Hochschule Karlsruhe
University of
Applied Sciences

Fakultät für
Informatik und
Wirtschaftsinformatik

HKA

Master Thesis

Name:	Fabian Burth
Topic:	Design and Implementation of a Security and Compliance Data Lake
Place of work:	SAP SE, Walldorf
Supervisor:	Prof. Dr.-Ing. Vogelsang
Co-examiner:	Prof. Dr. Körner
Deadline:	16/02/2023

Karlsruhe, 17/08/2022

The Chairman of the examination committee

Prof. Dr. Heiko Körner

Statutory declaration

I declare that I have composed the master thesis myself and without use of any other than the cited sources and aids. Sentences or parts of sentences quoted literally are marked as such; other references with regard to the statement and scope are indicated by full details of the publications concerned. The thesis in the same or similar form has not been submitted to any examination body and has not been published. This thesis was not yet, even in part, used in another examination or as a course performance. Furthermore I declare that the submitted written (bound) copies of the master thesis and the version submitted in digital format are consistent with each other in contents.

(Place, Date) (Fabian Burth)

Abstract

The importance of software is constantly growing and so is its complexity. Thus, large-scale enterprise software systems are usually not developed completely from scratch. Commonly required functionality like logging or serialization is often provided by already existing *open source software (OSS)*. Hence, modern software systems are composed of several components. These components usually have individual version updates, vulnerabilities, and licenses. Version updates of a component may affect the compatibility with other components. Vulnerabilities in one specific component may compromise the security of the whole software system. Violating license agreements may lead to litigations due to copyright infringement. Furthermore, one must consider that each component itself may be composed of several other components.

So, maintaining and monitoring large-scale enterprise software systems is an important part of the *Application Lifecycle Management (ALM)* and poses a considerable challenge to software companies. The common way to tackle these risks nowadays is by incorporating *Software Composition Analysis (SCA)* tools into the application development process. These tools analyze applications and retrieve information like vulnerabilities, licenses, and *software bill of materials (SBOM)*. But this approach often still has its flaws. The information extracted by these tools is frequently treated like logs and hence of limited value for future usage. Additionally, in larger companies different development teams often come up with point-to-point solutions of integrating the tools tightly coupled to their CI/CD pipeline.

The main objective of this thesis is the design and prototypical implementation of a *Security and Compliance Data Lake (SCDL)*, which provides a standardized way of integrating even multiple different SCA tools loosly coupled to CI/CD pipelines and to store the extracted information. By offering an *Application Programming Interface (API)*, it then should enable consumers to query this information on different levels of aggregation to answer questions that might not even have been known at the time the SCA was performed. A recent and popular example for such a question is "Which components contain log4j?".

This *Security and Compliance Data Lake* will build upon the *Open Component Model (OCM)*, an open standard to describe the SBOM with so called *Component Descriptors*. These *Component Descriptors* also describe how to access sources and resources. It thereby provides an entry point for the execution of SCA tools.

Contents

List of Figures

List of Tables

Acronyms

PoC proof of concept

SCDL Security and Compliance Data Lake

Chapter 1

Introduction

> *"[T]he trust we place in our digital infrastructure should be proportional to how trustworthy and transparent that infrastructure is"*
> Executive Order on Improving the Nation's Cybersecurity [1]

1.1 Motivation

The introductory quote above initially sounds pretty intuitive and self evident. Probably everyone in the IT industry would be able to agree on this. Yet, the panic and public outcry unleashed in the software industry after the vulnerability in Log4j was discovered, a popular and widely used logging library, goes to show how far the statement strives from reality. The vulnerability is rated with the highest severity possible since it enables *Remote Code Execution (RCE)*, which in other words allows an attacker to run any code on the machine using Log4j [2]. After the discovery, IT specialists all over the world had to identify which of their applications and systems were using vulnerable versions of the library before they could even start dealing with the vulnerability. This increases the response time and subsequently the risk exposure significantly. But what is it, that makes this such a difficult task?

Modern software systems are composed of numerous software components, such as Log4j, and these components may themselves be composed of other software components and so on. This adds several layers of complexity obfuscating the dependencies of a software system. The manufacturing industry has been dealing with such issues in their supply chains for years. But these companies usually maintain close relationships and contracts with their suppliers, perhaps even exchanging *bills of materials (BOM)*. Thus, the companies can easily keep track of each individual part in their products by accumulating the BOMs of their suppliers and they will even be informed about problems affecting this parts by their suppliers. This is different

from the software industry. Especially when it comes to *open source software (OSS)*, companies do not have a contract with the supplier. On the contrary, they frequently might not even know the maintainer of the software component. Subsequently, they also will not be informed about issues such as vulnerabilities with the components.

The software industry has come up with approaches to deal with this issue and implemented measures to proactively monitor applications and detect known vulnerabilities in its components. With growing complexity of software and changing development and deployment landscapes, these approaches are not sufficient and fail to answer questions such as which systems use vulnerable versions of Log4j. Why and how they fail in these situations will be further examined in the course of this thesis. But as "[s]oftware is eating the world"[3] and thus, as companies, devices of our everyday life, cars and even medical and military devices rely on software, as the impact of software on our privacy and also physical security increases, so does the responsibility of companies providing software and the gap between trust and transparency becomes less acceptable.

In an effort to address this issue, the US government has also published an Executive Order on Improving the Nation's Cybersecurity, which will require every company supplying software to the government to provide an *Software Bill of Materials (SBOM)* [1, 4]. As previously discussed, this is only possible if every company in the supply chain provides an SBOM for their software components. Therefore, the downstream impact of this Executive Order will most likely affect the entire industry.

1.2 Goals

The goals of this thesis aim to provide a research based practical contribution to closing the gap between the trust and transparency placed into our digital infrastructure [1].

To make this more concrete, the main goal of this thesis is to develop a *proof of concept (POC)* implementation of a *Security and Compliance Data Lake(SCDL)*. As the name already suggests, this is an application for storing information about software components. Such information might be about the occurrence of known vulnerabilities, as already indicated in the previous paragraphs, but also about licenses or dependencies. This kind of information about software components will be called metadata from here on.

Thus, the goal is to design and develop a central application for storing and querying software metadata, which is able to cope with the complexity and require-

ments of modern development and deployment landscapes. Therefore improving the transparency of the entire software supply chain and enabling companies to answer questions such as which applications, systems or even landscapes in a company contain a certain vulnerability.

Thereby, this thesis also contributes to the knowledge concerning metadata stores and respective API design by answering following research question:

> *What are the challenges arising during the development of a database application for the central metadata management of enterprise software systems and how may these challenges be solved?*

Furthermore, the application may support companies in fulfilling the requirements resulting from the recently announced Executive Order.

1.3 Scope

As the goal section already stated, this thesis is about designing and implementing a central application for software metadata management and its integration into modern development and deployment landscapes.

It is not about developing new ways of vulnerability detection. It is rather about monitoring systems and keeping track of already known vulnerabilities. This is just as important since a large amount of security incidents are caused by attackers exploiting such known vulnerabilities [5]. Therefore, neither does this thesis compete with current security testing techniques which are used to find vulnerabilities in applications [6], nor does it discuss these in detail. From the perspective of the central metadata store, each security testing technique applied may be viewed as a potential data source. But storing known vulnerabilities is still just one, although probably the most popular, use case of this central metadata store. A major design goal was to keep it extensible, so that all kinds of metadata may be stored.

1.4 Environment

This thesis is written in cooperation with SAP. As of today, with a total revenue of €27.34 billion, SAP is the third largest software company in the world after Microsoft and Oracle [7]. While also having gained some attention with *Business to Customer (B2C)* products like the *Corona-Warn-App*, its core products are *Business to Business (B2B)* enterprise software solutions. Originally, SAP grew around its *Enterprise Resource Planing (ERP)* system. Today, the company is also adopting *Internet*

of Things (IoT) technologies and *Artificial Intelligence (AI)* to provide advanced analytics for its customers and to maximize the value of its software products [8]. Besides, SAP is also investing heavily to push its products and customers to the cloud. Therefore, the company maintains partnerships with Amazon, Microsoft, Google and Alibaba as infrastructure providers.

The department this thesis is written with is developing and maintaining the *SAP Gardener*. SAP Gardener is *SAP's own managed Kubernetes service* which enables SAP itself as well as SAP customers to ship their applications to all of these different infrastructures using a unified deployment underlay. Since SAP Gardener offers the possibility to configure practically every detail, neither SAP nor its customers need to rely on the managed Kubernetes service of each hyperscaler with their individual perks and restrictions, but can leverage the functionality of a fully configurable kubernetes cluster without actually having to fully configure it [9].

1.5 Structure of the Thesis

In order to achieve the goals laid out before, the thesis builds upon following structure:

Foundation: In this chapter, basic terminology and existing concepts in software metadata management are established.

State of the Art: In the previous motivation and goal sections, it is already mentioned that existing measures for monitoring software component metadata such as vulnerabilities are insufficient in some cases. Therefore, to further motivate this thesis, this chapter will briefly introduce the state of the art approach and point out its limitations. The identification of this problems is crucial for the successful design of the Security and Compliance Data Lake.

Context at SAP Gardener: As mentioned, this thesis is written with the SAP Gardener team. It is greatly inspired by the problems they are facing and by the approaches they already implemented to overcome limitations of the state of the art. Thus, this chapter introduces their proposed standard, the Open Component Model, and its capabilities. Based on that, their development and deployment landscape is explained. Finally, an architecture for integrating a central metadata store such as the one designed and developed in this work into their current environment and thereby overcoming all these limitations is presented.

System Design: The system design chapter covers the conception of the data model, the selection of a database and the design of the API. Thereby, it especially focuses on giving detailed information about the ideas and motives that lead to specific design decisions. Besides, alternatives that have been considered and the respective reasons to not follow through with them will be an essential part of this section.

Implementation: Although the implementation is a major and time intensive part of the thesis, this chapter is kept short. The code base has grown quite big and discussing it in detail would be out of scope. Thus, this chapter focuses on explaining the high level architecture and solely examines some particularly interesting parts of the implementation. To make this tangible, it is accompanied by some showcasing of the actual functionality.

Results: Finally, the design decisions and the implementation as well as the knowledge gained through this work is evaluated and the research question is revisited.

Chapter 2

Foundations

This chapter provides the basic knowledge necessary to understand the following chapters as well as other research and literature regarding metadata management. It therefore explains several important concepts and abbreviations around *Software Identification*, *Open Source*, *Open Source Licensing* as and *Vulnerability Management*. Furthermore *Software Bill of Materials* and existing *Software Bill of Material Standards* are introduced. This is important to understand the requirements of the Executive Order mentioned in the introduction and to put everything into context.

2.1 Software Identification

Uniquely identifying a piece of software is a frequent concern when managing applications on enterprise scale. Thus, a standardized naming convention would be required. Unfortunately, pretty much every packet manager and tool uses its own convention.

Package URL (purl)
Package URL, or purl for short, is one of the most widely adopted attempts to standardize those existing approaches. As described in the projects GitHub Repository, it therefore specifies an URL string, a purl, which should be able "to identify and locate a software package in a mostly universal and uniform way across programming languages, package managers, packaging conventions, tools, APIs and databases" [10]. This Package URL consists of seven components [10]:

```
scheme:type/namespace/name@version?qualifiers#subpath
```

Listing 2.1: Package URL

```
pkg:docker/cassandra@sha256:244fd47e07d1004f0aed9c
pkg:github/package-url/purl-spec@244fd47e07d1004f0aed9c
pkg:deb/debian/curl@7.50.3-1?arch=i386&distro=jessie
pkg:npm/foobar@12.3.1
```

Listing 2.2: Package URL Examples

Each component is separated by a different specific character to allow for unambiguous parsing. The `scheme` (Required) is the constant value "pkg" which may be officially registered as an URL scheme in the future. `type` (Required) refers to a package protocol such as maven or npm. `namespace` (Optional) may be some type-specific prefix such as a Maven groupid, a Docker image owner or a Github user or organization. The `name` (Required) and `version` (Optional) are the name and version of the software. `qualifiers` (Optional) is also type-specific and may be used to provide extra qualifying data such as an OS, architecture or a distribution. With the `subpath` (Optional) one may specify a subpath within a package, relative to the package root [10].

Another convention to uniquely identify packages is the *Common Package Enumeration (CPE)* format which is primarily used in the context of vulnerability management ecosystem. Therefore, the standard will be discussed in the following section.

