Materials concept holds great promise. Its failure to live up to this promise reflects the general immaturity of the technologies designed to use them. This leaves open the possibility for development in a range of areas. The principal goal should be to refine how SBOM-based platforms like Dependency-Track parse the voluminous data it is tasked with managing. Reducing reporting inaccuracies is critical to its overall utility for DevSecOps. The way the platform interprets the cacophonic tidal-wave of module and component data needs to be rethought. One possible solution could be exploring more sophisticated algorithmic solutions to improve the precision of vulnerability analysis. Developers need to give serious thought to how manual interventions can be reduced in the environment without deleterious effects on accuracy.

A major functionality gap identified in the analysis concerns the integration of known backported fixes into data sources. There is a pressing need to explore and implement mechanisms which seamlessly incorporate information regarding backported security fixes into the platform. Such integration would promote a more comprehensive and accurate view of the software landscape.

Investigating ways to integrate baselines derived from known operating system patches into Dependency-Track will improve accuracy as well as reduce the need for manual tinkering. This requires SBOM integration into the operating system update ecosystems. Dependency-Track's capacity to track and manage vulnerabilities associated with specific patches could also stand to be improved. The issue of network security is not going away any time soon. The goals associated with the SBOM project are worthy pursuits. But it is not quite ready for prime-time.

7 Sources

Ajienka, N., Capiluppi, A., & Counsell, S. (2017, November). Managing hidden dependencies in OO software: A study based on open-source projects. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (pp. 141-150). IEEE. https://doi.org/10.1109/ESEM.2017.21

Akiyama, M., Shiraishi, S., Fukumoto, A., Yoshimoto, R., Shioji, E., & Yamauchi, T. (2023). Seeing is not always believing: Insights on IoT manufacturing from firmware composition analysis and vendor survey. *Computers & Security*, *133*, 103389. https://doi.org/10.1016/j.cose.2023.103389

Anchore. (2022, April 20). *Open-source container security with Syft & Grype*. Retrieved January 18, 2024, from https://anchore.com/opensource/

Anwar, A., Abusnaina, A., Chen, S., Li, F., & Mohaisen, D. (2021). Cleaning the NVD: Comprehensive quality assessment, improvements, and analyses. IEEE Transactions on Dependable and Secure Computing, 19(6), 4255-4269. https://doi.org/10.1109/TDSC.2021.3125270

Al-Badareen, A. B., Selamat, M. H., Jabar, M. A., Din, J., & Turaev, S. (2010, November). Reusable software components framework. In *European Conference of Computer Science (ECCS 2011)* (pp. 126-130). Puerto De La Cruz: NAUN. https://dl.acm.org/doi/10.5555/1961414.1961435

Allman, M., Paxson, V., & Terrell, J. (2007, October). A brief history of scanning. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement* (pp. 77-82). https://doi.org/10.1145/1298306.1298316

Arnott, D., & Pervan, G. (2012). Design science in decision support systems research: An assessment using the Hevner, March, Park, and Ram Guidelines. *Journal of the Association for Information Systems*, *13*(11), 1. https://doi.org/10.17705/1jais.00315

Arora, A., Wright, V. L., & Garman, C. (2022). *SoK: A framework for and analysis of Software Bill of Materials tools* (No. INL/JOU-22-68388-Rev000). Idaho National Laboratory (INL).

Arora, A., Wright, V. L., & Garman, C. (2022a). Strengthening the security of operational technology: Understanding contemporary bill of materials. *Journal of Critical Infrastructure Policy*, *3*(INL/JOU-22-66300-Rev001). https://doi.org/10.18278/jcip.3.1.8

Ashby, M. F., Shercliff, H., & Cebon, D. (2018). *Materials: Engineering, science, processing and design*. Butterworth-Heinemann.

