# 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

# Chapter 1

# Introduction

## 1.1 Motivation

## 1.2 Goals

The goal of this thesis is the design of a software solution

## 1.3 Environment

## 1.4 Structure of the Thesis

# Chapter 2

# Foundations

## 2.1 Terminology

This section explains several abbreviations and standards which are frequently mentioned alongside the topics *Vulnerability Management* and *Open Source Licensing* and are thus essential for the further understanding.

### 2.1.1 Software Identification

Uniquely identify 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" [1]. This Package URL consists of seven components [1]:

```
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 [1].

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.

### 2.1.2 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" [9].

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 [9].

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.

### 2.1.3 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.

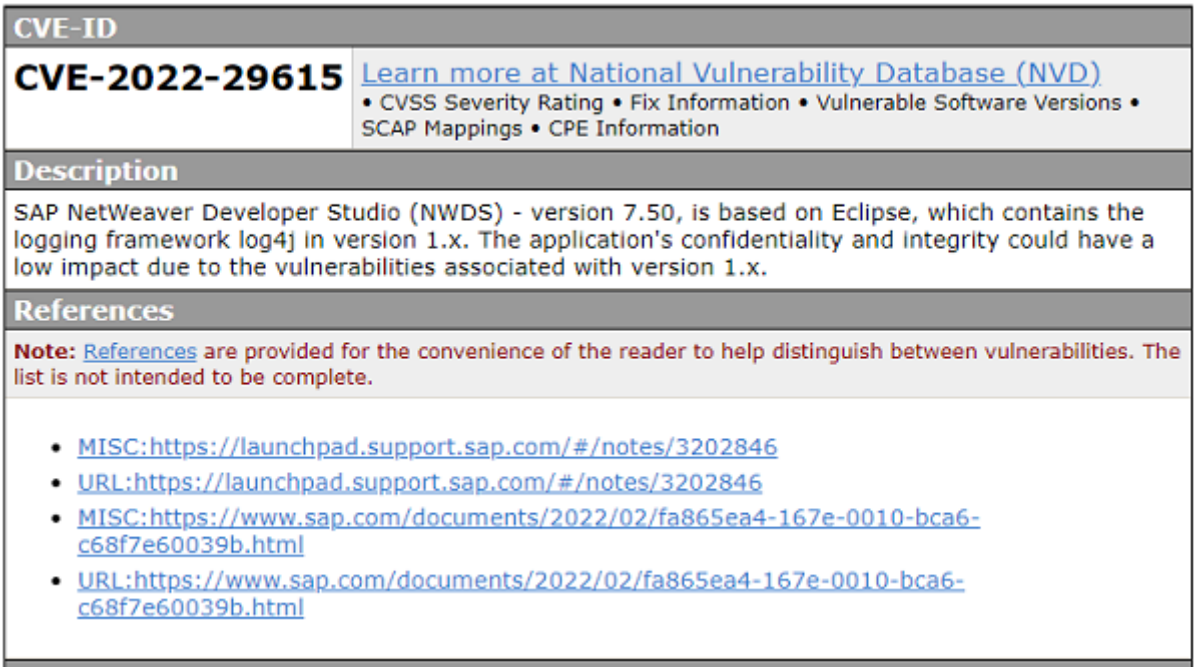
#### Vulnerability

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)."[2]

#### 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 [3].

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.



**CVE-ID**

**CVE-2022-29615** Learn more at National Vulnerability Database (NVD)
• CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information

**Description**

SAP NetWeaver Developer Studio (NWDS) - version 7.50, is based on Eclipse, which contains the logging framework log4j in version 1.x. The application's confidentiality and integrity could have a low impact due to the vulnerabilities associated with version 1.x.

**References**

**Note:** References are provided for the convenience of the reader to help distinguish between vulnerabilities. The list is not intended to be complete.

- MISC:https://launchpad.support.sap.com/#/notes/3202846
- URL:https://launchpad.support.sap.com/#/notes/3202846
- MISC:https://www.sap.com/documents/2022/02/fa865ea4-167e-0010-bca6-c68f7e60039b.html
- URL:https://www.sap.com/documents/2022/02/fa865ea4-167e-0010-bca6-c68f7e60039b.html

Figure 2.1: CVE Record
Source: [4]

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 [5].

### 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 [6].

**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.

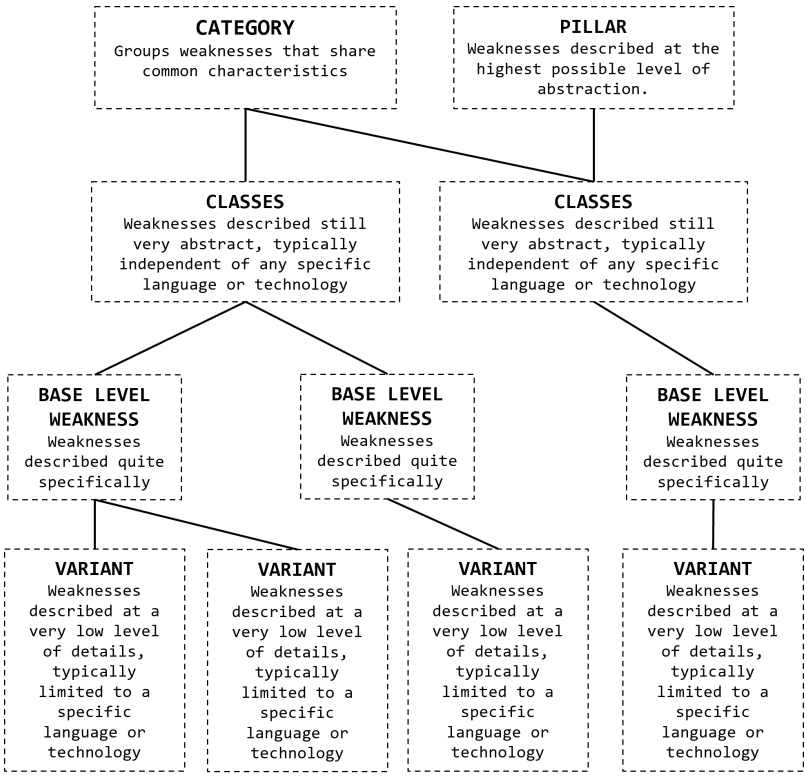



Figure 2.2: CWE Hierarchy
Source: Own Representation

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* [7].

