Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Alex Mantel

Risk Management System Prototype based on an Extended Time-To-Compromise (TTC) Metric

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Alex Mantel

Risk Management System Prototype based on an Extended Time-To-Compromise (TTC) Metric

Bachelorthesis eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Klaus-Peter Kossakowski
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 14. Juli 2018

Alex Mantel

Thema der Arbeit

Risk Management System Prototype based on an Extended Time-To-Compromise (TTC) Metric

Stichworte

IT-Sicherheit, Metriken, Time to Compromise, Modellierung, Risiko Management, Prototyp, β-Time-To-Compromise, Schwachstellenmanagement, CVE, CPE, Version

Kurzzusammenfassung

Ziel dieser Arbeit war es ein Riskomanagementsystem zu entwickeln, welches sich der Metrik β-Time-To-Compromise (Andrej Zieger (2018)) und einer *Advisory Datenbank* bedient. Von der Auflistung der Anforderungen bis hin zur Implementation wurde ein Riskomanagement Prozess begleitet (Stoneburner u. a. (2002)). Ein System soll demnach für eine Bewertung persistent modelliert werden und nicht, wie es bisher üblich ist, aus flüchtigen Schwachstellenscans. Die Arbeit hat den Anspruch für Sicherheitsbeauftragte, Riskomanager und Critical Emergency Response Teams interessant zu sein.

Alex Mantel

Title of the paper

Risk Management System Prototype based on an Extended Time-To-Compromise (TTC) Metric

Keywords

IT-Security, Metrics, Time to Compromise, Asset Modeling, Risk Management, Prototype, β-Time-To-Compromise, Vulnerability Management, CVE, CPE, Version

Abstract

Goal of this work was the creation of a risk management system, with the use of β-Time-To-Compromise metric (Andrej Zieger (2018)) and an *advisory datbase*. During the process from requirements analysis till the end of the iplementation we aim to support a risk management process (Stoneburner u. a. (2002)). A system is rated by a persistent model, not as common by a impersistent vulnerabiliy scan. This work is relevant for security managers, risk managers and critical emergency response teams.

Contents

List of Figures

Listings

Acknowledgments

I never lose. I either win or I learn.

Nelson Mandela

Thank you for your patience and love in every situation,
for having bad fights and good laughs
and for standing by me.

Anna Mantel
Eleonora Mantel
Alexander Mantel
Niko Thielebein
Mirco Helbing

I am grateful to my thesis mentor Andrej Zieger for the mutual understanding und sympathy.

1 Introduction

Taking care of computer systems within the context of it-security is a task we want to make easier. New releasing products, workflows, requirements and the discovery of new vulnerabilities are continuously changing factors of that task. Depending on the domain, people are not always aware of software which is deprecating or becoming vulnerable. Also not security related domain experts, do tend to underestimate the risk of compromising. The common way of vulnerability management is using scanners e.g. metasploit in order to rate eventually found vulnerabilities. According to the impact of an asset, those vulnerabilities are not weighted in a reasonable way. Thus they indicate a potential vulnerability of an asset, not the actual risk of compromising. Our approach is modeling an asset and estimate the risk of that asset, by assigning an impact to it. That way asset informations and vulnerabilities are kept persistent and not temporary within the scan context. Which promises a faster vulnerability detection. For the risk estimation a metric named β-Time-To-Compromise is used, because of its simplicity.

1.1 Goal

Objective of this work is the creation of a risk management system named *ettcs*. *ettcs* is an acronym for Extended Time-To-Compromise System. This system is supporting a risk management process, by offering two interfaces to the user. First the asset modeling and impact assigning. Second a risk overview, based on β-Time-To-Compromise and control recommendation. All that using an advisory database, which is providing details about vulnerabilities of specific products. The resulting software product is a prototype, as a first implementation iteration. Part of that iteration is the description, requirements analysis, design and implementation. Finding and naming problems and possible flaws is the actual goal of this prototype.

1.2 Thesis structure

This work is organized as follows. Section 2 describes the process *ettcs* is supporting. It also provides an overview of the terminology, it-security metrics and details of vulnerability management. Within section 3, the requirements analysis, we describe a given advisory database, list requirements regarding the risk management process and analyse an existing vulnerability dashboard which is based on CVSS (Mell u. a. (2007)). Those requirements are considered in the system design, which is described in section 4. Not only the general architecture and its components descriptions finds explanation here, also used frameworks are introduced. The general implementation of the prototype, which is described in section 5, contains partial code information. Also our way to manage microservices for this prototype is introduced, as well as a brief description of every component implementation and an interface overview. Finally we evaluate and conclude with problems occurred during the creation process of the prototype, in the last section 6 and give a brief outlook of future related work.

2 Vulnerability management

First of all we give an introduction about the terminology of vulnerability management. Also we will have a look at related it-security metrics including the Time-To-Compromise and its extension, the β-Time-To-Compromise.

2.1 Background

Within this section we introduce well known abbreviations and ubiquitous words within vulnerability management, as well as their purpose and issues.

2.1.1 Risk

Risk within the context of information technology systems has been described by Stoneburner u. a. (2002, Section 3) as follows:

> *Risk* is a function of the *likelihood* of a given threat-source's exercising a particular potential *vulnerability*, and the resulting *impact* of that *adverse event* on the organization.

Within this thesis such an *adverse event* is the purposed exploitation of an vulnerability for a computer system. So the only risk we handle is the risk of an compromising asset. Also defined by Stoneburner u. a. (2002) is a process to manage risks within information technology systems:

1. *System Characterization* is the description of the asset.
2. *Threat Identification* is naming a possible threat. As mentioned before, in our case it is the *adverse event*, that a person is exploiting a vulnerable asset. For instance a machine hosting a customer database becomes compromised.
3. *Vulnerability Identification* is the active search of vulnerabilities.

4. *Control Analysis* of current and planed actions to minimize the *likelihood* and therefore the possible *impact*.
5. *Likelihood Determination* of the probability that the threat is actually occurring.
6. *Impact Analysis* is the possible damage that might happen regarding the loss of three criteria: Integrity, Availability and Confidentially.
7. *Risk Determination* known as the combination of *Likelihood Determination* and *Impact Analysis*. Determined within a *Risk-Level Matrix*, resulting into three possible results: *high*, *medium* and *low*.
8. *Control Recommendations* changing the next *Control Analysis*. Quite common is an upgrade or a temporary disable of the component.
9. *Results Documentation* - "Once the risk assessment has been completed [...], the results should be documented in an official report or briefing."

2.1.2 Risk Management- and Vulnerability Management System (VMS)

Vulnerability management systems like nmap (Lyon (2009)), metasploit (Maynor (2011)) or w3af (Riancho (2011)) are also named as vulnerability scanners or audit tools. Those systems do help with the process step *Vulnerability identification*, but not with the actual risk handling since they take no asset impact into account. Also a setup, manual run and interpretation of the resulting CVSS scores are often required. Which takes too much effort for efficient risk management. That said, non-technical stakeholders do need support.

2.1.3 Common Vulnerabilities and Exposure (CVE)

Luis Alberto Benthin Sanguino (2017) gives a good introduction about Common Vulnerabilities and Exposures (CVEs): "CVE, [...] is a method used to assign identifiers to publicly known vulnerabilities found in IT products and to provide information (e.g., affected products) about the vulnerabilities. Across organizations, anti-virus vendors, and security experts, CVE has become the de facto standard to share information on known vulnerabilities and exposures." Within this work we do use CVE and vulnerability as a synonym. Identifying products which are vulnerable to a known vulnerability is done by CPEs.

2.1.4 Common Platform Enumeration (CPE)

A CPE Name from now on simply called CPE, specified by Andrew Buttner (2009) is an abbreviation for Common Platform Enumeration. Currently there are two CPE specifications for version 2.2 and 2.3 (Brant A. Cheikes (2011)) available. In order to enable matching of it-products and give them a common name, CPEs have been specified. Within that order are those fields separated by a colon as shown in listing 2.1. In that listing we two CPE a common known software product named Windows Vista. Its vendor Microsoft does not supply a version nor update informations. Thus those fields are blank, as we can see colons next to each other.

```
cpe:/o:microsoft:windows:vista::x86-enterprise
cpe:/o:microsoft:windows_vista:::x86-enterprise
```

Listing 2.1: CPE 2.2 examples. First one is has been deprecated by the second one. The version field has been wrongly used.

Also specified is a CPE deprecation process, which allows to mark an CPE as invalid and state a replacement, as it can be seen in listing 2.1. However, the first CPE wrongly uses the version field to give information about the operating system. Deprecated through the deprecation process, it is invalid and has been replaced by the second CPE. To address a few purposes of this deprecation process. A product can be renamed or an existing CPE might contain a typo. It does hold following information as specified by Andrew Buttner (2009):

1. **Part** specifies what kind it is, separated into three sections. It is either a **h**ardware component, **o**perating system part or an **a**pplication part.
2. **Vendor** is the vendor or supplier of the platform.
3. **Product**s common known name.
4. **Version** to state a release.
5. **Update** is used for update or service pack information.
6. **Edition** is used to determine the products edition. For instance Windows XP *pro*. Therefore equal CPEs with different are different products.

We do note that the version and update are fields to mark a history for a product. In fact both fields do hold information about a release state.

2.1.5 CPE dictionary & the National Vulnerability Database (NVD)

The united states of America run an public database which lists known vulnerabilities and also a public database for CPE entries. This National Vulnerability Database is abbreviated to NVD and contains informations about vulnerabilities, their related CPEs and a CVSS scoring. Similar to Section 2.1.3 gives Andrew Buttner (2009) a great introduction: "Its purpose is to provide a source of all known CPE Names as well as bind descriptive prose and diagnostic tests to a CPE Name."

2.2 Matching CPEs and CVEs

Matching CPEs and CVEs is not easy as specified by Andrew Buttner (2009). As described by Luis Alberto Benthin Sanguino (2017), there are different issues:

- **CVE entries without CPE entries** are a problem, since VMS do only use CPEs to match CVEs to CPEs.
- **Software Products without assigned CPE** does concern us, because we might have software running without such an identification.
- **CPE Dictionary Deprecation Process** makes it harder to match, since values of the identifie and therefore the identifier itself, r might change.
- **NVD Synchronization** is the exchange of information between the official CPE dictionary and the NVD. So there are at least two seperate services creating CPEs. Mistakes can be made on both sides.

A lot of effort is placed into CPE care. We say that the current CPE specification are representing CPEs as domain keys, since they hold the name of the product, the version and more information. Having a look from the database perspective we say that, in fact every CPE is an actual database row, **without** referential integrity and an unique identifier. We also say that the second state *Software Products without assigned CPE* can not be avoided, since it-companies do write, maintain and manage their own software or software modules.

A system named Inventory Vulnerability Analysis (IVA) has also been introduced by Luis Alberto Benthin Sanguino (2017). That system matches CPEs without total correctness.

2.3 IT-Security metrics

Within this section we give a brief overview of some IT-security metrics, including CVSS, time-to-compromise metric and its extension the β-TTC.