Barclay, I., Preece, A., Taylor, I., & Verma, D. (2019). Towards traceability in data ecosystems using a bill of materials model. *arXiv preprint arXiv:1904.04253*. https://doi.org/10.48550/arXiv.1904.04253

Barnett, R. J., & Irwin, B. (2008, October). Towards a taxonomy of network scanning techniques. In *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: Riding the wave of technology* (pp. 1-7). https://doi.org/10.1145/1456659.1456660

Barrett, R., Kandogan, E., Maglio, P. P., Haber, E. M., Takayama, L. A., & Prabaker, M. (2004, November). Field studies of computer system administrators: analysis of system management tools

and practices. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work* (pp. 388-395). https://doi.org/10.1145/1031607.1031672

Bauer, A., Harutyunyan, N., Riehle, D., Schwarz, GD. (2020). Challenges of tracking and documenting open-source dependencies in products: A case study. In: Ivanov, V., Kruglov, A., Masyagin, S., Sillitti, A., Succi, G. (eds) *Open-source systems. IFIP advances in information and communication technology*, vol 582. Springer, Cham. https://doi.org/10.1007/978-3-030-47240-5_3

Benz, C. R., & Newman, I. (2008). *Mixed methods research: Exploring the interactive continuum*. SIU Press.

Bicaku, A., Schmittner, C., Tauber, M., & Delsing, J. (2018, May). Monitoring industry 4.0 applications for security and safety standard compliance. In *2018 IEEE Industrial Cyber-Physical Systems (ICPS)* (pp. 749-754). IEEE. https://doi.org/10.1109/ICPHYS.2018.8390801

Booth, W., Colomb, G., & Williams, J. (2008). *The craft of research*. University of Chicago Press.

Borrego, M., Douglas, E. P., & Amelink, C. T. (2009). Quantitative, qualitative, and mixed research methods in engineering education. *Journal of Engineering education*, 98(1), 53-66. https://doi.org/10.1002/j.2168-9830.2009.tb01005.x

Burns, B. (2018). *Designing distributed systems: Patterns and paradigms for scalable, reliable services*. O'Reilly Media, Inc.

Carmody, S., Coravos, A., Fahs, G., Hatch, A., Medina, J., Woods, B., & Corman, J. (2021). Building resilient medical technology supply chains with a software bill of materials. *npj Digital Medicine*, *4*(1), 34. https://doi.org/10.1038/s41746-021-00403-w

Case, J. M., & Light, G. (2013). Emerging research methodologies in engineering education research. *Journal of Engineering Education*, *100*(1), 186-210. https://doi.org/10.1002/j.2168-9830.2011.tb00008.x

Chen, J., Nair, V., Krishna, R., & Menzies, T. (2018). 'Sampling' as a baseline optimizer for search-based software engineering. *IEEE Transactions on Software Engineering*, 45(6), 597-614. https://doi.org/10.1109/TSE.2018.2790925

Chen, Y., & Sun, X. H. (2006, September). Stas: A scalability testing and analysis system. In *2006 IEEE International Conference on Cluster Computing* (pp. 1-10). IEEE. https://ieeexplore.ieee.org/document/4100388

Cichon, W. (2021, August 17). *Securing SDLC - Vulnerability in 3rd party components*. LinkedIn. Retrieved January 17, 2024, from https://www.linkedin.com/pulse/securing-sdlc-vulnerability-3rd-party-components-wojciech-cichon/

Creswell, J.W. (2002*). Research design: Qualitative, quantitative and mixed method approaches*. Sage Publications.

Coffey, K., Smith, R., Maglaras, L., & Janicke, H. (2018). Vulnerability analysis of network scanning on SCADA systems. *Security and Communication Networks*, *2018*. https://doi.org/10.1155/2018/3794603

Cox, J., Bouwers, E., Van Eekelen, M., & Visser, J. (2015, May). Measuring dependency freshness in software systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*

(Vol. 2, pp. 109-118). IEEE. https://doi.org/10.1109/ICSE.2015.140

Crnkovic, I., Sentilles, S., Vulgarakis, A., & Chaudron, M. R. (2010). A classification framework for software component models. *IEEE Transactions on Software Engineering*, *37*(5), 593-615. https://doi.org/10.1109/TSE.2010.83

Deb, D., Dey, R., & Balas, E. (2019). *Engineering Research Methodology: A Practical Insight for Researchers: Vol. 153*. Springer. https://doi.org/10.1007/978-981-13-2947-0

Dependency-Track | Software Bill of Materials (SBOM) analysis. (2023). OWASP. Retrieved January 17, 2024, from https://dependencytrack.org/

