FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

MSc thesis in Computer Science

Lasse Halberg Haarbye — `ninjalf2@gmail.com`
Gustav Brieghel Jensen — `gustavbjensen@hotmail.com`

Security analysis of the Maritime Connectivity Platform

Advisor: Michael Kirkedal Thomsen

Submitted: September 2, 2019

Abstract

This thesis concerns itself with IT security in the maritime sector, specifically the Maritime Connectivity Platform (MCP). The thesis gives a detailed security assessment of the various threats and vulnerabilities that could pose a risk to a platform like the MCP. The analysis gives the developers of MCP an overview of the identified risks as well as an indication of which to prioritise first. Additionally, the assessment concludes that fuzz testing is a good way to tackle many of the identified risks such as injection attacks, broken authentication and access control, sensitive data exposure, and DDoS attacks. A virtual lab environment is provided which simulates the MCP Identity Registry, making testing more convenient. We present RESTFuzzer, a REST API fuzzing framework based on the Kitty framework. Using this framework, we build a fuzzer specifically for the MCP Identity Registry. Our fuzzer yielded some interesting results that the developers of MCP, in addition to the risk assessment, may use to prevent bugs or vulnerabilities. In particular, the fuzz testing revealed that requests occasionally take over a second to process. Our results showed that certain requests more often trigger those slow requests than others. Whether this can be exploited in a DDoS attack is not yet determined.

Dansk resumé

Dette speciale omhandler IT-sikkerhed i den maritime sektor, helt specifikt “the Maritime Connectivity Platform (MCP)”. Dokumentet inkluderer en detaljeret sikkerhedsvurdering af de adskillige trusler og sårbarheder der kan udgøre en risiko for en platform som MCP. Analysen giver udviklerne en oversigt over de identificerede risici såvel som en indikation af hvilke der bør prioriteres først. Udover dette konkluderer den at fuzztesting er en god måde at tackle mange af de identificerede risici såsom injection-angreb, fejlkonfigureret autentificering og adgangskontrol, eksponering af følsomme data, og DDoS-angreb. Et virtuelt testmiljø blev sat op for at simulere MCP Identitetsregistret, og senere blev et fuzztestingværktøj, som er målrettet MCP Identitetsregistrets REST API, præsenteret. Vores fuzztesting gav nogle interessante resultater som MCPs udviklere, udover risikovurderingen, kan bruge til at forebygge fejl og sårbarheder. Det væsentligste resultat fra fuzztesten var, at vi har fundet ud af at visse requests har større sandsynlighed for at have længere responstid end andre. Om dette kan udnyttes i et DDoS angreb vides endnu ikke med sikkerhed.

Preface

This Master's thesis is submitted in fulfilment of the master programme in Computer Science (*Datalogi*) at the University of Copenhagen, for Lasse Halberg Haarbye and Gustav Brieghel Jensen.

Contents

Introduction

1

Cybersecurity is an ever-evolving field as cybercriminals are becoming more and more creative. Today, almost everything has been digitised, and computer systems play a critical role in keeping sensitive data safe and providing everyday functionality in society. In recent years, the lack of security has been highlighted by several devastating attacks on companies and governments. Notably, in the summer of 2017, the Danish company A.P. Moeller-Maersk was affected by a virus by the name of NotPetya [1]. The virus originated from Russia and was targeted towards Ukraine, but it was so aggressive that it ended up affecting large companies, including Maersk, pharmaceutical company Merck, delivery company FedEx, construction company Saint-Gobain, food producer Mondelēz, manufacturer Reckitt Benckiser, and even Russian oil company Rosneft, in each case resulting in nine figure costs. The total damage costs of the NotPetya attack, as estimated by the White House, was more than 10 billion dollars. As former Homeland Security adviser Tom Bossert put it: “The weapon’s target was Ukraine. But its blast radius was the entire world.”

Unlike NotPetya, which was only meant for destruction, cyberattacks are also carried out for lucrative purposes. Cybercriminals are deploying methods ranging in complexity from phishing attacks, which are fraudulent attempts to retrieve personal information, credentials, or credit card information, to global ransomware attacks demanding large sums from citizens to unlock their data. It is often the case for worldwide cyberincidents, as with NotPetya and Stuxnet, which were intended for targets in Ukraine and Iran, respectively, that the malware manages to spread far beyond its initial target [1, 2]. This emphasises the importance of computer security, even if you are not considered a target.

In this thesis, we explore the area of cybersecurity within the maritime sector. In particular, we will be carrying out a security analysis of the identity registry subcomponent of the Maritime Connectivity Platform, a software framework targeted towards the maritime industry. The analysis will describe the most prominent risks for this particular platform and will serve as the basis for our choice to develop a fuzz tester designed to eliminate or reduce some of the identified risks.

Additionally, we will set up a virtual lab environment that will simulate the MCP Identity Registry where the fuzz tester can be deployed and its practicality evaluated. The lab is designed to be easily reproducible and extendable, utilising the operating-level virtualisation tool, Docker, and its extension, Docker Compose.

1.1 Learning objectives

The learning objectives of this thesis are the following:

1. Make a detailed analysis of a system's security / Give an analysis of a system based on a relevant model.
2. Identify security threats in a highly distributed maritime system as well as propose potential solutions or mitigation strategies to these.
3. Give an assessment of the severity and potentiality of security vulnerabilities and suggest improvements to the design.
4. Set up a virtual lab environment that simulates MCP and apply tools to automate the detection of vulnerabilities.

1.2 Scope

This thesis will cover a security analysis of the identity registry of the Maritime Connectivity Platform (MCP).

The thesis is focused on black box penetration testing, taking on the perspective of potential attackers. We will primarily be focusing on fuzzing the REST API exposed by the identity registry. The analysis is carried out on version 0.9.0-RC1 of the MCP Identity Registry.

1.2.1 Limitations

We will not go into white box penetration testing and techniques such as static code analysis, because memory and unmanaged system calls are handled by the Java Virtual Machine (JVM), making buffer overflow, heap overflow, and other memory related attacks inapplicable. We will not be doing any analysis/testing on the well-established framework Keycloak, which is being used as an identity broker for the MCP Identity Registry.

Our testing will only include the identity registry and not any other components of the MCP. This also means that the MCP management portal, which is a front-end to the identity registry and currently still in beta, will not be covered.

1.3 Structure

The rest of this thesis is structured as follows:

- **Chapter 2** gives background information about MCP and introduces basic fuzz testing concepts.
- **Chapter 3** details an IT security risk assessment of the MCP Identity Registry.
- **Chapter 4** describes how we have set up the virtual lab environment in which the MCP Identity Registry runs.
- **Chapter 5** details the implementation of our fuzz tester that we use for testing the MCP Identity Registry.

- **Chapter 6** describes our experiments using our fuzzer and documents the results.
- **Chapter 7** discusses our results, our fuzzer implementation, and our IT security risk assessment.
- **Chapter 8** concludes and summarises our work.

Background 2

The maritime sector is present throughout most of the world and plays an important role in modern society. The sector is mainly concerned with the transportation of people and goods by means of vessels/ships.

In Denmark, the sector is particularly large: 19 % of Denmark's total export comes from maritime activity [3]. In a report from Danish Shipping from June 2018, named "Facts and Figures", Denmark is listed as number six in the "top ten operator nations", only surpassed by USA, Singapore, China, Japan, and Greece as number one [4]. The top ten list indicates how much tonnage is commercially controlled from the respective countries. In an earlier report from 2017, Denmark was listed as number seven, and since then, the GT (gross tonnage) increased by more than 2 million [5].

Some of the large Danish maritime companies are Maersk, Norden and DFDS. Since 1996, Maersk has been the largest container ship and supply vessel operator in the world [6]. Clearly, the maritime sector is important for Denmark and will most likely continue to be so.

Digitisation of the industry. Like so much of society, the maritime sector has to a large extent been digitised. Fewer employees are present on ships, fewer people are needed to operate a ship, and more functions are carried out automatically.

As more and more parts of the sector becomes digitised, the risk of cybercrime rises, and so does the need for cybersecurity. Crewless ships might even become a reality in the future, in which case the need for cybersecurity would be even higher [7].

2.1 Cybersecurity in the maritime sector

The need for cybersecurity in the maritime sector has been highlighted in recent times. A threat assessment was made for the maritime sector in Denmark by the Danish Defence Intelligence Service (DDIS) in a report from January 2019 [8]. In this report, the threat from cyberespionage and cybercrime is assessed as "very high". This threat assessment by the DDIS is referenced by the Danish Maritime Authority (DMA) in a cybersecurity strategy document called "Cyber and Information Security Strategy for the Maritime Sector" [9].

In this document, the DMA present their strategy for the near future in regard to cybersecurity [9]. The strategy is part of the danish government's efforts to reinforce Denmark's defences against cyberthreats. The maritime sector was designated as one of the sectors of particular importance, which is the reason for this document.

More specifically, cybersecurity within the maritime sector is defined as “the safety of navigation in Danish waters and the safety and security of Danish-flagged ships and their crews”.

The report references another report by Centre for Cyber Security in which it is assessed that the threats from cyberespionage and cybercriminals are especially significant in the maritime sector. The threat from cyberespionage is deemed very high because foreign states may use espionage as a way to gain industrial and business advantages. As for cybercriminals, the threat is deemed very high: in particular, cybercriminals might want to “blackmail public authorities, businesses and individuals (ransomware)”.

The main goal of the strategy is the following: “security on board Danish ships and in Danish waters should not be compromised as a result of cyberattacks”.

It is thus established that cybersecurity is a big concern in the maritime sector. Now we will describe the Maritime Connectivity Platform which is a communication framework in the maritime industry with an emphasis on cybersecurity.

2.2 The Maritime Connectivity Platform

The Maritime Connectivity Platform (MCP) is a solution developed by the EfficienSea2 consortium which is led by the Danish Maritime Authority. The development of MCP started as an e-Navigation project and it was observed that the automatic identification system (AIS) was a major issue. The AIS is responsible, via transponders, for verifying that a ship or vessel is who and where they claim to be.

One of the goals of the MCP project is to provide “a communication framework enabling efficient, secure, reliable and seamless electronic information exchange between all authorised maritime stakeholders across available communication systems”. Further: “The objective of the Maritime Cloud is to provide a secure platform to enable maritime stakeholders to securely access technical services to gain further information for decision-making onboard and ashore during a voyage from berth-to-berth.” [10]

The platform has the following three core components:

Identity registry. In order to have proper communications, it is necessary for actors in the communication chain to prove their identity. A central component of the platform, the identity registry, is responsible for just that. “For secure and reliable identity management, it is the equivalent of a Central Person Registry or a Central Business Registry. It contains relevant information for authorised stakeholders, enabling verification of authenticity, integrity and confidentiality in information transfer processes. It provides a single login to all services.” [11]

Service registry. “For registering, discovering and using all relevant e-Navigation and e-Maritime services, commercial and non-commercial, authorised and non-authorised, for free and against payment. It can be seen as a sophisticated yellow pages phone book or the equivalent of an App Store on iPhone and Google Play for Android." [11]

Messaging service. "An information broker that intelligently exchanges information between communication systems connected to the platform, taking into account the current geographical position and communication links available to the recipient." [11]

An overview of the platform can be seen in Figure 2.1.

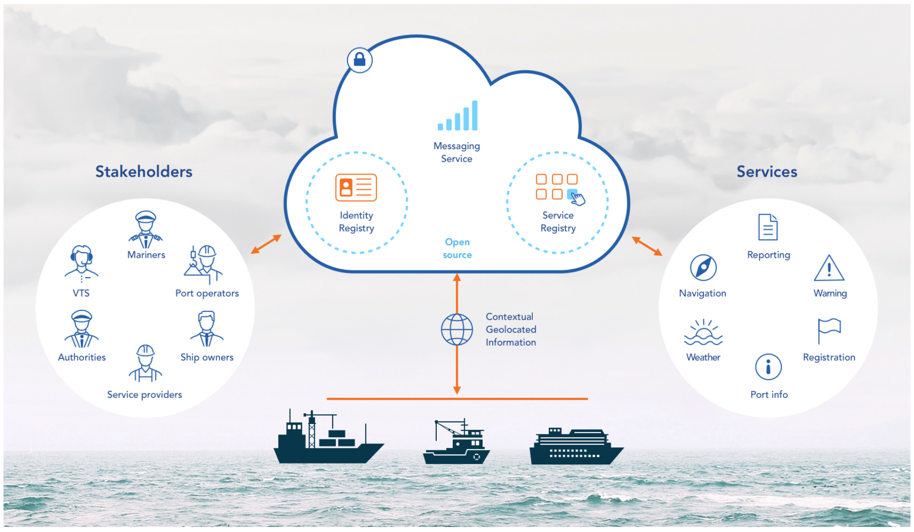


Figure 2.1: An overview of MCP, from [11].

At this point in time, the identity registry is the most developed component. Furthermore, it is a component in which security is highly important since it deals with authentication and authorisation. For that reason, we will focus on the identity registry component in our analysis and describe it in more detail in the following sections.

2.3 Identity registry

The identity registry is one of the core components of MCP. It is an open-source project written in Java and can be accessed on GitHub.[1] Essentially, it is a REST API that supports creating, deleting, updating, and retrieving certain entities that relate to MCP. Below is an explanation of the key entities [12]:

Organisation: "An entity such as an institution, company or association that has a collective goal and is linked to an external environment".

Vessel: "Any floating object used for the carriage of people or goods".

Service: "Services refer to digital services. For example, a weather service that is available to other services for machine to machine communication".

[1] The source code of the identity registry is available at GitHub: `https://github.com/MaritimeConnectivityPlatform/IdentityRegistry`

User: Mainly refers to human users, as opposed to other entities like devices.

Device: "Devices can be any number of entities that are not covered by the other entity types. It could for example be a lighthouse, an ECDIS[2] or a server that needs to be able to authenticate itself."

Organisations are the centrepiece entities of the registry. All the other entities — vessels, services, users and devices — each belong to a specific organisation. This can also be seen in the API specification: nearly all API paths start with `/org/{orgMrn}` which means the request takes some organisation identifier as input.

The API is meant both for use by front-end applications (the Management Portal) or by firing off REST requests manually. In this thesis we only access the API by firing off raw HTTP requests, since we do not have access to the Management Portal.

Ironically, the identity registry does not handle identity management. Instead, it utilises an existing solution called Keycloak.

2.3.1 Keycloak

Keycloak is an open source single sign-on identity broker that the identity registry uses to alleviate the pressure of identity and access management. The product is developed by JBoss, a division under Red Hat. Hence, it is almost certain that Red Hat already has deployed proper security measures. Consequently, we will not focus much on it throughout.

2.3.2 Swagger/OpenAPI 2.0 specification

The identity registry API is described by a Swagger/OpenAPI 2.0 specification that can be obtained by accessing the path `/v2/api-docs` on the server that hosts the identity registry, e.g. `http://localhost:8443/v2/api-docs`.

An OpenAPI/Swagger specification is an API description format for REST APIs. It details available endpoints and the operations (HTTP methods) for each. It also describes input parameters, authentication methods, contact information, software licence, and more [13].

All the available paths and methods of the identity registry are listed in Table 2.1.
Out of those 46 items, 43 relate to the five different entities (organisations, vessels, services, users, and devices) listed previously. The remaining three are about certificates.

It is worth noting that the table above lists all available operations in the API, but they can be accessed in two different ways. Either by using OIDC (OpenID Connect) or x509 authentication (certificates). Depending on which method is used, either `/oidc` or `/x509` must be prepended to the paths in Table 2.1. This means that in total, there are $46 \cdot 2 = 92$ different paths in the API (double the amount shown in the table). Since the requests are the exact same regardless of authentication method, we decided to not show the

[2] An ECDIS, or Electronic Chart Display and Information System, is an information system used for nautical navigation.

HTTP methods	Path (curly braces indicate variables)
GET	/certificates/crl/{caAlias}
POST	/certificates/ocsp/{caAlias}
GET	/certificates/ocsp/{caAlias}/**
POST	/org/apply
GET	/org/id/{orgId}
GET	/org/unapprovedorgs
GET PUT DELETE	/org/{orgMrn}
GET	/org/{orgMrn}/acting-on-behalf-of
POST	/org/{orgMrn}/agent
GET PUT DELETE	/org/{orgMrn}/agent/{agentId}
GET	/org/{orgMrn}/agents
GET	/org/{orgMrn}/approve
GET	/org/{orgMrn}/certificate/issue-new
POST	/org/{orgMrn}/certificate/{certId}/revoke
POST	/org/{orgMrn}/device
GET PUT DELETE	/org/{orgMrn}/device/{deviceMrn}
GET	/org/{orgMrn}/device/{deviceMrn}/certificate/issue-new
POST	/org/{orgMrn}/device/{deviceMrn}/certificate/{certId}/revoke
GET	/org/{orgMrn}/devices
GET POST PUT DELETE	/org/{orgMrn}/logo
POST	/org/{orgMrn}/role
GET	/org/{orgMrn}/role/available-roles
GET	/org/{orgMrn}/role/myroles
GET PUT DELETE	/org/{orgMrn}/role/{roleId}
GET	/org/{orgMrn}/roles
POST	/org/{orgMrn}/service
GET	/org/{orgMrn}/service/{serviceMrn}
GET PUT DELETE	/org/{orgMrn}/service/{serviceMrn}/{version}
GET	/org/{orgMrn}/service/{serviceMrn}/{version}/certificate/issue-new
POST	/org/{orgMrn}/service/{serviceMrn}/{version}/certificate/{certId}/revoke
GET	/org/{orgMrn}/service/{serviceMrn}/{version}/jbossxml
GET	/org/{orgMrn}/service/{serviceMrn}/{version}/keycloakjson
GET	/org/{orgMrn}/services
POST	/org/{orgMrn}/user
GET PUT DELETE	/org/{orgMrn}/user/{userMrn}
GET	/org/{orgMrn}/user/{userMrn}/certificate/issue-new
POST	/org/{orgMrn}/user/{userMrn}/certificate/{certId}/revoke
GET	/org/{orgMrn}/users
POST	/org/{orgMrn}/vessel
GET PUT DELETE	/org/{orgMrn}/vessel/{vesselMrn}
GET	/org/{orgMrn}/vessel/{vesselMrn}/certificate/issue-new
POST	/org/{orgMrn}/vessel/{vesselMrn}/certificate/{certId}/revoke
GET	/org/{orgMrn}/vessel/{vesselMrn}/services
GET POST PUT DELETE	/org/{orgMrn}/vessel/{vesselMrn}/vesselImage
GET	/org/{orgMrn}/vessels
GET	/orgs

Table 2.1: API calls in the MCP Identity Registry. Note that all paths must be preceded by either `/x509/api` or `/oidc/api`.

full request paths for both methods. We will now look closer at the entities in the registry.

To create an organisation, a POST request must be sent with a JSON object as body with the following parameters:

- Address (string),
- Country (string),
- Email (string),
- MRN (string),
- Name (string), and
- URL (string).

An MRN is a special identifier for the various entities that are present in the registry. Its precise specification and syntax can be seen at `https://www.iana.org/assignments/urn-formal/mrn`. Some examples of valid MRN values are[3]:

- `urn:mrn:mcl:org:imo`
- `urn:mrn:mcl:org:iala`
- `urn:mrn:mcl:org:dma`
- `urn:mrn:mcl:org:portofrotterdam`
- `urn:mrn:mcl:org:vts-oeresund`
- `urn:mrn:mcl:org:amsa@iala`

This MRN format will later become more important as we start testing.

In order to do security testing of the identity registry, we have decided to use fuzz testing as our primary means. This is ideal because there are many different paths to be tested, and each request has certain parameters that can be altered according to the OpenAPI specification. This specification makes it an obvious choice to implement a fuzzer that tests all the different requests with fuzzed inputs.

