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 was already mentioned that existing measures for monitoring software component metadata such as vulnerabilities are insufficient in some cases. That is explained in this chapter. It therefore starts with a brief introduction of the currently established state of the art approach. To make the issues more tangible, it then provides an explanation of the development and deployment landscape at SAP Gardener and follows up with an analysis of how this approach fails in such settings. The identification of this problems is crucial for the successful design of the Security and Compliance Data Lake.

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. Again 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. This considers functionality and performance of the Security and Compliance Data Lake as well as the knowledge gained throughout this work, revisiting the research question.

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.

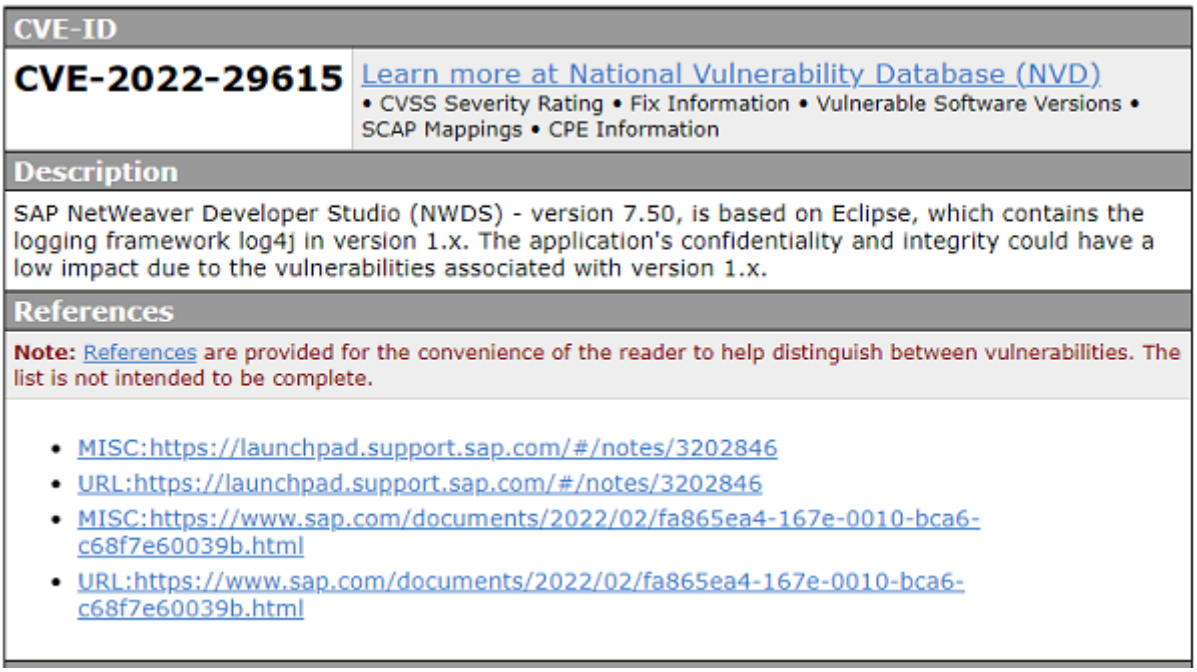

Figure 2.1: CVE Record
Source: [14]

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.