2.3.1 Common Vulnerability Scoring System (CVSS)

Common Vulnerability Scoring System (CVSS) is a metric which is specifying the possible danger of a vulnerability. It is the result of the base metric group, temporal metric group and the environmental metric group (Mell u. a. (2007)). Ignoring the CVSS version, CVSS became the standard in vulnerability ranking. It supplies a scoring between 0.0 and 10.0. It provides an answer to the question: *How dangerous is this specific vulnerability?* To point it out, note that CVSS does not take an system into account.

2.3.2 Time-To-Compromise (TTC)

Time-To-Compromise defined by McQueen u. a. (2006) is a it-security metric which rates the security-state of an asset. In contrast the CVSS metric we did mention before is rating vulnerabilities. It states the time until an *adverse event* (Section 2.1.1) occurs. To sum possible scenarios which lead to such an event, three possible compromise processes have been described. Process 1 for an existing vulnerability and an existing exploit by hand, as the most simple attack. Process 2 for an existing vulnerability without an existing exploit. Process 3 without an existing vulnerability nor an existing exploit, also known as a zero day exploit. Note that TTC is measured as a daily unit. The TTC-metrics gives an answer to the question: *How long does it take to compromise a system with v vulnerabilities, when the total amount of known vulnerabilities is k and the attacker has a skill of s?* Following definition is a summary of McQueen u. a. (2006) made by Andrej Zieger (2018).

Definition 1 (Original Time-to-compromise (TTC) (see McQueen u. a. (2006))) *Let $\mathcal{S} = \{novice, beginner, intermediate, expert\}$ be a set of discrete skill levels. Let the number of vulnerabilities of the component be $v \in \mathbb{N}$, and let $s \in \mathcal{S}$ be the skill level of the adversary, and $k \in \mathbb{N}$ the total number of vulnerabilities, then the* time-to-compromise

$$TTC : \mathbb{N} \times \mathcal{S} \times \mathbb{N} \rightarrow \mathbb{R}$$

is defined as

$$TTC(v, s, k) = t_1 \cdot P_1 + t_2 \cdot (1 - P_1) \cdot (1 - u) + t_3 \cdot u \cdot (1 - P_1)$$

with

- $t_1 = c_1$ *as the average time it takes to tune an available exploit,*
- $P_1 = 1 - e^{-v \cdot \frac{m(s)}{k}}$ *as the probability to have an exploit at hand,*
- $t_2 = c_2 \cdot E(v, s)$ *as the time to write an exploit times the estimated tries it takes,*
- $t_3 = (\frac{1}{f(s)} - 0.5) \cdot c_3 + c_2$ *as the time to find/wait for a new usable vulnerability and creation of a working exploit, and*
- $u = (1 - f(s))^v$ *as the probability that process 2 is unsuccessful.*

The estimated tries in process 2 as a function of vulnerabilities and skill are defined as follows:

$$E(v, s) = f(s) \cdot \left(1 + \sum_{t=2}^{v - v \cdot f(s) + 1} \left[t \cdot \prod_{i=2}^{t} \left(\frac{v \cdot (1 - f(s)) - i + 2}{v - i + 1}\right)\right]\right)$$

with f and m as function of skill:

- $m : \mathcal{S} \to \mathbb{N}$ *giving the number of readily available exploits for the given skill level,*
- $f : \mathcal{S} \to [0, 1]$ *providing the fraction of vulnerabilities usable at the given skill level,*

with the following constants:

- $c_1 = 1\text{d}$ *as the time to tune and use a readily available exploit*
- $c_2 = 5.8\text{d}$ *as the time to develop a new exploit*
- $c_3 = 32.42\text{d}$ *as the avg. time for a new vulnerability to occur*

Fixing the TTC

Andrej Zieger (2018) found problems regarding TTC and improved the TTC defined before, by doing following steps:

1. Making TTC continuous by replacing the four given skills $\{$novice, beginner, intermediate, expert$\}$ with an interval of $[0, 1]$.
2. Making TTC monotonous by adjusting the function for estimation of tries E (See definition 1) to result in natural numbers only.

$$\xi(a, v) = \frac{a}{v} \cdot \left(1 + \sum_{t=2}^{\lfloor v \cdot (1 - \frac{a}{v}) \rfloor + 1} \left[t \cdot \prod_{i=2}^{t} \left(\frac{v \cdot (1 - \frac{a}{v}) - i + 2}{v - i + 1}\right)\right]\right)$$

$$E(s,v) = \xi(\lfloor f(s)\cdot v\rfloor, v)\cdot(\lceil f(s)\cdot v\rceil - f(s)\cdot v) + \xi(\lceil f(s)\cdot v\rceil, v)\cdot(1 - \lceil f(s)\cdot v\rceil + f(s)\cdot v)$$

2.3.3 β-Time-To-Compromise (βTTC)

Same as the TTC defined before, the βTTC defined by Andrej Zieger (2018), is a metric which provides information about the security-state of an asset. It takes two arguments already known from TTC. First one v which is the amount of known vulnerabilities for an asset and as the second one, k which is the total amount of vulnerabilities.

Comparing TTC to βTTC, the parameter s (skill) is not passed as an argument to β-TTC, as you can see in Definition 2. Instead a β-distribution with the values $\alpha = 1.5$ and $\beta = 2$ is used as skill. The area under the curve of the β-distribution as the skill distribution, multiplied with the TTC is a abstraction of the classic TTC. By doing so all skills of attackers are attacking at the same time, with a certain probability.

Definition 2 (β-time-to-compromise (see Andrej Zieger (2018)))

$$\beta TTC(v,k) = \int_0^1 TTC(v,s,k)\cdot \mathrm{Beta}_{1.5,2.0}(s)ds$$

This metric gives an answer to the same question for the TTC (Section 2.3.2) without the variable s for the skill: *How long does it take to compromise an asset?*

2.3.4 Accepting that time is running

By using any TTC metric we do generate a static number. When ever a vulnerability is added, this number decreases, meaning that the systems compromising risk increased. However, we should also keep in mind that the risk of a running system is increasing over time as well. In other words: The chances for a system to become compromised are increasing over time.

Therefore we suggest to decrease the TTC over time. Since McQueen u. a. (2006) described the TTC to be in a *daily* unit, we decrease the TTC daily. However, another question comes into our mind, when we follow this suggestion: *At which point of time does the risk start, that an asset becomes compromised?* Zieger verbally approached the time of the last published CVE. Using the oldest known vulnerability might be a good idea. Recollecting that the TTC does actually work without a known vulnerability, we would have a discontinuity. For now, we leave this question open. A discussion can be found in Section 3.2.7.

2.3.5 Typed Time-To-Compromise

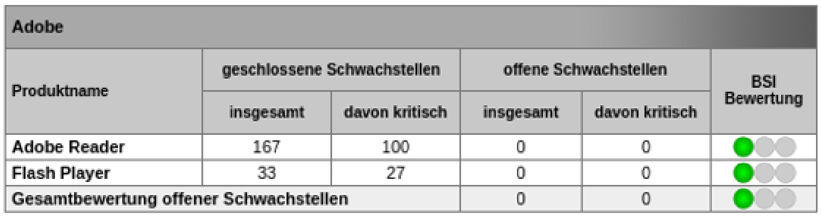
Also introduced by Andrej Zieger (2018) is a Time-To-Compromise typed by the three impact types confidentially, availability and integrity. A fourth type for *execution* has been introduced as well. All those typed TTC metrics rely on a subset of the vulnerabilities. Such a subset is partitioned by the type of a vulnerability.

2.4 BSI-Schwachstellenampel

The German agency *Bundesamt für Sicherheit in der Informationstechnik (BSI)* is hosting a system called *BSI-Schwachstellenampel* (Vulnerability-trafficlight). That system shows informations on a defined set of software products, grouped by vendors. It rates those systems on the current state of their vulnerabilities by the CVSS (Section 2.3.1) metric. Every row has a traffic light which visualizes the risk for the product. Within BSI (2012) three thresholds have been defined for the traffic light:

- If there exists one critical vulnerability with a CVSS score bigger then 7.0, the light turns red.
- It turns yellow when an minor-critical vulnerability with a CVSS between 4.0 and 6.9 does exist.
- Else the light is green. It represents the state of not existing vulnerability or minimal-critical vulnerabilities.

However, those vulnerabilities are determined for the most recent version or update of a product. In figure 2.4 we see an example of the vendor Adobe with the two products Adobe Reader and Flash Player. So this light gives an answer to the following question: *Exists a unpatched, critical or minor-critical vulnerability for this product at this moment?*



Adobe					
Produktname	geschlossene Schwachstellen		offene Schwachstellen		BSI Bewertung
	insgesamt	davon kritisch	insgesamt	davon kritisch	
Adobe Reader	167	100	0	0	🟢⚪⚪
Flash Player	33	27	0	0	🟢⚪⚪
Gesamtbewertung offener Schwachstellen			0	0	🟢⚪⚪

Figure 2.1: Screenshot of BSI-Schwachstellenampel specified for a few Adobe products

3 Requirements analysis

After we gave an introduction into vulnerability management, including a few it-security metrics, we do analyse given components and existing software. An advisory database containing vulnerability and product information belongs to such existing software. Same as the already introduced BSI-Schwachstellenampel.

3.1 Given advisory database

The only component we have to analyse of the actual state is an advisory database provided by the DFN CERT Services GmbH. A postgres database of version 9.3. It does hold information about CPEs, vulnerabilities including CVSS scores and also the advisories itself. In figure 3.1 a brief overview of the following tables is given. Note that not all tables of the database are shown. Tables we actually need are: `cpe_vendor` as the vendors of `cpe_products`. `cpe_cpe` for CPEs itself, is related to `authoring_vulnerability` trough the table `authoring_vulnerabilitycperelation`. So a vulnerability can occur in more then one CPE and a CPE can has more then one vulnerability. Some tables have a field with a name ending to `displayname`, which are fields for UI elements of another application. Those displayfields are not complete for every `cpe_cpe`. Another field worth stating is the `comparsion` within `authoring_vulnerability`. It contains order relations, to make vulnerability-range specifications possible. Those are useful for specifying which CPEs are effected. No write permissions will be granted on the advisory database in production.

As described before, CPEs do have a deprecation process. That process is taken care of within table `cpe_cpe` trough the fields `deprecated_by_id` and `deprecated_on_date`.

With that given advisory database we do have the luck, that we do not have to match CVEs and CPEs on our own, since `authoring_vulnerabilitycperelation` does state the relation already.