2.2 Open Source Software and Licensing

Open Source Software is widely used in modern software development. As to what Open Source Software actually is, the *Open Source Initiative* defined a set of rules, the *Open Source Definition* specifying distribution terms that the license of a software must comply with. Without going into detail and examining all of these terms, this definition generally ensures that such software "can be freely accessed, used, changed, and shared (in modified or unmodified form) by anyone" [11].

This description on its own may give the impression that there is no need to keep track of what *OSS Licenses* are used within an enterprise application. But there are still some limitation that *OSS Licenses* can put to the usage, especially the distribution of the software. Specifically, the so called *copyleft* principle and corresponding licenses might pose a challenge to some companies. While OSS generally may be used for commercial purposes, this principle dictates that if a company or individual distributes the software or a derivative, it has to be done under the same license it has been received under [11].

In the context of this work, OSS will be used within other OSS components, so *copyleft* is not an issue. But anyway, there is still a risk involved in using OSS. A company might distribute initial software versions under an OSS License until there is some kind of customer lock-in and then switch to another more restrictive license for further versions. Also, since there might not exists precedent cases regarding the legal interpretation of some specific OSS licenses, there is a risk of expensive law suits.

2.3 Vulnerability Management

Due to the widespread use of OSS, vulnerability management is an increasingly important topic. Since many organizations use the same software components, vulnerabilities often become publicly known. Also, the open source nature allows attackers to get precise information about the vulnerability itself. Thus, tracking publicly known vulnerabilities in an organizations software products is a crucial capability to keep them secure.

Vulnerability
According to the National Institute of Standards and Technology (NIST) a vulnerability is "A weakness in the computational logic (e.g., code) found in software and hardware components that, when exploited, results in a negative impact to confidentiality, integrity, or availability. Mitigation of the vulnerabilities in this context typically involves coding changes, but could also include specification changes or even specification deprecations (e.g., removal of affected protocols or functionality in their entirety)."[12]

Common Vulnerabilities and Exposures (CVE)
CVE was created by MITRE in 1999. Initially, the acronym was meant to stand for *Common Vulnerability Enumeration*. As described by the authors of the original whitepaper, many security tools and advisories used their own vulnerability identifiers. In order to integrate several of these, one had to manually compare and eventually relate the vulnerabilities to each other. Thus, a common naming convention and common enumeration of vulnerabilities was needed [13].

To solve this issue, the *CVE Program* assigns unique identifiers, so called *CVE IDs*, to each vulnerability. All the identified vulnerabilities are then maintained as *CVE Records* in the *CVE List*. A minimal *CVE Record* consists of a *CVE ID*, a

description of the vulnerability and at least one public reference. It does not include technical data, information about risks, impacts or fixes. An example for such a CVE Record is shown in figure 2.1 below.

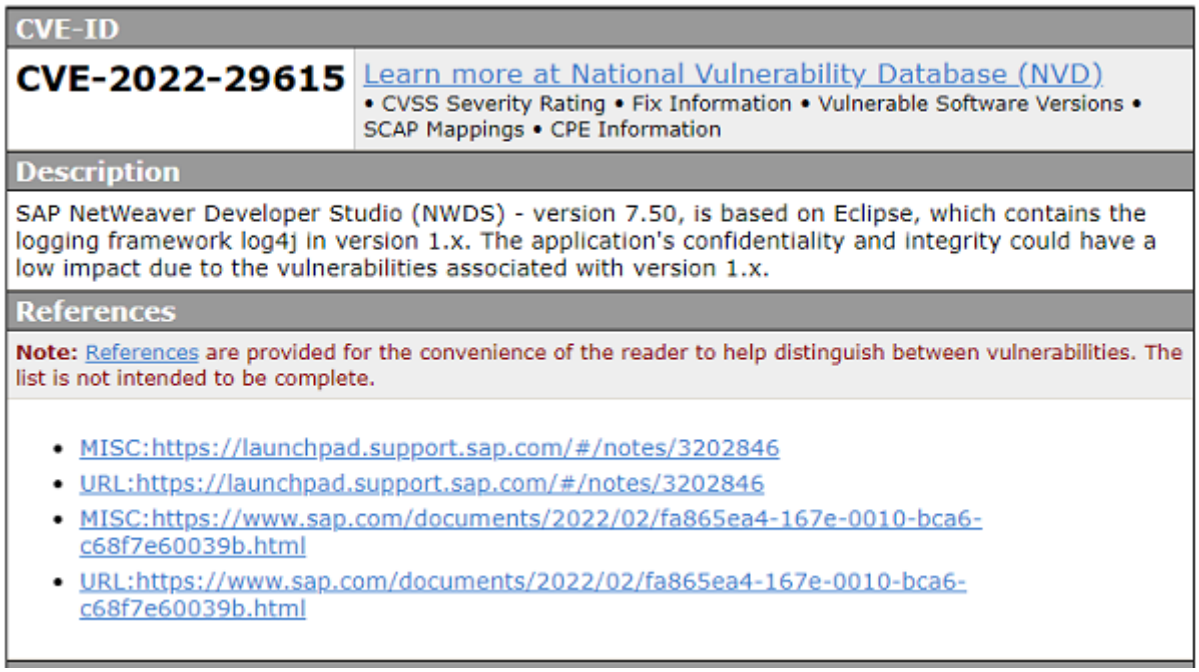

Figure 2.1: CVE Record
Source: Based on [14]

In order to make sure that all vulnerabilities listed in the *CVE List* are unique and maintained properly, *CVE IDs* can only be assigned and *CVE Records* can only be published by MITRE and several partner organizations, so called *CVE Numbering Authorities (CNA)*. Thus, to add a new vulnerability to the *CVE List*, the discoverer has to report it to a CNA [15].

Common Vulnerability Scoring System (CVSS)

CVSS is an open framework for rating vulnerabilities. It is owned and managed by the non-profit organization *Forum of Incident Response and Security Teams (FIRST)*. The framework captures the main characteristics of a vulnerability to produce a numerical score between 0.0 and 10.0 reflecting its severity.

Therefore CVSS is composed of three metric groups: Base, Temporal, and Environmental. The *Base Score* considers the intrinsic characteristics of a vulnerability that are constant over time and assumes the worst case impact across different environments. These characteristics take into account exploitability metrics like attack complexity but also impact metrics like confidentiality impact. The *Base Score* is typically calculated by the organization maintaining the vulnerable product or by

third party analysts on their behalf. The *Temporal Score* considers characteristics that may change over time but not across environments. Such a characteristic would be the maturity of exploit code. The *Environmental Score* considers the relevance of a vulnerability in a specific environment. *Temporal* and *Environmental Scores* are typically calculated by the consumers of the components to adjust the vulnerability rating to their organizations use case and environment. These Scores can then be used for internal risk management [16].

Common Weakness Enumeration (CWE)

CWE is a community-developed list of software and hardware weaknesses maintained by MITRE. Each weakness in the least is assigned a *CWE ID*. The list represents weaknesses on different levels of abstraction. This is conceptually shown by figure 2.2.

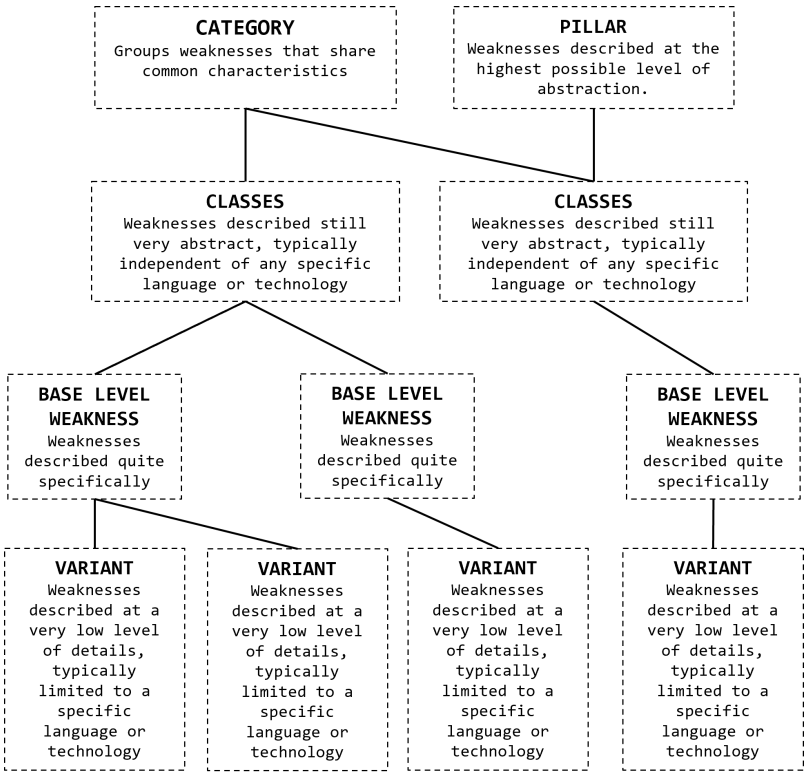

Figure 2.2: CWE Hierarchy
Source: Own Representation

So there exist relationships between elements on different levels of abstraction. As shown *Classes* are usually member of a *Category* and might also be the child, hence a more concrete description, of a *Pillar*. Additionally to what is shown in the figure, these relationships may also skip levels of hierarchy. Thus, a *Base Level Weakness* may be a direct member of a *Category* or a child of a *Pillar*. An example

for a Base Level Weakness is *CWE-478: Missing Default Case in Switch Statement* which is a member of *Bad Coding Practices* and a child of the class *Incomplete Comparison with Missing Factors*, which is a child of the pillar *Incorrect Comparison* [17].

Common Platform Enumeration (CPE): The CPE specification originally created by MITRE and now maintained by NIST provides a naming scheme for IT assets such as software. It may be used to uniquely determine a specific software and its version. This way a CPE enables cross referencing to other sources of information. The commonly used CPE naming scheme is structured as follows:

```
cpe:2.3: part : vendor : product : version : update : edition :
   language : sw_edition : target_sw : taget_hw : other
```

Listing 2.3: CPE Formatted String Binding

Thereby `part` may be *a* for applications, *o* for operating systems, and *h* for hardware devices. `edition` is a legacy attribute in the current version of the specification and may be omitted where not required for backward compatibility. The attributes after `edition` were newly introduced in this version and are referred to as *extended attributes*. `sw_edition` should characterize a particular market or class of users a product is tailored to (e.g. online), `target_sw` a software computing environment (e.g. linux), `target_hw` the instruction set architecture (e.g. x86), and `language` the language supported in the user interface [18].

National Vulnerability Database (NVD)

The NVD is a database of vulnerabilities owned and maintained by NIST. In the paragraph about CVE, it was mentioned that the *CVE Records* do not contain technical data, information about risks and impact, or fixes. The NVD feeds from the *CVE List* and uses the information provided in the *CVE Records* to perform further analysis. As a result, a NVD entry exists for each *CVE ID* and provides a *CVSS Base Score*, a *CWE ID* and a *CPE ID* [12].

Thus, NVD combines all the aforementioned standards and concepts to provide thorough and concise human and machine-readable information about vulnerabilities. *CPE IDs* identifying a particular software version in use may be queried against a NVD API to automatically check for known vulnerabilities. The *CVSS Base Score* is a valuable foundation for internal risk assessment and the *CWE ID* helps to quickly understand the type of a vulnerability.

2.4 Software Bill of Materials

A *Software Bill of Materials* is an inventory of the components used in a software. It ideally contains all direct and transitive components and their dependencies, so it is in other words pretty much the dependency graph of a software [19, 4].

As consequence to an *Executive Order on Improving the Nation's Cybersecurity*, the *National Telecommunications and Information Administration (NTIA)* published a document describing the minimum requirements for SBOMs [1, 4]). According to this document, these are:

Data Fields (Metadata)	Baseline information about each component: Supplier, Component Name, Version of the Component, Other Unique Identifiers, Dependency Relationship, Author of SBOM Data, Timestamp of SBOM creation
Automation Support	Automatic generation and machine-readability to allow for scaling across the software ecosystem.
Practices and Processes	Implementation of policies, contracts and arrangements to maintain SBOMs.

Table 2.1: Minimum Elements of a SBOM
Source: [4]

The goal of the *Data fields* is to sufficiently identify the components to track them through the supply chain and map them to other data sources, such as vulnerability and license databases. The *Automation Support* provides the ability to scale across the software ecosystem. The *Practices and Processes* ensure the maintenance by integration into the ALM. SBOMs thereby increase software transparency, providing those who produce, purchase and operate software the means to perform proper risk assessments [4].