Next, we will describe what fuzz testing is and how we will use it.

2.4 Fuzz testing

Fuzz testing, also known as "fuzzing", is a testing method that generates "random" data and feeds it to some program or service that receives input. Typically, the goal is to cause unexpected behaviour – for example, the program crashing. Sutton et al. put it quite simply in their book [14]: "In a nutshell, fuzzing consists of throwing everything but the kitchen sink at a target and monitoring the results.". Later, they give a more precise definition: "we define fuzzing as a method for discovering faults in software by providing unexpected input and monitoring for exceptions" [14, p. 22].

[3] From `https://developers.maritimeconnectivity.net/identity/index.html`

2.4.1 The fuzzing process

Sutton et al. also describe the general fuzzing process with five steps [14, pp. 27-28], outlined here:

1. **Identify target.** This means to identify which program/target you will fuzz test.

2. **Identify inputs.** In order to perform fuzzing, there needs to be some input vector where we can supply our fuzzed data. This step involves finding these input vectors and choosing the appropriate ones for testing. An input vector might be a file read by the application, command line arguments, or HTTP requests sent to some server.

3. **Generate fuzzed data.** This can be random data, predetermined values, mutation of existing data, or generated according to some protocol. This step is often carried out automatically by the fuzzer.

4. **Execute fuzzed data.** This step follows up on the previous step by sending the fuzzed data to the application, whether it be through TCP packets or command line arguments to a program, or by other means.

5. **Monitor for exceptions.** In order for fuzzing to be useful, we need to be able to know when an input caused some sort of exception in the application. Ideally, we can pinpoint which exact input caused an exception. In the case of sending HTTP requests, each response can be examined for anomalies. In binaries, it is common to monitor for segmentation faults (crashes).

2.4.2 Data generation

In order to do fuzzing, data needs to be generated such that it can be fed to the target application. Generally, there are three main methods of data generation: the random, the mutation-based, and the protocol-based method.

Random method. The random method is self-explanatory — it involves generating random data and throwing it at the target. The benefit of this method is that it is very simple and barely requires any preparation. Also, this method can be effective for finding very simple bugs such as buffer overflows. The disadvantage of this method is that it fails at finding more intricate bugs that only would be triggered by more fitting data.

As an example, here is a very easy way of firing off a thousand random bytes to a binary on a Linux-based system:

```
head -c 1000 /dev/urandom | ./myprogram
```

The above can be put in a for-loop, and the program's exit code can be checked as a way of monitoring for exceptions. That would be a very simple but working fuzzer using the random method. It shows how easy it is to create a basic random fuzzer. Naturally, it would require more work on other targets such as web servers.

Mutation-based method. The mutation-based method involves taking some existing data that is expected by the target application and modifying it in various ways.

As an example, suppose your target application is a PDF-reader. In this case, throwing random data at the target will not find very many bugs because most files will be rejected for not following the PDF file format. Instead, if we have a mutation-based fuzzer that takes in a sample of valid PDF-files and modifies them in various ways, e.g. by doing bit flips, we are more likely to discover bugs in the target.

Protocol-based method. Following the example of a PDF-reader as our target, suppose we want to generate semi-valid PDF-files and change bits and pieces to see how the program reacts. Using this method, we build PDF-files from scratch following the PDF file format protocol, and then adding our own tweaks during the process.

2.4.3 Monitoring for exceptions

In order to be successful with fuzzing, we need to know when we caused an unusual reaction in the target. The way of doing this depends on the target being fuzzed:

Web applications. For web applications, one way of monitoring for exceptions is to check for timeouts. If you send a request and the server stopped responding shortly after, that may be a good sign you caused unexpected behaviour. Also, in many cases the server will send a response for each request you send, as in the case of the HTTP protocol. For HTTP testing, a 500 Internal Server Error can sometimes be a good sign that you caused unexpected behaviour.

Local applications. When testing a binary or other software that runs locally, a typical way of monitoring for exceptions is watching for certain signals sent by the OS. For example, the OS will send a signal to the program if it attempts to divide by zero or it tries to read/write to memory to which it has no access.

The popular fuzzer tool American fuzzy lop monitors for exceptions exclusively when the program is terminated by the OS, for example by segmentation faults (SIGSEGV).

However, some programs have exception handlers in place and will deal with such signals in their own way, which can prevent the fuzzer from detecting the fault. Furthermore, there is no guarantee that the OS will send a signal when the program elicits "unexpected behaviour".

2.4.4 Web application/HTTP fuzzing

Since we will be fuzzing a web application in this thesis, we will now dive deeper into fuzzing web application targets. This is based off of [14].

When fuzzing web applications, inputs can be a bit more complex. Many web applications use the HTTP protocol, so one might say that one input corresponds to an HTTP request. And in each HTTP request, there are many elements that can be fuzzed, such as:

- the request headers (keys, delimiters, values),
- the request body/payload (in case of POST/PUT requests), and
- the URL and any parameters or key-value pairs that it may contain.

Performance bottlenecks. When fuzzing binaries, performance, in terms of speed, is usually quite high. This is often not the case with web application fuzzing due to the overhead of packet transmission.

We have now gone through the basic principles behind fuzzing. After our risk assessment and after we have set up the virtual lab in the following chapters, we can then begin to apply this background information in Chapter 5.

Analysis

3

In this section we will carry out a security analysis and risk assessment of the MCP Identity Registry. We have described the risk assessment methodology that we will be applying to the MCP Identity Registry. We will be combining theory from the book “Computer Security: Principles and Practice” [15] with the OWASP top 10 list of most critical risks of web application security [16], giving us a strong theoretical foundation backed by real world data.

3.1 Risk assessment methodology

The background for our risk assessment is based on the section “Security Risk Assessment” in [15, p. 481-502], and we will apply the methodology described in this section when analysing the MCP Identity Registry in section Section 3.2. Note that we have not considered legal or regulatory concerns as we do not have sufficient knowledge about the judicial system, neither on a national nor global scale. To carry out a risk assessment, we will need to:

1. identify assets to be protected,
2. identify threats to these assets, and
3. propose counter measures to these threats.

The three key points above will be used by the organization in charge of the system to decide what measures must be taken to reduce an identified risk or accept the risk if deemed necessary. The process must be iterated upon and updated regularly due to the constant changes in technology and risk environment.

Many formal standards exist for assessing IT security risks, but we will focus on ISO 13335, which lists the following four approaches for identifying and mitigating security risks:

Baseline approach. The goal of the baseline approach is implementing a basic general level of security using baseline documents and industry best practice, with minimum cost. The level of security is achieved through an initial investment into creating guidelines and protocols. This protects against the most common threats, and resources are conserved as a result of no subsequent analysis being carried out.

Informal approach. The informal approach is an informal non-structured analysis usually carried out by internal experts or external consultants.

The main advantage is that this approach can be carried without the need to teach individuals new skills, allowing for a quick and cheap security analysis. A major disadvantage, however, is that some risks can be overlooked or assessed inappropriately, due to the lack of a formal process, or due to biased views of the individuals carrying out the analysis.

Detailed risk analysis (or formal approach). The detailed risk analysis is a formal structured process that provides better assurance that most significant risks are considered and analysed. The process involves identification of assets, identification of threats to these assets, likelihood of the risk occurring, and the following consequences. The main disadvantage is in terms of a significant cost; time, resources and expertise are all needed to perform this kind of analysis.

Combined approach. The combined approach combines one or more of the previously mentioned approaches into one solution. The goal is to provide reasonable protection across all categories as quickly as possible, giving a more flexible solution than any of the other approaches alone.

The approach used is chosen based on the size of the organisation and the resources available to allocate to the security risk analysis.

We will focus on the detailed risk analysis approach, because it utilises a formal process that we can follow and document. With this approach as the basis of our analysis, we can continue on to briefly determine the role of assets.

3.1.1 Asset identification

An **asset** is something that needs to be protected, because it holds a value; sometimes insignificant, sometimes vital for the success of the organisation. Hence, it is necessary to first identify assets and then estimate their value to the organisation. This part is crucial, as overlooking assets can affect other assets that actually *have* been considered, and leave them vulnerable too. This often happens if assets are not clear to the individuals carrying out the risk assessment, and therefore it is often recommended to plan interviews with personnel that is more qualified to identify assets. In our case, we will be identifying the assets ourselves as we have been working with the MCP Identity Registry for a long time now and have a good understanding of how it works.

The way we have chosen to estimate the value of assets is by determining whether they provide one or several of the key qualities of the CIA triad:

- Confidentiality
- Integrity
- Availability

Since assets have value, there is the inevitable danger of losing them or having their value decreased. Read on to see how we describe this danger.

3.1.2 Threat and vulnerability identification

The **threat source**, which is either natural or human-made, accidental or deliberate, is the origin of the threat. Natural threats are often caused by natural disasters whereas human-made deliberate threats are often the acts of hackers and cybercriminals. Human-made accidental threats are, for example, an employee entering incorrect information into the system, causing it to malfunction. Note: the relationship between a threat and an asset is many-to-many, i.e. a threat may target multiple assets and one asset may be subject to multiple threats.

If a threat source is human-made and the type of threat is deliberate, we can consider the following about the threat source:

Motivation: why would they target this specific asset?

Capability: how skilled are they and how much expertise would they need to exploit this vulnerability?

Resources: how much time, money, and resources do they have available?

Probability of attack: what is the likelihood of our assets being attacked and how often?

Deterrence: is there anything that would deter the attacker from exploiting this threat?

A **vulnerability** is a weakness in the system that a threat source can exploit and consequently poses a risk. Note that both a vulnerability and a threat source must be present for an asset to be subject to risk. Most of the formal standards in [15] include checklists than can help identify common threats and vulnerabilities and suggest techniques to determine their relevance. However, the formal standards often cost money to get a copy of, so we based our vulnerabilities and threat identification on the OWASP top 10 list of most critical risks of web application security [16].

Distinction between risk and threat. The two terms risk and threat are often used interchangeably but in our analysis, just like in [15], we will use the following definitions of threat and risk:

- A **threat** is the potential for a threat source to exploit a vulnerability of an asset, compromising the security of it and causing problems for the owner of the asset.

- A **risk** is seen as the likelihood that a threat source exploits a vulnerability of the asset, combined with the negative resulting consequences.

Below, we will explain how likelihood and consequences are assessed and combined into a risk level.

Rating	Likelihood description	Expanded definition
1	Rare	May occur only under exceptional circumstances and may be deemed as “unlucky” or very unlikely.
2	Unlikely	Could occur at some time but not expected given current controls, circumstances, and recent events.
3	Possible	Might occur at some time, but just as likely not. It may be difficult to control its occurrence due to external influences.
4	Likely	Will probably occur in some circumstance and one should not be surprised if occurred.
5	Almost certain	Is expected to occur in most circumstances and certainly sooner or later.

Table 3.1: Risk Likelihood. Table 14.2 from [15].

3.1.3 Risk analysis

Having identified assets, threats and vulnerabilities, we now need to analyse the risks based on this information. The first step is examining existing controls and based on that, estimating the likelihood of occurrence for each threat source to exploit a vulnerability of an asset.

Since we *cannot* estimate a risk rating with a formula, given that precise probabilities are unknown and assets often intangible, we must use a different approach. Likelihood is estimated by specifying one of the five different likelihood ratings that are described in Table 3.1. Naturally, there will be some uncertainty in this estimation and the reasoning should be discussed and documented accordingly.

After that, the risk consequences have to be determined. Usually this is done by the owner(s) of the assets and not the risk analyst(s), but in this case, we will have to take on this role as well. The consequences are estimated using the qualitative descriptions in Table 3.2.

Finally, a risk level is assigned based on the combination of likelihood and consequences. The risk level matrix can be seen in Table 3.3.

For documentation purposes, the whole analysis is combined into a risk register which is a prioritised list of assets along with its threats and vulnerabilities, risk likelihood, risk consequences, and risk level. This list may change over time as a result of several iterations on the risk assessment process. It is worth noting that the list is not necessarily sorted according to risk level. Some risks are faster, cheaper, and easier to address than others, and so they can be prioritised higher. See Table 3.4 for an example risk register.

3.1.4 Risk treatment

We have chosen to skip risk treatment as this is a response to the risk analysis and a task for the organisation itself.

Rating	Consequence	Expanded definition
1	Insignificant	Generally, a result of a minor security breach in a single area. Impact is likely to last less than several days and requires only minor expenditure to rectify. Usually does not result in any tangible detriment to the organisation.
2	Minor	Result of a security breach in one or two areas. Impact is likely to last less than a week but can be dealt with at the segment or project level without management intervention. Can generally be rectified within project or team resources. Again, does not result in any tangible detriment to the organization, but may, in hindsight, show previous lost opportunities or lack of efficiency.
3	Moderate	Limited systemic (and possibly ongoing) security breaches. Impact is likely to last up to 2 weeks and will generally require management intervention, though should still be able to be dealt with at the project or team level. Will require some ongoing compliance costs to overcome. Customers or the public may be indirectly aware or have limited information about this event.
4	Major	Ongoing systemic security breach. Impact will likely last 4-8 weeks and require significant management intervention and resources to overcome. Senior management will be required to sustain ongoing direct management for the duration of the incident and compliance costs are expected to be substantial. Customers or the public will be aware of the occurrence of such an event and will be in possession of a range of important facts. Loss of business or organizational outcomes is possible, but not expected, especially if this is a once-off.
5	Catastrophic	Major systemic security breach. Impact will last for 3 months or more and senior management will be required to intervene for the duration of the event to overcome shortcomings. Compliance costs are expected to be very substantial. A loss of customer business or other significant harm to the organization is expected. Substantial public or political debate about, and loss of confidence in, the organization is likely. Possible criminal or disciplinary action against personnel involved is likely.
6	Doomsday	Multiple instances of major systemic security breaches. Impact duration cannot be determined and senior management will be required to place the company under voluntary administration or other form of major restructuring. Criminal proceedings against senior management is expected, and substantial loss of business and failure to meet organizational objectives is unavoidable. Compliance costs are likely to result in annual losses for some years, with liquidation of the organization likely.

Table 3.2: Risk Consequences. Table 14.3 from [15].

	Consequences					
Likelihood	Doomsday	Catastrophic	Major	Moderate	Minor	Insignificant
Almost certain	E	E	E	E	H	H
Likely	E	E	E	H	H	M
Possible	E	E	E	H	M	L
Unlikely	E	E	H	M	L	L
Rare	E	H	H	M	L	L

Risk level	Description
Extreme (E)	Will require detailed research and management planning at an executive/director level. Ongoing planning and monitoring will be required with regular reviews. Substantial adjustment of controls to manage the risk is expected, with costs possibly exceeding original forecast.
High (H)	Requires management attention, but management and planning can be left to senior project or team leaders. Ongoing planning and monitoring with regular reviews are likely, though adjustment of controls is likely to be met from within existing resources.
Medium (M)	Can be managed by existing specific monitoring and response procedures. Management by employees is suitable with appropriate monitoring and reviews.
Low (L)	Can be managed through routing procedures.

Table 3.3: Risk Level Determination and Meaning. Table 14.4 from [15].

Asset	Threat / Vulnerability	Existing Controls	Likelihood	Consequence	Level of Risk	Risk Priority
Internet router	Outside hacker attack	Admin password only	Possible	Moderate	High	1
Destruction of data center	Accidental fire or flood	None (no disaster recovery plan)	Unlikely	Major	High	2

Table 3.4: Example Risk Register, detailing likelihood, consequence, risk level, and existing controls for each threat to an asset. Table 14.5 from [15].

3.2 Risk assessment of the MCP Identity Registry

Here we apply the methodology that we established in Section 3.1. We first identify our assets and estimate their value and the severity of the consequences, should these assets be compromised. After that, threats and vulnerabilities are identified, and risk likelihood and consequences estimated. Finally, based, on this information, a risk level is assigned, and everything is documented in a risk register.

3.2.1 Assets

Database information. The identity registry stores organisational data, personal user information, credentials, certificates, etc., in an associated database. The asset possesses the first two qualities of the CIA triad, confidentiality and integrity. This is because some identity information must often be kept confidential and the integrity of the information must not be compromised as the whole system revolves around verifying identities.

The identity service itself. Through its REST API, the identity registry allows for functionality such as verifying identities, creating and revoking certificates, governing details about organisations, vessels, users, and devices, managing permissions, etc. The primary value of this asset facilitates itself as availability, as it needs to be running optimally to provide users with the service.

3.2.2 Threats and vulnerabilities

We have identified several human-made deliberate threats. We have not identified any vulnerabilities directly, but we have identified places where there is potential for them to sit unnoticed now or where they could appear in the future. It is detailed more in the risk analysis in Subsection 3.2.3.

As it stands now, the identity registry is hosted only on a single virtual machine in the Amazon cloud. This suboptimal deployment, that we know is not the intended and final solution, may make the identity information subject to several additional threats, especially those of the natural and accidental human-made kind. For this reason, we have chosen not to include threats such as "accidental termination of virtual machine", "bankruptcy of Amazon", and "(insert natural disaster here) hits Amazon data centre". If the system is set up in a manner so that it is distributed across the globe, synchronising the information between instances, like it is meant to be, many of these threats will disappear.

The threats and potential vulnerabilities, as described by to the OWASP 2017 report [16], as well as some other well-known vulnerabilities, are explained on a general level below.

Injection

Injection attacks occur when untrusted data is sent to an interpreter or parser without being sanitised. If the data is blindly inserted into a query or command that gets executed, it can alter the query or command to do something completely different. If vulnerable, it could leave an attacker with remote code execution abilities or complete control over a database.

Motivation: an attacker may want to alter, delete, or gather information for nefarious purposes and profit.

Capability: these types of attacks are fairly easy to deploy.

Resources: requires only a single machine and no more than a couple of minutes of tinkering once a vulnerability is found.

Probability: the probability of this kind of attack being used is high, even if a bug has not been discovered yet, it is fair to assume that there are bots roaming around the web looking for injection vulnerabilities.

Deterrence: the attacker could be prosecuted if identified, but these attacks can be carried out very anonymously over the internet.

Broken Authentication

Some systems come with factory default credentials (including the MCP Identity Registry), thus permitting credential stuffing and brute force attacks. Credential stuffing is basically just trying a bunch of likely usernames and passwords such as "root" and "password" or "admin" and "admin". Brute-forcing refers to trying as many possibilities as possible and as fast as possible.

Motivation: an attacker could be interested in finding out how much of a system they can compromise with relatively little effort, either out of curiosity or profitable reasons.

Capabilities: very easy; credential stuffing can be accomplished by novice hackers; brute forcing requires some, but not much, experience with brute force tools.

Resources: very cheap in time and effort; tools and exploits can be crafted in terms of hours, perhaps minutes; exploitation tools will subsequently run continuously without intervention by attacker.

Probability: almost certain; anyone can do it. Automated bots roam the internet utilising credential stuffing and brute forcing.

Deterrence: fear of prosecution may deter some low-level adversaries but others have no reason not to try.

Sensitive Data Exposure

In many web applications, sensitive data is present in an insecure format, either in storage or in transit. This could be credit card information, passwords, usernames, emails, and other personal information. If this data can be accessed by an unauthorised person, it is classified as sensitive data exposure.

Motivation: attackers may target sensitive data exposure for high financial gain through blackmail, for making a point, or for personal reasons.

Capabilities: varies; depends what kind of attack is used to access the sensitive data.

Resources: in the worst case, very little resources, time, and skill are required; very sensitive data could be obtained through broken authentication, a very easily exploitable security vulnerability, as mentioned above.