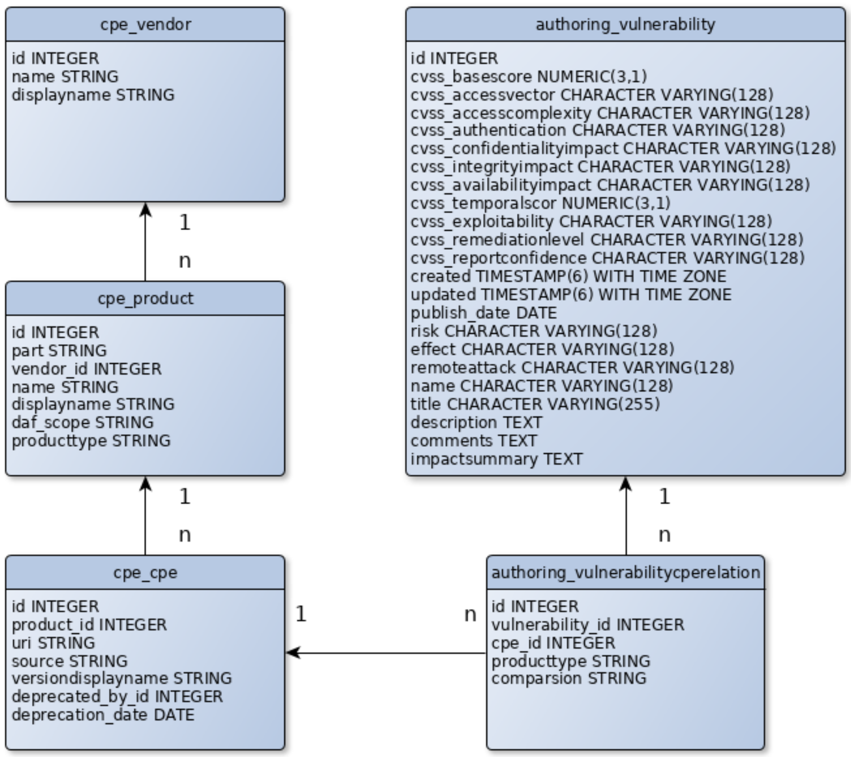

cpe_vendor
id INTEGER
name STRING
displayname STRING
1
n
cpe_product
id INTEGER
part STRING
vendor_id INTEGER
name STRING
displayname STRING
daf_scope STRING
producttype STRING
1
n
cpe_cpe
id INTEGER
product_id INTEGER
uri STRING
source STRING
versiondisplayname STRING
deprecated_by_id INTEGER
deprecation_date DATE
1
n
authoring_vulnerability
id INTEGER
cvss_basescore NUMERIC(3,1)
cvss_accessvector CHARACTER VARYING(128)
cvss_accesscomplexity CHARACTER VARYING(128)
cvss_authentication CHARACTER VARYING(128)
cvss_confidentialityimpact CHARACTER VARYING(128)
cvss_integrityimpact CHARACTER VARYING(128)
cvss_availabilityimpact CHARACTER VARYING(128)
cvss_temporalscor NUMERIC(3,1)
cvss_exploitability CHARACTER VARYING(128)
cvss_remediationlevel CHARACTER VARYING(128)
cvss_reportconfidence CHARACTER VARYING(128)
created TIMESTAMP(6) WITH TIME ZONE
updated TIMESTAMP(6) WITH TIME ZONE
publish_date DATE
risk CHARACTER VARYING(128)
effect CHARACTER VARYING(128)
remoteattack CHARACTER VARYING(128)
name CHARACTER VARYING(128)
title CHARACTER VARYING(255)
description TEXT
comments TEXT
impactsummary TEXT
1
n
authoring_vulnerabilitycperelation
id INTEGER
vulnerability_id INTEGER
cpe_id INTEGER
producttype STRING
comparsion STRING

Figure 3.1: Datamodel: Partial entity relationship model of the advisory database

3.2 Supporting the process

We give an overview of how steps of the process introduced in Section 2.1.1 shall be supported by *ettcs*. Following sections do describe the user interaction with *ettcs*. The last process step *Results Documentation* will be skipped by us.

3.2.1 System Characterization

As described in Section 2.1.1, the first step of the risk management process is the *System Characterization*. The system we want to characterize is a computer with running software components. From now on we name such a system an asset.

Within the process a user shall model an asset. To make it identifiable for humans, a unique name and a description are required for the creation. After the initialization components representing a part of the asset might be added.

Adding a component is done by searching for a product with release informations like version and update. But as we figured out before in Section 2.2, not all software products do have a CPE assinged. On first thought, we have the urge to use CPEs to clarify what software that component is.

Modeling that asset shall be extendable in the context computer networks (McQueen u. a. (2006)).

3.2.2 Threat Identification

The step *Threat Identification* can be skipped, since the only *adverse event* we handle is, as already mentioned, the exploitation of a vulnerability. Within the prototype implementation as in this work, we will not take care of other threats.

3.2.3 Vulnerability Identification

Identification of vulnerabilities shall be dropped for the *ettcs* user completely. This is possible, since CERTs and other organizations are gathering information of those vulnerabilities. Using an advisory database, the NVD or any other vulnerability source for identification of new vulnerabilities, we do not have to identify them on our own. Since those vulnerabilities do hold informations about CPEs, *ettcs* shall use that link. That way it only fetches such asset vulnerabilities and makes a manual identification redundant.

3.2.4 Control Analysis

Taking current controls and planned controls into account, is not part of *ettcs*.

3.2.5 Likelihood Determination

Stoneburner u. a. (2002) described a way to partition likelihood into high, medium and low. However, by calculating the TTC of an asset, we in fact do calculate the likelihood. So within *ettcs* the likelihood is determined by βTTC. As we already mentioned in Section 2.3.4, we shall keep in mind, that for every passing day the chances are increasing. By doing that, the user is not part of this process step anymore.

3.2.6 Impact Analysis

The simplified question is "*What is the possible damage when this asset is compromised?*" or even simpler "*How important is this asset?*".

In every organization are assets with more and less business impact. However, the ones we rate an higher impact are simply more important and shall be treated with more carefulness. In order to give the modeler the possibility to rate an impact of an compromised asset, we supply an importance of an asset. So *ettcs* shall not actually help with the *Impact Analysis* process, but helps clarifying which assets have been rated with an importance. Also we ignore the fact, that for a proper analysis we shall categorize for the three aspects integrity, availability and confidentially. We do justify our ignorance, with the fact that the Typed TTCs (Section 2.3.5 shall be implemented in future implementations. However implementing those would be too much for this work and thus it is procrastinated.

Summing up:

- An assets impact is rated by an `importance`.
- `importance` hints the possible impact of a compromised system.
- There are CRUD (Create, Read, Update & Delete) operations for importance.

3.2.7 Risk Determination

Using the importance as impact and the βTTC for the likelihood, we do combine them to *risk*. That combination must be shown to the stakeholders through a dashboard or scorecards.

However, as described in Section 2.3.4 we have an increasing risk over time. We still have to define at which point the risk starts. Remembering the simplified question: *At which point of time does the risk start, that an asset becomes compromised?* Instead of using the time of the last published CVE, we decide to let the user enter a date since when the system has been patched for the last time. Another option is the date of the oldest patch of all components within an asset. For this prototype we choose the first option because of simplicity.

3.2.8 Control Recommendations

The whole goal of advisory management is the actual control recommendation. Most advisories do contain a simple message to either switch off the software, switch off a certain feature or patch the system.

Ideally *ettcs* would find a corresponding advisory, if such an advisory exists. However, instead of matching such an advisory we simplify our task by answering the question: *Exists a newer version of the it-product?*

3.3 Analysis of BSI-Schwachstellenampel

What can we learn from the BSI-Schwachstellenampel which has been introduced in Section 2.4? First of all, the general overview is decent. The traffic light for every product allows to see the danger on first sight. A disadvantage is the grouping by vendors. Overall rating the vendors is not helpful for an estimation of the asset or vulnerability risk. We group a similar view by assets. Also we use the products version/update for the rows, because we might decide not to update a product in production. Another point we do criticize is the column *geschlossene Schwachstellen* (*closed vulnerabilities*), since it gives no security related information to a stakeholder.

We sum up those requirements for the risk presentation:

- An asset represents a host.
- A component is the representation of a running product release.
- A component does belong to an asset.
- A whole asset will be rated trough a traffic light, by its `importance`.
- The `importance` holds two threshold to define which result of the βTTC is *high*, *medium* or *low*. By doing so, we combine the estimated likelyhood with the impact resulting in a risk.
- If a product is vulnerable to an known CVE, that CVE shall be noted.
- There is an overview of all assets in a single view.

3.4 Dashboard and ScoreCards

Before we did define a few requirements for the risk presentation. Whilst mentioning those requirements, we actually talked about dashboards only. However, Shikha Shahi (2018) mentioned that it is a lot of effort for companies to introduce a dashboard and that is questionable from of the information-security perspective to introduce such a public information system. Instead or additional to a dashboard solution, do ScoreCards offer enough privacy, with the same amount of informations. According to Shikha Shahi (2018) shall a ScoreCard fit on a DIN A4 page and contains graphs, e.g. a sparkline.

In figure 3.2 we can see a sparkline for a imaginary TTC history of an asset. That scenarios date is the 19th May 2018 and the running TTC arrived zero. Within that sparkline we can see

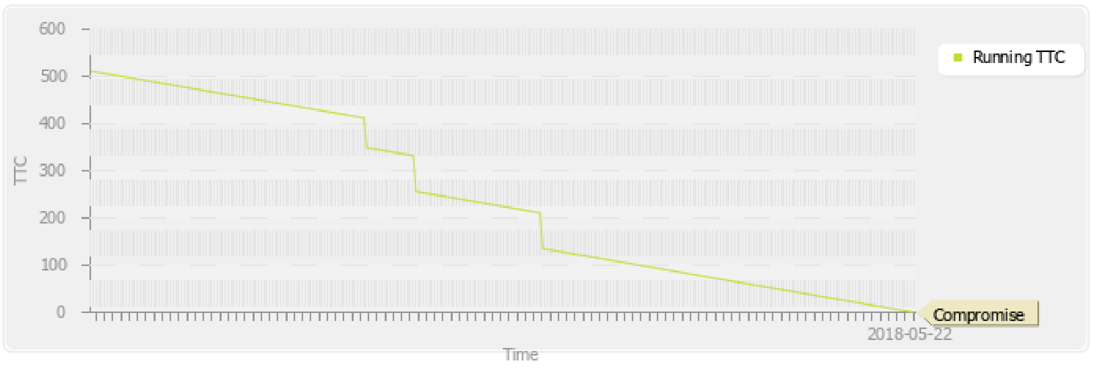


Figure 3.2: Fictional sparkline for TTC

three kinks, indicating at which time vulnerabilities have been published. At the three kinks of that sparkline we can see at which time vulnerabilities have been published.

4 Software design

This chapter gives insight about the design of the prototype. We discuss a few frameworks, including Ruby on Rails, Pythons Django and so on. We have a look at the general architecture, its components and the used frameworks.

4.1 Design goals

What is important to us, while planing the software we are going to write?

- *Low coupling of components and high cohesion of those*
  After prototyping, we might want to keep some components. On the other hand, it is likely that components might get replaced or removed. In order to ease that task, those components shall depend as less as possible and as much as needed to each other. To do so, we separate as much as needed and place
- *Adaptive for distributive systems*
  Asset models are sensible informations. Depending on the organization, there might exist restrictions or policies depending on that data. We set the goal to build *ettcs* distributive, so components hosting sensible data can run within such organizations. Also in future we might want to interact with various vulnerability management or any other systems.
- *Integration of the provided advisory database*
  The provided advisory database must find place and use within the architecture.