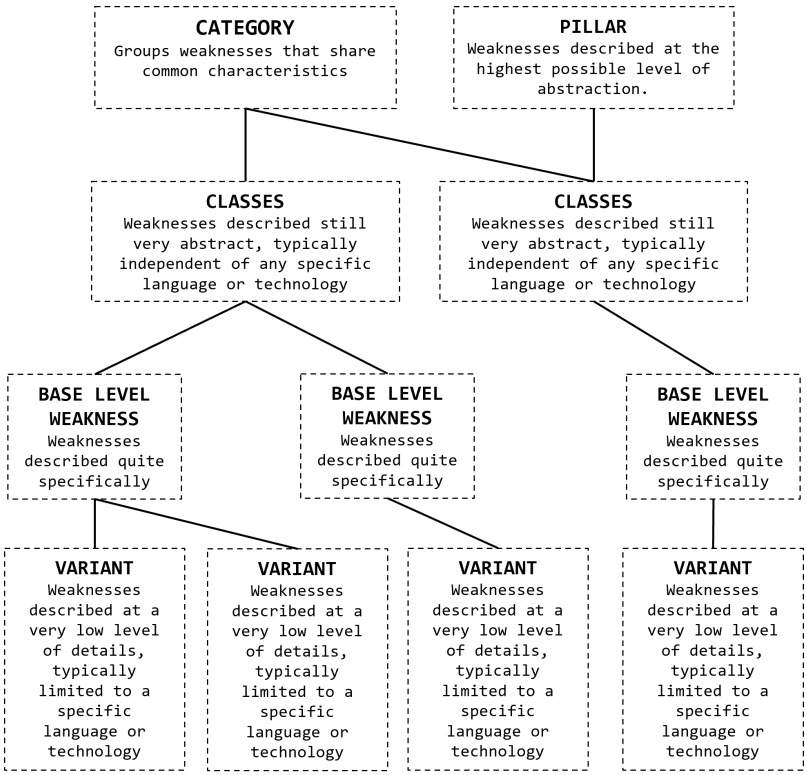

Figure 2.2: CWE Hierarchy
Source: Own Representation

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.

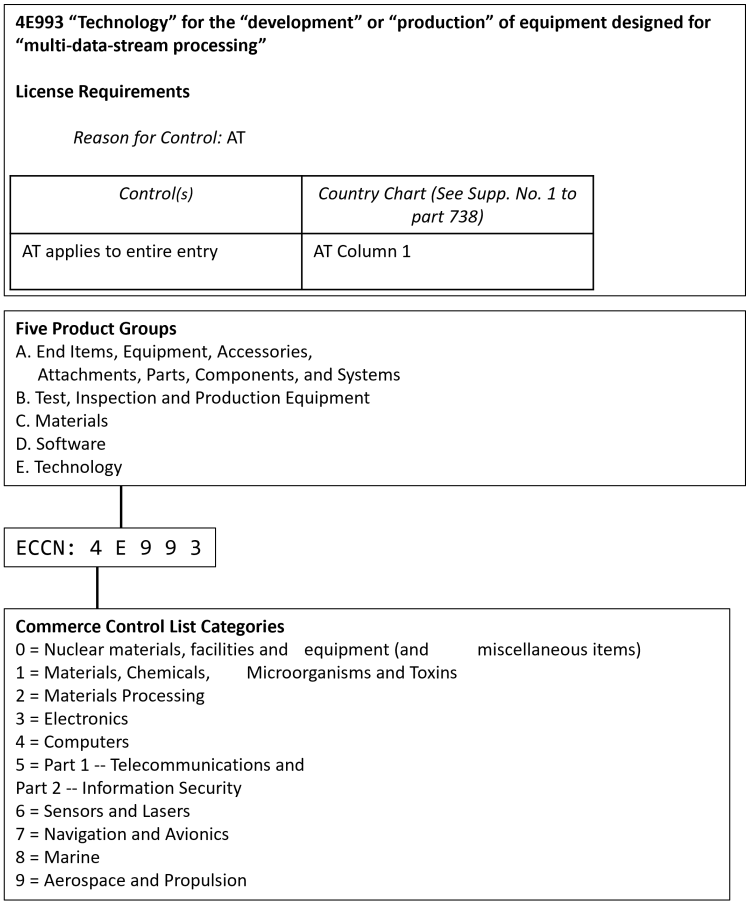

Figure 2.4: ECCN Structure
Source: [24]

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].