Duboc, L., Rosenblum, D., & Wicks, T. (2007, September). A framework for characterization and analysis of software system scalability. In *Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering* (pp. 375-384). https://doi.org/10.1145/1287624.1287679

Edgar, T., Niddodi, S., Rice, T., Hofer, W., Seppala, G., Arthur-Durett, K., Engels, M., & Manz, D. (2019). Safer and optimized vulnerability scanning for operational technology through integrated and automated passive monitoring and active scanning. *Journal of Information Warfare*, 18(4), 125–155. https://www.jstor.org/stable/26894697

Eggers, S. L., Simon, T. B., Morgan, B. R., Bauer, E. S., & Christensen, D. (2022). *Towards a software bill of materials in the nuclear industry* (No. INL/RPT-22-68847-Rev000). Idaho National Laboratory (INL. https://doi.org/10.2172/1901825

Eickelmann, N. S., & Richardson, D. J. (1996, March). An evaluation of software test environment architectures. In *Proceedings of IEEE 18th International Conference on Software Engineering* (pp. 353-364). IEEE. https://doi.org/10.1109/ICSE.1996.493430

Elbadawi, K., & Yu, J. (2011, July). Improving network services configuration management. In *2011 Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN)* (pp. 1-6). IEEE. https://doi.org/10.1109/ICCCN.2011.6006050

Executive Office of the President. (2021). *Executive Order 14028 on Improving the Nation's Cybersecurity*. Retrieved from https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity

Foreman, P. (2019). *Vulnerability management*. CRC Press.

Gandhi, R., Germonprez, M., & Link, G. J. (2018, January). Open data standards for open-source software risk management routines: An examination of SPDX. In *Proceedings of the 2018 ACM International Conference on Supporting Group Work* (pp. 219-229). https://doi.org/10.1145/3148330.3148333

Ghosh, A., Nashaat, M., & Miller, J. (2019). The current state of software license renewals in the IT industry. *Information and software technology*, *108*, 139-152. https://doi.org/10.1016/j.infsof.2019.01.001

Graham, R. (2019, January 14). *Software Bill of Materials (SBOM) - Does it work for DevSecOps?* Retrieved January 21, 2024, from https://cybersecurity.att.com/blogs/security-essentials/software-bill-of-materials-sbom-does-it-work-for-devsecops

Haff, G. (2020). *The state of enterprise open source*. Red Hat. Retrieved January 17, 2024, from https://www.redhat.com/rhdc/managed-files/rh-enterprise-open-source-report-detail-f21756-202002-en.pdf

Herman, M. (2023). *cool-projects/sbom at main · mherman-fi/cool-projects*. GitHub. Retrieved January 26, 2024, from https://github.com/mherman-fi/cool-projects/tree/main/sbom

Hendrick, S. (2022). *Software Bill of Materials (SBOM) and Cybersecurity Readiness.* The Linux Foundation. Retrieved January 13, 2024, from https://8112310.fs1.hubspotusercontent-na1.net/hubfs/8112310/LF%20Research/State%20of%20Software%20Bill%20of%20Materials%20-%20Report.pdf

Hutton, J. L., & Williamson, P. R. (2000). Bias in meta-analysis due to outcome variable selection within studies. *Journal of the Royal Statistical Society Series C: Applied Statistics*, 49(3), 359-370. https://doi.org/10.1111/1467-9876.00197

Holsing, N. F., & Yen, D. (1999). Software asset management: Analysis, development and implementation. *Information Resources Management Journal (IRMJ)*, *12*(3), 14-26. https://doi.org/10.4018/irmj.199907010

Höfer, A., & Tichy, W. F. (2007). Status of empirical research in software engineering. In *Empirical Software Engineering Issues. Critical Assessment and Future Directions: International Workshop, Dagstuhl Castle, Germany, June 26-30, 2006. Revised Papers* (pp. 10-19). Springer. https://doi.org/10.1007/978-3-540-71301-2_3

Jayawickrama, W. (2006, October). Managing critical information infrastructure security compliance: A standard based approach using ISO/IEC 17799 and 27001. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"* (pp. 565-574). Berlin, Heidelberg: Springer. http://dx.doi.org/10.1007/11915034_125