4.2 Architecture

Within this section we create and discuss about software architecture for *ettcs* . First we statisfy previously stated requirements, then move to the architecture step by step.

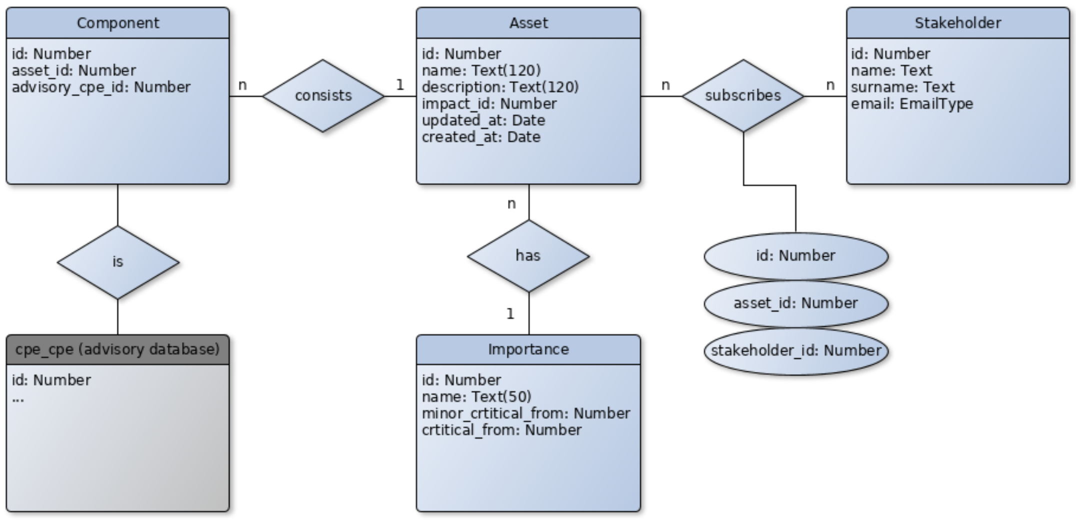


Figure 4.1: Overview of new entities

4.2.1 Adding entities

In figure 4.1 we list the entities to add, resulting from the previously defined requirements. As most important we have the asset entity, which is related to all other entities. It is the container for all running software an has various components running. Those components represent the product on a specific version or update. We simplify the asset modeling task, by using a CPEs supplied from the advisory database. We do rate the whole asset by giving it an specific importance. Every importance holds two fields **minor_critical_from** and **critical_from**. Their values are used by the traffic light, similar to the BSI-Schwachstellenampel (Section 2.4). When the TTC is lower critical threshold, the light is red. When the TTC is lower than the minor-critical threshold and bigger than the critical threshold, the light is yellow. The light is green when TTC bigger then the minor-critical threshold. As the last entity we introduce the stakeholder. Its purpose is adding a recipient for the scorecards.

We group those entities in order to shape them into set boundaries for component contexts. While the advisory database is an external service, we put all their entities into in a separate context. We group the component and asset, because components do aggregate to their asset. Importance is also a part of that context, because an impact without is questionable. Thus the stakeholder is the single, new entity context. However, we also make the relation *subscribes* to an entity we name *Subscription*, as it can be seen in figure 4.2.

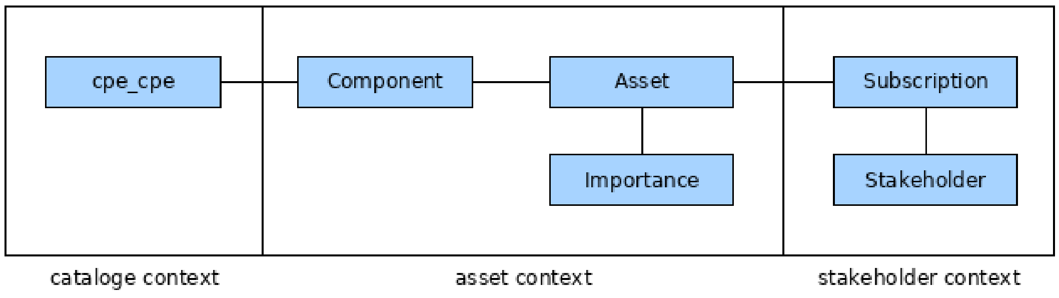


Figure 4.2: Grouped entities by context

4.2.2 Adding components

Tasks like modeling an asset and rating it with an importance, such as adding a stakeholder and subscribing it to an asset are requirements, specified before. However, not listed as an requirement but pretty important to us, is the ease of configuration. Having a single access point for such a configuration is our first design decision. We name it **configuration_service**

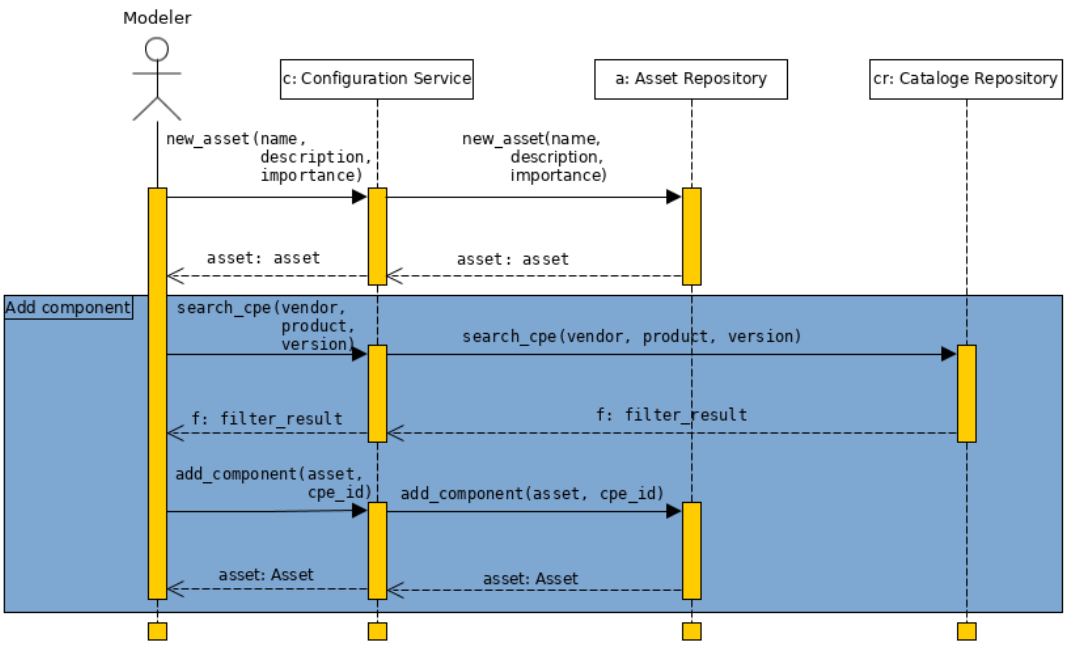

Modeler
c: Configuration Service
a: Asset Repository
cr: Cataloge Repository
new_asset(name, description, importance)
new_asset(name, description, importance)
asset: asset
asset: asset
Add component
search_cpe(vendor, product, version)
search_cpe(vendor, product, version)
f: filter_result
f: filter_result
add_component(asset, cpe_id)
add_component(asset, cpe_id)
asset: Asset
asset: Asset

Figure 4.3: Sequence diagramm of the modeling process

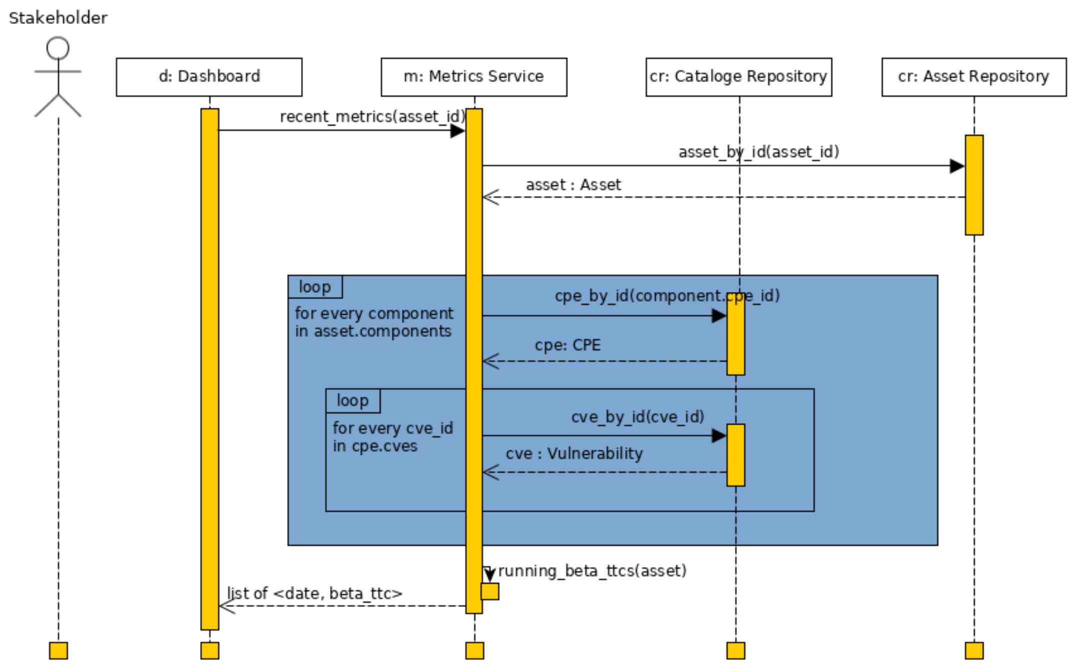


Figure 4.4: Sequence diagramm of fetching βTTC and CVE informations

and specify for now, that it provides a website for those tasks. For the entities stakeholer, importance and asset shall be CRUD operations provided.

One of the most complex communications between the components is the creation of an asset with components. In figure 4.3 we can see such a creation with the search of a component. Note that *filter_results* purpose is providing all informations we want to show, after filtering. We do so to minimize traffic and implementation effort. The three repository services **asset_repository**, **stakeholer_repository** and **catalog_repository** are instances of database adapters for the contexts described in figure 4.2. Note that the catalog offers only read-only operations.

We also specify that the **dashboard**, listed within the requirements is providing the informations trough a website. Justifying that choice with the argument, that we do not know if that system shall run for the stakeholders in a private matter only or in a public place. Also the metrics calculation is done by a separate component named **metrics**.

The last service we describe is the **scorecard_service**. The only purpose is sending an email notification to subscribed stakeholders. That email notification contains a ScoreCard, which is a PDF containing the current dashboard informations.