Due to this Executive Order, SBOMs are now required for all U.S. federal software procurements. This does not only affect direct software vendors of the U.S. government [1]. As a consequence to this Executive Order, every organization that is downstream from the U.S. government in the supply chain may be required to provide SBOMs for its products. Thus, this will be a crucial capability for most software vendors.

There are three data formats mentioned in the minimum elements document which are interoperable, able to fulfill the requirements and either human- and

machine-readable. Those are the *Software Package Data eXchange (SPDX)*, *CycloneDX* and *Software Identification (SWID)* tags [4]. SAP uses yet another SBOM format, called Open Component Model (OCM), which does not fulfill the minimum requirements. The OCM will be discussed further in one of the following sections. SPDX is the most mature standard. It has laid out a lot of groundwork for the more recent CycloneDX. Thus, to get an better and more concrete understanding of SBOMs, SPDX will be examined in more detail.

SPDX License List and License Identifiers

SPDX is an initiative founded in 2010 and hosted at *The Linux Foundation*. In 2021 the SPDX specification even became an ISO standard [20]. The initiative focuses on solving challenges regarding the licenses and copyrights associated with software packages. SPDX therefore assembles licenses and exceptions commonly found in OSS in the *SPDX License List*. More precisely, this list includes a standardized short identifier, the full name, the license text, and a canonical permanent URL for each license and exception. By incorporating this *SPDX License Identifiers* in source on file level, one enables automation of concise license detection, even if just parts of an OSS project are used. Furthermore, SPDX provides *Matching Guidelines* to ensure that e.g. a "BSD 3-clause" license in a LICENSE file of an OSS project with different capitalization or usage of white space than the master license text included in the *SPDX License List* is still identified as "BSD 3-clause" license.

SPDX Documents

At the heart of the SPDX initiative are the *SPDX Documents* which leverage the *SPDX License List* and *SPDX License Identifiers* to describe the licensing of a set of associated files, referred to as *Package* in the context of SPDX. A *SPDX Document* provides means to describe information about the document creation, the package as a whole, individual files, snippets of code within an individual file and other licenses that are not contained in the *SPDX License List* but are still relevant for the package, relationships between *SPDX Documents*, and annotations, which in a way are comments within an *SPDX Document*. The concept of relationships is a rather new addition to the specification. It is particularly useful if one has an SPDX Document describing a binary. Explicitly capturing relationships like "generated from" these source files and "dynamically linking" these libraries allows for a complete licensing picture.

These documents may be represented in one of the following five file format: tag/value (.spdx), JSON (.spdx.json), YAML (.spdx.yaml), RDF/xml (.spdx.rdf),

and spreadsheets (.xls) [21, 22].

To give a more concrete idea of the basic concepts of *SPDX Documents*, an example from the SPDX GitHub repository will be briefly examined [23]. Therefore figure 2.3 below shows the directory structure of a “Hello World” project in C.

Figure 2.3: C Project Directory Structure
Source: [23]

Listing 2.4 shows a corresponding *SPDX Document*. Some tag:value pairs which are less relevant for the overall understanding are deliberately omitted to contain the length of the example.

```
SPDXVersion: SPDX-2.2
DataLicense: CC0-1.0
SPDXID: SPDXRef-DOCUMENT
DocumentName: hello
DocumentNamespace: https://swinslow.net/spdx-examples/example1/hello-v3
Creator: Person: Steve Winslow (steve@swinslow.net)
Created: 2021-08-26T01:46:00Z

##### Package: hello
PackageName: hello
SPDXID: SPDXRef-Package-hello
PackageDownloadLocation: git+https://github.com/swinslow/spdx-examples.git#example1/content
PackageLicenseConcluded: GPL-3.0-or-later
PackageLicenseInfoFromFiles: GPL-3.0-or-later
PackageLicenseDeclared: GPL-3.0-or-later
PackageCopyrightText: NOASSERTION

FileName: /build/hello
SPDXID: SPDXRef-hello-binary
FileType: BINARY
LicenseConcluded: GPL-3.0-or-later
LicenseInfoInFile: NOASSERTION
FileCopyrightText: NOASSERTION

FileName: /src/Makefile
SPDXID: SPDXRef-Makefile
FileType: SOURCE
LicenseConcluded: GPL-3.0-or-later
LicenseInfoInFile: GPL-3.0-or-later
FileCopyrightText: NOASSERTION

FileName: /src/hello.c
SPDXID: SPDXRef-hello-src
FileType: SOURCE
LicenseConcluded: GPL-3.0-or-later
LicenseInfoInFile: GPL-3.0-or-later
FileCopyrightText: Copyright Contributors to the spdx-examples project.

Relationship: SPDXRef-hello-binary GENERATED_FROM SPDXRef-hello-src
```

```
Relationship: SPDXRef-hello-binary GENERATED_FROM SPDXRef-Makefile
Relationship: SPDXRef-Makefile BUILD_TOOL_OF SPDXRef-Package-hello
```

Listing 2.4: SPDX Document

Most of the tag:value pairs are self-explanatory, but some might require some explanation. The *Concluded License* is the license the SPDX file creator has concluded as the governing license of a package or a file. *License Information from Files* contains a list of all licenses found in a package and the *Declared License* is the license declared by the authors of the package [22]. Additionally, listing 2.4 illustrates how the concept of relationships may be used.

It is also worth mentioning that the concept of *Packages* in SPDX as a set of associated files is really rather loose. Thus, describing the project in figure 2.3 as two separate packages, one for source and one for binary, optionally in the same or also in two separate *SPDX Documents* would be completely conform with the specification as well.

Since SPDX has been around for so long and is an accepted ISO standard, there exists a lot of useful tooling. It is therefore quite easy to automate tasks like producing, consuming, transforming and validating *SPDX Documents* [21].

2.5 Regulations

Vulnerabilities and licenses are not the only risks associated with enterprise software. Companies also need to consider governmental regulations since violations may lead to fines that have critical impact on the business.

Export Control Classification Number (ECCN)
The ECCN is a 5 character alpha-numeric designation used to determine whether an so called dual-use item needs an export license from the U.S. Department of Commerce in order to legally export it. Dual-use items are items that may be used for civil as well as military purposes. The ECCN gives some information about the product, as shown in figure 2.4 below.

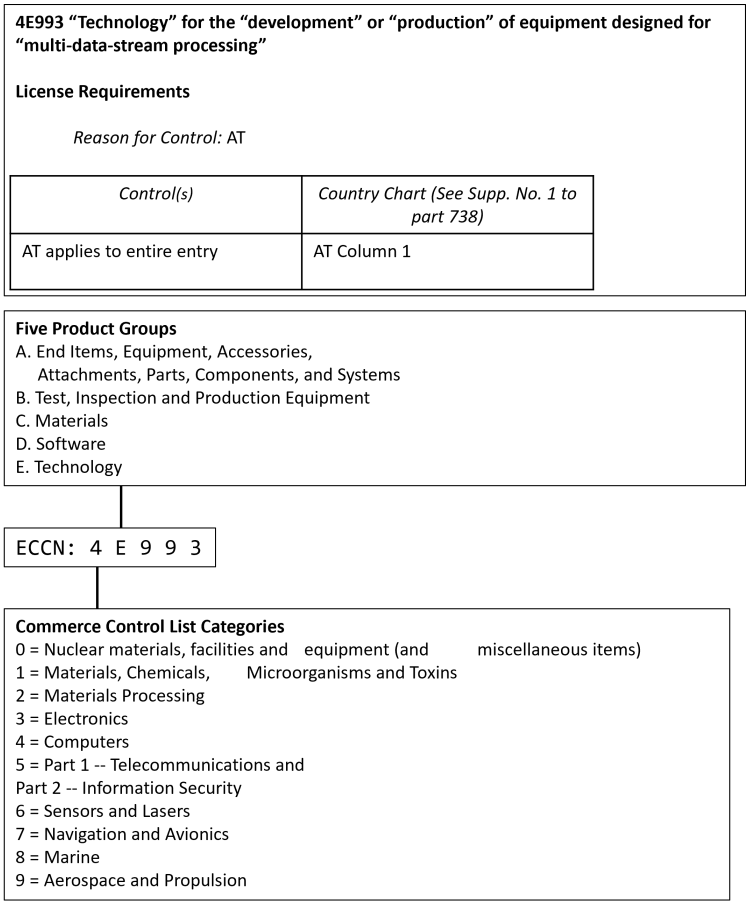

Figure 2.4: ECCN Structure
Source: Based on [24]

All ECCNs are listed in the *Commerce Control List (CCL)*. The entries in the list contain information about why an item might be under export control regulations. In the top of figure 2.4 is a snippet of such an entry. The reason for the export regulations of products with the classification number 4E993 is AT, which is the abbreviation for Anti-Terrorism [24].

Chapter 3

State of the Art

This chapter gives an overview of the state of the art approach of integrating security and compliance measures into the *software development life cycle (SDLC)*. That includes a brief introduction of practices such as *Continuous Integration and Continuous Delivery (CI/CD)* and *DevOps*. In the end, the limitations of this current approach are discussed, thereby further motivating this thesis.

3.1 Software Development Life Cycle

Since the emerging of cloud technologies, the software industry, especially also the major software companies like Microsoft, Amazon and SAP shifted their business model from *software as a product (SaaP)* to *software as a service (SaaS)*. This gave companies the opportunity to frequently release updates without adhering to a rigid distribution cycle. So they could react to feedback much faster and improve their software continuously [25, 26].

To keep up with this new pace, the SDLC had to be adjusted as well. So developers started to *continuously integrate (CI)* their code after they conducted some changes while at the same time automatically testing the software. Thus, always keeping it up to date and in a shippable state.

As an extension of that idea, the software is also automatically and *continuously delivered (CD)* to testing or even production environments, accelerating the process even more [27].

A quick side note - also after reviewing some literature, there seems to be some disagreement whether to distinguish continuous delivery and continuous deployment [27, 25]. The articles and papers that do differentiate define continuous delivery as automatically delivering to production-like environments for evaluation and testing. But there are still some manual steps necessary to actually deploy into production

or customer environments [27, 28, 29].

DevOps, a combination of "development" and "operations", is a practice which also resulted from this change of the SDLC. Since the concept of CI/CD merges areas of development and operations, the corresponding roles were combined to reduce communication overhead and misunderstandings [29]. DevOps is usually reliable for the setup and maintenance of the *CI/CD Pipeline*. Hence, they automate the steps required for the continuous integration and delivery, such as building and testing the software after a developer integrated his code changes and moving it to the next step after the previous tests and checks are passed.

3.2 Integrating Security and Compliance Measures

In practice, in order to conduct CI/CD, DevOps usually sets up a so called *CI/CD Pipeline* for the project. This includes a set of tools to automate build, test and deployment of the software. The most popular and widely used example is Jenkins [30]. But essentially, CI/CD pipelines are just simple stage-gate systems. Thus, the process is divided into a number of stages. Between each stage is a quality gate, such as a set of automated tests. The software has to pass this quality gate in order to being able to move to the next stage [31].

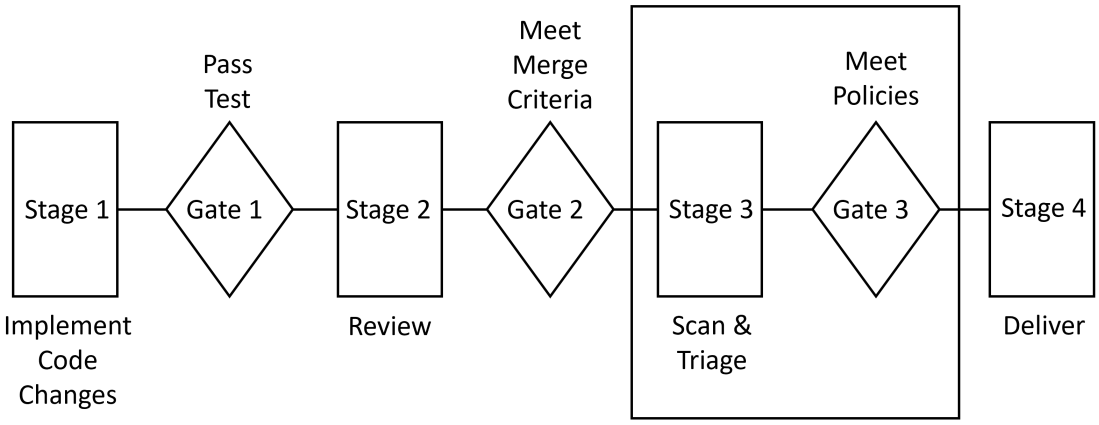

Figure 3.1: CI/CD Pipeline as Stage-Gate System
Source: Based on [31]

Figure 3.1 is an abstract technology agnostic illustration of a CI/CD pipeline as a stage-gate system. This also shows the currently established state of the art approach to reduce software supply chain risks such as problematic licenses or vulnerabilities in dependencies. Security and compliance scans are conducted as part of the CI/CD pipeline. The results of these scans may also be triaged before they are finally checked against a defined set of policies, forming another quality gate *before delivery*.