Probability: very likely due to the high value of sensitive data and potentially low effort required to access it.

Deterrence: it is likely that no cybercriminals will be deterred from attempting to access sensitive data as the value gained far outweighs the risk of being caught.

XML External Entities (XXE)

References inside XML documents can refer to entities outside the documents, for example files, file shares, and can be used for things such as port scanning and remote code execution.

Motivation: attackers may want to find a way to execute arbitrary code and take over the system.

Capabilities: takes fair amount of skill to probe the target system using XXE to find and exploit a vulnerability.

Resources: takes considerable amount of time, both to probe target system and to craft an exploit.

Probability: possible although improbable that an attacker will attempt to probe the system.

Deterrence: there is some deterrence in the sense that the time and resources required are not worth the risk of being caught, and the risk only increases as a probing or attempted attack goes on.

Broken Access Control

Similar to Broken Authentication, when the restrictions on what users are authorised to do are not configured correctly, the users may gain access to view confidential information or abuse functionality.

Motivation: less motivation from employees/users to exploit accidental authorisation than an adversary might have.

Capabilities: no additional capabilities required except knowledge of the system's functionality.

Resources: no additional resources required.

Probability: unlikely due to lack of motivation and personal gain.

Deterrence: company may have surveillance, keep logs, etc., to deter employees from abusing broken authorisation.

Security Misconfiguration

Examples of security misconfiguration include but are not limited to:

- failing to set up firewall rules to block unrelated traffic,
- deploying software with incorrect or incomplete configurations, and
- failing to hide revealing debug/error messages.

This kind of vulnerability is particularly abundant because it can manifest itself in so many ways.

Cross-Site Scripting (XSS)

Cross-site scripting is when an attacker inputs something on a website that is displayed back to themselves or other clients later. JavaScript code can be inserted that steals various information such as authorisation cookies, personal user information, and so on. The dangerous thing about this is that all the attacker needs to do is input malicious code and then wait for users to visit the site, or manually supply them with a seemingly innocent URL. Most often, this is an attack that targets the visitors of the site rather than the site itself. To probe a site for XSS flaws, all the attacker needs to do is include some JavaScript code in input fields that proves or disproves the existence of a flaw.

Motivation: an attacker may want to collect all kinds of information for each visitor and store it in a database that can either be sold or used later to target a visitor individually.

Capabilities: takes insignificant amount of skill to probe site for XSS vulnerabilities; moderate amount of skill needed to exploit them.

Resources: insignificant amount of time needed to probe for XSS flaws.

Probability: high, because of the low skill and resource requirements.

Deterrence: some individuals, like site owners, may be able to see (in the database or in log files) that the site is being probed for XSS flaws and can potentially identify an attacker. Additionally, the destination of the data collection can potentially be tracked.

Insecure Deserialisation

Deserialisation, which refers to loading data that was serialised on one machine into memory on another, can result in remote code execution, and other serious attacks such as injection attacks and privilege escalation attacks, if not done correctly. An example could be that a cookie is saved in a client's web browser in some format that the particular programming language can read into memory. The username and cookie are stored, potentially along with authorisation information. The attacker can change this information to gain access to previously inaccessible functionality [16].

Motivation: an attacker may want to seise control of a system to include in a botnet or to gain access to the network where the system resides.

Capability: complicated; requires knowledge about target system's use of deserialisation as well as exploit crafting.

Resources: takes some time to understand the target system and find an exploit. Does not require access to source code, as the deserialised data is often saved on disk or transferred over some communication channel and can easily be inspected.

Probability: unlikely unless target is a high-value asset.

Deterrence: the attacker could be prosecuted if identified, but identification is unlikely as exploits can be deployed remotely.

Using Components with Known Vulnerabilities

The JVM, like any other software, is prone to security holes, and when these are found and patched, it is essential to update the Java software on the systems. Similarly, this is not only true for the software itself but also the protocols it uses and the systems it runs on. For example, a vulnerability in the TLS protocol could compromise the communication between the identity registry and the clients, while a vulnerability in the operating system on which the identity registry runs, could leave an attacker in full control of the identity registry.

Insufficient Logging & Monitoring

When logging and monitoring are not applied to a sufficient degree, the time to respond to an attack or incident is often over 200 days [16]. Attackers can then stay inside the system or network for longer periods of time and even spread to other systems.

Social engineering

Not all people are equally proficient in the tech world and this means some people can be tricked into giving up credentials themselves or installing malware that will eventually get hold of the credentials automatically. It often happens through phishing emails designed to lure gullible people into clicking malicious links or attachments. There is no real solution to the problem, but mitigation can be achieved by making users of the system aware of the risks and coaching management to adhere to industry best practices such as the principle of least privilege, limiting damages should someone compromise their credentials.

Motivation: social engineering is a popular choice for cybercriminals to make easy money.

Capabilities: can be as simple as sending poorly written emails to enough people. That is, trivial skills are needed.

Resources: the time needed depends on the length of the email, which, in fact, does not need to be very long to be effective.

Probability: certain.

Deterrence: none; can be carried out remotely.

DDoS

Of course, the possibility of a DDoS attack is always there since, at the very least, all you need is more bandwidth than the target system can handle. Such attacks are usually carried out by attackers with large botnets but recently, with the discovery of amplification attacks like the 1.35Tb/s GitHub DDoS in 2018 [17], the necessary bandwidth for an attack is now only a fraction of the target system's capacity. Other than hiring a company to handle a potential DDoS attack, there is little to do, both in terms of prevention and mitigation.

Motivation: motivations of DDoS attacks can range from cybercrime to cyberactivism to cyberterrorism.

Capabilities: can range from simple skills, for example knowing how to order an attack on the deep web, to complex skills if carrying out such an attack. In the latter case, large botnets are typically required, meaning that the attacker must already have compromised many other victim (now zombie) machines.

Resources: building up a large botnet will take considerable time and resources. Ordering a DDoS attack seems to cost a few hundred dollars (in a cryptocurrency, naturally) according to a quick Google search.

Probability: not improbable once high-value clients have signed up to use the registry.

Deterrence: lack of lucrative gain may deter most attackers.

Man in the middle attack

A Man in the Middle (MitM) attack is the act of covertly listening in on traffic that goes between a client and a server, thus potentially snooping on confidential information. In general, as long as all communication between client and server is encrypted, an MitM attack is unlikely. However, in 2013, there was an incident where the browser on Nokia phones would decrypt the user's HTTPS traffic in order to speed up loading times, effectively giving Nokia clear-text access to the traffic [18].

Motivation: an attacker may want to listen in on activities and traffic to gather confidential information that can be sold or otherwise used for nefarious purposes.

Capability: complicated; requires knowledge about target system as well as known exploit in protocol.

Resources: takes some time to understand the target system and find an exploit, either in the target system or in an underlying protocol, if one exists.

Probability: unlikely because protocol exploits will likely be used on high-profile targets instead.

Deterrence: the attacker could be prosecuted if identified, but identification is unlikely as MitM attacks are covert in nature.

3.2.3 Risk analysis

The MCP Identity Registry is deployed on an Amazon EC2 instance, so there is no synchronisation between systems or data centres. Whether or not the system is backed up regularly if at all is unknown, but we will assume there is some basic backup routine deployed like database dumps. The risk analysis is based on what we know about how the system is deployed right now.

Our resulting risk register is not to be seen as an exhaustive list of risks, but rather, a ranking of the most significant ones as pointed out by OWASP [16]. In some of the rows we have determined that, in case this vulnerability is actually present in the system, it would leave multiple assets compromised. For

Asset	Threat/Vulnerability	Existing Controls	Likelihood	Consequence	Level of Risk	Risk Priority
Database information & identity service	Security Misconfiguration	Firewall most likely set up in a relaxed fashion	Likely	Major	Extreme	1
Database information & identity service	Using Components with Known Vulnerabilities	Unknown	Possible	Major	Extreme	2
Identity service	DDoS attacks	Single (and possibly low-spec) machine	Possible	Major	Extreme	3
Database information	Injection	Model based DB library	Unlikely	Major	High	4
Database information	Sensitive Data Exposure	Swagger specification contains many GET requests	Likely	Minor	High	5
Database information & identity service	XML External Entities	No XML input, no SOAP frameworks used	Rare	Major	High	6
Database information & identity service	Broken Authentication	Few, organised places with predefined credentials	Unlikely	Major	High	7
Database information	Cross-Site Scripting	Inputs inserted (potentially unsanitised) into database	Unlikely	Major	High	8
Database information & identity service	Insecure Deserialization	No serialized data as input or saved in databases or on disk	Rare	Major	High	9
Database information & identity service	Broken Access Control	Few, organised places with predefined credentials	Possible	Minor	Medium	10
Database information & identity service	Social engineering	Keycloak manages authorisation	Possible	Minor	Medium	11
Database information	Man in the Middle Attack	HTTPS used everywhere	Rare	Moderate	Medium	12
Database information & identity service	Insufficient Logging & Monitoring	Identity registry and Keycloak employ logging	Possible	Insignificant	Low	13

Table 3.5: Risk register for the MCP Identity Registry, detailing likelihood, consequence, risk level, and existing controls for each threat to an asset.

example, if a vulnerability grants an attacker remote arbitrary code execution abilities, all assets (at least on that particular system) would essentially be compromised. We have analysed the risk consequences and level accordingly. The risk register can be seen in Table 3.5.

It is worth pointing out that we do not have access to details about how the system is deployed, as it is currently running in beta. Below is our explanation for the risk levels that we obtained and documented in the risk register.

Risk priority 1 The highest priority, security misconfiguration, has the likelihood rating of “likely” because there are very many things that can be configured incorrectly, not only the software, but also on operating system level. The consequences can range from minor data spills to an attacker seising complete control over the system and as a result, the “major” consequence rating has been applied. This amounts to an extreme risk level.

Risk priority 2 The likelihood of using components with known vulnerabilities, is rated as “possible” because the MCP Identity Registry is already using a vast number of components and libraries, and the probability that some vulnerability will be found in a library that is used, is then that much higher. The consequences are rated as major, again, because the worst-case scenario is that an attacker gains root access on the system, and this again results in an extreme risk level.

Risk priority 3 DDoS attacks are always a possibility and may be the result of cybercrime, cyberactivism, and cyberterrorism. With regard to consequences, if the identity registry were to be subject to a lasting DDoS attack, the availability of the service would be compromised, effectively halting the system for as long as the attack lasts. So, with this moderate likelihood and the major consequences, we reach an extreme risk level.

Risk priority 4 Injection attacks exhibit an “unlikely” likelihood rating, because the Java implementation uses a high-level model-based library to interact with the database, significantly reducing the risk of SQL injections. Better still, the implementation in general uses many well renowned high-level libraries, further reducing the risk of other types of injection attacks.

Nonetheless, the major consequences of an attack are significant enough that this threat/vulnerability receives a high risk level.

Risk priority 5 Sensitive data exposure proves to have a high risk level. This is, despite the moderate consequences, because of the high likelihood. The minor consequences can be attributed to the fact that confidential information may be compromised but the system as a whole does not cease to function and no data is irretrievably lost. Additionally, the data that is exposed is not chosen by the attacker, as this vulnerability is not exploited through an offensive attack in the first place, but rather a careful probing of what information is accessible. The high likelihood stems from the fact that the identity registry provides many ways to retrieve information from the database using the HTTP GET method. Should any of these GET method handlers be coded incorrectly, there may be a possibility for data exposure.

Risk priority 6 XML External Entities are rated as “rare” because the existing environment, at a quick glance, does not fulfil the terms for XXE vulnerabilities to arise, as described under existing controls. Still, because the code base is quite extensive and the consequences would be major (e.g. remote code execution), a thorough combing of the source code is justified. The "high" risk level is appropriately assigned.

Risk priority 7 Broken authentication is given the likelihood "unlikely" because authentication and credentials are handled by Keycloak. For example, custom password policies can be defined with settings such as minimum length, hashing algorithm and iterations, required characters, expiry date, and more.[4] Giving a strict password policy will significantly reduce the probability of a successful brute-force or credential stuffing attack, especially the minimum length setting.

Still, the MCP Identity Registry comes with many settings related to certificates and these must also be configured correctly. Hence, all the configuration files with predefined credentials and authorisation settings need to be thoroughly inspected. This is warranted as the major consequences greatly outweigh the low effort required to diminish the risk. Accordingly, a risk level of "high" is applied.

Risk priority 8 Cross-site scripting is rated as "possible", the middle likelihood rating. The reason for this is that the identity registry has many different methods for accepting input from a client through its REST API. The possibility of a cross-site scripting flaw *existing* is thus quite prominent. However, it is must be noted that the flaw will not be exploitable unless the result that is displayed back to a client is somehow parsed or interpreted, which, as of right now, it is not. The major consequences can be attributed to how this type of exploit can spy on people, act on their behalf, and even alter or remove functionality (by for example redirecting people), compromising confidentiality, integrity and availability. Hence, a risk level of "high" is assigned.

Risk priority 9 Insecure deserialisation is given a "rare" likelihood rating as nothing suggests that the identity registry stores or transmits any serialised data, and if so, it is managed by a high-level Java library. Combined with the major consequences (e.g. remote code execution) we still obtain a "high" risk level.

Risk priority 10 Broken access control (or broken authorisation), has the likelihood rating "possible". Despite Keycloak handling identity and access management, the chance of accidentally misconfiguring permissions is quite high. The reasoning for rating consequences as "minor", unlike the "major" rating for **Risk priority 7**, is that gaining unauthorised access more likely to happen yet *less likely* to be exploited. For example, an employee having permissions incorrectly configured could be unaware of it, and even if found out, they might be unwilling to exploit it or even take measures to have it fixed. Additionally, the exploitable functionality might be contained to a specific area. In conclusion, we end at a medium risk level.

Risk priority 11 Social engineering is an ever-evolving category of its own and therefore is given the likelihood rating "possible". Despite the extreme

[4] `https://www.keycloak.org/docs/3.0/server_admin/topics/authentication/password-policies.html`

prevalence of social engineering in the modern world, the reason for not rating the likelihood higher, is that the MCP Identity Registry is not likely to be targeted directly, but instead indirectly by, for example, phishing credentials off of users where their e-mail might reveal their relation to MCP. The consequences are estimated to be minor as higher ranking personnel is less likely to fall into these traps and thus, only lower authorisation levels can be accessed. In conclusion, a "medium" risk level is applied.

Risk priority 12 Man in the Middle (MitM) attacks reach a medium risk level. The consequences are rated as "moderate" because, despite the attacker not being able to choose which information is compromised (similarly to **Risk priority 5**), the attacker may through longer periods collect information from many victims at once if a systemic vulnerability is exploited. The rare likelihood rating is based on the fact that MitM attacks require a vulnerability in either the identity broker, in this case Keycloak which is we don't gauge to contain such vulnerabilities, or in the underlying traffic's encryption.

Risk priority 13 Insufficient logging and monitoring is not likely to be the cause of an exploit but rather the reason some attack, whether ongoing or already completed, is not found. Thus, the possible likelihood and the insignificant consequences result in a low risk level.

Above, we have identified several risks that could potentially be limited with the use of a fuzz tester, several of them with a risk level of "high" or above. These include sensitive data exposure, broken authentication and access control, DDoS attacks, and injection. As we discussed in Section 2.4, a fuzz tester can be used to catch a wide range of different vulnerabilities and bugs. Based on this risk assessment as well as the background on fuzz testing, we conclude that the MCP Identity Registry would benefit from being subjected to fuzz testing.

In the following chapter, we will set up a virtual environment in which the MCP Identity Registry can be simulated, as well as be tested upon by our fuzz tester implementation, which we will introduce in Chapter 5.

Lab environment

4

One of the goals of this thesis is to “set up a virtual lab environment that simulates MCP”. In this chapter we describe how our lab environment is set up, with detailed instructions on how to replicate it. This will enable others to reproduce our results, and hopefully, build upon our work. As previously discussed, we focus solely on the identity registry.

Section 4.1 gives an overview of the components in the identity registry and how they interact. In Section 4.2, we describe the Docker setup we created for our virtual lab environment. Finally, in Section 4.3, we describe how to run our lab environment.

4.1 Structure of components

The MCP Identity Registry as a whole consists of the identity registry component itself and a Keycloak instance. Depending on the scale of the setup, these components can use external databases and/or reverse proxies that unwrap the HTTPS traffic into HTTP requests to the APIs.

Below is a diagram that shows a real world example of how the components communicate with each other.

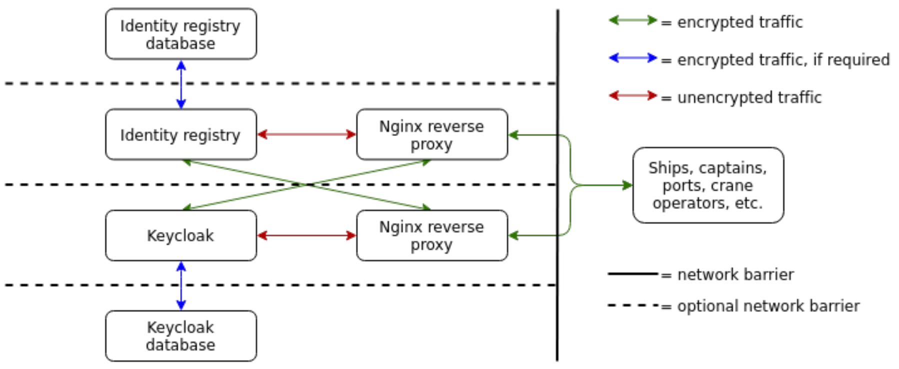


Figure 4.1: Communication between the different components and subcomponents of the identity registry. The figure demonstrates how no unencrypted traffic leaves a component on its own separate network.

As we can see in Figure 4.1, each core service is separated to its own network, and perhaps even its own machine. This means that the unencrypted traffic never leaves the network (or machine), and that all traffic that does go across networks will be encrypted, thanks to the reverse proxies.

4.2 Docker setup

To ensure that our setup is reproducible, we have chosen to use Docker as the backbone of our lab environment. Docker is a tool to create, run, and distribute applications as images to ensure all dependencies are shipped along with the application.

We have used Docker Compose[5], a tool that simplifies orchestration of multi-container Docker setups. It works by having a central `docker-compose.yaml` file that defines the different services that depend on each other. This file can also modify the behaviour of a service by

- changing its start command and/or parameters,
- setting environment variables,
- mounting host directories to guest containers (often with initialisation scripts or configuration files),
- attaching containers to different networks, and
- mapping network ports.

To avoid complicating our setup, we have used only images from the Docker Hub. The versions of the Docker images are locked to versions that we have tested with and that we know are working, so using our docker-compose.yaml along with our resources will ensure a setup exactly identical to ours.

4.2.1 Network separation

As opposed to the example in Section 4.1, our setup is a little different. Looking at Figure 4.2, we can see that the network segmentation is not as extensive. This is both due to how Docker works, but also because massive network segmentation is not necessary for a development environment. As seen in Figure 4.2, the different containers are set up so that they can only communicate with other services on their network. This means that databases, for example, which do not need to be accessible by the user, are on a back-end network and only communicate with the front-end. This simulates the real world where back-end services are hidden from the user. If, instead, it was desired to have a production-ready Docker setup, the example in Figure 4.3 with the following network segmentation could be used:

1. A private network — where all back-end containers (databases, keycloak instance, identity registry) can communicate together but not with the outside world.
2. An intermediary network — where the containers that need to connect to the private network can be connected with another container that also needs to be on the public network. This is to avoid that containers are not on the private and public network at the same time.