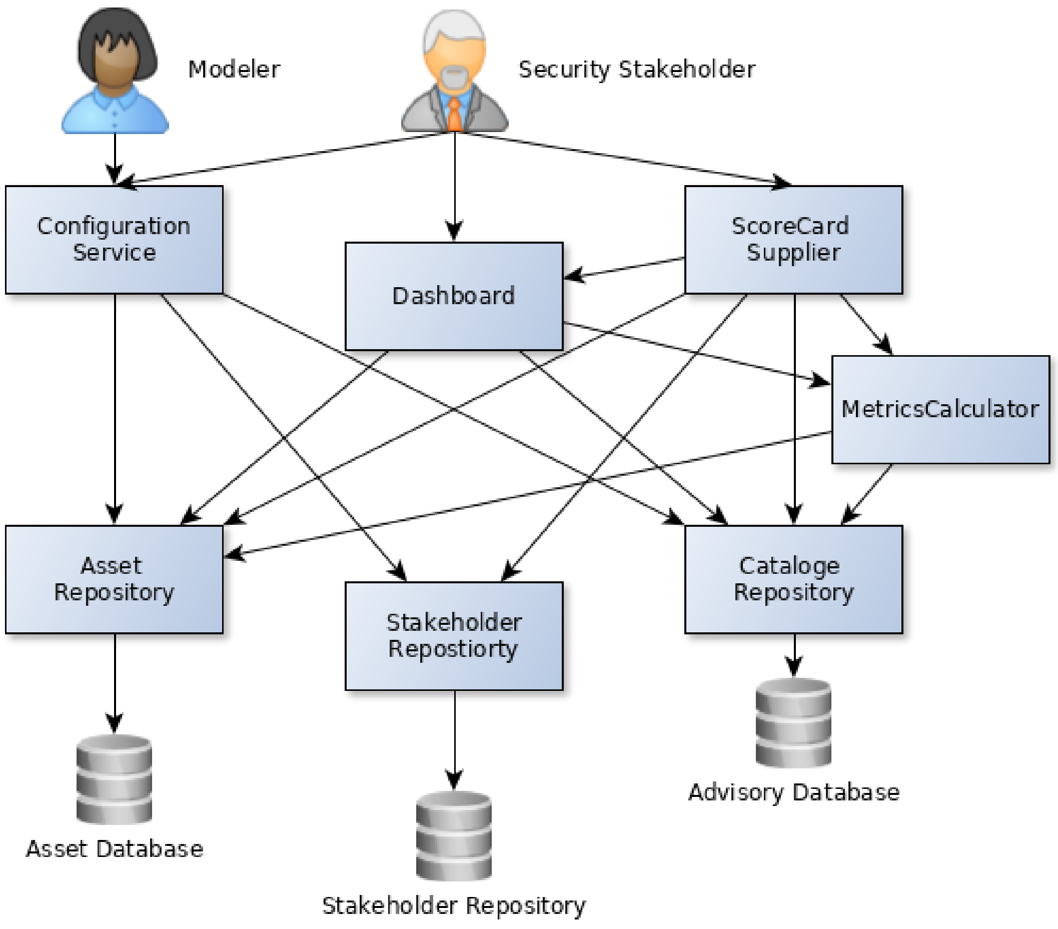


Figure 4.5: Overview of all microservices

An overview of all components and its dependencies specified before, is given in figure 4.5. Since the services **configuration_service**, **dashboard** and **scorecard_service** are there for human interaction, we specify them as interface components.

4.2.3 Choosing a microservice architecture

Since we did set as a design goal to be distributive, we choose a microservice architecture. It is quite common to use RESTful APIs for microservices (Newman (2015)), we just adapt to that decision, because we have no need to do different.

For the repository microservices, we use the classic pattern (Fredrich (2012)). That patterns corresponding CRUD (**C**reate, **R**ead, **U**pdate and **D**elete) equivalences are shown in table 4.2.3.

CRUD	Method	Endpoint	Requestbody	Responsebody
C	POST	/entities	entity to create	created entity with assigned id
R	GET	/entities		all entities
R	GET	/entities/:id		entity with passed id
U	PUT	/entities/:id	entity to update	
D	DELETE	/entities/:id		

4.3 Frameworks & tools

Until now made functional design decisions. Following we give a brief introduction into frameworks we use and the choice of using them.

4.3.1 Web- & RESTful API

What is the best Web- and RESTful API framework in Ruby, for us to prototype with? In order to choose a RESTful API for prototyping, we do compare Ruby on Rails and Sinatra. Ruby on Rails (Bächle und Kirchberg (2007)) is a popular, full stack web-framework. It provides an Object Relational Mapper, named ActiveRecord out of the box. It is also the de facto standard to prototype websites. Rails relies heavily on convention over configuration. Another great point is, that we can patch an application during runtime, without reloading. However, Sinatra (Harris und Haase (2011)) is a plain Web- and RESTful API framework. Same as Rails, it provides templating, a simple routing technique and allows patching during runtime. Within figure 4.1 we see a simple web application, containing the routing to root "/" and returning a "*Hello, World*" within the response body.

```
require 'sinatra'

get '/' do
  "Hello, world!"
end
```

Listing 4.1: Code: "Hello, world!" in Ruby's Sinatra

Since we decided to build microservices, choosing a full stack framework is oversized; even for prototyping. We apply the same argument pattern for the two python frameworks `django` and `flask`. Similar to Ruby on Rails does Django provide a full stack functionality. Flask on the other hand is a simple web framework similar to Ruby's Sinatra.

So for microservices using Ruby, we choose Sinatra as a RESTful API framework. In case of python flask is used.

4.3.2 Object Relational Mapper

What is the best database adapter for us to prototype with? Before in Section 4.3.1, we listed advantages and disadvantages of Rails and Sinatra. Now we discuss which ORM (Object Relational Mapper) we shall use. *ActiveRecord* and *Sequel* are popular Ruby ORMs.

ActiveRecord as part of Rails, also relies on convention over configuration (Pytel u. a. (2007)). Within ActiveRecord the model itself is also the repository. Also the models name is the database tables name.

Adapting to our given advisory database might get out of hand with configuration over configuration.

Sequel on the other hand, is a light weight ORM, without a lot restrictions. Ruby's Hash implementation is used heavily used by Sequel. In listing 4.2 is an example session shown (Evans (2018)). We do choose Sequel, because of it simplicity and flexibility.

```ruby
require 'sequel'

DB = Sequel.sqlite # memory database, requires sqlite3

DB.create_table :items do
  primary_key :id
  String :name
  Float :price
end

items = DB[:items] # Create a dataset
# Populate the table
items.insert(:name => 'abc', :price => rand * 100)
items.insert(:name => 'def', :price => rand * 100)
items.insert(:name => 'ghi', :price => rand * 100)

# Print out the number of records
puts "Item count: #{items.count}"
puts "The average price is: #{items.avg(:price)}"
```

Listing 4.2: Code: Simple use of Ruby's Sequel with a SQLite3 database

4.3.3 wkhtmltopdf

As mentioned before, we do need a functionality to convert the webpage of our dashboard into a PDF, in order to send it to its stakeholders. For converting HTML pages to PDF files, a tool named `wkhtmltopdf` (Truelsen) is used. An example usage is found in listing 4.3, Given arguments specify rotation of the resulting PDF, the source website and the result file.

```
# using vertical paper alignment and a remote address
wkhtmltopdf -O Landscape 'http://www.dfn-cert.de/' 'dfn-cert.pdf'
```

Listing 4.3: Usage of `wkhtmltopdf`

4.3.4 rufus-scheduler

The ScoreCard service has to check the risk of all assets frequently. We manage this frequency with the Ruby gem `rufus-scheduler`. This gem allows us to write cronjobs in Ruby notation. Listing 4.4 contains an example provided by Mettraux (2018) on the project site, in the wild.

```
# quickstart.rb

require 'rufus-scheduler'

scheduler = Rufus::Scheduler.new

scheduler.in '3s' do
  puts 'Hello... Rufus'
end

scheduler.join
# let the current thread join the scheduler thread
```

Listing 4.4: Usage of `rufus-scheduler`

4.3.5 Technology stack

Within figure 4.3.5 we can see all choosen frameworks and applications. When trying to access a wide spectrum of users, like in online shops there is a need to support as many browsers as possible. Since we have no such a need, we set the browser requirements to a minimum version supporting HTML5. HTML5 supplies a date picker out of the box (Pilgrim (2010)). For the cascade style sheet, we do use W3CSS. It is responsive out of the box and easy to handle.

Since the ScoreCard service is actually sending emails, a email client might be shown on the figure.

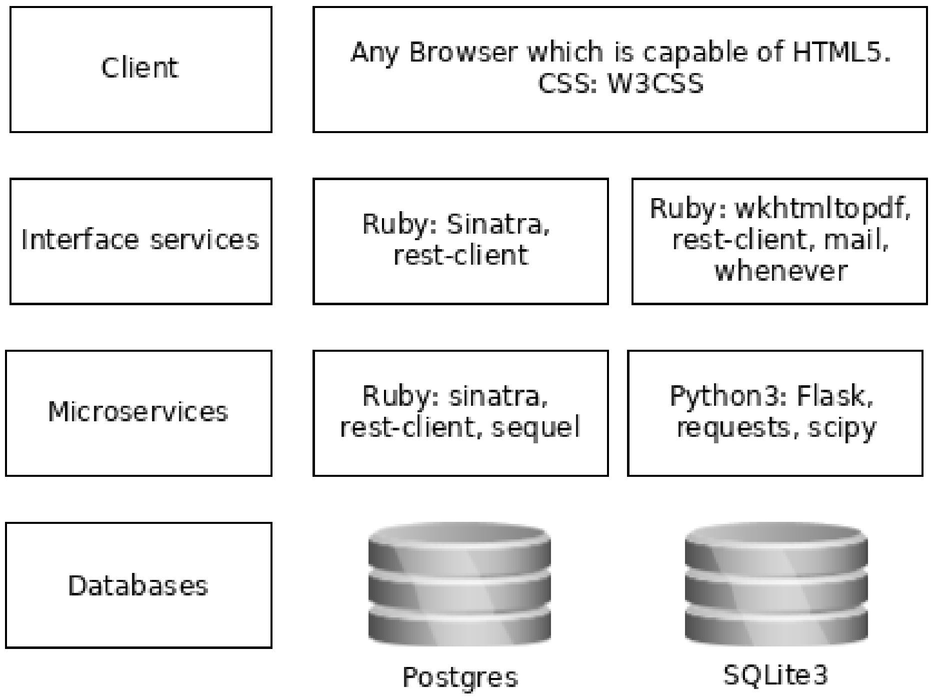

Client
Any Browser which is capable of HTML5.
CSS: W3CSS
Interface services
Ruby: Sinatra, rest-client
Ruby: wkhtmltopdf, rest-client, mail, whenever
Microservices
Ruby: sinatra, rest-client, sequel
Python3: Flask, requests, scipy
Databases
Postgres
SQLite3

Figure 4.6: Technology stack with used frameworks

5 Implementation

Within this chapter we do describe our way to orchestrate microservices and implemented parts of the dashbaord, catalog service, asset model service and the β-TTC. Service, microservice and component are used as synonyms.

5.1 Project structure

Our projects hierarchy is organized as follows. For every component is a single subfolder given, as we can see in figure 5.1. Every component folder contains a *start.sh* script for managing this instance. Depending on the microservice, a *Gemfile* is given in order to state the dependencies. Repository services using *SQLite3* contain the database file, which do end with *´*.db´*.