Jones, R., & Tate, L. (2023). Visualizing Comparisons of Bills of Materials. *arXiv preprint arXiv:2309.11620*. https://doi.org/10.48550/arXiv.2309.11620

Kloeg, B., Ding, A. Y., Pellegrom, S., & Zhauniarovich, Y. (2024). Charting the Path to SBOM Adoption: A Business Stakeholder-Centric Approach. In *19th ACM ASIA Conference on Computer and Communications Security (ACM AsiaCCS), July 1–5, 2024, Singapore* (pp. 1–14). ACM. Retrieved January 13, 2024, from https://zhauniarovich.com/publication/2024/kloeg2024charting/kloeg2024charting.pdf

Klukken, P., Parsons, J. R., & Columbus, P. J. (1997). The creative experience in engineering practice: Implications for engineering education. *Journal of Engineering Education*, 86(2), 133–138. https://doi.org/10.1002/j.2168-9830.1997.tb00276.x

Krsul, I. V. (1998). *Software vulnerability analysis*. Purdue University.

Laan, S. (2017). *IT infrastructure architecture: Infrastructure building blocks and concepts third edition*. Lulu.com.

Larabel, M. (2020). *GitStats - linux*. Phoronix. Retrieved January 30, 2024, from https://www.phoronix.com/misc/linux-eoy2019/index.html

Larabel, M. (2020a). *GitStats - systemd*. Phoronix. Retrieved January 30, 2024, from https://www.phoronix.com/misc/systemd-eoy2019/index.html

Lau, K. K. (2006, May). Software component models. In *Proceedings of the 28th International Conference on Software Engineering* (pp. 1081-1082). https://doi.org/10.1145/1134285.1134516

Li, X., Chang, X., Board, J. A., & Trivedi, K. S. (2017, January). A novel approach for software vulnerability classification. In *2017 Annual Reliability and Maintainability Symposium (RAMS)* (pp. 1-7). IEEE. https://doi.org/10.1109/RAM.2017.7889792

Lipner, S. (2010). Security development lifecycle. *Datenschutz Und Datensicherheit - Dud*, 34(3), 135–137. https://doi.org/10.1007/s11623-010-0021-7

Malkawi, M., Özyer, T., & Alhajj, R. (2021, November). Automation of active reconnaissance phase: an automated API-based port and vulnerability scanner. In Proceedings of the 2021 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (pp. 622-629). https://doi.org/10.1145/3487351.3492720

Martin, R. A. (2020, July). Visibility & control: Addressing supply chain challenges to trustworthy software-enabled things. In *2020 IEEE Systems Security Symposium (SSS)* (pp. 1-4). IEEE. https://doi.org/10.1109/SSS47320.2020.9174365

Martínez, J., & Durán, J. M. (2021). Software supply chain attacks, a threat to global cybersecurity: SolarWinds' case study. *International Journal of Safety and Security Engineering*, *11*(5), 537-545.

McKenzie, C. R. (2004). Hypothesis testing and evaluation. *Blackwell handbook of judgment and decision making*, 200-219. Blackwell.

National Telecommunications and Information Administration. (2021, April 28). *Software component transparency*. United States Department of Commerce. Retrieved January 12, 2024, from https://www.ntia.gov/other-publication/2021/ntia-software-component-transparency

National Vulnerability Database. (2024). CVE-2023-7028. Retrieved January 26, 2024, from https://nvd.nist.gov/vuln/detail/CVE-2023-7028#range-10238564

Nguyen, P., Durlauf, S., & Tikalsky, M. (2023). *Software Bill of Materials: A Catalyst to a More Secure Software Supply Chain* (Doctoral dissertation, Acquisition Research Program). https://dair.nps.edu/handle/123456789/5026

NTIA Multistakeholder Process on Software Component Transparency Framing Working Group. (2019). *Framing Software Component Transparency: Establishing a Common Software Bill of Material (SBOM)*. United States Department of Commerce. Retrieved January 13, 2024, from https://www.ntia.gov/files/ntia/publications/framingsbom_20191112.pdf

Paik, I., & Park, W. (2005). Software component architecture for an information infrastructure to support innovative product design in a supply chain. *Journal of organizational computing and electronic commerce*, *15*(2), 105-136. https://doi.org/10.1207/s15327744joce1502_2

Perez, J. (2023). *State of open-source report: Open-source usage, market trends & analysis*. Openlogic by Perforce. Retrieved January 17, 2024, from https://www.openlogic.com/system/files/openlogic-2023-state-of-open-source-report.pdf

Perlman, R. (2000). *Interconnections: Bridges, routers, switches, and internetworking protocols*. Addison-Wesley Professional.