[5] `https://docs.docker.com/compose/`

3. A public network — where containers that are exposed to the outside world reside.

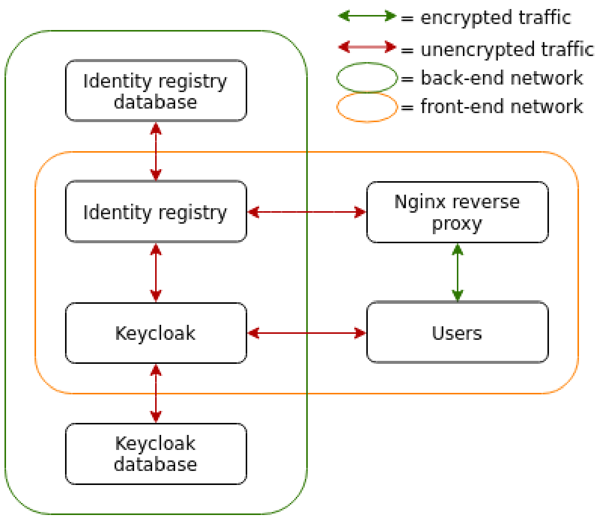


Figure 4.2: Communication between the different components in our identity registry Docker setup.

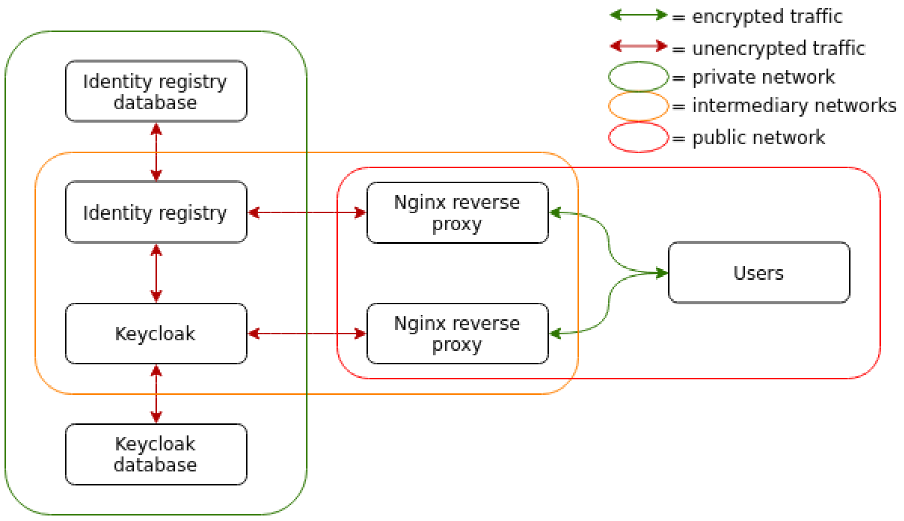


Figure 4.3: Communication between the different components in an optimal identity registry Docker.

4.2.2 Images

These are the Docker images used in our setup:

- `mariadb:10.3.14` (for keycloak database)

- `mysql:5.7` (for identity registry database)
- `dmadk/mc-identity-registry-api:0.9.0-RC1`
- `dmadk/keycloak-mysql-mc-ha:0.9.0-RC1`
- `nginx:1.14.2` (for reverse proxy)
- `munkyboy/fakesmtp` (for sending fake emails, used by identity registry. No version tag; not provided by author)
- `dwdraju/alpine-curl-jq` (for inserting data into Keycloak and the identity registry after startup)

4.2.3 Data created at startup

In order to be able to test most functionality of the identity registry, certain data structures need to be created first. We therefore follow the instructions from the README on GitHub.[6] The following data is inserted into the identity registry:

- An organisation with URN `urn:mrn:mcl:org:dma` named "Danish Maritime Authority" is created.
- A "Role" is created within that organisation so that the role name `ROLE_ORG_ADMIN` (organisation admins) is granted to users who have `MCADMIN` permissions.
- A user for the above organisation is created with URN `urn:mrn:mcl:user:dma:dma-employee` with `MCADMIN` permissions.
- A certificate is issued for the new user so that authorisation can be done with certificate files instead of through authorisation headers.
- Two new vessels are added to the created organisation, one identified by MRN `urn:mrn:mcl:vessel:dma:poul-loewnoern` and another by MRN `urn:mrn:mcl:vessel:dma:jens-soerensen`.
- A new device is added to the organisation, identified by MRN `urn:mrn:mcl:device:dma:hirtshals-fyr`.
- A new service is added, identified by `urn:mrn:mcl:service:instance:dma:party`.

In total, this means that every data structure in the identity registry has been created when the Docker setup has finished. This also acts as a way of ensuring that the identity registry is working as it should in the Docker setup.

[6] `https://github.com/MaritimeConnectivityPlatform/IdentityRegistry#insert-data`

4.3 Instructions to run

The Docker setup requires at least 2 GB of memory (or 3 GB if running with authorisation tokens). Software-wise, it is required that relatively new versions of Docker and Docker-compose be installed. We tested with the following versions and later:

- Docker Compose version 1.17.1, build unknown,
- Docker version 19.03.1, build 74b1e89.

Also, the following line must be added to `/etc/hosts`:

```
127.0.0.1       keycloak                keycloak
```

The reasoning for this is that our fuzzer (see Section 5.3) will contact the Keycloak instance (through localhost) to obtain an authorisation token. Since the hostname in the Docker setup is "keycloak" and not "localhost", an incorrect token is generated. With the above line, the "Host:" HTTP header will be "keycloak" but the request will still be sent to localhost.

To run the main program, change into the directory where the `docker-compose.yaml` resides and run the command

```
sudo docker-compose up
```

This will fetch all the required images and then start the services.

Note that we have had issues during startup with hanging for several minutes on Arch Linux. Our testing on Ubuntu and Red Hat derivatives, on the other hand, went smoothly. Setting up the lab environment from scratch on a DigitalOcean VPS is described in Appendix A1.

The virtual environment that we set up in this chapter will be used to allow our fuzz tester, which we will introduce in the next chapter, to continuously run on the MCP Identity Registry in a controlled environment.

Implementation

5

In this chapter. we start by introducing existing fuzzer tools and our reasons for creating our own fuzzer in Section 5.1. Then, we introduce Kitty, a Python framework for creating fuzzer tools, in Section 5.2. Having the necessary fuzzing background knowledge, we then describe our own fuzzer called RESTFuzzer, which uses the Kitty framework, in Section 5.3. Finally, in Section 5.4, we provide instructions on running our fuzzer.

5.1 Introduction

As part of our security analysis of MCP, we decided to do fuzz testing on the API exposed by the identity registry service.

In order to perform fuzzing of the identity registry, we needed some tool that allows for fuzzing web services. Traditional well-known fuzzer tools such as `American Fuzzy Lop` (AFL)[7] and others only work on binaries and are thus not suitable in our case.

We found several minor projects on the web for fuzzing web services, but none of them fulfilled our requirements. Examples of existing fuzzer tools for web services include APIFuzzer[8] and API-fuzzer[9] (same names, different projects). A promising project named RESTler [19], which fulfils our requirements, also exists but its source code is not available at the time of this writing.

Therefore, we set out to create our own fuzzer. We designed it so it works on a Swagger/OpenAPI 2.0 specification that is provided by the identity registry. As a bonus, the tool can work on any other service that provides a Swagger/OpenAPI 2.0 specification. A Swagger/OpenAPI specification is a JSON/YAML file that lists all endpoints of a REST API interface, including details such as possible return codes and expected inputs and parameters[10].

One of the main contributions of this thesis is the resulting product – `RESTFuzzer`, a Python framework that easily allows fuzzing of REST APIs with little setup. The code is written in Python 3.

[7] `http://lcamtuf.coredump.cx/afl/`

[8] `https://github.com/KissPeter/APIFuzzer`

[9] `https://github.com/Fuzzapi/API-fuzzer`

[10] `https://swagger.io/specification/`

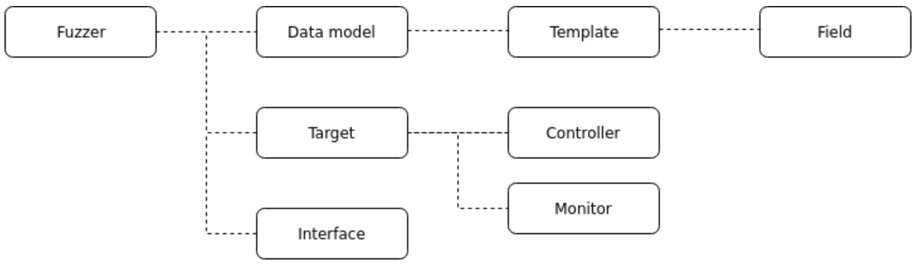


Figure 5.1: A structural overview of the components of a fuzzer created with the Kitty framework.

5.2 The Kitty framework

In order to refrain from reinventing the wheel, we decided to build on top of a framework that does the heavy lifting but still allows great customisation and control.

The Kitty framework is a modular and extensible fuzzing framework written in Python. Its goal is to allow easy fuzzing of unusual targets such as serial connections and other non-TCP/IP protocols. Kitty is an open-source framework written in Python and available on GitHub[11]. Note that Kitty by itself is not a fuzzer, but a framework that can be used to build a fuzzer.

A fuzzer written using Kitty consists of three core components: the data model, a target, and an interface, as can be seen in Figure 5.1. We will now describe them along with their subcomponents in more detail.

5.2.1 Data model

The data model defines what kind of data the fuzzer will generate. Data is represented by so-called “fields” that can be rendered into a value, most often bytes. Kitty comes with many built-in fields that can be subclassed for additional functionality. Each field, if defined as fuzzable, can mutate a finite number of times, altering the output value the next time it is rendered. Each of the built-in fields includes some interesting mutations in addition to its default value that was provided when the field was instantiated. For instance, a float comes with `inf`, `-inf`, `0.`, `-0.`, `NaN`, and `-NaN`.

Kitty is generally well-suited for generation-based fuzzing because it is quite straightforward to create a data model that represents the data exchanged in some protocol.[12] This makes it suitable for generating e.g. HTTP requests, because some parts can be made static, while we can choose which parts of the request we want to fuzz and how they should be fuzzed.

More generally, there are two types of data models in Kitty: low-level models and high-level models:

Low-level model. The low-level model allows for defining single payloads, constructed from multiple fields. Some elementary built-in fields for the low-level model are the following:

[11] https://github.com/cisco-sas/kitty

[12] https://kitty.readthedocs.io/en/latest/data_model/overview.html

1. `Static` — static data that is never fuzzed. This can be used for parts of the input that should never be altered. For example, in HTTP requests, you might never want to change or remove the `HTTP/1.1` part that comes after the path.
2. `String` — a string with predefined mutations, buffer overflow strings, command and SQL injections, etc.
3. `BitField` — integers. Some convenient aliases are defined such as `UInt8`, `SInt8`, `UInt16`, `SInt16`, etc.
4. `Float` — an arbitrary length floating point number.
5. `Delimiter` — a field used to represent delimiters such as colons, commas, and dashes.
6. `Dynamic` — a field where mutations are determined at run time.

Fields can be enclosed in container fields. Some notable built-ins are the following:

1. `Meta` — a field that does not render to any value, no matter which fields it contains. A useful field storing metadata.
2. `Pad` — pads each field with data; padding data and padding length are both user-specified.
3. `Repeat` — repeats the enclosed fields.
4. `OneOf` — renders only one of the enclosed fields at a time, until all of the specified fields have been rendered.
5. `Template` — the outer field that encloses all other fields. This field is important because it is the outermost field that corresponds to e.g. a fuzzable HTTP request for a specific URL.

For example, we might define a `Template` that contains an HTTP request to a specific URL that then fuzzes some query parameter in the URL. Several `Template`s can be bundled together and connected with a high-level model which is then directly used by the fuzzer.

Furthermore, fields can use specific encoders to alter the way the values are rendered. For example, a `String` initialised with the value "`hello world`" field can render to the ASCII bytes representing "`hello world`", or it can be rendered to Base64, which would yield "`aGVsbG8gd29ybGQ=`". This particular encoder could be useful in HTTP requests where the content type is `application/ocsp-request`, which usually requires the body to be Base64-encoded.

High-level model. The high-level model describes how a sequence of messages (`Templates`) looks like to the fuzzer [13]. The standard high-level model in Kitty is called `GraphModel`. In the most basic example, a series of `Templates` can be connected to the `GraphModel` independently. In that case, the fuzzer

[13] `https://kitty.readthedocs.io/en/latest/data_model/overview.html`

will test all the connected `Templates` one by one until each `Template` has been exhausted (meaning that all possible mutations were tested).

More complex behaviour is also possible with the `GraphModel` — templates can be connected to each other. Suppose we have three templates: A, B, and C. It is then possible to connect them in such a way that Template B is only tested after Template A, and Template C only after Template B. We will now look at an example to see the data model concepts in action.

Example: fuzzing HTTP requests

Suppose we have some REST API that manages blog posts, where a single post consists of a title and a body. Let's create a template that creates a new blog post for that API. Listing 5.1 shows Python code that generates a Kitty template for the path `/api/blog/create` with fuzzed JSON content:

Listing 5.1: Example of a Kitty template that fuzzes an HTTP request for creating blog posts. Tested with Python 3.

```python
from kitty.model import Static, String, Container, Template, \
    SizeInBytes, BitFieldAsciiEncoder

start = Static("POST /api/blog/create HTTP/1.1\r\n")
# Make the body a single container, so we can use it for SizeInBytes
body = Container([Static('{"body":"'),
                  String("my content"), # fuzzable
                  Static('", "title":"'),
                  String("my title"), # fuzzable
                  Static('"}')])
headers_start = Static("Host: localhost\r\n" +
                       "Content-Type: application/json\r\n" +
                       "Content-Length: ")
headers_end = Static("\r\n\r\n")
# The content-length value depends on the rendered size of the body
# which changes on every mutation
cnt_len_val = SizeInBytes(body, 32, encoder=BitFieldAsciiEncoder('%d'))
headers = Container([headers_start, cnt_len_val, headers_end])
tmpl = Template([start, headers, body])
n_mutations = 2
for _ in range (n_mutations):
    print("---begin request---")
    print(tmpl.render().tobytes().decode())
    tmpl.mutate()
    print("---end request---\n")
```

The output of above Python script is shown below in Listing 5.2.

Listing 5.2: Output of running the Python script from Listing 5.1.

```
---begin request---
POST /api/blog/create HTTP/1.1
Host: localhost
Content-Type: application/json
Content-Length: 41
```

```

{"body":"my content", "title":"my title"}
---end request---

---begin request---
POST /api/blog/create HTTP/1.1
Host: localhost
Content-Type: application/json
Content-Length: 51

{"body":"my contentmy content", "title":"my title"}
---end request---
```

The above output shows two valid HTTP requests; one before mutating, and one after. It can be noticed how the "`body`" field has changed after mutation, while the rest of the request has remained the same. This showcases how straightforward it is to generate fuzzable HTTP request templates in Kitty. The above requests are in fact quite similar to the ones we will be using when fuzzing the MCP Identity Registry.

The above covers data generation, but the data also needs to be transmitted to the target (in our case, an HTTP server). We will cover this next.

5.2.2 Target

The target defines the target machine and protocol. This is a requirement for the fuzzer to know where to send the fuzzed data, and how to receive a response, if possible/appropriate.

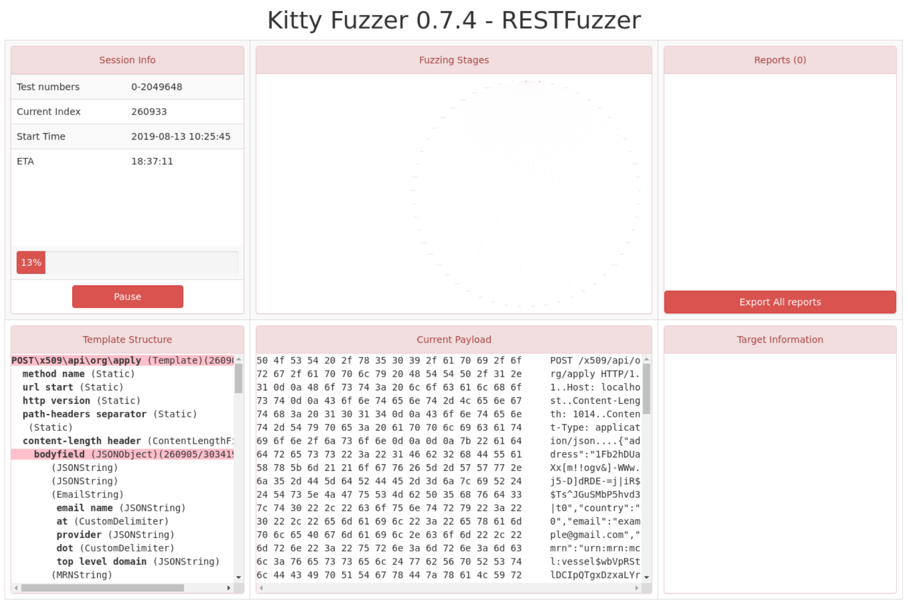
The target can additionally define a controller and a monitor. The controller is in charge of "controlling" the target, that is, to make sure the target is ready to be fuzzed. This could be starting a virtual machine, installing software, killing a process, and so on. Only one controller can be used per target but it is not required.

The monitor can monitor the state of the target, e.g. memory and CPU usage, network traffic, serial output, and so on. Multiple monitors can be attached to a target but they are also optional.

In our HTTP fuzzing example, the target would be responsible for sending each data payload (HTTP request) in the form of a TCP packet and being able to receive a response.

5.2.3 Interface

An interface is a user interface that makes it possible to monitor the fuzzer (not the target) as it runs. A default web interface is included with Kitty, which should be sufficient for most cases. It can be seen in Figure 5.2.


Kitty Fuzzer 0.7.4 - RESTFuzzer

Session Info	
Test numbers	0-2049648
Current Index	260933
Start Time	2019-08-13 10:25:45
ETA	18:37:11

Template Structure

```
POST\x509\api\org\apply (Template)(2609
  method name (Static)
  url start (Static)
  http version (Static)
  path-headers separator (Static)
   (Static)
  content-length header (ContentLengthF
    bodyfield (JSONObject)(260905/30341
       (JSONString)
       (JSONString)
       (EmailString)
        email name (JSONString)
        at (CustomDelimiter)
        provider (JSONString)
        dot (CustomDelimiter)
        top level domain (JSONString)
       (MRNString)
```

Current Payload

```
50 4f 53 54 20 2f 78 35 30 39 2f 61 70 69 2f 6f     POST /x509/api/o
72 67 2f 61 70 70 6c 79 20 48 54 54 50 2f 31 2e     rg/apply HTTP/1.
31 0d 0a 48 6f 73 74 3a 20 6c 6f 63 61 6c 68 6f     1..Host: localho
73 74 0d 0a 43 6f 6e 74 65 6e 74 2d 4c 65 6e 67     st..Content-Leng
74 68 3a 20 31 30 31 34 0d 0a 43 6f 6e 74 65 6e     th: 1014..Conten
74 2d 54 79 70 65 3a 20 61 70 70 6c 69 63 61 74     t-Type: applicat
69 6f 6e 2f 6a 73 6f 6e 0d 0a 0d 0a 7b 22 61 64     ion/json....{"ad
64 72 65 73 73 22 3a 22 31 46 62 32 68 44 55 61     dress":"1Fb2hDUa
58 78 5b 6d 21 21 6f 67 76 26 5d 2d 57 57 77 2e     Xx[m!!ogv&]-WWw.
6a 35 2d 44 5d 64 52 44 45 2d 3d 6a 7c 69 52 24     j5-D]dRDE-=j|iR$
24 54 73 5e 4a 47 75 53 4d 62 50 35 68 76 64 33     $Ts^JGuSMbP5hvd3
7c 74 30 22 2c 22 63 6f 75 6e 74 72 79 22 3a 22     |t0","country":"
30 22 2c 22 65 6d 61 69 6c 22 3a 22 65 78 61 6d     0","email":"exam
70 6c 65 40 67 6d 61 69 6c 2e 63 6f 6d 22 2c 22     ple@gmail.com","
6d 72 6e 22 3a 22 75 72 6e 3a 6d 72 6e 3a 6d 63     mrn":"urn:mrn:mc
6c 3a 76 65 73 73 65 6c 24 77 62 56 70 52 53 74     l:vessel$wbVpRSt
6c 44 43 49 70 51 54 67 78 44 7a 78 61 4c 59 72     lDCIpQTgxDzxaLYr
```

Figure 5.2: An overview of the web interface used to monitor a fuzzer's progress.