5.1.1 Managing mircoservices with *compose.sh* and tmux

Docker (Anderson (2015)) is a great tool to orchestrate, deploy and ship software. It became the de facto standard in the microservices world. However, since shipping software is not part of our prototype implementation, we do not have a real need for Docker.

Instead of using docker as a container manager we do use *tmux*, a terminal multiplexer to orchestrate the microservices. Using tmux, all services will run on the same space, with same resources on one single host. We wrote a simple script named `compose.sh` which iterates over all directories within the project and calls a `start.sh` script. It allows us to *start*, *restart* and *stop* all services.

That `start.sh` is started within a tmux session. The script itself is simply starting the service. With those scripts we did replace `docker-compose` for prototype purposes. A similar approach can be found in the wild, using `pmux` (programmable tmux) (Jenson (2015)). Instead of using bash scripts, pmux uses a JSON file for configuration, holding various parameters for orchestration. We decide to not use pmux, because it seems exorbitant for our purposes.

In listing 5.1 we list tmux's equivalent commands compared to docker. Note that within that listing services and containers are used as synonyms.

```
├── asset_repository
│   ├── asset_models.db      # SQLite3 database file for the assets
│   ├── Gemfile              # Common ruby dependency file
│   ├── Gemfile.lock         # Currently installed dependencies with version
│   │                        # and their dependencies
│   ├── lib                  # Folder with content of the code
│   ├── Rakefile             # Common ruby task scripts
│   ├── start.sh             # Start script introduce later
│   ├── test                 # Test files for this service
│   └── vendor               # Installed dependencies
├── cataloge_repository      # Similar to 'asset_repository'
├── compose.sh               # A partial docker-compose replacement, more later
├── configuration_service    # Similar to the dashboard
├── dashboard
│   ├── Gemfile
│   ├── Gemfile.lock
│   ├── lib
│   │   ├── asset_repository_adapter.rb
│   │   ├── cataloge_repository_adapter.rb
│   │   ├── metrics_adapter.rb
│   │   ├── public           # Classic HTML/public folder containing JavaScript,
│   │   │                    # images and CSS
│   │   ├── router.rb        # File to route HTTP requests to methods
│   │   └── views            # Folder containing HTML templates to render
│   ├── Rakefile
│   ├── start.sh
│   ├── test
│   └── vendor
├── LICENSE
├── metrix
│   ├── asset_repository_adapter.py
│   ├── beta_ttc.py
│   ├── cataloge_repository_adapter.py
│   ├── models
│   ├── packages
│   ├── router.py
│   └── start.sh
├── README.md
├── scorecard_supplier
└── stakeholder_repository
```

Figure 5.1: Overview of the projectstructure

```
# list running containers
tmux ls
docker ps

# start a specific service. in this example 'catalog_repository'.
cd catalog_repository;
  tmux new-session -d -s "catalog_repository" ./start.sh "start";
  cd ..
docker start "catalog_repository"

# start within container 'catalog_repository' a bash shell.
# not needed with tmux bacause we do run within the same environment.
docker exec -t -i "catalog_repository" /bin/bash

# stop service/container
tmux kill-session -t "catalog_repository"
docker kill "catalog_repository"

# start all services/containers
./compose.sh start
docker-compose up

# stop all services/containers
./compose.sh stop
docker-compose down
```

Listing 5.1: Code: Simple service management with tmux

5.2 Services

Within this section we describe details about the implemented services. At the end of every service description, we discuss service related issues and problems occurred during the implementation.

5.2.1 Stakeholder Repository and Asset Repository Service

We list these services in one section, because they have the same technical structure. Both repositories supply CRUD operations for their assigned entities, which have been described in section 4.2.1. Those CRUD operations fulfil the presented RESTful API operations described in section 4.2.3. Also both services do have a *SQLite3* database backend. The initialization of the database in the *Stakeholder Service* is shown in listing 5.2.

```
if ARGV.include? "init"

  DB.create_table :stakeholders do
    primary_key :id
    String      :name
    String      :surname
    String      :email
  end

  DB.create_table :subscriptions do
    primary_key :id
    foreign_key :stakeholder_id, :stakeholders
    # id of advisory productversion
    Integer     :asset_id
  end

end
```

Listing 5.2: Code: Snippet of router.rb in stakeholder_repository

Our **stakeholder_repository** does supply those URIs for the entity *stakeholder* and *subscription*. Similar does the **asset_repository** supply it for *asset*, *component* and *importance*

```
# StakeholderRepository#update
# Hash -> Hash
def update(stakeholder)
  @stakeholder_policy.verify(stakeholder)
  @stakeholders.where(id: stakeholder["id"].to_i).
                update(name: stakeholder["name"],
                       surname: stakeholder["surname"],
                       email: stakeholder["email"])

end
```

Listing 5.3: Code: Snippet of stakholder_repository.rb

In listing 5.3, see the update implementation of the CRUD operations. `@stakeholders` is a Sequel representation of the stakeholders table, on which SQL-like operations are executed.

Instead of implementing models, we do use Ruby's Hash implementation. For model validation we do use the `verify` method, which is a implementation of the policy pattern (Evans (2004)). In fact `@stakeholder_policy` contains policy methods only. For instance the verification of the email type.

5.2.2 Catalog Repository Service

The **catalog_repository** supplies *GET* requests only, because of the read-only permission of the advisory database. Within table 5.2.2 we see all those supplied resources. Corresponding from the read-only access ony are possible. Note that in no case a request body has been supplied and therefore not shown within the table.

Method	Endpoint	Request Parameters	Responsebody
GET	/cpes/:id	vendor, product, version	cpe with related vulnerabilities
GET	/products/:id		product containing a list of possible vulnerabilities
GET	/vulnerabilies/:id		vulnerability informations
GET	/vulnerabilies		vulnerability statistics including the total amount of all vulnerabilities

CPE search

In order to search within the advisory database for CPE entries, we filter those with the SQL statement in listing 5.4. Since such a query would be complicated and inefficent to write in Sequel notation, we pass it a postgres SQL statement. We join the tables *cpe_cpe*, *cpe_cpeproduct* and *cpe_cpevendor*. After joining those, we filter with the passed parameters for vendor, product and version. Those parameters can be seen within the where clause as `:vendor`, `:product` and `:version`. We pass that query to Sequel, the parameters are prepared to prevent SQL injection attacks. Also, a modeler must not model assets with deprecated CPEs. Within the where clause, as the last conjunction we ensure that no deprecated CPEs will be shown when filtering.
That query might be optimized by restrict the sets before joining them.

```
select cpe_cpe.id                      as cpe_id,
       cpe_cpe.uri                     as cpe_uri,
       cpe_cpe.versiondisplayname      as cpe_version,
       prod_vendor.vendor_id           as vendor_id,
       prod_vendor.vendor_displayname  as vendor_displayname,
       prod_vendor.vendor_name         as vendor_name,
       prod_vendor.product_id          as product_id,
       prod_vendor.product_name        as product_name,
       prod_vendor.product_displayname as product_displayname
from cpe_cpe full outer join (
  select cpe_cpeproduct.id             as product_id,
         cpe_cpeproduct.name           as product_name,
         cpe_cpeproduct.displayname    as product_displayname,
         cpe_cpevendor.id              as vendor_id,
         cpe_cpevendor.displayname     as vendor_displayname,
         cpe_cpevendor.name            as vendor_name
  from cpe_cpeproduct full outer join cpe_cpevendor
  on cpe_cpeproduct.vendor_id = cpe_cpevendor.id
) as prod_vendor on cpe_cpe.product_id = prod_vendor.product_id
where (vendor_name like '%' || :vendor || '%'
       or vendor_displayname like '%' || :vendor || '%') and
      (product_name like '%' || :product || '%'
       or product_displayname like '%' || :product || '%') and
      (cpe_cpe.versiondisplayname like '%' || :version || '%'
       or cpe_cpe.uri like '%' || :version || '%') and
       cpe_cpe.deprecated_by_id is null"
```

Listing 5.4: Code: CPE filter with argument preperation

Fetching vulnerabilities and successor CPEs

Whilst we do not use the `comparsion` field provided by the advisory database, for finding CPEs which are threatened by a vulnerability, a workaround is needed. We set following rule:

Every vulnerability of every successor CPEs, are also vulnerabilities for a specific vulnerability.

We implement this rule by first fetching all successor CPEs, then merge all vulnerabilities of the successor CPEs into a set. That way we have a unique set for a CPE. All that is done within the class `CpeRepository`. Also the determination of successor CPEs is done in this class.

We find the successor CPEs of a CPE by selecting all CPEs then selecting those which have a higher version number. Selecting those CPEs is predicated with the method `newer_version`?. Its implementation can be found in listing 5.5. Similar implementations can be found for ruby in the wild (`https://github.com/dazuma/versionomy`). However, we later argue why we will drop this in another iteration.

The method `newer_versions` within `catalog_repository` takes a given CPE and returns a list of CPEs for the same, linked product with the fact of a higher, lexical version number. This will also compare characters at the end of the version.

```ruby
#   "1.1.1"  > "1.1"
#   "1.1.1a" > "1.1.1"
#   "1.1.1b" > "1.1.1a"
#
# str x str => bool
def self.newer_version?(new_version, old_version)
  return false if new_version.nil? || new_version == ""
  return true  if old_version.nil? || old_version == ""

  splited_new_version = new_version.split(".")
  splited_old_version = old_version.split(".")
  splited_new_version.each.with_index do |new_num, index|
    old_num = splited_old_version[index]
    # if old_num does not exist, left one is 'longer' thus newer :)
    if     !old_num           then return true
    elsif new_num > old_num then return true
    elsif old_num > new_num then return false
    end
  end
  false
end
```

Listing 5.5: Code: An order implementation named `newer_version?`

Also we want to state that this must be changed in future versions. As with a release of the operating system openSUSE Brown (2017) wrote as follows:

“On behalf of the openSUSE Board and Leap Release Management I am pleased to announce the next version of openSUSE Leap after 42.3 will be: openSUSE Leap 15”

So the successor of version 42.3 is 15. This fact leads to a need of a version database which contains version links as successors.

5.2.3 Metrics Service

We do use the βTTC implementation provided by Andrej Zieger (2018) for the **metrics_service**. The service contains a python module for βTTC calculation. In order to supply a running TTC, we need to keep the vulnerabilities and the daily risk increasion in mind. In fact we calculate all values for the sparkline on demand.