Triage is a term derived from medicine where it describes the classification and prioritization of patients if there are not enough resources to treat them all immediately. In software testing or scanning, it refers correspondingly to the process of (re-)rating and (re-)classifying issues found during the scan in the context of its occurrence within the particular project. Common issues include problematic licenses or vulnerabilities. For the latter, the triage may be done based on the CVSS introduced in the previous chapter. Policies may then prevent shipping of software that contains vulnerabilities with a CVSS that exceeds a certain threshold or specific blacklisted licenses.

Unfortunately, there is not a lot of academic literature on this specific topic to back up the claim that this is the state of the art approach. Therefore, for the purpose of this work, the offerings of the major vendors of corresponding tools, hence *Software Composition Analysis (SCA)* tools were deemed as a suitable source. SCA tools usually scan source code repositories or binary files, analyze the dependencies and match them against vulnerability or license databases.

According to a a market research of Forrester from 2021, based on a combination of market presence, current offering and strategy, these are Mend (formerly known as WhiteSource), Synopsys, Sonatype and Snyk [32]. All of these tools advertise their smooth integration into the CI/CD pipeline, to be more precise into the version control systems and build tools, as another quality gate [33, 34, 35, 36]. This should in general be a suitable representation of what the industry is requesting, although it is to mention that most of these vendors also offer other integration options. Another popular one for the SCA tools that scan binary files are the artifact repositories.

Additionally, it is undeniable that it makes a lot of sense to integrate these scans as another quality gate as part of the CI/CD pipelines into the repositories. At this point, all the files of a project are easy accessible for these tools. Besides, it is a commonly known principle in process management, that a defect becomes more expensive the later it is found.

3.3 Producing Software Bill of Materials

As a solution to increasing the supply chain transparency, the US government decided that analogous to other industries their software vendors have to provide SBOMs. Subsequently, the NTIA conducted research and published several documents on the topic. One of those being the minimum elements for SBOMs discussed in the foundations chapter. As mentioned there, one of the minimum requirements is

automation support. Therefore, the NTIA conducted a survey of existing SBOM formats and standards considering how to integrate them into the SDLC [37].

In this document the NTIA also suggests to leverage existing tools such as version control systems, build tools, code scanners and binary analysis tools to generate SBOMs [37]. Thus, the tools already integrated into the CI/CD pipelines as established in the previous paragraphs. Since these SCA tools obtain some of the most important SBOM information like dependencies and licenses anyway, they can also provide these results in a specific SBOM format. Therefore, all of the previously mentioned SCA tool vendors quickly adjusted and do provide such options now.

3.4 Limitations

This approach is intuitive and might be sufficient for the scope of a single development team which deploys its software to a single infrastructure. But the limitations really start to show if this scope is exceeded.

While the quality gate approach is not wrong, it is insufficient. Once the software passed the corresponding quality gate, there are no more scans. But new vulnerabilities in one of the dependencies may still be discovered after this point in time. These would then go undetected until the respective quality gate is reached again with the next version of the software. Besides, the main software that is being developed might depend on additional third party software at runtime. Consequently, this third party software has to be delivered alongside the main software. But the third party software might never pass any of the companies CI/CD pipelines and thus never pass the quality gates. Therefore, it would never be scanned and its vulnerabilities would again go undetected. Furthermore, with the quality gate approach, the scans are frequently also treated as such. Thus, developers hope for their code to pass the scan and discard the results of it does. Finally, with this approach, each development team has to configure, integrate and maintain this scanning tools on their own in each of their pipelines. In larger companies, this often also leads multiple opinionated CI/CD pipelines with additional point to point integration for such compliance tools and individual reporting dashboards as an attempt to overcome the existing limitations of this approach.

To solve these issues, the *compliance scans have to be decoupled from the CI/CD pipeline*. A quite popular solution to this is to integrate the them with the artifact repositories. Thereby, even vulnerabilities in components that do not pass an internal CI/CD pipeline are detected. But of course this only works with the binary scanners, which usually provide less precise results. Besides, this might also get tedious and

end up in multiple point to point solutions in cases where the artifacts are distributed over several different artifact repositories. Another possible option is based on the production of SBOMs during the compliance scans. When initially passing the quality gate, an SBOM can be generated, which may be used as access point to conduct further compliance scans. The tools of Mend and Synopsys already provide such an option. The problem here is, that the components in the generated SBOMs usually have tool specific identifiers. Therefore, each tool can only conduct scans based on their own SBOM. So both of this approaches are not ideal, and even if they were, there would still be a problem.

One might imagine a situation where the artifacts, which comprise the final product and are built from different internal as well as third party repositories, are stored in multiple different artifact repositories. From these repositories the artifacts are deployed into multiple different production environments. But naturally, there are different versions of the final product which are comprised of different versions of the artifacts. Given a scan detects a vulnerability in a specific version of an artifact in the artifact repository, how would one go about telling which environments contain this particular version of the artifact? Of course, the information about which product version contains what version of the artifacts is stored somewhere, the SBOMs for example. And the information where this product version is deployed is most certainly also stored somewhere else. But ultimately, there is no easy way to answer this question without skimming through this different data sources. This concludes the final and central issue.

Software metadata is distributed over the entire software development life cycle. Thus, there is a need for a central application for storing and querying software metadata.

Chapter 4

Context at SAP Gardener

As mentioned in the introduction, this work is written in cooperation with SAP. In fact, the limitations mentioned in the previous chapter are deducted from the limitations the SAP Gardener team struggles with themselves. Subsequently, this chapter introduces the *Open Component Model (OCM)*, SAP Gardeners proposed standard to decouple the compliance scans from the CI/CD pipeline. Based on this knowledge, an overview of the development and deployment landscape of SAP Gardener is given. Finally, the suggested integration of the Security and Compliance Data Lake, as a central application for storing and querying software metadata, with the existing standard and landscape is presented. Thereby, this chapter provides a reference architecture on how to overcome the major limitations of the current state of the art approach.

4.1 Open Component Model

The OCM is an SBOM format created and used by SAP Gardener. It does not fulfill the minimum requirements as defined by the NTIA. But this is due to the fact that the OCM has a different focus than SPDX or CycloneDX. While those two were deliberately designed to be a bill of materials, thoroughly listing the inventory of a software, the OCM was specifically developed to decouple CI from CD and thereby overcome related limitations such as the ones mentioned in the previous chapter. Following is a technical explanation of the OCM deducted from the specification [38] and an internal presentation [39]. Even though this explanation focuses on understanding the rationale behind the design decisions of the proposed standard rather than technical completeness, it may still be a little hard to grasp at times. This is due to the abstract nature of the OCM. Therefore, emphasis and examples are used where possible. Also, while the textual description really explains the OCM as

the abstract model that it is, figure 4.1 below shows an actual *Component Descriptor*, the serialization format of the OCM. This may also support in understanding the abstract concepts.

```yaml
component:
  name: github.com/gardener/etcd-druid
  version: v0.15.0
  repositoryContexts:
    - baseUrl: eu.gcr.io/sap-se-gcr-k8s-private/cnudie/gardener/development
      componentNameMapping: urlPath
      type: ociRegistry
  provider: internal
  sources:
    - name: github_com_gardener_etcd-druid
      access:
        type: github
        repoUrl: github.com/gardener/etcd-druid
        ref: refs/tags/v0.15.0
        commit: b8a7ba5493134f5b3645363bfa830874065e3509
      version: v0.15.0
      extraIdentity: {}
      type: git
      labels:
        - name: cloud.gardener/cicd/source
          value:
            repository-classification: main
  resources:
    - name: etcd-druid
      version: v0.15.0
      type: ociImage
      access:
        type: ociRegistry
        imageReference: >-
          eu.gcr.io/sap-se-gcr-k8s-public/eu_gcr_io/gardener-project/gardener/etcd-druid:v0.15.0-mod1
      digest: null
      extraIdentity: {}
      relation: local
      labels: []
      srcRefs:
        - identitySelector:
            name: github_com_gardener_etcd-druid
            version: v0.15.0
          labels: []
  componentReferences:
    - name: etcd
      componentName: github.com/gardener/etcd-custom-image
      version: v3.4.13-bootstrap-8
      digest: null
      extraIdentity:
        imagevector-gardener-cloud+tag: v3.4.13-bootstrap-8
      labels:
        - name: imagevector.gardener.cloud/images
          value:
            images:
              - name: etcd
                repository: eu.gcr.io/gardener-project/gardener/etcd
                resourceId:
                  name: etcd
                sourceRepository: github.com/gardener/etcd-custom-image
                tag: v3.4.13-bootstrap-8
  labels: []
signatures: []
```

Figure 4.1: Component Descriptor
Source: Based on [38]

In the context of the OCM, a *Component* is a software intended for a purpose which is identified by a *globally unique name*. This definition is still pretty vague. But this is on purpose and it will become clear why throughout this section.
A *Component Version* is identified by the *globally unique name* of the corresponding *Component* and a *version*. A *Component Version* may contain *Component References* and *Artifact References*. Furthermore, it has a *Repository Context*, a *Labels* and a *Provider* property. *Repository Context* provides a formal description of the repositories that store a representation of this *Component Version* and how to access them. *Labels* is really just a container for additional metadata. It therefore is an array of objects with the properties *name* and *value*. The *name* is a string while *value* may be a string, array or map, arbitrarily nested. *Provider* specifies the company or organization providing the *Component Version*.
A *Component Reference* is a reference to another *Component Version*. This initially sounds simple but there is actually a pitfall. A *Component Reference* is not identified by the *Identity* of another *Component Version*, thus by the *globally unique name* and the *version*. Each *Component Reference* has a *Component Version-local Identity* which additionally to the *globally unique Identity* of the referenced *Component Version* contains a *name* and an *extra identity*. This is due to the fact that a *Component Reference* does not have a strict predefined semantic. An example might help to understand this. Imagine one *Component Version* describing a version of a web server and another *Component Version* describing a version of an entire landscape of a REST application (as already mentioned, the definition of a *Component* is very loose). Now the REST application *Component Version* may use the web server *Component Version* twice, once as a classic HTTP server and once as a HTTP load balancer. Therefore, one *Component Reference* could have the *name* "http server" and the other "load balancer". Due to this definition of *Component Reference Identity*, the OCM provides the semantic capabilities to express such a situation. If the *name* is also not sufficient to uniquely identify a *Component Reference* within a *Component Version*, perhaps because the same *Component Version* is used as a HTTP server twice, the *extra identity* may be used to provide further distinction. This is a flat map of key-value-pairs. Besides, every *Component References* may also have a container for additional metadata, thus a *Labels* property.
Artifact is used as an umbrella term for *Sources* and *Resources*. *Sources* are usually the input for the build process of *Resources*, typically some source code. Subsequently, *Resources* are usually built from *Sources* and are capable of doing something. Thus, *Resources* are typically executables or OCI Images. *Artifact References* have a *Component Version-local Identity*, similar to *Component References*. There is one

significant different between the two which may easily lead to confusion. In the *Component Reference Identity*, the *version* is part of the referenced *Component Versions Identity*. In the *Artifact Reference*, the *version* is only part of the *Artifact Reference Identity*. What this means in practice is, if the *version* within a *Component Reference* changes, it is actually referencing a different *Component Version*. On the other hand, if the *version* within a *Artifact Reference* changes, the referenced *Artifact*, so the executeable or OCI Images, may still be exactly the same. Reciprocal, even if the *version* is the same in two *Artifact References*, the referenced *Artifacts* may have different versions. So there is no concrete coupling between the physical *Artifact*, so the executeable or the OCI Image, and the *Artifact Reference*. The *version* here is just another property to semantically distinct *Artifact References* within a *Component Version*, like *name* and *extra identity*. As a consequence, a *Artifact Reference* with the *name* "web server and *version* "v1.2.8" could reference an nginx v1.1.3 executeable and another *Artifact Reference* within the same *Component Version* with the *name* "web server" and *version* "v2.0.0" could reference an apache v2.4.54 executeable. This is an extreme case, supposed to show that there actually is now real coupling. In practice, the *name* and *version* usually do correspond to the underlying *Artifact*. And to do further distinctions in cases where a *Component Version* references the same physical *Artifact* twice, as an example twice nginx v1.1.3, once as an executeable for ARM platforms and once for x86-based platforms, the *extra identity* is used. The actual reference or rather coupling to the executeable or OCI Image is provided and *access* properties of the *Artifact Reference* .The *access* property, which is a map of key-value-pairs which has a required *type* property, provides a formal description of how and where to access the *Artifact*. Thereby, the *type* defines the access method, usually by specifying the repository type. The other key-values pairs in the map then depend on that type and may reference a particular commit by specifying the repository URL and a commit hash. Furthermore, all *Artifact References* have a *type* and *Labels* property, and specifically *Resource References* additionally have a *digest*, *relation* and *srcRefs* property. As already mentioned, a *Resource* may be an executable or an OCI Image. This can be expressed by the *type*. In the case of *Sources*, the *type* usually refers to the kind of source code management system. The *digest* is an object defining a *hash algorithm*, a *normalization algorithm* and the *hash* of the *Resource*. The *relation* property may have the value "local" or "external". The value "local" means that the *Resource* is derived from *Sources* contained in this *Component Version*. These *Sources* may then be referenced in the *srcRefs* property. This is an array of objects with a map type property called *IdentitySelector*, specifying the *Identity of the corresponding Resource Reference*, and

Labels.