**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 [8].

**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* [2].

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.

### 2.1.4 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 [10, 11].

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 [12, 11]). 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: [11]

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 [11].

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 [12]. 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 [11]. 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 [13]. 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) [14, 15].

To give a more concrete idea of the basic concepts of *SPDX Documents*, an example

from the SPDX GitHub repository will be briefly examined [16]. Therefore figure 2.3 below shows the directory structure of a “Hello World” project in C.

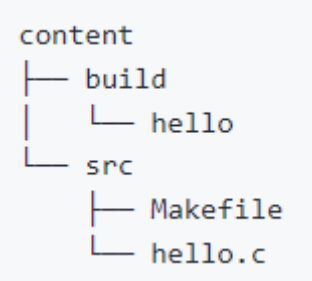



Figure 2.3: C Project Directory Structure
Source: [16]

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 [15]. 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* [14].

### 2.1.5 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.

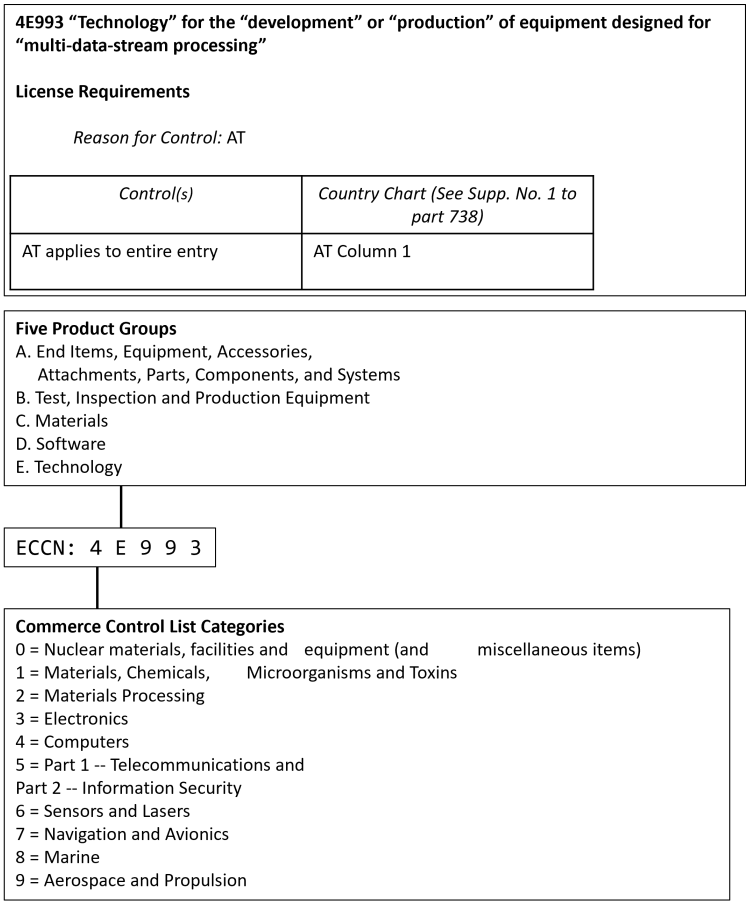



Figure 2.4: ECCN Structure
Source: [17]

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 [17].

A lot of these standards could be explained in much more detail (at the moment, they are kept just as long as currently deemed necessary to understand the further topics).

## 2.2 Context at SAP Gardener

Builds, Repositories, Gardener Servicelet Pattern (Session mit Uwe)

### 2.2.1 SAP Gardener

### 2.2.2 SAP Gardener Deployment Scenario

### 2.2.3 Open Component Model

## 2.3 Integration into the Application Lifecycle

# Chapter 3

# State of the Art

## 3.1 Common Approach of the Industry

## 3.2 Current Approach at SAP Gardener

## 3.3 Problems of Existing Approaches

# Chapter 4

# 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.

## 4.1 Requirements

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

### 4.1.1 Functional Requirements

| **A1: API** | | | |
|---|---|---|---|
| **Ref.#** | **Functionality** | **Category** | **Priority** |

Table 4.1: API Requirements

The SCDL must be able to consume data from multiple different data sources over the internet. The SCDL must be able to provide a SBOM of the components to the user.

## 4.2 Data Model

## 4.3 Database

## 4.4 API

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

# Chapter 5

# Implementation

# Chapter 6

# Results

## 6.1 Evaluation of the Security and Compliance Data Lake

## 6.2 Conclusion and Outlook