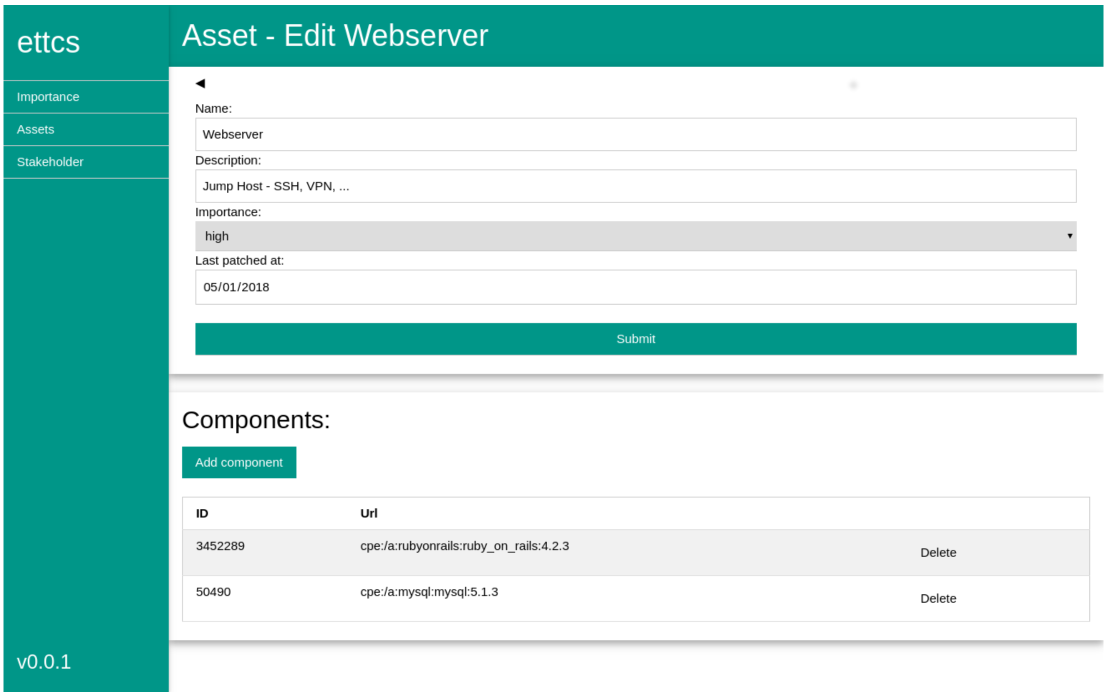
```python
# calculates beta_ttcs beginning from assets last_patched_at till now.
# asset - to calculate beta_ttc of
# total_amount_of_vulnerabilities - known as beta-TTCs k
# returns a Map of dates in a string format like '2018-03-30'
#          and the beta_ttc for that date.
def running_beta_ttcs(asset, total_amount_of_vulnerabilities):
    date_format = '%Y-%m-%d %H:%M:%S %z'
    last_patched_at = datetime.strptime(asset["last_patched_at"],
                                        date_format)
    last_patched_at = last_patched_at.replace(tzinfo=None)
    delta = timedelta(days=1)
    i = 0
    accu = {}
    now = datetime.now()
    while last_patched_at < now:
        last_patched_at = last_patched_at.strftime('%Y-%m-%d')
        vulnerability_quantity = vulnerability_quantity_at_date(asset,
                                                                 now)
        beta_ttc_result = beta_ttc(vulnerability_quantity,
                                   total_amount_of_vulnerabilities)
        accu[updated_at_str] = beta_ttc_result - i
        last_patched_at = last_patched_at + delta
        i = i + 1
    return accu
```

Listing 5.6: Code: Calculation of all βTTCs of the past days

One disadvantage of this code is, that when we dont keep track of what happened before the **last_patched_at** field. Which leads to a loose of information of perviously calculated TTC values. Thus the dashboard sparkline for the running TTC is never growing, resulting in a partial use of it. Storing daily calculated TTCs is a better approach. Those calculated values might be helpful for data analysts.

5.2.4 Configuration Service

This service interacts with all non interface services. For every services it communicates with, has been an adapter written.



Name: Webserver
Description: Jump Host - SSH, VPN, ...
Importance: high
Last patched at: 05/01/2018

ID	Url	
3452289	cpe:/a:rubyonrails:ruby_on_rails:4.2.3	Delete
50490	cpe:/a:mysql:mysql:5.1.3	Delete

Figure 5.2: Screenshot: Editing an asset

Within figure 5.2 we see the form for editing an already created asset. Changing the name, description, select another importance, edit the date when the asset has been patched the last time. Also we can add and remove components of that asset. Also a sidebar has been added to the left hand side, containing the entities *Importance*, *Asset* and *Stakeholder*.

5.2.5 ScoreCard Service

The ScoreCard service is the only microservice, which is not offering a RESTful API. When the time passed by, In listing 5.7 the daily poll is shown, which is checking if the stakeholders need to be notified. The method `notify_stakeholders` generates a PDF using `wkhtmltopdf` of the Dashbaord. It also fetches all subscribed stakeholders from the **stakeholder_service** and sends the generated PDF to those.

The combination of `wkhtmltopdf` and a on JavaScript relying chart framework like chartkick, leads to a problem. Resulting in a missing sparkline chart in the ScoreCard. While we make a requirement to the client, that the browser is able to run JavaScript and HTML5, the tool wkhtmltopdf is not able to plot because it has no JavaScript engine running. There are two general possibilities to handle it for a successor version of *ettcs* prototype. The first option might be to rely on another chart framework, which serves plotted images. This option is gives us the opportunity to keep the state of simply serving the dashboard as a ScoreCard.

```ruby
require 'rufus-scheduler'

scheduler = Rufus::Scheduler.new

scheduler.every '1d' do
  asset_repository_adapter.all_assets.each |asset|
    beta_ttc = metrics_adapter.calculate_metrics(asset)
    if minor_critical?(asset, beta_ttc) || critical?(asset, beta_ttc)
      notify_stakeholders(asset)
    end
  end
end

scheduler.join
```

Listing 5.7: Code: Snippet of ScoreCard

5.2.6 Dashboard Service

Within Figure 5.3 on page 38, we can see the asset overview of the dashboard. The assets are sorted by their ids and simple overview of the current security state is provided. A more detailed description of the security state can be found on within Figure 5.4. Not only all informations which are found within that overview are provided, also a detailed overview of the components is given. Also the field `newer_version_available?` can be found there, as the support of the process step *Control Recommendations*.

5.3 General Implementation Critique

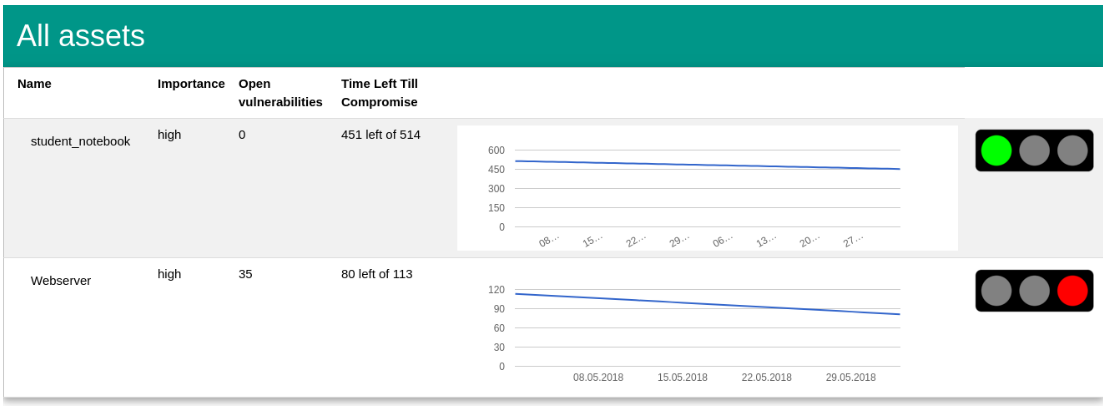
Whilst we list critique within the sections of the components, we also had issues regarding all components. Such critique is listed below:

1. *Missing HATEOAS*
   While we specified that the services must fulfil RESTful APIs, we did not achive that goal. Regarding to Fowler (2010) a RESTful API has to provide *Hypermedia Controls*, also known as HATEOAS. Instead of passing the IDs of an object, the corresponding link to

request the resource shall be provided. Which has not been implemented by us. In fact we achived level 2.

2. *Missing component tests*
   Whilst prototyping we did add unit tests for parts of the components. For instance the introduced method **newer_version?** Although it has been described before, that relying on such a method for ordering releases is a bad idea. However, when iterating another prototype or continue working on this one, component tests are required.

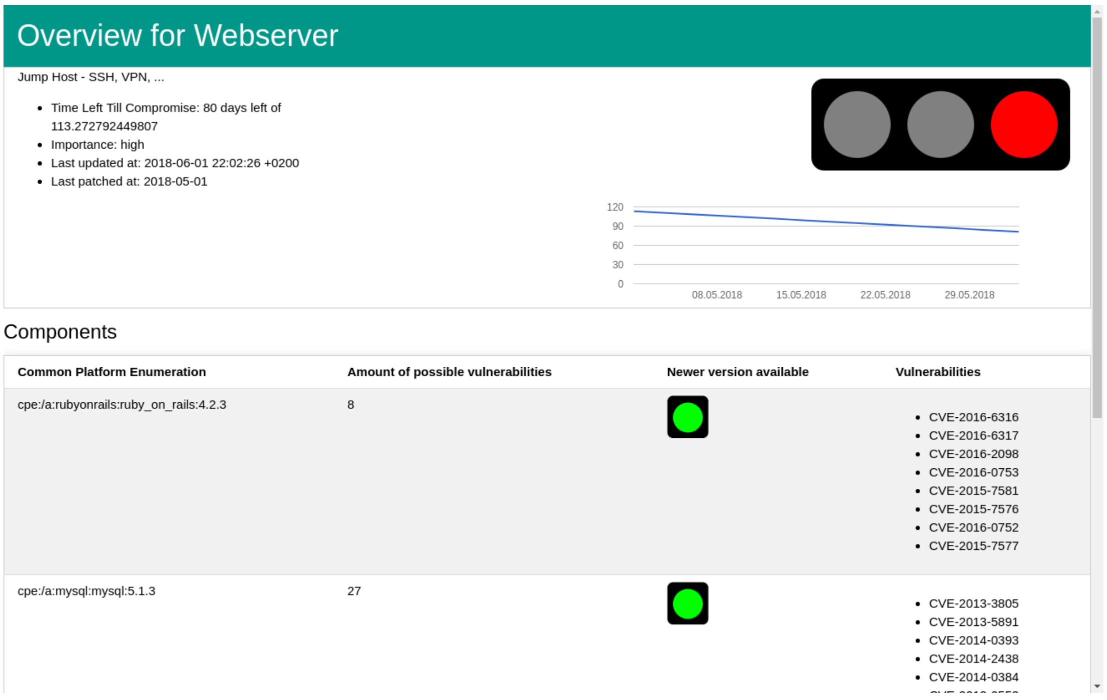
3. *Leak of performance*
   Although not listed within the requirements, running into performance issues must be avoided. But calculating TTC values for past dates is an time consuming task, which depends on the last patched date. The longer that date passed, the longer does the calculation need. To counter this problem, we could introduce redundancy to *Metrics Service*, which is our bottleneck.



All assets

Name	Importance	Open vulnerabilities	Time Left Till Compromise		
student_notebook	high	0	451 left of 514		🟢⚫⚫
Webserver	high	35	80 left of 113		⚫⚫🔴

Figure 5.3: Screenshot: Overview of all assets within the dashboard



Overview for Webserver

Jump Host - SSH, VPN, ...