As stressed several times throughout the last paragraphs, both *Component References* as well as *Artifact References* have *Component Version-local Identities*. This means, while the *Identity* of *Components* and respectively *Component Versions* contains a *globally unique name*, which makes the whole *Identity* globally unique and therefore *Component Versions* globally uniquely identifiable, the *Identity* of *Component and Artifact References*, thus the combination of *name*, *component name*, *version* and extra identity or just *name*, *version* and possibly *extra identity* respectively, is only guaranteed to be unique within the scope of a *Component Version*. As a consequence, to globally uniquely address the abstract concepts of *Component and Artifact Reference*, the combination of *Component Version Identity* and *Component or Artifact Reference Identity* is necessary. Thus, the combination of the globally unique *component name* and the *component version* plus respective local *Component or Artifact Reference Identity*.
As mentioned and shown in the beginning of this section, to *serialize* this abstract concept, OCM defines a format called *Component Descriptor*. A *Component Descriptor* thereby is the serialized form of a *Component Version*. It may be expressed in YAML as well as JSON. Thus, this is *machine readable* representation of a *Component Version*. The explanation also sporadically mentioned some abstract data types, mainly to make the concepts more tangible. In cases the data type was omitted, one may assume it is a string. Nevertheless, some of these strings have to adhere to specific patterns. Such information was omitted on purpose to avoid clutter. To obtain this information and get a technically complete specification of the standard, refer to the official documentation [38].

4.2 Capabilities of the Open Component Model

After this abstract description, this brief section revisions the major concepts of and points out the most important capabilities enabled by the Open Component Model.

Components and *Component Versions* respectively are an abstract concept, defined in a way that allows to describe anything, from single resources to entire application landscapes. Due to the loose semantic of the *Component References*, *Component Versions* may even be used to form arbitrary groups of other *Component Versions*. Thus, a *Component Version* named "SAP Gardener CI/CD" could be used to group all *Component Versions* used by the SAP Gardener CI/CD team. But especially, *Component Version* are capable of describing configurations of deployment environments. The *Component References* and *Artifact References* are then spanning

a graph over all *Component Versions* and thereby indirectly over all *Artifacts* deployed in a deployment environment. By traversing this graph, it is therefore possible to get all *Resources* deployed in the environment. Since both, *Resources* as well as *Sources*, through the *access* property, provide a machine readable way to automatically find and download them, compliance scans with binary and even source code scanners may be conducted asynchronously, independent of any CI/CD pipeline.

There are some further aspects to the Open Component Model, that are not as important for this work, but are interesting nonetheless and therefore shall still be briefly discussed here. Besides enabling asynchronous compliance scans, the aforementioned concept allows for an entire holistic deployment automation based on a *Component Version* and thereby for a complete decoupling of CI from CD. A fact that was not explicitly mentioned before, while almost all properties of a *Component Descriptor* are immutable in the context of a *Component Version*, the *access* properties are exchangeable. This is especially important in complex development and deployment landscapes as it provides additional flexibility. Due to national restrictions, one might be obligated to deploy resources from artifact repositories located in the respective country. Hence, different *Component Repositories* may store *Component Descriptors* for a specific *Component Version* with different *access* properties. Thus, the *access* properties may be exchanged transparently without affecting the deployment automation.

While the Open Component Model thereby solves almost all limitations of the current state of the art approach, it does not provide the means to easily answer where Log4j is deployed. *Thus, there is still a need to store this information in a queryable manner alongside other possibly relevant metadata.*

4.3 Development and Deployment Landscape at SAP Gardener

So SAP Gardener is SAP‘s own managed Kubernetes service which enables to easily setup a kubernetes cluster in the cloud of one of the hyperscalers as well as on premise. In order to provide this service, the team has to maintain a sophisticated development and deployment landscape, which leverages the capabilities of the OCM. Therefore, this section provides a concrete example how to decouple CI from CD using the OCM.

A representation of the CI part of the development landscape is illustrated in figure 4.2 below.

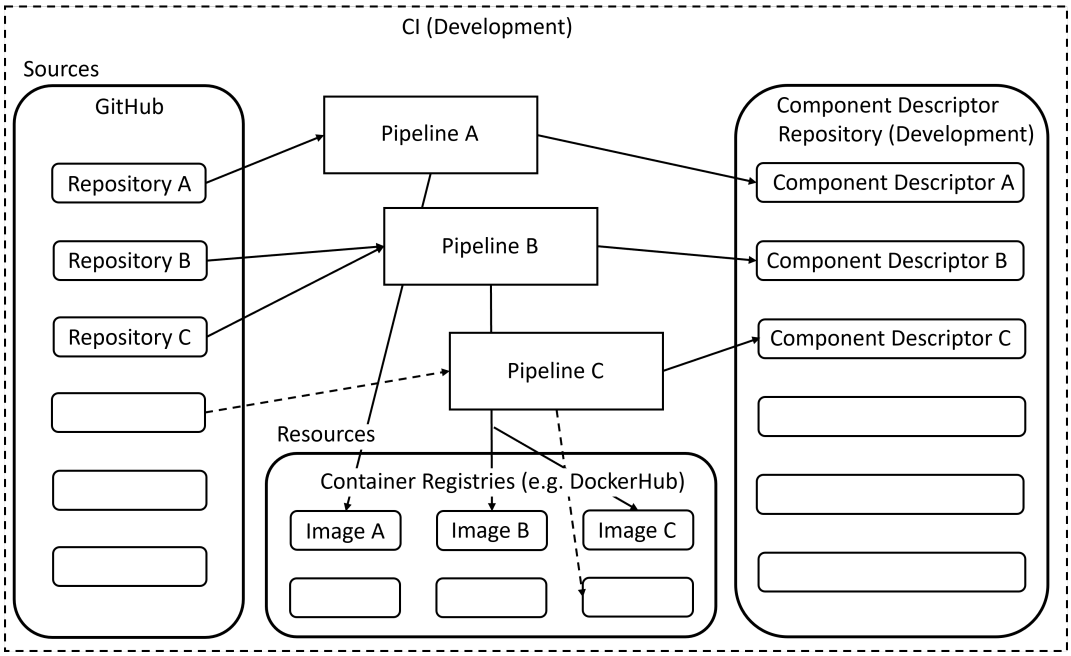

Figure 4.2: Gardener CI
Source: Based on [39]

As shown, there are several sources organized in git repositories hosted on GitHub. Furthermore, there are pipelines for creating the Component Descriptors. These pipelines may build OCI Images from the sources, as indicated by the unbroken lines to Pipeline A and Pipeline B. Instead of actually building, these pipelines may also just use references to existing open source OCI Images and optionally their corresponding sources, as indicated by the dashed arrows. In fact, more than half of the OCI Images powering SAP Gardener are only referenced and not built within the pipeline. Finally, as a result of the pipeline processing, the Component Descriptors are published into the Developments Component Descriptor Repository.

The next illustration in figure 4.3 is a representation of how this CI system is connected to the CD system.

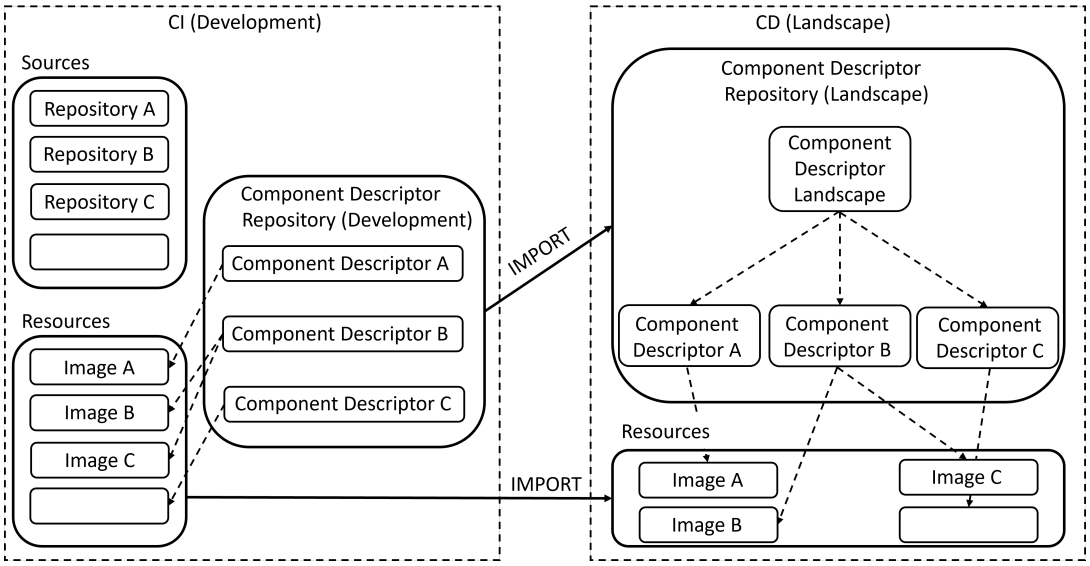

Figure 4.3: Gardener CD
Source: Based on [39]

The left side, CI (Development) is a condensed view of figure 4.2. The dashed arrows represent references. Since the references to the sources stay exactly the same, these dashed arrows are omitted to avoid even more clutter.

So whenever a new Component Version is released through the CI system, the CD system discovers and imports the corresponding Component Descriptors together with the referenced resources into Component Descriptor and Artifact Repositories of the respective landscape. Thereby, while essentially still referencing the same OCI Images, the access property of these Resource References of the replicated Component Descriptors is adjusted to the OCI Images in the landscapes Artifact Repository. Also, as shown in the illustration, in SAP Gardener, an additional Component Descriptor for a Landscape Component gets automatically created. It references all the Component Versions in a given snapshot of SAP Gardener. Every time a new Component Version is released in the CI system, a new Component Version of this Landscape Component is created.

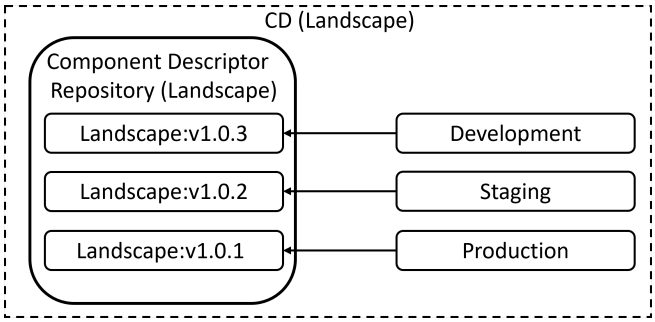

Figure 4.4: Gardener Deployments
Source: Based on [39]

Finally, figure 4.4 indicates how this setup and particularly the Landscape Component is leveraged to actually deploy an instance of the SAP Gardener. Therefore, the Component References of a Landscape Component Version are traversed and the referenced resources deployed. As usual, the Production deployment is based on an older Landscape Component Version than the Development deployment.

4.4 Integration of the Security and Compliance Data Lake into the SAP Gardener Landscape

As already indicated in the previous chapters, although the Security and Compliance Data Lake is an application for storing all kinds of metadata, the initial primary use case is storing the data resulting from compliance scans, thus dependencies, vulnerabilities and licenses. Below figure 4.5 shows the architecture designed for the integration into this complex environment of SAP Gardener.