Pospisil, O., Blazek, P., Fujdiak, R., & Misurec, J. (2021, November). Active scanning in the industrial control systems. In *2021 International Symposium on Computer Science and Intelligent Controls (ISCSIC)* (pp. 227-232). IEEE. https://doi.org/10.1109/ISCSIC54682.2021.00049

Puppet. (2024). Welcome to Puppet Bolt. Retrieved January 18, 2024, from https://www.puppet.com/docs/bolt/latest/bolt.html

Rebaiaia, M. L., & Kadi, D. A. (2013). *Network reliability evaluation and optimization: Methods, algorithms and software tools* (Vol. 79, pp. 5-7). CIRRELT. Retrieved January 12, 2024, from https://www.cirrelt.ca/documentstravail/cirrelt-2013-79.pdf

Rezaei, R., Chiew, T. K., Lee, S. P., & Aliee, Z. S. (2014). Interoperability evaluation models: A systematic review. *Computers in Industry*, 65(1), 1-23. https://doi.org/10.1016/j.compind.2013.09.001

Robinson, M. A. (2010). An empirical analysis of engineers' information behaviors. *Journal of the American Society for information Science and technology*, 61(4), 640-658. https://doi.org/10.1002/asi.21290

Robinson, M.A. (2016). Quantitative Research Principles and Methods for Human-Focused Research in Engineering Design. In: Cash, P., Stanković, T., Štorga, M. (eds) *Experimental Design Research*. Springer, Cham. https://doi.org/10.1007/978-3-319-33781-4_3

Roberts, R. (2016). Understanding the validity of data: A knowledge-based network underlying research expertise in scientific disciplines. *Higher Education*, *72*, 651-668. https://doi.org/10.1007/s10734-015-9969-4

Ross, P. T., & Zaidi, N. L. B. (2019). Limited by our limitations. *Perspectives on Medical Education*, 8(4), 261–264. https://doi.org/10.1007/s40037-019-00530-x

Ruohonen, J. (2019). A look at the time delays in CVSS vulnerability scoring. *Applied Computing and Informatics*, 15(2), 129-135. https://doi.org/10.1016/j.aci.2017.12.002

Safianu, O., Twum, F., & B, J. (2016). Information System Security Threats and Vulnerabilities: Evaluating the human factor in data protection. *International Journal of Computer Applications*, 143(5), 8–14. https://doi.org/10.5120/ijca2016910160

Sangal, N., Jordan, E., Sinha, V., & Jackson, D. (2005, October). Using dependency models to manage complex software architecture. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (pp. 167-176). https://doi.org/10.1145/1094811.1094824

Santos, N., Rodrigues, R., & Ford, B. (2012, December). Enhancing the OS against security threats in system administration. In *ACM/IFIP/USENIX International Conference on Distributed Systems*

Platforms and Open Distributed Processing (pp. 415-435). Springer. https://doi.org/10.1007/978-3-642-35170-9_21

Security backporting Practice. (n.d.). Red Hat. Retrieved January 27, 2024, from https://access.redhat.com/security/updates/backporting

Sehgal, V. V., & Ambili, P. S. (2023, April). A Taxonomy and Survey of Software Bill of Materials (SBOM) Generation Approaches. In *Analytics Global Conference* (pp. 40-51). Springer. https://doi.org/10.1007/978-3-031-50815-8_3

Sibal, R., Sharma, R., & Sabharwal, S. (2017). Prioritizing software vulnerability types using multi-criteria decision-making techniques. *Life Cycle Reliability and Safety Engineering*, *6*, 57-67. https://doi.org/10.1007/s41872-017-0006-8

Software Identification Ecosystem Option Analysis. (2023). In *Cybersecurity and Infrastructure Security Agency*. CISA. https://www.cisa.gov/sites/default/files/2023-10/Software-Identification-Ecosystem-Option-Analysis-508c.pdf