5.3 RESTFuzzer

Having just covered the basics of fuzzing in the Kitty framework, we will now describe how we built our own fuzzer, `RESTFuzzer`.

To adhere to the Kitty framework structure, some files are needed to create and start a fuzzer. We have not created all the possible components described above, as only a target, data model, and interface are strictly needed. The basic structure can be seen in Figure 5.3.

5.3.1 Parser

The `parser.py` module is responsible for generally parsing and manipulating the Swagger/OpenAPI specification[14].

An OpenAPI 2.0 specification is roughly structured as follows:

- Basic information about the API (name, title, contact information, version, licence, hostname).
- Tags: a list of tuples consisting of `name` and `description`. These tags can then be referenced by certain paths (requests).
- Paths: a central part of the specification that defines information for each possible URI, most notably the HTTP method, required and optional parameters and their location, and finally, the expected responses. The expected type and format for each parameter is presented as simple types such as integers, floating point numbers, strings or Boolean values. If the parameter types are of more complex types, like JSON arrays or

[14] It is limited to version 2.0 of the specification

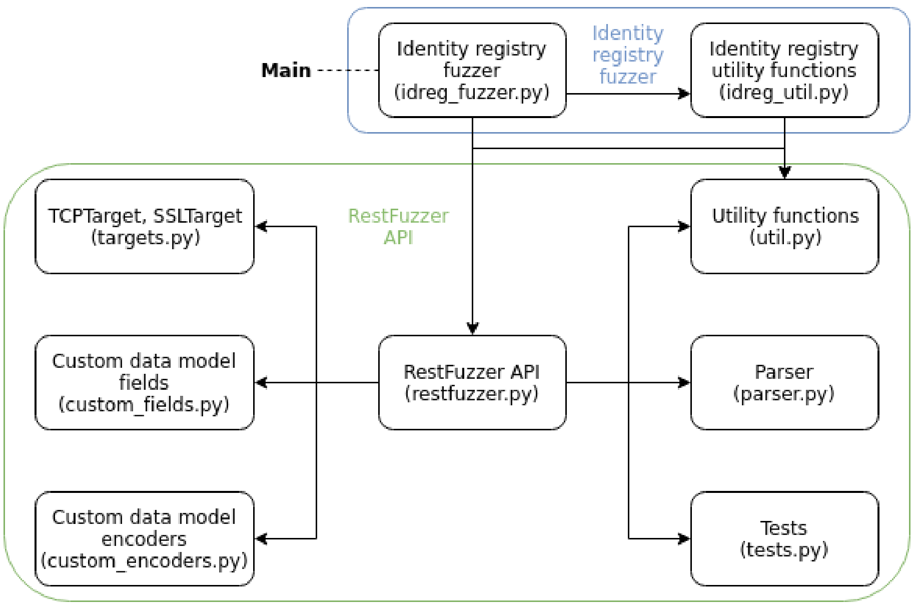

Main
Identity registry fuzzer (idreg_fuzzer.py)
Identity registry fuzzer
Identity registry utility functions (idreg_util.py)
TCPTarget, SSLTarget (targets.py)
RestFuzzer API
Utility functions (util.py)
Custom data model fields (custom_fields.py)
RestFuzzer API (restfuzzer.py)
Parser (parser.py)
Custom data model encoders (custom_encoders.py)
Tests (tests.py)

Figure 5.3: A structural overview of the RESTFuzzer API.

objects, they can refer to schema objects (see point below) that define the subparameters of the object or the expected types of the array elements. These objects may also be reused for several paths, reducing the size of the specification.

- Definitions: a dictionary of special objects ("schemas") that may be used by paths in the API, either in the request body or response. For example, in the identity registry, there is a definition for `Organization` which has the required items `adress`, `country`, `email`, `mrn`, `name` and `url`, as well as their types.

The specification can contain many other fields, but those above are the most important ones – in particular, paths and definitions.

Appendix A2 shows actual examples of a path and a schema object from the identity registry, see Listing A2.1 and Listing A2.2.

Once these are parsed we can start to assemble them into Kitty templates that can then be used by our fuzzer.

5.3.2 Targets

Our targets, specified in `targets.py` are modified versions of basic TCP and SSL targets found in the Katnip repository.[15]

The targets, `TCPTarget` and `SSLTarget`, implement the methods `_send_to_target` and `_receive_from_target`, respectively, which are responsible for delivering payloads to the target server and also receiving responses. This is done

[15] Katnip is an accompanying repository to Kitty which provides some additional fuzzing tools including but not limited to controllers, monitors, and targets.
`https://github.com/cisco-sas/katnip`

via socket programming. `SSLTarget` inherits from `TCPTarget` and overrides the method `_get_socket` where it “wraps” the socket to support SSL using `ssl.wrap_socket()`.

5.3.3 Custom fields

Moreover, we have extended some of the data model fields included with Kitty. In `custom_fields.py`, we have added fields to represent JSON objects and arrays and implemented functionality so that they will render all possible combinations between their child fields. Furthermore, since mutation count has such a great effect on run time, we limited the number of mutations for simple types by subclassing Kitty's built-in `String`, `Float`, `SInt32`, and `SInt64` fields, and only included the interesting edge cases for each particular type. We will now go over our own custom fields used by our fuzzer.

Integer fields. We have defined two integer fields, `SInt32` and `SInt64`, that we use for fuzzing integers. They have the following edge cases:

- 1,
- 0,
- -1,
- the minimum possible size for the particular integer, and
- the maximum possible size for the particular integer.

String field. For string types, we defined our own field, `JSONString`, that subclasses the built in `String` field. It renders as four different fixed strings that could trigger interesting behaviour:

- a string consisting of 750 consecutive `A`'s,
- `420000000` (a big integer),
- `0` (the number zero), and
- the empty string.

Furthermore, it renders three different random strings of varying length. One of the random strings consists of all ASCII letters, digits and printable symbols, another consists only of ASCII letters, and the third only of digits. This way, we introduce some randomness into our testing. This is to cover bugs that could be triggered by e.g. special symbols that were not tested by the five fixed strings above.

ThoroughContainer. By default, containers and templates in Kitty are rendered in a “one at a time” manner. Suppose we have this template with two fuzzable string fields:

```
tmpl = Template([Static("http://"),
                 JSONString("www"),  # fuzzable
                 Static(".example."),
                 JSONString("com")]) # fuzzable
```

Its first initial value will be `http://www.example.com`. When mutating, the fuzzer will start fuzzing the first fuzzable field in the Template, which is the `www` field:

```
while tmpl.mutate():
    tmpl.render().tobytes().decode()

'http://AAA(...)AAA.example.com' # note: 750 A's
'http://420000000.example.com'
'http://0.example.com'
'http://.example.com'
"http://JFvx+ca?(,1e;m.BPZID)$LxJK!s]UFT[c&6M'b,=S^!e!@cI51#7QNa&
    isqzBrh@NAcv.&Hk`t0h4M_fNt#q%-1x.example.com"
'http://BglwLdFquuEkBEqIjGfBCtZjYbSesFsDpc.example.com'
'http://013512487920.example.com'
```

Once all seven possible `JSONString` values have been tested for that field, it will reset to its default value, "`www`". The fuzzer will then move on to fuzzing the next fuzzable field, in this case the `com` part:

```
'http://www.example.AAA(...)AAA' # note: 750 A's
'http://www.example.420000000'
'http://www.example.0'
'http://www.example.'
'http://www.example.CIE)c&Xh,;;n^`G$8;y`O-MC4K]v54<>h[E~#8m%FGb}N-xW~_H
    `6B+3G/*TKuC<9at1N3A}Vw'
'http://www.example.rCcYSIfCZTVzHPdOnZP'
'http://www.example.422589270186136442236652592408170561719856114215333'
```

This means that the `www` field will remain the same while the fuzzer is testing out different values of the `com` field. Likewise, the `com` field remains unchanged while fuzzing the `www` field.

We reason that this is not optimal because some bugs might only be triggered when a multitude of fields have certain values. Therefore, we decided to create a container field `ThoroughContainer` which solves this problem by testing every single possible combination of mutations of individual fields. In the example above, it would test every possible combination of the two fuzzable fields, instead of going through one field while the other is at its default value. This means that the template in this particular example would have $7 \cdot 7 = 49$ different test cases instead of $7 + 7 = 14$.

This is great for examples like this where the total combination count is within reasonable limits. However, if you have a template with many different fuzzable fields, each having many mutations, the number of combinations becomes astronomical. The result is a never-ending fuzz test. To prevent that, we have built the class with a built in constant, `MAX_COMBINATIONS`. The default value of that constant is $200,000$. When the total number of combinations surpasses this constant, the class uses a different method of mutating fields, which we call "the random method".

The random method works the following way: on each mutation, up to two fields in the container are selected at random and mutated. That way, a number of different "states" are tested in a random fashion. This does of course not test all combinations but it does cover a lot of different states that would not have been tested by the default `Container` mutation method in Kitty.

JSON Objects. In order to generate HTTP bodies, we created a field called `JSONObject` which is capable of rendering JSON objects as they are typically included in e.g. POST requests where the content-type is `application/json`. The field takes as input a dictionary mapping from key names (e.g. "body" in above example) to fields (e.g. `Static` or `JSONString`). It inherits from `ThoroughContainer` so that it tests all possible combinations of fuzzable fields if possible. Here is an example of a JSONObject field:

```
JSONObject({'body': Static('Hello World'), 'title': Static('Cool Title')
    , 'version':SInt32(42)})
```

And here is what it would initially render to:

```
{"body":"Hello World","title":"Cool Title","version":42}
```

JSON Arrays. Arrays are often part of JSON objects. An example:

```
{"numbers":[1,2,3,4,5]}
```

Our `JSONArray` type takes as input a list of fields and renders them enclosed in square brackets with commas in between. It inherits from `ThoroughContainer` so that all combinations are tested.

Authorisation token field. In order to be able to fuzz APIs that require authorisation, an authorisation header can be necessary. For this purpose, we created a special field that takes as input a function which generates an authorisation header. We then use this field in our identity registry fuzzer which uses a login from Keycloak. The field also provides functionality to generate a new token periodically if necessary. Here is an example of an authorisation token:

```
Authorization: Bearer eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUI(...)
```

Content-Length field. HTTP requests that contain data (in the body) should always have a "Content-Length" header which specifies the length of the data in bytes. Since the data in the body is typically fuzzed, its length can vary, and the content length value should vary accordingly. To address this issue, we created a field called `ContentLengthField` which takes as input a field that should correspond to the body in the HTTP request. Every time the request is fuzzed, this field then renders the correct content length based on the newest rendered body.

Query strings field. Since part of the data we are fuzzing is URLs and some of those URLs can contain query string parameters, we needed to add support for that too. The field `QueryStringsField` takes a dictionary that maps keys to fields, for example {`'status':JSONString('failed), 'test':SInt32(32)`}. It could then be used in a URL like so:

```
www.example.com/getThing?status=failed&test=32
```

Multipart body field. In order to support file uploads via HTTP, we use `Content-Type: multipart/form-data` in the headers. The specific method is described in detail at `https://tools.ietf.org/html/rfc2388`.

This field takes as input a boundary (used in the multipart/form-data protocol), a name, a filename and a field that should render the actual file content. Those variables are all needed to create a body containing a file according to the aforementioned protocol.

5.3.4 Custom encoders

In `custom_encoders.py` we define some encoders that are either necessary for some of our custom fields, or for encoding values in a particular format. For example, the Swagger/OpenAPI 2.0 specifies the format "byte", which must be encoded as Base64, and "binary", which must be encoded as raw binary data.

5.3.5 Utility module

The `util.py` module provides various utility functions that are too general and therefore do not belong in any particular module. They are usually used across multiple modules.

In particular, there are methods for parsing HTTP requests and responses such that the individual components are given in a dictionary. A request is split up into method, path, headers and body; a response is split up into HTTP status code, headers and body. This is useful for logging and processing the results.

It also features a class called `SimpleReportManager` with the purpose of creating and managing an SQLite database that stores all test results. It creates three tables: `reports`, `metadata`, and `templates`. The `reports` table stores the HTTP request and response for each individual test. The `metadata` table is a key-value store that is used to store information such as time when fuzzing started and ended, number of templates, and settings used. The `templates` table maps template ID's to names such that template ID's stored in the reports table can be mapped to their names.

5.3.6 Tests

The `tests.py` module defines various unit tests. The most thoroughly tested modules are `custom_fields.py` and `util.py` due to their "simple" behaviour. All tests can be run simply by running `python3 tests.py` from the command line.

The `tests.py` file contains the following test classes, each of which is responsible for testing their own respective module:

TestParser tests whether the parsing of the identity registry results in the correct number of templates, definitions and paths,

TestUtil tests some of our utility functions from the `util` module,

TestCustomEncoders tests that our null value encoder and Boolean value encoder work as expected, and

TestCustomFields the biggest test module that verifies that our custom fields function as they should.

5.3.7 REST API fuzzer

In `restfuzzer.py` we define a class, `RESTFuzzer`, that will fuzz a specified REST API. This class in itself can be used as an API for anyone who wishes to fuzz a REST API. It takes the location of a Swagger specification, target host and port as input. Optionally, a key and certificate file can be specified as authentication when fuzzing over HTTPS.

5.3.8 Identity registry fuzzer

Here, we have used the RESTFuzzer API from `restfuzzer.py` to build a fuzzer. The `idreg_fuzzer.py` has much functionality tailored specifically to fuzz the MCP Identity Registry. For example, for many, if not all, of the paths that require valid MRN and certificates would result in bad requests. These have to be handled manually because the format of an MRN value is not specified in the Swagger specification and consequently the parser just generates generic sample data. The generic data has been replaced with actual MRN and certificate values instead.

5.3.9 Identity registry utility functions

Since our fuzzer needs a lot of functionality in addition to what the RESTFuzzer API provides, we have defined a new utility module, `idreg_util.py`. This is because we want to keep the RESTFuzzer API separate and because the content does not apply to any other modules other than our own fuzzer.

5.4 Instructions to run

First, the Kitty framework should be installed. This can be achieved using pip:

```
sudo pip3 install kittyfuzzer
```

However, it requires a small change in the source code to function optimally, namely the `kitty/fuzzers/server.py` file. The change fixes a bug[16] where the response callback function is called before the first response is received. We have submitted a pull request and the author has acknowledged the issue. In the meantime, the bug may remain on the master branch for a little while. See Listing 5.3 for the required patch. The location of the file on the file system can vary but it can be found with this short Python snippet:

```
import kitty.fuzzers.server as x; print(x.__file__)
```

Listing 5.3: Patch to fix a bug in the Kitty fuzzing framework

```
diff --git a/server.py.old b/server.py
index 4952cd8..0e6a26f 100644
```

[16] `https://github.com/cisco-sas/kitty/issues/84`

```
--- a/server.py.old
+++ b/server.py
@@ -66,12 +66,12 @@ class ServerFuzzer(BaseFuzzer):
         self._test_info()
         resp = None
         for edge in sequence:
-            if edge.callback:
-                edge.callback(self, edge, resp)
             session_data = self.target.get_session_data()
             node = edge.dst
             node.set_session_data(session_data)
             resp = self._transmit(node)
+            if edge.callback:
+                edge.callback(self, edge, resp)
         return self._post_test()

     def _transmit(self, node):
```

Note that if the response callback is already called at the end of the for-loop, the patch is not necessary.

After these steps, the identity registry fuzzer can be run by going to the `src` directory and running the following command: `python3 idreg_fuzzer.py`. Make sure to first start the services as described in Section 4.3. Run the program with the `-h` switch to see usage help and additional advanced options. Setup from scratch on a DigitalOcean VPS is described in Section A1.

Having just described our fuzzer implementation, we will use it for some experiments in the next chapter.

Experiments

6

In this chapter, we describe our experiment setup and the results that came from it. To set up experiments, we use the identity registry fuzzer we implemented and described in the previous chapter. We start by describing the experiments we chose and the reasoning behind them in Section 6.1. Then, we document the results we gathered from processing the data files that the tests produced in Section 6.2.

6.1 Setting up experiments

We chose to do three separate fuzzer runs on the identity registry:

1. Authorisation enabled, using certificates (only using paths beginning in `/x509` because those are meant for use with certificates).
2. Authorisation enabled, using authorisation headers (only using paths beginning in `/oidc` because these are meant for use with authorisation headers).
3. No authorisation (using only `/oidc` paths).

The reasoning behind these three experiments is the following. In the identity registry, most requests require some level of authorisation, for example when creating new users or vessels – those operations require one to be authorised in order to make changes within the organisation in which these new entities are created.

Therefore, in two out of the three runs, we authorise all requests with the admin user. This means that all requests are done with the highest possible authority, so all requests should be authorised. We do two separate runs with authorisation because there are two ways to authorise: via certificates (using `/x509` paths) and via authorisation headers (using `/oidc` paths).

In the fuzzing run where we authorise with certificates, we only test the paths beginning with `/x509`. In the run where we authorise with headers, we only use `/oidc` paths. In the third and final run without authorisation, we use the paths that begin with `/oidc` but do not use any authorisation header in the requests.

The reason for also doing a run with no authorisation is to determine if it would be possible to access, insert or modify any data which otherwise should be reserved for authorised users.

The reason for only using either `/x509` or `/oidc` paths and not both is that they access the exact same API but only the authorisation method differs;

therefore, we thought it would be superfluous to test both sets of paths in all runs.

Furthermore, in the runs where we use authorisation, we remove templates that use the `DELETE` method as well as templates that revoke certificates. The reason for this is that those particular templates might result in deletion of the certificate that we use for authorisation or the initial organisation created, which would mean that less requests would have an impact.

6.1.1 Configuration

The experiments were run on DigitalOcean Virtual Private Servers (or Droplets as they call them) with 1 vCPU, 2 GB memory, except the one using authorisation tokens, which needed 3 GB to complete. The exact DigitalOcean setup is described in Appendix A1.

We configured our fuzzer so that each test would take roughly 24 hours on a DigitalOcean setup. The duration of a run depends on the total number of mutations, i.e. the total number of tests in a run. We can influence the total number of tests by modifying how many mutations each of our custom fields has (for example, `JSONString` which is used quite often). Since a lot of the mutations are spent on modifying POST requests that contain JSON objects, our special `ThoroughContainer` field is used often and is thus responsible for a large portion of the mutations due to the way it functions.

The class `ThoroughContainer` contains two parameters, `MAX_COMBINATIONS` and `RANDOMIZED_FACTOR`, which impact how many mutations it will end up trying. We used the following values in our tests:

- `MAX_COMBINATIONS = 200000`
- `RANDOMIZED_FACTOR = 60`

These particular numbers have no special meaning and are merely used to influence how long a fuzzer run should take.

The total number of tests, templates, and the duration of each test can be seen in Table 6.1. The difference in number of templates and tests is due to the runs using authorisation having some templates removed (templates using DELETE method or templates that remove certificates).