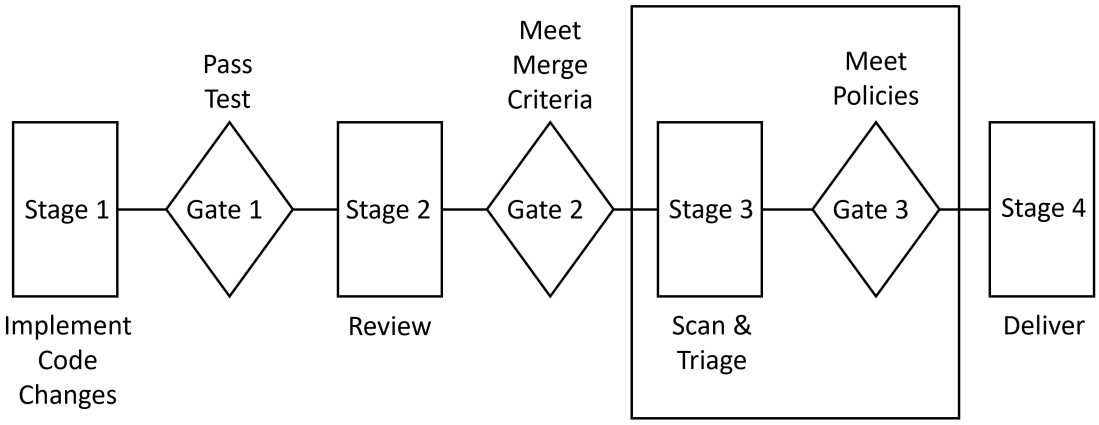

Figure 3.1: CI/CD Pipeline as Stage-Gate System
Source: Based on [31]

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 Open Component Model deducted from the specification [38] and an internal presentation [39]. Even though this explanations 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 ?? below shows an actual *Component Descriptor*, the serialization format of the OCM. This may also support in understanding the following paragraphs.

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 number*. Furthermore a *Component Version* contains *Component References*, *Artifacts*, *Labels* and a *Provider*. *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 it actually has some pitfalls. A *Component Reference* is not identified by the *Identity* of another *Component Version*, thus by the *globally unique name* and the *version number*. Additionally to the *Identity of the Component Version*, the Identity of the *Component Reference* also contains a *name* for the reference. 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* might use the web server *Component Version* twice, as a classic HTTP server and 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. Additionally, *Component References* also contain *Labels*.
Artifact is used as an umbrella term for either a *Source* or a *Resource*. *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. *Artifacts*, so *Sources* as well as *Resources*, have common properties. They both have a component-local *Identity* composed of required *name* and *version* properties and an optional *extra identity*. The optional *extra identity* property is necessary for similar reasons

as the *name* property in *Component References*. There might be a situation where a *Component Version* contains a specific *Resource* such as an executable twice, one for ARM platforms and the other one for x86 based platforms. To enable even further distinctions, the *extra identity* is a map of key-value-pairs. Component-local 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 *Artifacts*, thus the combination of *artifact name*, *artifact version* and possibly *artifact extra identity*, only has to be unique within the scope of a *Component Version*. As a consequence, to globally uniquely address the abstract concept of an *Artifact* in OCM, the combination of *Component Version Identity* and *Artifact Identity* is necessary. Thus, the globally unique *component name* and the *component version* plus the *artifact name*, *artifact version* and possibly the *artifact extra identity*. Besides the identity properties, *Artifacts* also have a *type* and *access* property and contain the well-known *Labels*. 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, e.g. git. Finally, the *access* property, which is a map of key-value-pairs which again has a required *type* property, provides a formal description of how and where to access the content. Thereby, the *type* defines the access method, usually by specifying the repository type, e.g. github. 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 commit hash. A *Resource* additionally has a *digest*, *relation* and *srcRefs* property. The *digest* is ann 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 property *IdentitySelector*, specifying the identity of the corresponding *Resource* and the *Labels* property.

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 Development and Deployment Landscape at SAP Gardener

So SAP Gardener develops SAP‘s own managed Kubernetes service which enables its to easily setup a kubernetes cluster in the cloud of one of the hyperscalers as well as on premise. Therefore, the team has to employ a rather sophisticated development and deployment landscape.

An exemplary representation of the CI part of the development landscape is illustrated in ?? below.