Soomro, T. R., & Hesson, M. (2012). *Supporting best practices and standards for information technology.* Infrastructure Library. Retrieved January 14, 2024, from https://digitallibrary.aau.ac.ae/bitstream/handle/123456789/547/Supporting%20Best%20Practices%20and%20Standards%20for%20Information%20Technology%20Infrastructure%20Library.pdf?sequence=1&isAllowed=y

Statista. (2024, January 9). *Common IT vulnerabilities and exposures worldwide 2009-2024*. Retrieved January 17, 2024, from https://www.statista.com/statistics/500755/worldwide-common-vulnerabilities-and-exposures/

Stalnaker, T., Wintersgill, N., Chaparro, O., Di Penta, M., German, D. M., & Poshyvanyk, D. (2023). BOMs away! Inside the minds of stakeholders: A comprehensive study of bills of materials for software systems. *arXiv preprint arXiv:2309.12206*. https://doi.org/10.48550/arXiv.2309.12206

Stenberg, D. (2023, June 12). *NVD damage continued*. Retrieved January 21, 2024, from https://daniel.haxx.se/blog/2023/06/12/

Stenberg, D. (2023a, August 26). CVE-2020-19909 is everything that is wrong with CVEs. Retrieved January 21, 2024, from https://daniel.haxx.se/blog/2023/08/26/cve-2020-19909-is-everything-that-is-wrong-with-cves/

Stoddard, J. T., Cutshaw, M. A., Williams, T., Friedman, A., & Murphy, J. (2023). *Software Bill of Materials (SBOM) Sharing Lifecycle Report* (No. INL/RPT-23-71296-Rev000). Idaho National Lab (INL). https://doi.org/10.2172/1969133

Syafrizal, M., Selamat, S. R., & Zakaria, N. A. (2022). Analysis of Cybersecurity Standard and Framework Components. *International Journal of Communication Networks and Information Security (IJCNIS)*, *12*(3). https://doi.org/10.17762/ijcnis.v12i3.4817

Synopsys. (2022). *2022 Open-source security risk report*. Synopsis. Retrieved January 17, 2024, from https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-2022.pdf

Trim, P., & Lee, Y. I. (2016). *Cyber security management: a governance, risk and compliance framework*. Routledge.

U.S. Department of Commerce. (2021). *The minimum elements for a Software Bill of Materials (SBOM) pursuant to Executive Order 14028 on Improving the Nation's Cybersecu*rity. https://www.ntia.doc.gov/files/ntia/publications/sbom_minimum_elements_report.pdf

Vacca, J. R. (2007). Threats and Vulnerabilities. In *Practical Internet Security* (pp. 145–175). Springer. https://doi.org/10.1007/978-0-387-29844-3_7

van Kervel, S.J.H. (2011). High quality technical documentation for large industrial plants using an enterprise engineering and conceptual modeling-based software solution. In: De Troyer, O., Bauzer Medeiros, C., Billen, R., Hallot, P., Simitsis, A., Van Mingroot, H. (eds) Advances in Conceptual Modeling. Recent Developments and New Directions. ER 2011. *Lecture Notes in Computer Science*, vol 6999. Springer. https://doi.org/10.1007/978-3-642-24574-9_55

Williams, C. (2007). Research methods. *Journal of Business & Economics Research (JBER)*, 5(3). https://doi.org/10.19030/jber.v5i3.2532

Wirth, A. (2022). Log jam: lesson learned from the Log4Shell vulnerability. *Biomedical Instrumentation & Technology*, 56(3), 72–76. https://doi.org/10.2345/0899-8205-56.3.72

Xia, B., Bi, T., Xing, Z., Lu, Q., & Zhu, L. (2023). An empirical study on software bill of materials: Where we stand and the road ahead. *arXiv preprint arXiv:2301.05362*. https://doi.org/10.48550/arXiv.2301.05362

Zahan, N., Lin, E., Tamanna, M., Enck, W., & Williams, L. (2023). Software Bills of Materials are required. Are we there yet? *IEEE Security & Privacy*, *21*(2), 82-88. https://doi.org/10.1109/MSEC.2023.3237100

Appendices

Appendix 1. Bolt workflow