The *Data Collection Service* is the central component of this architecture. Its job is to fetch all the information that shall be stored in the *Data Lake*. In practice, this would be done based on policies, requiring to do such a fetching once a day or based on some kind of event. In order to actually fetch the information, the Data Collection Service sends a request to the *Access Service* (1).

The Access Service is a transparency layer already built into the *Component Repositories* as part of the OCM. Thus, if the Data Collection Service requests a set of Component Versions and their referenced Artifacts, the Access Service first fetches the corresponding Component Descriptors from the Component Repository (2). After receiving this information, the Access Service evaluates the access property in the referenced Sources and Resources and sends respective requests to the *Source* and *Resource Repositories* to fetch the Artifacts (3). Finally, the Access Service may return all the requested information to the Data Collection Service.

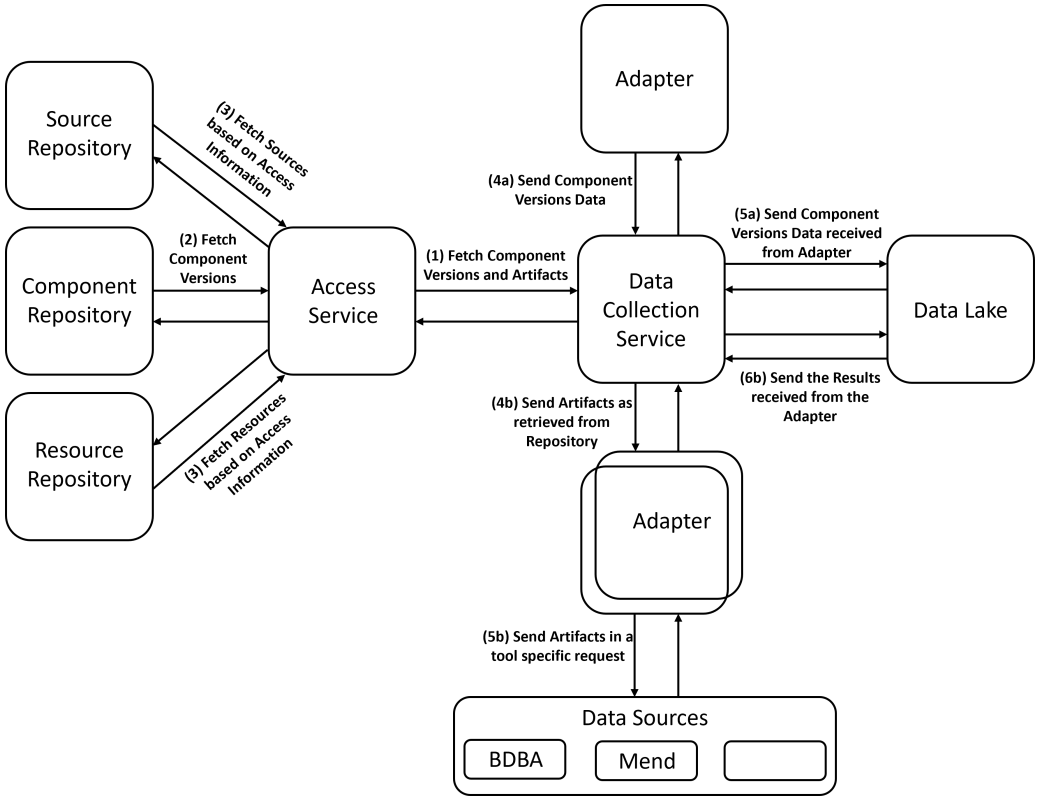

Figure 4.5: Data Lake Integration
Source: Own Representation

As shown in the figure, based on this information, there are two requests flows that may be triggered by the Data Collection Service (4a) and (4b). There is no required order to this, in practice, they may even be executed in parallel. The request flow illustrated above the Data Collection Service (4a) sends the Component Versions data, so the Component Descriptors, to an *Adapter* which adjusts and returns this data in the format required for the consumption by the Data Lake.

The request flow below the Data Collection Service (4b) distributes the Artifacts to a set of Adapters. Here, the Adapters initially wrap the Artifacts into the format required by a specific Data Source. In this case, there would be an Adapter for each compliance scanner. Upon receipt of the scan results, these Adapters also adjust the data to the format required for consumption by the Data Lake, before returning it to the Data Collection Service.

Finally, the Data Collection Service may send all the information, thus the information acquired from the Component Versions about Components, Resources, Sources and their relationships (5a) as well as the information from the scanning tools about dependencies, vulnerabilities and licenses within these Artifacts (6b) to the Data Lake.

There were no communication protocols mentioned so far. This is due to the

facts, that it does not really matter on the conceptual level and that the final implementation of this integration architecture is out of the scope of this work. But in practice, the communication between Data Collection Service and Access Service, between Adapters and Data Sources and between Data Collection Service and Data Lake will be over HTTP. The Adapters however will most likely not be implemented as discrete services but as components within Data Collection Service and therefore communicate over shared memory. Since it is very likely that additional Data Sources shall be added later on, the important part on the conceptual level is to still logically separate the components. Thus, there should still be well defined interfaces between the Adapters and the Data Collection Service.

Chapter 5

System Design

This section describes the design of the *Security and Compliance Data Lake*. It covers the conception of the data model, the selection of a database and the design of the API. Thereby, it especially discusses alternatives and focuses on giving detailed information about the ideas and motives that led to specific design decisions.

5.1 Requirements

Before actually going into the details of the systems design, the requirements have to be specified, since they are at the core of every design decision.

5.1.1 Functional Requirements

The below table 5.1 provides a condensed list of the functional requirements for the Security and Compliance Data Lake. Thereby, every requirement is described by a short and precise but abstract statement of what functionality the system must have and an additional explanation which also includes an example. Additionally, there is a column categorizing the requirements as priority 1 or 2.

Priority 1 is functionality deemed necessary for a central metadata store which should solve the limitations and problems identified in the previous chapters. Furthermore, priority 1 functionality is usually functionality that has to be considered in the design process and otherwise cannot be easily added without foundational remodeling.

Priority 2 functionality describes convenience features which are less urgent and may easily be added later on.

Requirements		
Ref.#	**Functionality**	**Prio.**
R.1	The SCDL shall be able to consume and store metadata from multiple different data sources. The SCDL shall be able to work with any kind of metadata about software components. Therefore, it has to be able to handle multiple different scanning tools as well as other kinds of data sources like build tools. As an example, it may have to consume data from BDBA, Mend but also Jenkins. Thus, it has to be considered that besides vulnerabilities and licenses, a variety of other metadata types may need to be added in the future.	1
R.2	The SCDL shall store the metadata from different data sources without aggregation[1]. Different tools that generally serve the same purpose may provide similar information. As an example, BDBA and Mend are both SCA tools and therefore provide overlapping results. To ensure that no data is lost, this information shall not be combined and aggregated before storing.	1
R.3	The SCDL shall provide the metadata from different data sources with aggregation[1]. As mentioned before, to ensure no data is lost, the data from different data sources shall be stored without aggregation. Anyway, to be consumed by a user, this data shall be aggregated. As an example, when querying all packages contained in a specific resource, the result returned by the SCDL shall not contain the same package twice in different representations, if it was identified by BDBA and by Mend. Instead, it shall contain an aggregated representation of the package. Thus, some kind of aggregation layer is needed which provides transparency regarding the data sources.	1

Continued on next page

[1] *aggregation* in this context means to merge the data about a package of e.g. a BDBA scan and a Mend scan to a single package entity instance

Requirements		
Ref.#	**Functionality**	**Prio.**
R.4	The SCDL shall provide a level of aggregation[2] to group sources and resources. As pointed out before, one problem also with SBOMs is the disconnection of the artifact metadata and the deployment information. To bridge this gap, an additional aggregation level for grouping artifacts is necessary. As an example, this additional aggregation level shall enable to group all resources contained in a specific deployment.	1
R.5	The SCDL shall enable users to query the metadata on different levels of aggregation[2] As an example, a user shall be able to query for all vulnerabilities in a specific resource, thus query on the aggregate level of resources. But a user shall also be able to query for all vulnerabilities in an entire specific deployment, thus querying on the aggregate level of deployments (querying on this level of aggregation enables to answer where Log4J is deployed).	1
R.6	The SCDL shall enable users to perform assessments. The relevance of specific pieces of information such as vulnerabilities or licenses depends on the usage context. As an example, while the internal usage of an altered OSS with a copyleft license is lawful, the distribution is not. Therefore, a possibility has to be provided to assess such pieces of information in the context of their occurrence.	2

Continued on next page

[2] *aggregation* in this context refers to the "whole/part" semantic of the word [40]. Thus, since resources and sources are comprised of packages, they are both aggregations of packages. On a model level, the same applies for the relationships between packages and vulnerabilities or licenses as well as between entire deployments and the deployed resources.

Requirements		
Ref.#	**Functionality**	**Prio.**
R.7	The SCDL shall provide common data aggregation and filter functions for the queries. As an example, a user shall be able to filter for the vulnerability with the highest CVSS within a resource or shall be able to get the count of vulnerabilities within a resource.	2
R.8	The SCDL shall enable users to query the metadata in the common SBOM formats. In order to be able to fulfill governmental requirements of the executive order mentioned in the Software Bill of Materials section, the SCDL has to provide a way to to query the metadata in the common SBOM formats. As an example, a user shall be able to query the SPDX document for a specific resource.	2

Table 5.1: Requirements

So, by fulfilling this functional requirements, the Security and Compliance Data Lake may actually serve as a central application for storing and querying software metadata. Thereby, solving the problem of metadata being distributed throughout the development life cycle and bridging the gap between artifact metadata and deployment information.

5.1.2 Non-functional Requirements

Since this shall be a prototypical implementation, there is a strong focus on fulfilling the functional requirements. Thus, no concrete limits regarding performance or scalability such as a maximum response time of 5 seconds or support for up to 1000 concurrent users are set here. Considering the novelty of the topic, there is very few reference data and therefore, such specifications would be premature. However, for a central metadata store which may prospectively power dashboard web applications for monitoring purposes, scalability and performance definitely have to be considered in design decisions already.

5.2 Data Model

The basic entities relevant in the software supply chain are artifacts, thus sources and resources, and the packages comprising these artifacts. The definition of sources and resources is still the same as introduced in the previous chapter. Resources are capable of doing something and are usually executables or OCI Images. Sources are the code the resources are built from. Compliance scanners usually scan entire source code repositories or binaries. Through different methodologies, these tools detect the packages contained in these scanned artifacts on a best effort basis. In the context of this work, a package is defined as functional unit contained in artifacts, whereby it is usually a collection of files forming a library which is imported in the source code. By subsequently matching these packages against different databases such as the NVD, introduced in section 2.3 "Vulnerability Management", known vulnerabilities and licenses are identified. To give a better idea of these results, figure A.1 in the appendix shows a snippet returned from the API of BDBA. The results on their own are useful already and provide interesting data about the above mentioned entities. But it is still loose metadata that lacks context information such as which deployments contain the corresponding entities. Therefore, an additional entity type to conduct further grouping is required. The OCM already introduced such an entity type, the component.

To conclude this, from a high level perspective, the important entities are *components*, *sources*, *resources*, *packages* and the information attached to these entities such as vulnerabilities and licenses. To generalize this and abstract away from specific data sources, the entity type representing these types of metadata is called *info snippet*. So these entity types are the basic building blocks for the data model. From here on, it is getting rather complex and abstract. To still keep the explanations tangible, below figure 5.1 already shows an *entity-relationship model (ERM)* describing the final and universal data model. This may be used as a reference point throughout the following paragraphs, discussing the design decisions leading up to the specific entities, relationships and cardinalities.