Fuzzer Run	Duration	Templates	Tests
1 (auth. with certificates)	21h21m36s	52	2,049,650
2 (auth. with headers)	18h4m0s	52	2,049,650
3 (no authorisation)	21h14s	66	2,098,428

Table 6.1: Overview of each of our three fuzzer runs.

Mutation count for each template Table 6.2 shows the number of mutations for each template in our runs. This table is important because it shows which templates are tested more thoroughly than others. In general, the more fuzzable parts a request has, the more mutations it gets. For this reason, all top ten templates are POST or PUT requests. These requests most often include JSON bodies with several fuzzable values, hence the high mutation count. It therefore makes sense that such requests have a higher test count than a simple GET request with only one parameter.

6.2 Results

The fuzzer, as it runs, saves every HTTP request/response pair to a database. We have analysed this data and generated some graphs to highlight interesting points. The graphs were generated by performing postprocessing on all three resulting SQLite databases produced for each run. The file `postprocess.py` that is included in the source code takes an SQLite database as input and produces graphs and useful information as output.

6.2.1 HTTP status codes

To get an overview of the fuzzer results, we found it helpful to see which HTTP status codes were encountered in responses most often. In Figure 6.1, the HTTP status codes for all the HTTP requests that were sent during the fuzzer runs can be seen. We will now comment on specific status codes seen in this figure.

400 Bad Request. Common for all three runs is the large amount of `400 Bad Request` responses. This is largely due to the fuzzable parts in each request, e.g. parameters in URLs and JSON objects in bodies. Since a lot of these values are altered to invalid ones during fuzzing, this results in many `400 Bad Requests`. This is reflected by many of the error messages seen in those responses. The following list shows the top three most common error messages included in `400 Bad Request` responses, for run number two (the run using authorisation headers):

1. `MRN is not in a valid format`: 291258 occurrences.
2. `must be a well-formed email address, MRN is not in a valid format`: 102942 occurrences.
3. `MRN is not in a valid format, must be a well-formed email address`: 91263 occurrences.

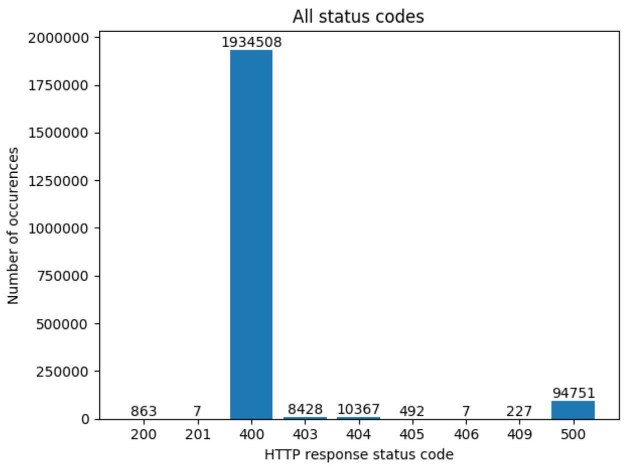
Most error messages in those `400 Bad Request` responses do indeed relate to errors such as the ones above, denoting that some parameter in the request was invalid.

401 Unauthorized. In the third run, we see a lot of `401 Unauthorized` codes. These are of course due to the fact that no requests in that run used any means of authorisation, and there are only a few paths in the identity registry API that do not require authorisation.

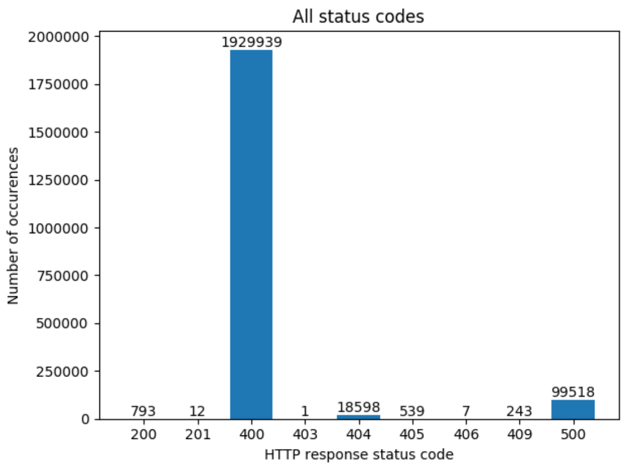
200 OK and 201 Accepted. Common for all three runs is the very small amount of "successful" status codes (`200 OK` and `201 Accepted`). In all runs, there are less than a thousand of those, which indicates that by far the large majority of all requests resulted in some unsuccessful status code. This is not necessarily a bad thing since our goal with fuzzing is to find security bugs.

Template	Mutations	Percentage
PUT\oidc\api\org\{orgMrn}	303,718	14.82
POST\oidc\api\org\apply	303,420	14.80
PUT\oidc\api\org\{orgMrn}\user\{userMrn}	302,584	14.76
PUT\oidc\api\org\{orgMrn}\service\{serviceMrn}\{version}	298,151	14.55
POST\oidc\api\org\{orgMrn}\user	297,718	14.53
POST\oidc\api\org\{orgMrn}\service	293,278	14.31
PUT\oidc\api\org\{orgMrn}\device\{deviceMrn}	44,099	2.15
PUT\oidc\api\org\{orgMrn}\vessel\{vesselMrn}	44,099	2.15
POST\oidc\api\org\{orgMrn}\device	39,233	1.91
POST\oidc\api\org\{orgMrn}\vessel	39,233	1.91
GET\oidc\api\org\{orgMrn}\service\{serviceMrn}\{version}	5,171	0.25
GET\oidc\api\org\{orgMrn}\service\{serviceMrn}\{version}\certificate\issue-new	5,171	0.25
GET\oidc\api\org\{orgMrn}\service\{serviceMrn}\{version}\jbossxml	5,171	0.25
GET\oidc\api\org\{orgMrn}\service\{serviceMrn}\{version}\keycloakjson	5,171	0.25
POST\oidc\api\org\{orgMrn}\vessel\{vesselMrn}\vesselImage	5,171	0.25
PUT\oidc\api\org\{orgMrn}\vessel\{vesselMrn}\vesselImage	5,171	0.25
GET\oidc\api\org\{orgMrn}\device\{deviceMrn}	5,164	0.25
GET\oidc\api\org\{orgMrn}\device\{deviceMrn}\certificate\issue-new	5,164	0.25
GET\oidc\api\org\{orgMrn}\service\{serviceMrn}	5,164	0.25
GET\oidc\api\org\{orgMrn}\user\{userMrn}	5,164	0.25
GET\oidc\api\org\{orgMrn}\user\{userMrn}\certificate\issue-new	5,164	0.25
GET\oidc\api\org\{orgMrn}\vessel\{vesselMrn}	5,164	0.25
GET\oidc\api\org\{orgMrn}\vessel\{vesselMrn}\certificate\issue-new	5,164	0.25
GET\oidc\api\org\{orgMrn}\vessel\{vesselMrn}\services	5,164	0.25
GET\oidc\api\org\{orgMrn}\vessel\{vesselMrn}\vesselImage	5,164	0.25
PUT\oidc\api\org\{orgMrn}\role\{roleId}	445	0.02
POST\oidc\api\org\{orgMrn}\role	441	0.02
PUT\oidc\api\org\{orgMrn}\agent\{agentId}	306	0.01
POST\oidc\api\org\{orgMrn}\logo	305	0.01
PUT\oidc\api\org\{orgMrn}\logo	305	0.01
POST\oidc\api\org\{orgMrn}\agent	302	0.01
GET\oidc\api\org\{orgMrn}\agent\{agentId}	302	0.01
GET\oidc\api\org\{orgMrn}\role\{roleId}	302	0.01
GET\oidc\api\org\{orgMrn}	298	0.01
GET\oidc\api\org\{orgMrn}\acting-on-behalf-of	298	0.01
GET\oidc\api\org\{orgMrn}\agents	298	0.01
GET\oidc\api\org\{orgMrn}\approve	298	0.01
GET\oidc\api\org\{orgMrn}\certificate\issue-new	298	0.01
GET\oidc\api\org\{orgMrn}\devices	298	0.01
GET\oidc\api\org\{orgMrn}\logo	298	0.01
GET\oidc\api\org\{orgMrn}\role\available-roles	298	0.01
GET\oidc\api\org\{orgMrn}\role\myroles	298	0.01
GET\oidc\api\org\{orgMrn}\roles	298	0.01
GET\oidc\api\org\{orgMrn}\services	298	0.01
GET\oidc\api\org\{orgMrn}\users	298	0.01
GET\oidc\api\org\{orgMrn}\vessels	298	0.01
POST\oidc\api\certificates\ocsp\{caAlias}	14	0.00
GET\oidc\api\certificates\crl\{caAlias}	8	0.00
GET\oidc\api\certificates\ocsp\{caAlias}**	7	0.00
GET\oidc\api\org\id\{orgId}	4	0.00

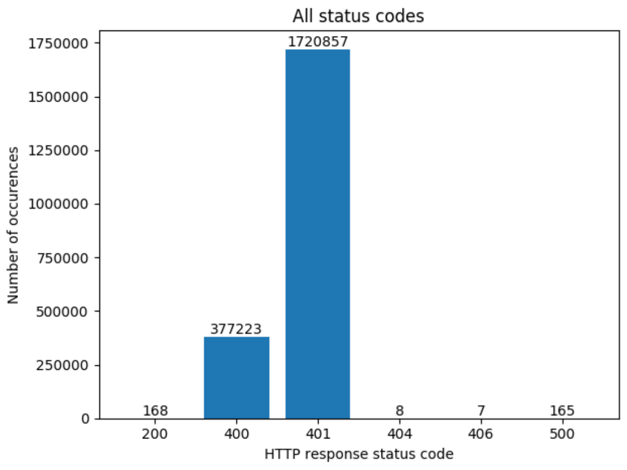
Table 6.2: Number of mutations in each template, sorted by most mutations.



(a) Run 1 (Auth certificates).



(b) Run 2 (Auth headers).



(c) Run 3 (No authorisation).

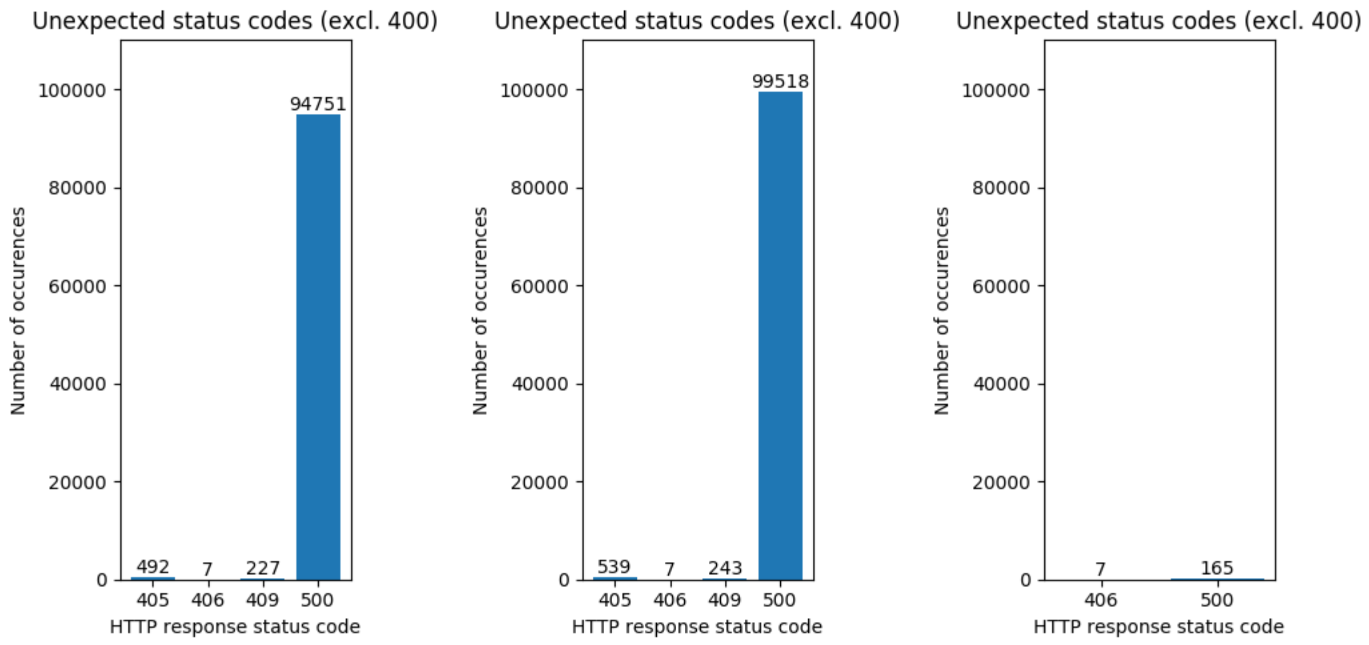
Figure 6.1: HTTP status codes for all three runs.

6.2.2 Unexpected HTTP status codes

The Swagger specification that the identity registry provides lists expected response codes for each method of each path. We consider an HTTP response code that was not in the list of expected status codes to be interesting, because it can be an indication of a problem. We have therefore gathered those responses.

Figure 6.2 shows unexpected response codes for all runs. We excluded `400 Bad Responses` from the plots because of their prevalence. The status code `500 Internal Server Error` in particular is interesting because it indicates that something went wrong during processing of a request.

It can be seen in this figure that by far, status code 500 was the most common "unexpected" status code, not including status code 400. Other than code 500, there are also occurrences of `405 Method Not Allowed`, `406 Not Acceptable` and `409 Conflict`. Those three status codes are not as interesting because they indicate that the request was rejected for a specific reason and probably did not trigger any bugs in the system. Status code 500 is more interesting because it can mean many different things depending on the included error messages (if present) in the response. We will take a closer look at some of the error messages seen in responses with status code 500 later.



(a) Run 1 (Auth certificates). (b) Run 2 (Auth headers). (c) Run 3 (No authorisation).

Figure 6.2: Unexpected HTTP status codes for all three runs (excluding `400 Bad Requests` because of their prevalence).

6.2.3 Unexpected HTTP status codes over time

The next figures, Figure 6.3 and 6.4 shows how the amount of unexpected responses develops over time. We will describe these further now.

The runs with authorisation. It can be seen in Figure 6.3 and 6.4 that the graphs for both runs with authorisation are roughly identical. It can also be seen that the graph rises linearly at some points but stalls at other times. This could indicate that some templates (paths) did not cause unexpected responses.

The run without authorisation. As we can see in Figure 6.5, all unexpected response codes were received in the first 200 minutes of the unauthenticated fuzzer run. This is because the path `/oidc/api/org/apply` is the fourth path in the Swagger specification, sorted alphabetically. This is the only path that does not require authentication and thus is the only one where the identity registry actually processes the request instead of returning a `401 Unauthorized`. For all three runs it can be seen that the number of unexpected responses and number of `500 Internal Server Error`s follow each other almost exactly.

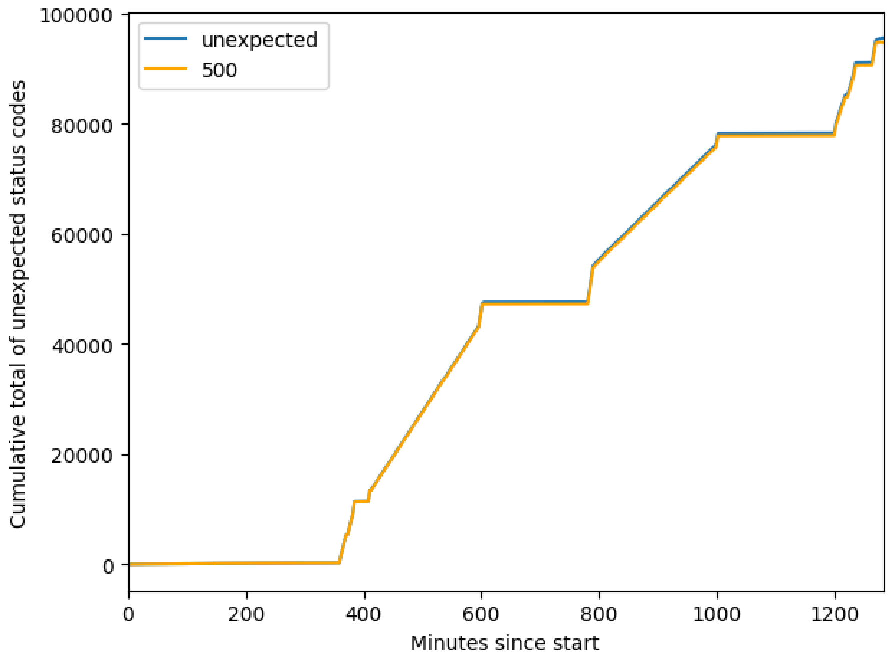


Figure 6.3: Unexpected responses over time – run 1 (Auth certificates).

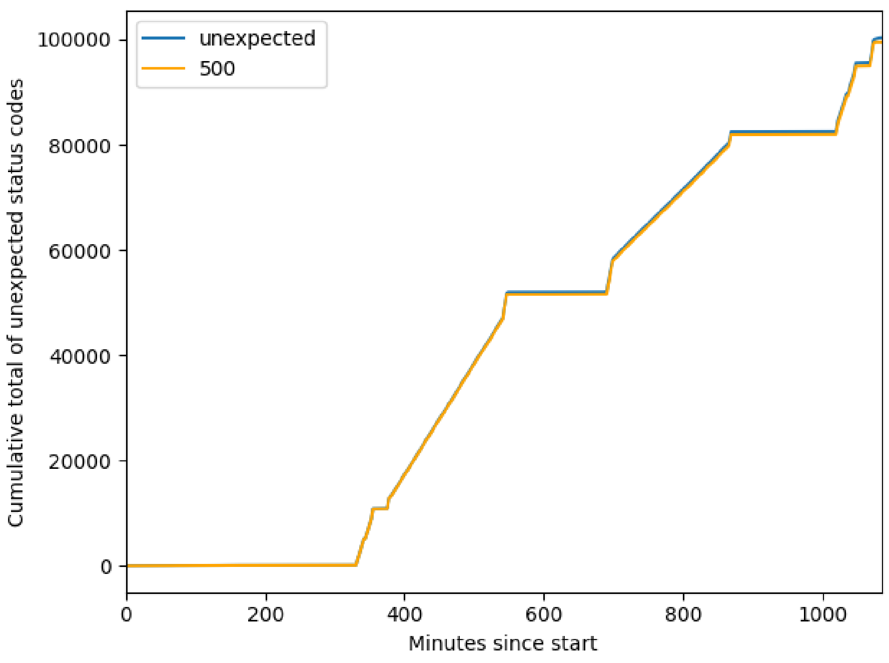


Figure 6.4: Unexpected responses over time – run 2 (Auth headers).