Now, after the foundations have been laid out and there is a general understanding of both the goal of the thesis and the department, this section puts those pieces together. It introduces the Open Component Model and explains how it fits into the SAP Gardener Deployment Scenario. Thereby, it also shows how the Open Component Model forms the basis for the Security and Compliance Data Lake.

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

As already mentioned, the OCM is an SBOM format, created and used by SAP Gardener. The OCM does not fulfill the minimum requirements. This is because it is designed to fulfill different requirements. SPDX and CycloneDX are deliberately designed to be a bill of materials, thoroughly listing the inventory of a software. Meanwhile the purpose of OCM is to enable a holistic delivery automation in a multicloud environment [39].

To really understand the rationale behind OCM, it is crucial to look at where the idea historically comes from. In the ABAP on-premise environment, every development had to fulfill certain conditions so it Now, the SAP Gardener managed kubernetes service is a cloud-native, open-source project. From early on, it consisted of about 60 *Open Container Initiative (OCI) Images*. Thus, there immediately was a need for automation of both, delivery and compliance. As a consequence, SAP Gardener began to develop what is known as OCM today.

SBOMs loose information without connection to the deployment

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 the API. Thereby, it especially focuses on giving detailed information about the ideas and motives that lead to specific design decisions.

5.1 Terminology

The System Design section makes use of a lot of heavily overloaded words which may lead to confusions and make it quite difficult to follow. To avoid this, the meaning(s) of those words in specific contexts will be specified in the following:

Component

Artifact

Package

5.2 Requirements

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

5.2.1 Functional Requirements

Requirements	
Ref.#	**Functionality**
R.1	The SCDL shall be able to consume and store data 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. Thus, one has to consider that besides vulnerabilities and licenses, a variety of other data types may need to be added in the future.
R.2	The SCDL shall store the data from different data sources *without aggregation*. Different tools that generally serve the same purpose may provide similar information. To ensure that no data is lost, this information shall not be combined and aggregated for storing (*aggregation* in this context means to e.g. combine the data about a package of a BDBA scan and a WhiteSource scan to an single package entity instance on database level).
R.3	The SCDL shall enable users to perform assessments. The relevance of specific pieces of information such as vulnerabilities depend on the use case. There the SCDL has to provide a possibility to assess such pieces of information in the context of its occurrence.
R.4	The SCDL shall enable users to query aggregated metadata of a component. As mentioned before, to ensure no data is lost, the data from different data sources shall be stored without aggregation (*aggregation* in this context means to e.g. combine the data about a package of a BDBA scan and a WhiteSource scan to an single package entity instance on database level). Anyway, to be consumed by a user, this data shall be aggregated. Thus, some kind of aggregation layer is needed.

Continued on next page

Requirements	
Ref.#	**Functionality**
R.5	The SCDL shall enable users to query the aggregated metadata on different levels of aggregation. The user shall be able to query a component and all its metadata as well as a package and all its metadata. Furthermore, information about vulnerabilities and licenses shall bubble up when querying (so *aggregation* in this context means to e.g. collect the vulnerabilities and licenses from packages contained in a component and perhaps also filter them based on the highest CVSS).
R.8	The SCDL shall provide means to request the SBOM. In order to be able to fulfill governmental requirements like the executive order mentioned in 2.4 about the Software Bill of Materials, the SCDL has to provide a way to request an SBOM.
R.9	Removal of old unused versions

Table 5.1: Requirements

5.2.2 Non-functional Requirements

Since this shall be a prototypical implementation, there is a strong focus on fulfilling the functional requirements. Anyway, performance definitely has to be considered in design decisions already, especially since the SCDL shall serve as the backend for a dashboard web application. Scalable (Wie viele Daten fallen im Gardener an? Wie lange wird das gut gehen? Datenarchivierung) –> PoC aufbauen und dann nachschauen!

200MB/Day pro Scanner

5.3 Data Model

5.4 Database

5.5 API

Tool agnostic, how does the api look like to abstract away from different tools

Chapter 6

Implementation