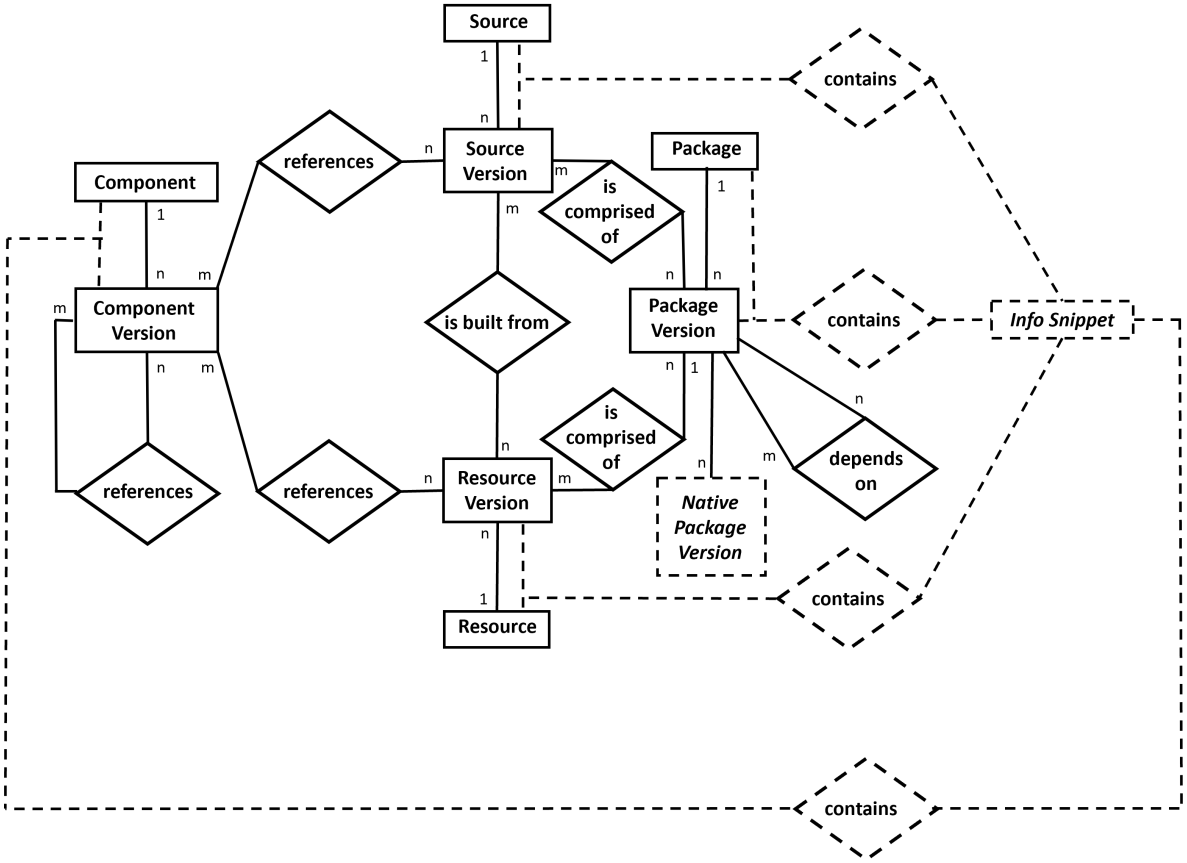

Figure 5.1: Data Model
Source: Own Representation

Even though the motivation behind every element may not be obvious immediately, the model as a whole should feel quite familiar and intuitive by now. An important notice at this point, the data model is inspired by the OCM. As mentioned above, especially the entity type *component* is lend from the OCM. Since this thesis is written within SAP Gardener, a seamless integration of the this Security and Compliance Data Lake such as described in the previous chapter is of course also a requirement, even though it is not explicit listed. But still, the Security and Compliance Data Lake is designed independent of the OCM. Thus, in theory it is entirely possible to use a different kind of component model. As an example, if one is able to express the concept of *components* and *artifacts* with the means of the SPDX standard, one could use SPDX instead of OCM to provide this structural information. Or, since SPDX is not optimal for this purpose, one could create and use an own component model, as long as it has the means to express *components* and *artifacts* (There is generally no necessity to distinct between *sources* and *resources*. *Sources* could be treated as *resources*, at the only cost of losing the connecting "is built from"-information between the two entities.).

5.2.1 Universal Data Model

Contrary to common ERMs, the one in figure 5.1 does not have any properties. There are two major reasons for this. Firstly, the just mentioned independence of a specific component model would hardly be possible if the data model would define fixed predefined properties for each entity type. Secondly, the different scanning tools provide a wide range of information about packages and other data sources but scanning tools may also be added. It is therefore practically impossible to foresee what properties may be needed. Besides, these may vary depending on the user of the Security and Compliance Data Lake.

Another special feature of above ERM are the entity types and relationships illustrated with dash lines. These represent classes of entity types and relationships. Since the whole set of data sources cannot be known upfront, the whole set of potentially required *Info Snippets* cannot as well. As already mentioned in several examples before, when adding a build tool as data source, an entity type *Build Information* may be needed. Also, the relationships of different *Info Snippet* entity types may vary. While a *Vulnerability* and a *License* is usually *contained* in multiple *Package Versions* leading to a (n:m)-relationship, a *Build Information* is usually associated to one *Resource Version* leading to a (1:n)-relationship. But generally, *Info Snippets* could be associated to any other entity type in the data model with any cardinality. This kind of flexibility is necessary to enable R.1 (consume and store metadata from multiple different data sources). The *Native Package Version* correspondingly illustrates the representation of a package, native to a concrete data source. Thus, instances of *Native Package Versions* may be *BDBA Package*, *Mend Package* or even *Jenkins Package*. So if all three data sources provide information about the exact same *Package Version*, each representation may be stored without a need to merge their properties before storing. Thereby, this enables R.2 (store metadata from different data sources without aggregation). Then, a set of properties commonly provided by all of the data sources may be aggregated on *Package Version* level, thereby also enabling R.5 (provide the metadata from different data sources with aggregation). So, all these *Native Package Versions* representing the exact same package are related to the same *Package Version* on the model level. As the different data sources may use different identifiers for the packages, the merging process cannot be triggered automatically. Hence, until a human defines that the *BDBA Package*, the *Mend Package* and the *Jenkins Package* are actually representations of the same package, no merging is done and each is related to a different *Package Version*.

So after explaining the special features of above ERM, the common entity types and relationships may be discussed. The basic entity type *component* is broken down into two distinct entity types, *Component* and *Component Version*. As immediately noticeable, this distinction is done for each of the basic entity types. *Component* is a purely abstract entity type. It merely groups all the versions of the same component together. Thereby, the *Component* may provide information about the semantics of this grouping such as whether this *Component* describes a specific deployment or whether it describes all software used by a department. Thus, information that is identical for all versions of this component and would have to be stored redundantly for each *Component Version* otherwise. Naturally, there are multiple *Component Versions* of each *Component*. Therefore, the (1:n)-cardinality here is self-explaining.

As established by the previously described grouping semantic of *components*, a *Component Version* may reference multiple other *Component Versions*. For example, a *Component Version* describing a specific version of a deployment may reference multiple other *Component Versions* such as *Component Versions* describing specific versions of a web server, a service and a database. Reciprocal, a *Component Version* may of course be referenced by multiple *Component Versions*. For example, a *Component Version* describing a web server may be referenced by several *Component Versions* describing different versions of the same deployment or entirely different deployments. Thus, this is a recursive (n:m)-relationship. There may also be a need to store additional occurrence specific metadata as properties of the *references*. Considering the above example, such occurrence specific metadata may provide information about the usage of the web server within the deployment, hence whether it is used as a HTTP server or as a load balancer. Together, these model elements fulfill requirement R.4 (provide an aggregation level to group sources and resources).

Furthermore, a *Component Version* may also reference multiple *Source Versions* and *Resource Versions*. As an example, the *Resource Versions* comprising the web server and the *Source Versions* from which the respective *Resource Versions* were built. As before, with the recursive relationship of *Component Versions*, the *Source Versions* and *Resource Versions* may of course also be referenced by multiple *Component Versions*, resulting in a (n:m)-relationship. Again, there may be a need to store additional occurrence specific metadata as properties of the *references*. Specifically, these *references* may be used to store *triage* information. As this *reference* describes the usage context of the *Artifact*, one may for example decide whether a copyleft license is or is not acceptable here. The relationship between *Source* and *Source Version* as well as between *Resource* and *Resource Version* is similar to the relationship between *Component* and *Component Versions*. But the abstract *Source*

and *Resource* entity types actually do have a concrete purpose but only preventing redundant storage of certain properties. In this case, these abstract entities may have properties to store *triage policies*. As an example, one may store that a specific vulnerability may be ignored for the usage of *Resource Versions* v1.0.0 to v1.2.3 of a respective *Resource* within *Component Versions* v1.4.2 to v1.4.12 of a specific *Component*. The *references* and the abstract *Source* and *Resource* entities thereby enable the fulfillment of requirement R.6 (enable users to perform assessments).

Resource Versions may also reference the *Source Versions* they are built from. A *Resource Version* may be built from multiple *Source Versions* and a *Source Version* may be used to build multiple *Resource Versions*. This also results in a (n:m)-relationship.

Since *Artifacts* are comprised of *Package Versions* and the same *Package Version* may occur in multiple *Artifacts*, both *Source Version* as well as *Resource Version* have a (n:m)-relationship to *Package Version*. Again, there may be a need to store additional occurrence specific metadata as properties of the *is comprised of* relationship.

The relationship between *Package* and *Package Version* is again similar to the relationship between *Component* and *Component Versions*. But as already explained *packages* are broken down even further into three different entity types and thereby three aggregate levels. Since several data sources may provide different representations, thus different *Native Package Versions*, of the same *Package Version*, the cardinality of this relationship is (1:n). *Package Versions* frequently *depend on* other *Package Versions* and so on. This may lead to quite long chains of dependencies. This has to be kept in mind as these transitive dependencies are also relevant when trying to answer the question, whether a certain *Artifact* or *Component* contains a specific *Package* such as Log4J, and its corresponding vulnerabilities.

Finally there is the *Info Snippet* class of entity types. As explained above, different *Info Snippet* entity types may have relationships to different entity types with different cardinalities.

5.2.2 Insights into the Development Process

At this point, some insight into the development process may be beneficial to understand this design decisions. The scope of this central data store was initially much narrower. The first PoC was actually strictly bound to the OCM. Therefore, the data model predefined the properties of *component* and *artifact* entity types. As it was bound to the OCM, there was no issue in doing so. But the data model also predefined the properties of the *package* entity types. In fact, the third aggregate

level for *packages*, *Native Package Version* did not exist at all. Instead, the *Package Version* entity type had a set of properties that was hoped to be common and harmonizable throughout all prospective data sources. At this time, the application was also tailored to only having scanning tools as data sources. Therefore, to define this set of properties, the API documentations of different scanning tools were analyzed for the common and most important properties, especially the ones of BDBA and Mend [41]. Additionally, to get a better understanding, both scanners were used on some artifacts to get some sample data. Then, the provided attributes were narrowed down and some interviews with developers were conducted. A huge effort was made here, as this was such an important decision. Also, instead of having the *Info Snippet* class of entity types, there was only a *Vulnerability* and a *License* entity type whose set of properties was defined in the same process. Besides the fact that there was already a substantial amount of disagreement between different developers which properties were actually required, by the time the PoC was finished, several new use cases were discovered that required additional properties and even entire additional entities to provide other information than about vulnerabilities or licenses.

This led to a change in perspective, interpreting the task of defining the right entity types with the right properties rather as a task to make the entire application extensible regarding the respective entity types and properties. But this flexibility and extensibility comes at the cost of a highly increased overall complexity. Apart from remodeling the data model, it also required a completely new architecture and completely different implementation. Thus, only after that, the scope widened drastically, also allowing other component models. Consequently, as a kind of disclaimer, the design process presented here is not exactly as chronological as it may initially seem, since it hides a complete development iteration leading up to the final concepts.

5.2.3 Application of the Data Model

Although a great effort was made to make the data model as tangible as possible, it may still be hard to grasp due to its abstract nature, omitting properties and introducing classes of entity types. Therefore, this section discusses how this data model could be applied. As reference component model, the OCM is used and as reference data sources, only BDBA is used. This thereby also reveals a problem that has to be faced when applying this data model to a real world use case. The figure 5.2 below shows the corresponding data model instance.