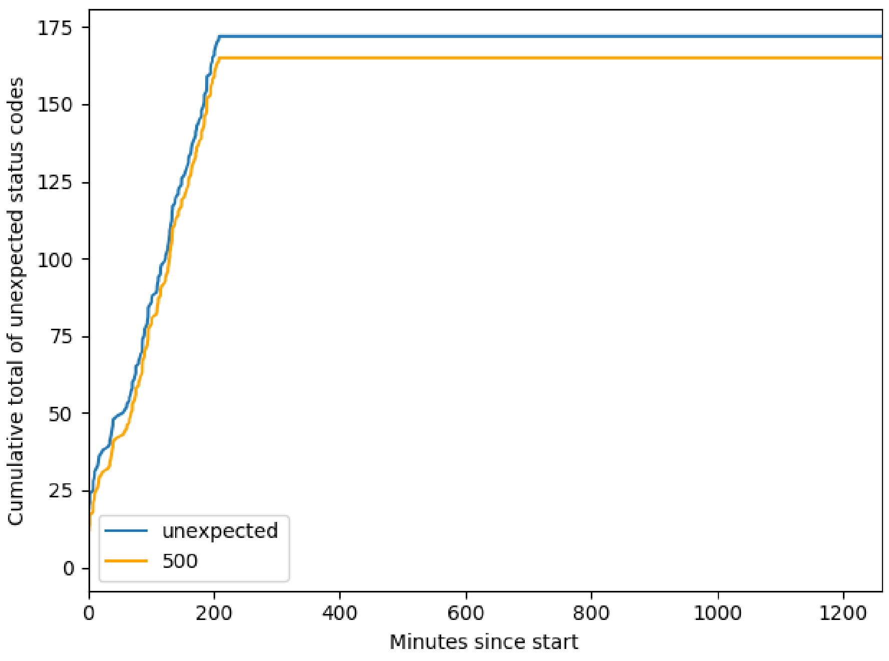


Figure 6.5: Unexpected responses over time – run 3 (No authorisation).

6.2.4 Templates causing most 500 Internal Server Error responses

The next figures (Figure 6.6, 6.7 and 6.8) show which templates caused most `500 Internal Server Error` responses. What can be seen here is that POST requests by far cause most of them. For the third run (the one without authorisation), it is also interesting that only two templates provoked 500 status codes. Again, this is explainable by the fact that most templates require authorisation to execute.

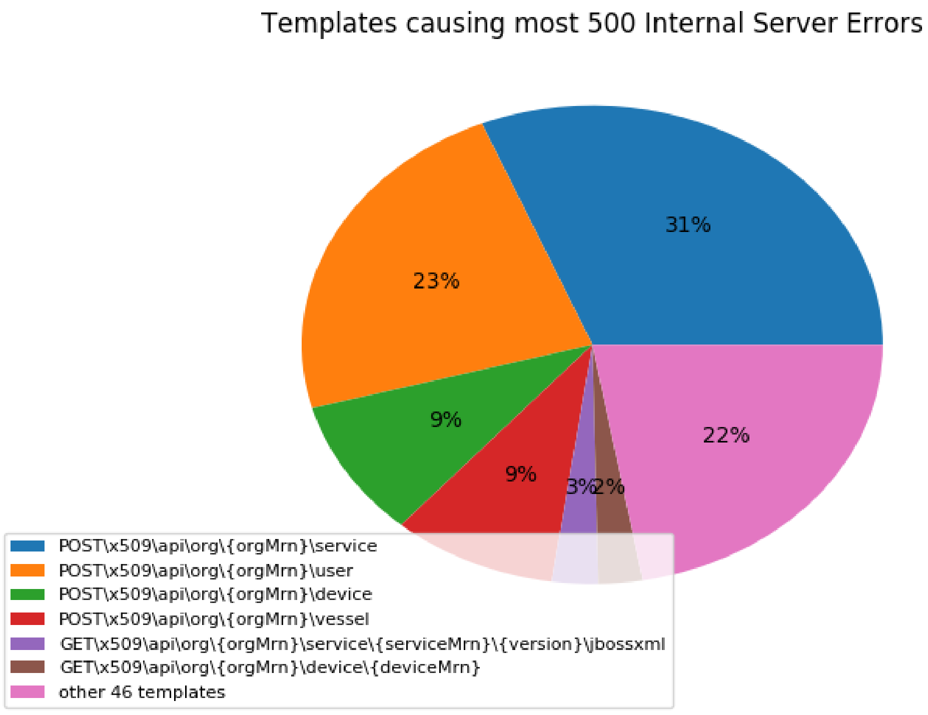


Figure 6.6: Templates causing most `500 Internal Server Errors` – run 1 (Auth certificates).

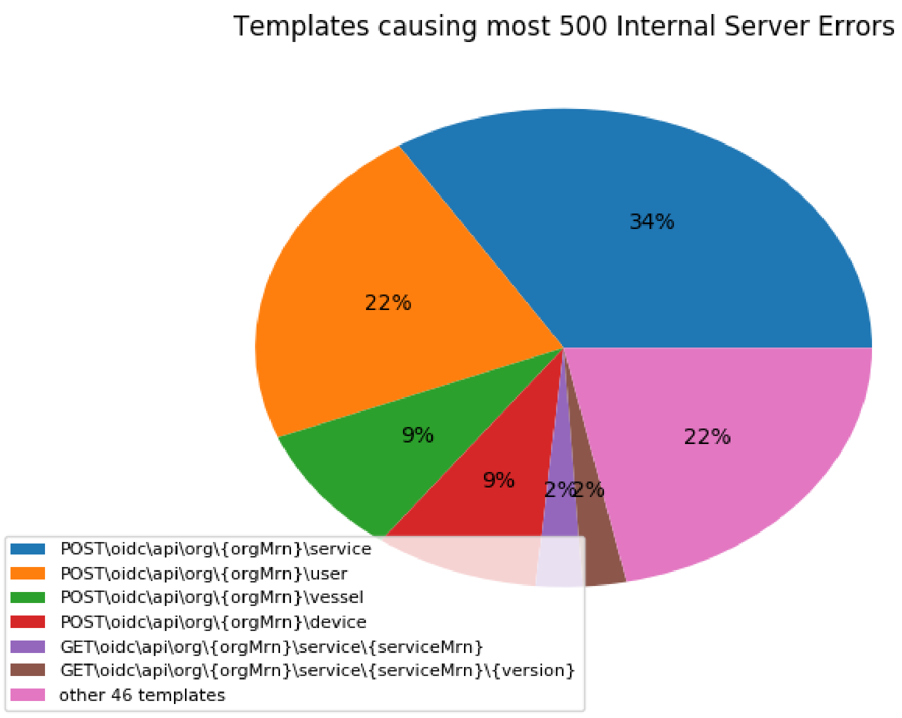


Figure 6.7: Templates causing most `500 Internal Server Errors` – run 2 (Auth headers).

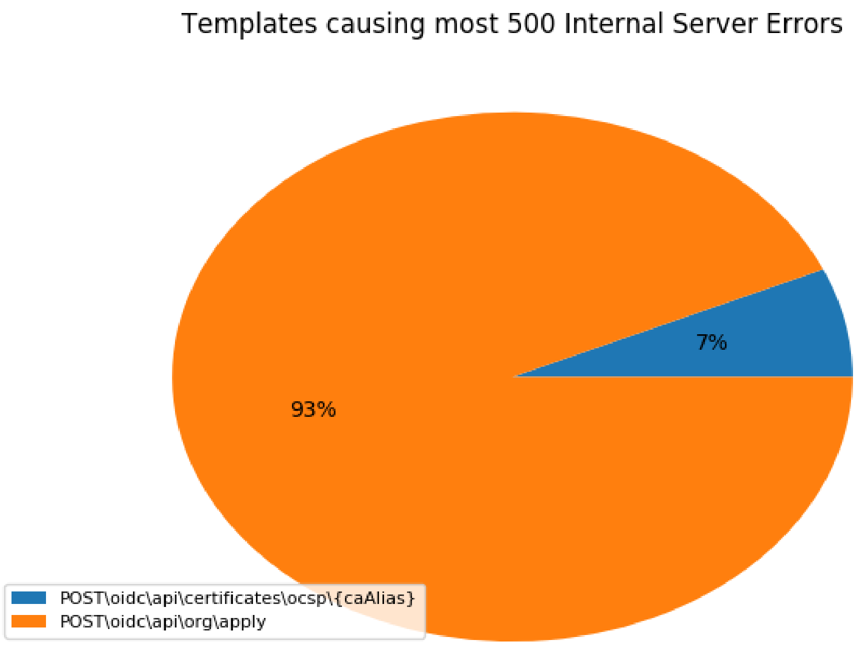


Figure 6.8: Templates causing most `500 Internal Server Errors` – run 3 (No auth).

6.2.5 Slow requests over time

We noticed during the fuzzer runs that sometimes, a request would hang and not receive a response for at least a second. For this reason, we created a plot that displays all "slow" requests as a function of time. Our definition of a "slow request" is one that takes at least a second between sending the request and receiving a response. The plot also shows each major template on the first axis, so that patterns more easily can be identified. Figure 6.9, 6.10, and 6.11 show these plots, one for each of our runs.

Note that not all template names have been shown on the plot due to space constraints. Instead, only the templates with the highest mutation counts are listed, because these take up most of the space on the graphs.

The plots show an interesting pattern: some templates seem to have a high concentration of requests taking longer than a second to process. An example is the first template in Figure 6.9, `POST/x509/api/org/apply`. At the same time, other templates did not cause any slow requests, for example the template `PUT/x509/api/org/{orgMrn}` in figure Figure 6.9 (the second template).

The runs with authorisation. Interestingly, the graphs for the two runs with authorisation (Figure 6.9 and 6.10) show similar patterns. The main difference between them is that the run that used authorisation headers (Figure 6.10) appears to have requests that took just around one second to complete scattered all over the duration of the run. It still has the same patterns as the previous run with some templates causing high concentrations of slow requests, typically between one and a half and two seconds.

Most likely, the first run with authorisation using certificates has the same pattern, but the slow requests scattered all over the run are not shown because they took less than one second. The difference in response times for the slow requests in these two runs is likely because we used a different DigitalOcean droplet for each of them (the one using authorisation headers required more RAM because it had to store more data, namely the authorisation headers).

The run without authorisation. The run that did not use authorisation, shown in Figure 6.11, mostly has occurrences of slow requests on the first big template: `POST/oidc/api/org/apply` (a request that does not require authorisation). As we explained earlier, this is because most paths in the API require authorisation, and thus, most requests are rejected immediately.

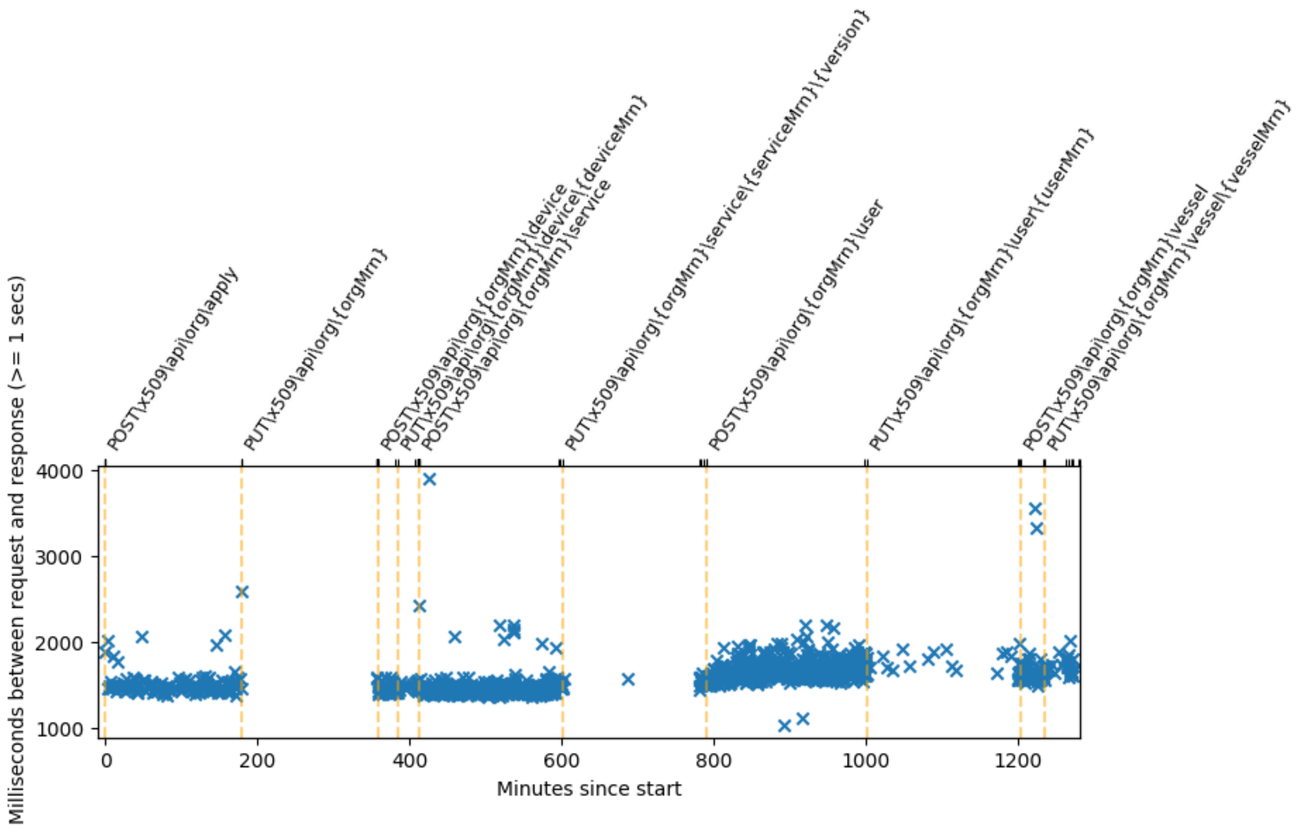


Figure 6.9: Slow requests over time – Run 1 (Auth certificates).

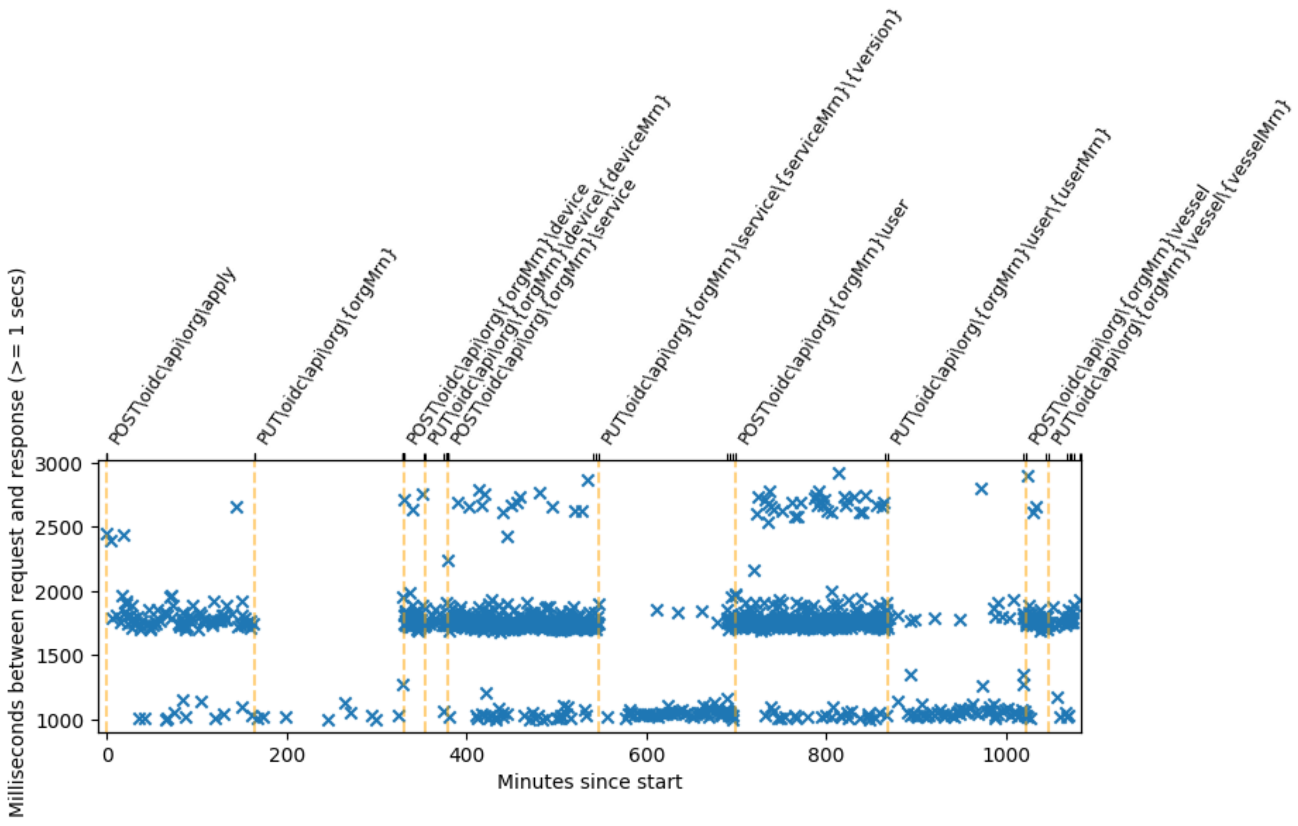


Figure 6.10: Slow requests over time – Run 2 (Auth headers).

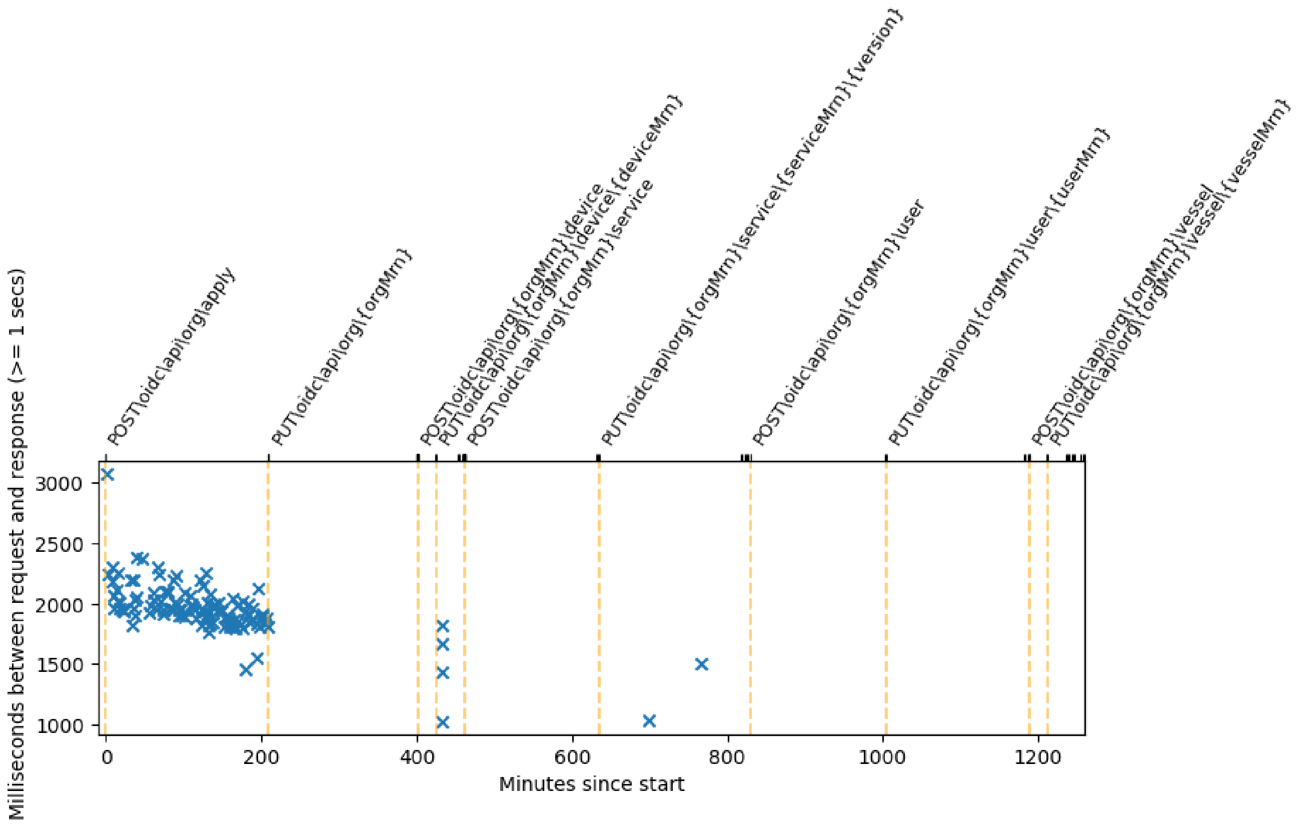
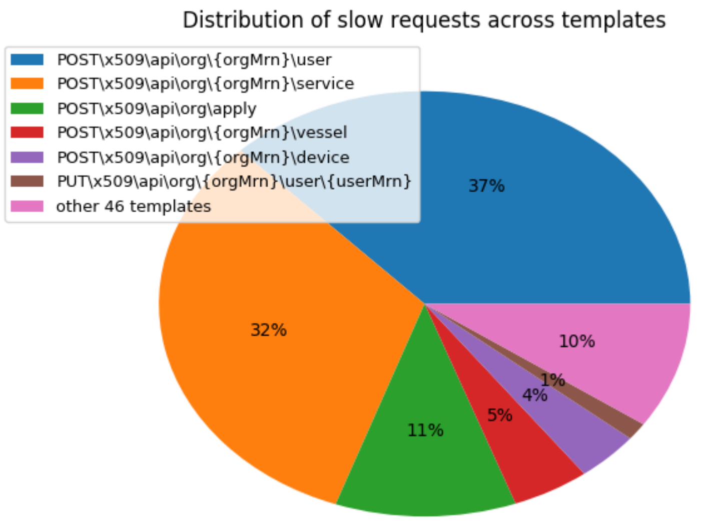


Figure 6.11: Slow requests over time – Run 3 (No authorisation).