- Time Left Till Compromise: 80 days left of 113.272792449807
- Importance: high
- Last updated at: 2018-06-01 22:02:26 +0200
- Last patched at: 2018-05-01

Components

Common Platform Enumeration	Amount of possible vulnerabilities	Newer version available	Vulnerabilities
cpe:/a:rubyonrails:ruby_on_rails:4.2.3	8	🟢	CVE-2016-6316, CVE-2016-6317, CVE-2016-2098, CVE-2016-0753, CVE-2015-7581, CVE-2015-7576, CVE-2016-0752, CVE-2015-7577
cpe:/a:mysql:mysql:5.1.3	27	🟢	CVE-2013-3805, CVE-2013-5891, CVE-2014-0393, CVE-2014-2438, CVE-2014-0384

Figure 5.4: Screenshot: Asset details within dashboard

6 Conclusion & Future Work

In this work we implemented a risk management system based on a Time-To-Compromise extension, named β-Time-To-Compromise. We did list requirements, designed and implemented *ettcs*.

6.1 Conclusion

During this work we analysed a risk management process and created a system to support this process. For this creation we listed requirements and specified which process steps have to be done by *ettcs* and which do need user interaction.

To keep the system flexible we choose a microservice architecture with two user interface components. As the first service, we described a service for the configuration of the process support. Modeling assets, giving them an importance or impact and let stakeholders subscribe to those assets are part of that service. Second service is a dashboard for supplying risk informations of those defined assets. A third service named ScoreCard checks the risk daily. If the TTC of a specific asset is smaller then a threshold specified within the importance, an e-mail will be send to the subscripted stakeholders.

There are four services not directly interacted by the user. Three repository services for the context of stakeholder, catalog and asset. The fourth component is responsible for calculating metrics for the modeled assets.

The use of *ettcs* did show that the project is still under development and not ready for production use yet. Encountering problems, for example realizing that the sparkline within the dashboard never grows, were the goal of this thesis. Another problem is located in the ScoreCard, which is a dashboard representation, leaking all sparklines. As the last issue we note performance issues, because of a backward TTC calculation.

6.2 Future Work

In order to make *ettcs* useable for production, we need to make some changes and extensions.

1. *Add user- and permission management*
   We built a system, without any authentication of identity. Those asset informations must be treated sensible. Giving a blueprint of the software landscape to an attacker has to be avoided. This explains the need for such authentication in production, including roles.

2. *Improve Control Recommendations*
   Stored advisories within the advisory database are currently not used within *ettcs*. Instead we provide the simple information if there exists a newer release. Extend *ettcs* to provide those advisories as control recommendations.

3. *Use typed TTCs*
   In section 2.3.5 we introduced Ziegers typed TTCs. Whilst the βTTC we used does not differentiate integrity, confidentially and availability, although this is needed within the process step *Impact Analysis*. Implement the use of typed TTCs in *ettcs*.

4. *Compare TTC values to real data*
   Andrej Zieger (2018) mentioned that "Nevertheless, the results of the time-to-compromise model [...] are not heavily supported and tested against real-world data." To validate the correctness of TTC, Zieger also approaches to set up a honeypot network and measure the real TTC. We cite this in order to clarify the importance of the βTTC validation.

5. *Extend the asset model*
   The current asset model represents a single host. Those hosts are connected through a network. Andrej Zieger (2018) already mentioned that in future we should "extend the formal model of time-to-compromise to a network and more complex systems". To do so, the asset model has to be extended for network purposes. Extend the asset model for network purposes.

6. *Simplify Asset Modeling*
   Some organisations already have information about the asset model, even if partial. Is the effort worth modelling an asset and keeping it up to date. *Where can we find information about assets in an organisation?*

   a) *Packet managers*
      contain detailed informations about the software product. We unify package and product in that context. Packages do provide dependencies, which is a great help for asset modeling. Installed packages indicate that a product is part of our asset. Write an adapter for packet managers. Modeling an asset by simply installing a package and its dependencies sounds fair.

b) *Enterprise architecture management*
EAM is the combination of five contexts. Business, process, integration, software and technical contexts named layers (Winter und Fischer (2006)). Those layers are modeled within a meta-model. Within the meta-model entities relate to other entities unbounded to their contexts. Thus a network-switch is related to a web server, whose is related to a process and so on. Within this work we worked on the technical and software context only. However, integreating to an EAM framework gives us an answer to the question: *How long does it take until an attacker compromised this workflow?*

c) *Use existing vulnerability management systems*
Although fingerprinting is not the way we want to go, it can help creating the asset. Those resulted fingerprints might be matched on the CPEs, to persist that asset for *ettcs.* Write adapters for vulnerability management systems.

7. *Create a product catalogue*
Managing CPEs is as already mentioned a pretty hard task. The deprecation process screams for an actual revision system. Often such products d have dependencies on other products, which are also not specified within CPEs. Such a product catalogue is also interesting for EAM. Create a public product catalogue compatible to the specifications of CPE 2.2 and CPE 2.3.

8. *Go Event-Driven*
Instead of polling frequently, reacting on events would increase the performance and allow a fluent processing of the risk. Change to a Event-Driven Architecture (Michelson (2006)) brings benefits. Such events include *the release of a new vulnerability* or a *change of an asset.* Adapt *ettcs* to an event-driven architecture.

9. *Store calculated TTCs*
Instead of recalculating passed values, storing those values is a better approach in perspective of performance, complexity and correctness. This informations might be also interesting for data analysts and fit into a data warehouse.

Bibliography

[Anderson 2015] Anderson, Charles: Docker [software engineering]. In: *IEEE Software* 32 (2015), Nr. 3, S. 102–c3

[Andrej Zieger 2018] Andrej Zieger, Klaus-Peter K.: The β-Time-to-Compromise Metric for Practical Cyber Security Risk Estimation. In: *11th Internaional Conference On It Security Incident Management And It Forensics*, IEEE, 2018. – ISBN 978-1-5386-6632-6

[Andrew Buttner 2009] Andrew Buttner, Neal Z.: Common Platform Enumeration CPE Specification. URL `http://cpe.mitre.org/files/cpe-specification_2.2.pdf`, 2009

[Bächle und Kirchberg 2007] Bächle, Michael ; Kirchberg, Paul: Ruby on rails. In: *IEEE software* 24 (2007), Nr. 6

[Brant A. Cheikes 2011] Brant A. Cheikes, Karen S.: Common Platform Enumeration CPE Specification. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.696.7787&rep=rep1&type=pdf`, 2011

[Brown 2017] Brown, Richard: *[opensuse-factory] openSUSE Leap's Next Major Version Number.* openSUSE mailing list. 2017

[BSI 2012] BSI: In: *Schwachstellenampel - Produktsicherheit auf einen Blick.* URL `https://www.allianz-fuer-cybersicherheit.de/ACS/DE/_/downloads/BSI-CS_028.pdf?__blob=publicationFile&v=3`, 2012. – [Online; accessed 12-July-2018]

[Evans 2004] Evans, Eric: *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley Professional, 2004

[Evans 2018] Evans, Jeremy: *Documentation for Sequel.* 2018. – URL `http://sequel.jeremyevans.net/documentation.html`

[Fowler 2010] Fowler, Martin: Richardson Maturity Model: steps toward the glory of REST. In: *Online at http://martinfowler.com/articles/richardsonMaturityModel.html* (2010), S. 24–65. – [Online; accessed 12-July-2018]

[Fredrich 2012] Fredrich, Todd: RESTful Service Best Practices. Pearson eCollege, 2012. – URL `http://www.restapitutorial.com/media/RESTful_Best_Practices-v1_1.pdf`

[Harris und Haase 2011] Harris, Alan ; Haase, Konstantin: *Sinatra: Up and Running: Ruby for the Web, Simply.* " O'Reilly Media, Inc.", 2011

[Jenson 2015] Jenson, Graham: In: *Testing microservices with pmux and TravisCI.* URL `https://maori.geek.nz/testing-microservices-with-pmux-and-travisci-8d3c42ce995c`, 2015. – [Online; accessed 12-July-2018]

[Luis Alberto Benthin Sanguino 2017] Luis Alberto Benthin Sanguino, Rafael U.: Software Vulnerability Analysis Using CPE and CVE. Mai 2017

[Lyon 2009] Lyon, Gordon F.: *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning.* USA : Insecure, 2009. – ISBN 0979958717, 9780979958717

[Maynor 2011] Maynor, David: *Metasploit toolkit for penetration testing, exploit development, and vulnerability research.* Elsevier, 2011

[McQueen u. a. 2006] McQueen, Miles A. ; Boyer, Wayne F. ; Flynn, Mark A. ; Beitel, George A.: Time-to-Compromise Model for Cyber Risk Reduction Estimation. In: Gollmann, Dieter (Hrsg.) ; Massacci, Fabio (Hrsg.) ; Yautsiukhin, Artsiom (Hrsg.): *Quality of Protection: Security Measurements and Metrics.* Boston, MA : Springer US, 2006, S. 49–64. – URL `http://dx.doi.org/10.1007/978-0-387-36584-8_5`. – ISBN 978-0-387-36584-8

[Mell u. a. 2007] Mell, Peter ; Scarfone, Karen ; Romanosky, Sasha: A complete guide to the common vulnerability scoring system version 2.0. In: *Published by FIRST-Forum of Incident Response and Security Teams* Bd. 1, URL `http://www.nazimkaradag.com/wp-content/uploads/2014/11/cvss-guide.pdf`, 2007, S. 23

[Mettraux 2018] Mettraux, John: *rufus-scheduler.* 2018. – URL `https://github.com/jmettraux/rufus-scheduler`

[Michelson 2006] Michelson, Brenda M.: Event-driven architecture overview. In: *Patricia Seybold Group* 2 (2006)

[Newman 2015] Newman, Sam: *Building microservices: designing fine-grained systems.* " O'Reilly Media, Inc.", 2015

[Pilgrim 2010] Pilgrim, Mark: *HTML5: Up and Running: Dive into the Future of Web Development.* " O'Reilly Media, Inc.", 2010

[Pytel u. a. 2007] Pytel, Chad ; Yurek, Jonathan ; Marshall, Kevin: *Pro Active Record: Databases with Ruby and Rails.* Apress, 2007

[Riancho 2011] Riancho, Andrs: w3af-web application attack and audit framework. In: *World Wide Web electronic publication* (2011), S. 21

[Shikha Shahi 2018] Shikha Shahi, Matthias H.: Sicherheitskennzahlen in der Praxis. 2018

[Stoneburner u. a. 2002] Stoneburner, Gary ; Goguen, Alice Y. ; Feringa, Alexis: SP 800-30. Risk Management Guide for Information Technology Systems. Gaithersburg, MD, United States : National Institute of Standards & Technology, 2002. – Forschungsbericht

[Truelsen] Truelsen, Jakob: *wkhtmltopdf–Convert html to pdf using webkit (qtwebkit)*

[Winter und Fischer 2006] Winter, Robert ; Fischer, Ronny: Essential layers, artifacts, and dependencies of enterprise architecture. In: *Enterprise Distributed Object Computing Conference Workshops, 2006. EDOCW'06. 10th IEEE International* IEEE (Veranst.), 2006, S. 30–30

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 14. Juli 2018 Alex Mantel