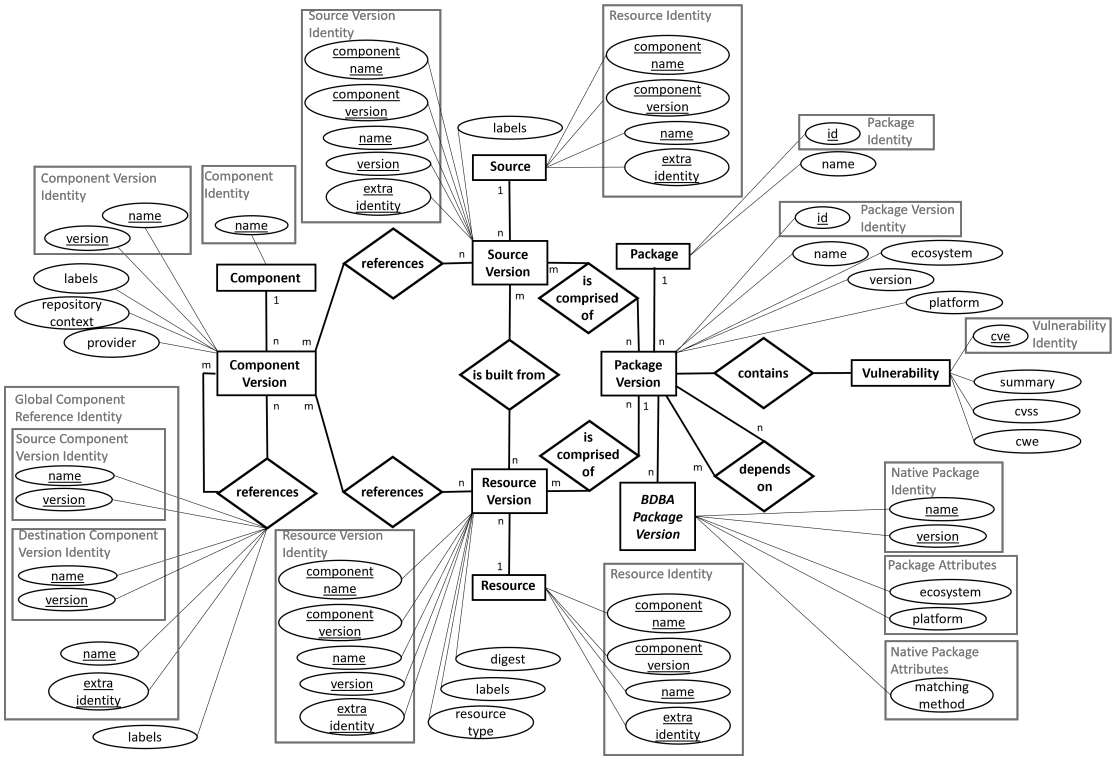

Figure 5.2: Example Data Model Instance
Source: Own Representation

The figure is very crowded. But this is necessary, as the purpose of this figure is to provide a concrete and thorough example of how the abstract universal data model may be applied to specific component models and tools.

The important aspect to point out here is the relationship between *Component Version* and *Source Version* and between *Component Version* and *Resource Version*. Although depicted as a relationship with (n:m)-cardinality in compliance with the universal data model, due to the *Component Version-Local Identity* of *Source Version* and *Resource Version*, these can actually only be (1:n)-relationships. There were detailed explanations about this in section 4.1 "Open Component Model". *Source Version* and *Resource Version* were referred to as *Source Reference* and *Resource Reference* because in the context of OCM, instead of actually representing the technical artifact, they only reference the technical artifact through their *access* property. The SAP Gardener team chose this initially rather confusing specification of *Source Versions* and *Resource Versions* with *Local Identities* since in practice, it is difficult to reliably determine whether two referenced technical artifacts are actually the same technical artifact.

The following sections further explain the issue and discuss different approaches of dealing with this artifact identity problem. They thereby point out the use cases and limitations of each approach. The terms *Resource Version* and *Source Version*

are from here on used as in the context of OCM, thus these terms are interchangeable with *Resource Reference* and *Source Reference*.

Uniform Resource Identifiers

The initial and most obvious approach is to treat technical artifacts just as components. Thus, assign a globally unique name and a version to each technical artifact. But the reason this works for components is that these are purely abstract or logical entities. Thus, there is no digital or rather technical twin such as source code or a binary that corresponds to a specific component. If there is, who guarantees that two *Resource Versions* with the same name and the same version actually point to the same binary? Or reciprocal, that two *Resource Versions* pointing to the same binary actually have the same name and version? Where and how would one look up this globally unique name of a *Resource Version* in the first place?

A common approach in this situations are URIs. As introduced in section 2.1 "Software Identification", by providing a specification of how this URI is composed, standards such as purl provide a way to create theoretically reproducible identifiers. Theoretically, because in practice composing this URIs still has to be done by people. Consequently, there is always room for interpretation and human error. Imagine a *Resource Version* in two different *Component Versions* referencing artifacts in two different repositories, for example `github.com/example/nginx` with tag `1.0.3` and `github.com/example/nginx-webserver` also with tag `1.0.3`. At this point, it is difficult to determine, whether these are technically the same artifact and should consequently have the same URI. Thus, this approach is unreliable and impractical.

Content-Addressable Uniform Resource Identifiers

In order to guarantee that two *Resource Versions* with the same URI actually point to the same binary and that two *Resource Versions* pointing to the same binary actually have the same URI, this URI has to be content-addressable. Therefore, a coupling of identity and location is necessary.

The straight forward approach to do this in practice is to use the URLs provided by the repositories. But as the example above shows, this approach is not really flexible, as the URLs have to be immutable.

The OCM specifies that the access property is variable. Still in section 4.3 "Development and Deployment Landscape at SAP Gardener", it is explained that images, thus technical artifacts, and component descriptors are copied from the development landscape into other landscapes and that the access property changes in this process. So the development landscape may serve as single source of truth

and the respective artifact URLs of the access property may be used as this content-addressable URI. Even though the URLs in the access property of the component descriptors in other landscapes may differ from the URI globally identifying the artifact, as this URI is used to copy the artifact to the location referenced by the URL in the new access property, it is the same per construction. So this approach is generally reliable and practical.

SAP Gardener did not consider this approach anyway, as it is more complex and globally unique identifiers for artifacts were not relevant for the original scope of the OCM, which was deployment automation.

Digests as Global Identifiers

So, another option to determine whether two artifacts are technically the same, even though they are located in different repositories is through calculating a suitable *normalized digest*.

A *digest* refers to a short, fixed-length string calculated through a hash function [42]. There are several features in which artifacts may vary but that are irrelevant for their comparison. For example, the exact same source code file may provide different digests when hashed with the exact same hash method, depending on whether it is hashed on a windows or a linux machine. This is due to the different line endings, thus "Carriage Return and Line Feed" on windows and "Line Feed" on linux. In order to calculate a normalized digest, the input data for the hash function is prepared so that such irrelevant features do not have an impact on the digest. So the input data is normalized.

This approach is generally reliable and clean, but there are again several problems. When looking at figure 5.2 above, there is already a digest property available for *Resource Versions*. As described in section 4.1 "Open Component Model", this digest property specifies a hash function, a normalization function and the corresponding digest value. Although the primary reason for this is that different kinds of resources, for example executeables and OCI Images, may need different kinds of normalization functions, this also means that the same artifact may have different digest values in different *Component Versions*.

Consequently, the digests need to be calculated in a standardized way, independent of the OCM. But even then, there is still another issue regarding the data model. There is no way to decide whether artifacts with different normalized digests may be different versions of the same artifact. Thus, the additional aggregate level, *Source* and *Resource*, required for triage policies is practically impossible to determine.

Local Identifiers

This is the approach used in the OCM. This concept was explained in detail in section 4.1 "Open Component Model". The detailed examination of the general artifact identity problem also showed implicitly why differentiating between the terms *Resource*, *Resource Version* or *Source*, *Source Version* and *Resource Reference* and *Source Reference* is so difficult from a modeling perspective.

After all, the most important aspect to point out here, is that although it is not intuitive, only having local identifiers for artifacts is still quite functional regarding the goals which shall be achieved by the data model.

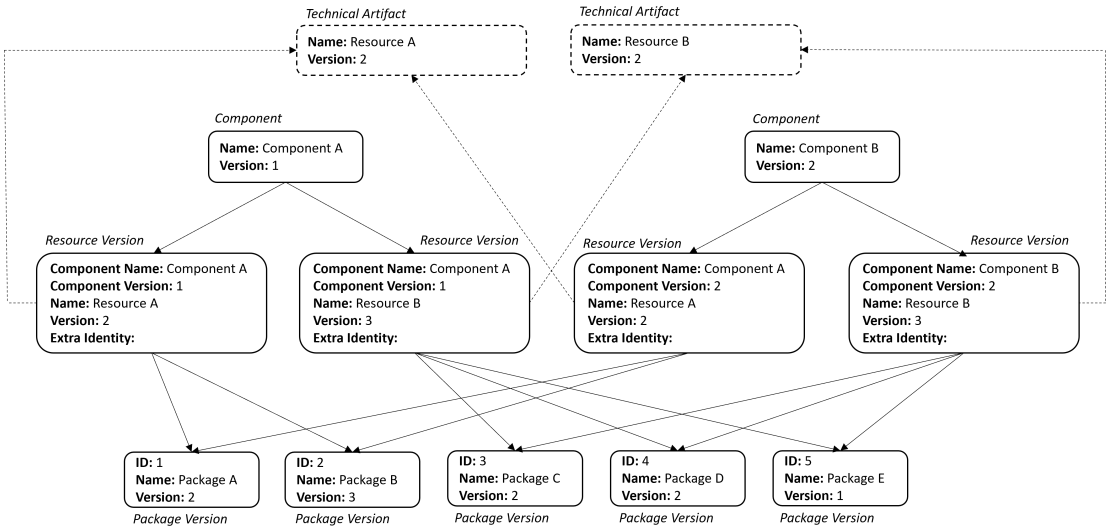

Figure 5.3: Local Artifact Identities
Source: Own Representation

Figure 5.3 shows two components that reference the exact same technical artifacts. But since the *Identity* of *Resource Versions* is local, the component name and version are part of the *Resource Versions Identity*, as also shown in figure 5.2. Thus, it appears as they are referencing different artifacts. But as the packages comprising the *Resource Versions* are identified by scanning the technical artifact accessed through the access property, they are the same for the *Resource Versions* which practically reference the same technical artifact.

So, for answering questions such as which deployment contains Log4J and its corresponding vulnerabilities, the local artifact identities do not make a difference. As in practice, it may also be assumed that within a *Component*, *Resource Versions* with the same name but different version numbers are in fact pointing to different versions of the same technical artifact, the *Resource* aggregate level is also kind of possible. But its scope and the scope of *triage policies* within these entities respectively is of

course limited to a certain *Component*. Furthermore, attaching *Build Information* and other *Info Snippets* to artifacts is difficult with this approach.

5.3 Database

After the application context and data model are defined, a suitable database has to be selected. Therefore, this section analyzes the most relevant database technologies regarding their applicability as a central data store based on the previously defined data model.

5.3.1 Relational Databases

The first database technology analyzed are relational databases. They are by far the most popular and widely used database technology. This is represented in Stack Overflow's 2021 developer survey [43]. Among the technologies discussed in detail here, it has also been around for the longest as the original paper introducing the relational model for databases by Edgar Codd was published in 1970 [44]. Therefore, the technology itself is very mature. everybody has know how. worth considering for every enterprise project.

A quick repetition of the foundations of relational databases. The name "relational database" stems from the mathematical definition of relation. Thus, given sets

5.3.2 Document Store Databases

5.3.3 Graph Databases

5.4 API

Chapter 6

Implementation

Appendix A

Additional figures

A.1 Black Duck Binary Analysis Result


```
{
    "lib": "glob",
    "objects": [
        "etcd-druid"
    ],
    "version": "v0.2.3",
    "vendor": "gobwas",
    "extended-objects": [
        {
            "name": "etcd-druid",
            "fullpath": [
                "etcd-druid_v0.13.0_github.com_gardener_etcd-druid",
                "sha256:3db0932b4abd14ebbb6b385a931c3b1ea74d2408124739a97265b8887f182ec5.tar",
                "etcd-druid"
            ],
            "timestamp": 1662401610,
            "size": 62745750,
            "sha1": "a86ccd14767d5adf117f9daf20586fadde8c2fdc",
            "confidence": 1,
            "matching-method": "go-mod-package",
            "binary-type": "elf-executable-x86_64",
            "package-type": "go-mod",
            "type": "go",
            "bd-id": "github.com/gobwas/glob",
            "detected_version": "v0.2.3"
        }
    ],
    "vulns": [],
    "tags": [
        "glob"
    ],
    "homepage": "https://github.com/gobwas/glob",
    "latest-version": "1.0.4",
    "codetype": "go",
    "coverity_scan": null,
    "license": {
        "name": "MIT",
        "url": "https://opensource.org/licenses/MIT",
        "type": "permissive"
    },
    "vuln-count": {
        "total": 0,
        "exact": 0,
        "historical": 0
    }
},
```


Figure A.1: Snippet of BDBA Analysis Result
Source: [34]