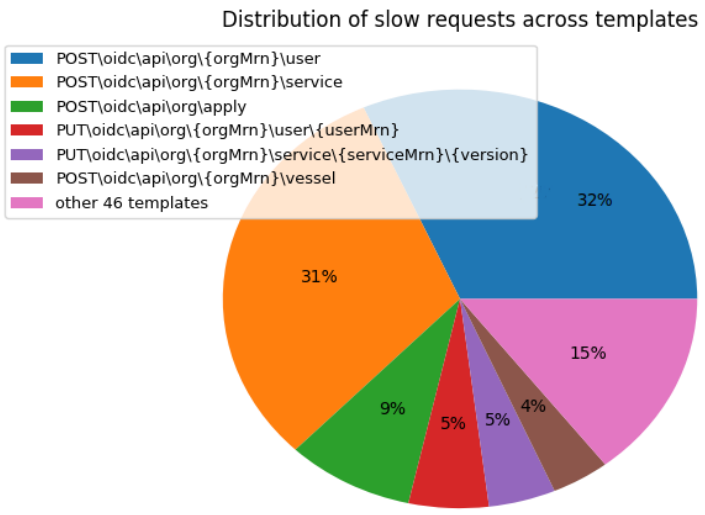
6.2.6 Slow requests distributed among templates

The next figure, Figure 6.12, shows which templates resulted in most slow requests. The figures for both runs with authorisation are somewhat similar: the top three templates remain the same, accounting in total for 80 percent in the run with certificate authentication and 72 percent in the run with authorisation tokens. Note that not all templates that caused slow requests are listed, because the slices in the pie would become very small, and it is sufficient to know which ones caused the most slow requests.

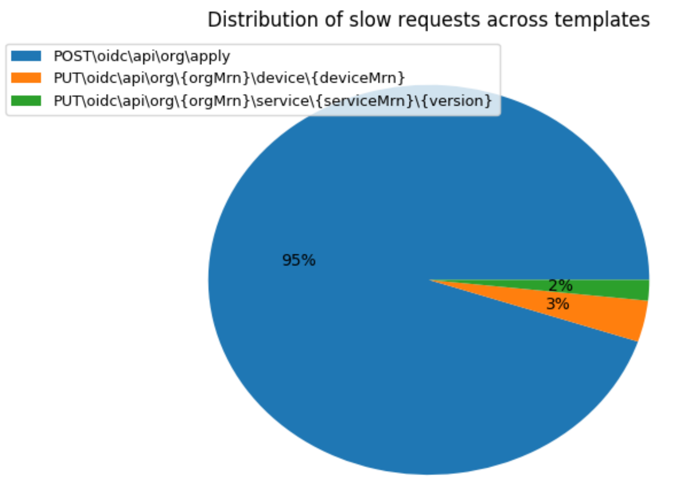
For the run without authorisation, only three templates caused slow requests. This can also be seen on the time plot in the previous graph (Figure 6.11). The template `POST/oidc/api/org/apply` caused 95 % of slow requests. This is explainable by the fact that the `org/apply` operation is one of the only requests in the API that does not require authorisation – anyone can apply to create a new organisation. On the other hand, it is interesting that the two other templates using the HTTP PUT method caused some slow requests. These two templates try to modify an existing device or service for a particular organisation, so they should not be processed without authorisation. Yet, they did cause some slow requests which is interesting.



(a) Run 1 (Auth certificates).



(b) Run 2 (Auth headers).



(c) Run 3 (No authorisation).

Figure 6.12: Slow requests distribution, part two.

6.2.7 500 Internal Server Error – error messages

We have already shown that a significant amount of requests resulted in `500 Internal Server Errors`. This particular HTTP status code is interesting for our purposes, for reasons mentioned earlier. We will now list the different kinds of error messages that were included with the HTTP responses that had the status code 500.

SQL error message. The most interesting error message, from a security point of view, is the following:

```
could not execute statement; SQL [n/a]; nested exception is
org.hibernate.exception.DataException: could not execute statement
```

This error message is interesting because it indicates that something went wrong in connection with some SQL query. SQL errors are interesting because they sometimes indicate the possibility of SQL injection. However, whether this is the case here is not certain. In any case, from a pure user experience standpoint, a REST API should not return this kind of technical and vague error message.

We will now look closer at which requests generated this SQL error message. The exact same error message occurred a number of times in the different runs:

- Run 1 (Authorisation using certificates): 7 occurrences
- Run 2 (Authorisation using headers): 10 occurrences
- Run 3 (No authorisation): 0 occurrences

Interestingly, all requests that caused this error message used the exact same request path and method:

```
POST /x509/api/org/urn:mrn:mcl:org:dma/role
```

It can be seen from this request URL that it operates on the existing organization that was created in the lab environment – the organisation with MRN value `urn:mrn:mcl:org:dma`. The request attempts to create a new role within that organisation.

This particular request also uses a JSON body consisting of two keys: `permission` and `roleName`. In all cases where this error message occurred, the `permission` value was set to a long string consisting of 750 consecutive `A`'s.

Here is an example of a request body that caused this error message (750 `A`'s):

```
{"permission":"AAA(...)AAA","roleName":"ROLE_ENTITY_ADMIN"}
```

In every occurrence of this error message, the value for the key `permission` was identical to the one above – only the value of `roleName` changed. Since this error message only occurred on long values of `permission`, this might hint at some string length check not being properly done in the identity registry before the command is processed.

Validation failed error message. The second most interesting error message is one such as this:

```
"Validation failed for classes
[net.maritimecloud.identityregistry.model.database.Role] during persist time
for groups [javax.validation.groups.Default, ]\nList of constraint
violations:[\n\tConstraintViolationImpl{interpolatedMessage='The value
'420000000' is not in the predefined list of accepted values!',
propertyPath=roleName, rootBeanClass=class
net.maritimecloud.identityregistry.model.database.Role, messageTemplate='The
value '420000000' is not in the predefined list of accepted values!'}\n]"
```

Messages such as the one above (with differing values) occurred a total of 40 times in the second run that used authorisation headers. The message is interesting because it appears to be one that was automatically generated by Java. It is doubtful that it hints at a security issue, but in any case, it is not good practice for API requests to return technical gibberish. So at the very least, this message indicates that some checks are not present in the identity registry which checks for invalid input and gives an easy-to-read error message, as it does in many other cases.

Other error messages. The top three most common error messages accompanying a `500 Internal Server Error` response are the following:

- `MRN is not a valid entity MRN!`: 99206 occurrences
- `javax.ws.rs.BadRequestException: HTTP 400 Bad Request`: 21 occurrences
- `Could not parse mail; nested exception is javax.mail.internet .AddressException: Domain contains illegal character in string ''example@gmail'com''`: 20 occurrences

Most other error messages relate to invalid values in POST/PUT requests, for example: `Could not parse mail ...`, `Malformed escape pair at index ...`, `badly formatted directory string`. Another interesting one is `'An error occured while trying to create user of the Organization on shared Identity Provider!'`, although it is unclear what exactly it implies in terms of security.

We have now described the main results of our three fuzzing experiments. In the next chapter, we will discuss those results as well as other subjects in this thesis.

Discussion

7

In this chapter, we will shed some light on the discussions and problematic situations we came across and solutions and/or compromises to these. We start by discussing our results from fuzz testing in Section 7.1. Next, we discuss our fuzzing strategy and important implementation details in Section 7.2. Finally, in Section 7.3 we discuss our virtual lab environment for MCP.

7.1 Fuzzing results

Overall, as documented in the previous chapter, we could not pinpoint any specific security bugs in the MCP Identity Registry with our fuzz testing. We did find two noteworthy results which are not demonstrable security issues, but have the potential of being so:

Slow requests: We found that some requests would take at least a second before receiving a response. While it looks like certain paths and methods result in more slow requests that others, we are not certain why this happens. Still, this could enable an attacker to overload the identity registry with such requests and possibly cause a Denial of Service (DoS).

More work needs to be done in order to find the exact reasons for those slow requests, and to determine whether they can be used to conduct a (D)DoS attack.

Requests causing SQL error messages: We found that a specific request caused `500 Internal Server Error` responses with this error message:

```
could not execute statement; SQL [n/a]; nested exception is
org.hibernate.exception.DataException: could not execute statement
```

This SQL error message only occurred when using a long string in the JSON body of the request. We have not determined whether this particular request is vulnerable to SQL injections. Error messages that mention SQL in some way can sometimes indicate an SQL injection vulnerability, which is why we found this interesting. More work needs to be done in order to ascertain whether this poses a security risk.

These two findings require more investigation in order to ascertain whether they are in fact security issues. For now, we can say that the fact that our fuzzer enabled us to get those results is a success because our fuzzer was able to provoke unusual behaviour in the identity registry. Yet, we cannot know for sure whether more security issues exist. Other fuzzing or testing methods might give different results, which we will discuss later.

7.2 Implementation details

In this section, we discuss choices we made when implementing our fuzzer as well as limitations and alternatives that should be considered.

7.2.1 Fuzzing strategy

When we implemented our REST API fuzzer (RESTFuzzer), we had simplicity and extensibility in mind. We wanted to create a framework which could be used to create REST API fuzzers in general, not just for the MCP Identity Registry. The one thing these different REST APIs have in common is an OpenAPI specification which details all possible requests and their parameters in the API. We thus needed a general way of taking such a specification and turning it into a list of Kitty templates which are then fuzzed by Kitty.

Our approach. For simplicity, our approach is to parse one request path and method at a time, one by one. This means that every single path-method pair is fuzzed as a complete unit, independent of any other path-method pair in the specification. This has its benefits and drawbacks. The benefits are the following:

- It is straight-forward to parse the specification and create a list of templates.
- Each path is fuzzed independently and extensively.
- The approach should be able to find rudimentary bugs (for example, insufficient input validation) in most APIs.

The disadvantages are the following:

- It will not find bugs that only occur after specific combinations of requests (for example, a bug might only happen after first creating a new object with a POST request and then modifying it right after with a PUT request). Because our approach is not aware of any IDs of newly created data, our fuzzer will in most cases not be able to send a PUT request to modify existing data.
- A large amount of requests will be ineffective because valid data is not supplied. This was reflected in our results, where we had a large amount of `400 Bad Request`s. In general, many GET, PUT and DELETE requests take as input an ID of the object that is being changed. With this approach, that ID will be a fuzzable entity and will in most cases not be an existing ID in the identity registry. The result is that the registry rejects the request because the ID was non-existent.

Other approaches. Our relatively simple approach cannot capture every possible security issue in an API and thus we have also looked into other fuzzing strategies as well. In particular, the feedback-driven "intelligent" fuzzer called RESTler, proposed by Atlidakis et al [19], shows promising results. Their method greatly reduces the number of 400 Bad Requests in an intelligent fashion by analysing dynamic feedback from responses continuously. Their approach is mainly different from ours in that they use a dynamic approach whereas our approach is static (our fuzzer does not alter its behaviour based on feedback).

Furthermore, RESTler focuses on fuzzing meaningful sequences of requests that operate on the same objects. This is in contrast to our approach which tests all request paths independently. We would like to have used RESTler on the MCP Identity Registry, but the source code was not available and the authors could not share it with us at the time. For that reason, we created our own fuzzer.

Number of mutations per template. In our implementation, small GET requests with only one fuzzable parameter might only get four mutations. To balance out mutation counts across all templates, it might be a good idea to test more mutations on those "simple" templates so that at least a hundred different values are tried. That might prove useful and is very little extra work compared to other templates which might take up hundreds of thousands of mutations.

In general though, we find it reasonable that our fuzzer gives templates with more fuzzable fields more mutations. Naturally, a POST request that creates some new object with several fuzzable properties should have more mutations than some GET request with just one fuzzable parameter.

Optional parameters. In OpenAPI specifications, parameters in requests can be in a list of "required" parameters. If a parameter is not in the list of required parameters, it is considered to be optional. In our implementation, we only consider required parameters. Naturally, this prevents our fuzzer from finding bugs that only would occur when also fuzzing these optional parameters. We decided not to consider optional parameters because it would make the implementation a lot more complex. However, for completeness, these optional parameters should be considered for future work.

7.2.2 Other testing methods

Other methods than fuzzing should be considered for testing the security of the MCP Identity Registry. In particular, load testing is a possibility. With load testing, one might spawn multiple clients/threads that all send requests to the identity registry continuously. To increase impact, the requests could be POST requests that create new data in the registry. Such a test can expose systems that scale badly with number of users. Since the identity registry in the long run is meant to be widespread among many users, we consider this kind of test to be very relevant.

7.3 Lab complexity

As mentioned in Section 4.1 our lab environment is not optimal for production. With Docker, however, it is completely possible to set up a test environment that could also be used in production. But, in order to avoid long debugging sessions, culminating from large configuration and build files, we agreed that limiting the complexity was the way to go. The first compromise we made was to leave only the identity registry behind a reverse HTTPS proxy, leaving the Keycloak instance as is. The reasoning behind this is that setting up the identity registry and the reverse HTTPS proxy with certificates and everything was quite tricky, causing issues that sometimes left us clueless as to what was wrong. Also, the identity registry was the main focus so we followed the recommended setup for that and left the Keycloak in a functional, but perhaps insecure state.

Another aspect of keeping the complexity in check was to use images that did not have to be built locally but could be pulled from the Docker Hub. This is because each local Docker image requires its own Dockerfile along with resources to build the image. Using ready-made images leaves us with less maintenance.

In the next chapter, we conclude our work in this thesis.

Conclusion 8

In this chapter, we conclude on the results of our work. In the section below is an overview of our conclusions and in Section 8.2 we give suggestions for future work. Section 8.3 contains a short summary of the main contributions of this thesis.

8.1 Conclusions

In Chapter 3, we provided a security assessment of the MCP Identity Registry based on a formal and structured process. Using a risk register, we have documented which areas have the highest priorities with regard to implementing and maintaining the best security. Notably, we found that several of the threats and vulnerabilities with a risk level of "high" or above could be discoverable using a fuzz tester. Thus, based on this analysis we concluded that a fuzz tester would be a beneficial tool to accompany the MCP Identity Registry.

In Chapter 4, we successfully constructed a virtual lab environment in which the MCP Identity Registry can run and be tested upon. As intended, the environment is easy to set up, use, and maintain.

We described how we created a fuzz tester in Chapter 5 as well as how our solution, RESTFuzzer, can be used to fuzz test REST APIs with the purpose of finding security vulnerabilities or bugs.

In Chapter 6 we described the experiments that we conducted using our fuzz tester and presented some interesting correlations between the URI of a request and the duration required to process it. The correlation could be a sign of a potential DDoS vulnerability. Additionally, some response codes came along with error messages that could indicate SQL injection vulnerabilities. Similarly some error messages indicated that there were scenarios that were not handled properly in the code, whether a bug or not. It is possible that these findings may lead the developers on track to avert such vulnerabilities before it is put into production. Additionally, this tool, although still having room for improvement, can be used in the future to continuously evaluate the security of both the identity registry as well as other REST APIs.

8.2 Future work

We have concluded our work and commented on the results we have obtained but as always, there is the possibility of improvement. Hence, we have proposed areas where the work could be continued or where our compromises could be turned into full-fledged solutions instead.

8.2.1 Analysis

As mentioned before, the security risk assessment is a continuous process that must be iterated upon to keep up with the changes as development of MCP carries on and with technology in general. Ideally, the assessment will be used by the developers to try to mitigate or eliminate risks that the identity registry is exposed to.

8.2.2 Lab setup

As we discussed in Section 7.3, the virtual lab environment could, and perhaps *should*, be enhanced to mimic what a production environment would look like. Running the security tests in this environment instead would also prevent the discovery of bugs that would only be present in a test environment. To match the structure in Figure 4.3, a reverse proxy should be assigned to the Keycloak instance in addition to the already present one for the identity registry. The networks should be segmented further to limit how many services are on the same network. This way, only encrypted traffic is allowed to the outside world and services become more isolated, preventing direct access between some of them.

8.2.3 Implementation

The implementation of our fuzzer can be extended with functionality to mimic what has been achieved by the authors of [19]. That is, to intelligently create requests that make more sense than blindly testing all possibilities. In addition to that, the following items should be addressed in future work.

Parsing OpenAPI specifications. There exist some special cases in OpenAPI specifications which are not parsed correctly in our implementation, if at all:

- Rendering of array values inside query parameter strings (inside the URL) needs to be done in a special way. For example, a specific key can contain multiple values in a URL. Example of correct usage: `example.com/get?status=0&status=1&status=2`. In our implementation, that example would render as `example.com/get?status=[0,1,2]` which is incorrect in most cases.
- Some string types in a specification can be defined by a custom regex. Our implementation ignores the regex and instead uses our default `JSONString` type.

The above examples are not a problem when fuzzing the MCP Identity Registry, but can become relevant with other APIs. Specifically, the above issues were discovered when using our RESTFuzzer on an example API called Petstore provided by Swagger.[17]

[17] `https://petstore.swagger.io/`

Testing other APIs. The implementation of RESTFuzzer should be tested on other APIs than the MCP Identity Registry that provide OpenAPI specifications. This is to verify that it parses other specifications correctly and handles all cases elegantly. This task might prove difficult because there is a large amount of functionality with varying complexity supported by OpenAPI specifications.

Verifying that the fuzzer can find bugs. The effectiveness of our fuzzer framework should be verified. This can be done by taking an existing API, for example the Petstore API provided by Swagger, and introduce faults (for example, SQL injection vulnerabilities) deliberately and see if our fuzzer would trigger the inserted faults, perhaps by looking for special `500 Internal Server Errors`.

Implementing other types of tests. As mentioned in Chapter 7, there are other fuzzing methods (such as the feedback-driven method used by [19]) and other kinds of tests that are relevant to security testing. These could be implemented and taken into account for more thorough testing in the future.

8.3 Summary

All in all, we consider our risk assessment, as well as the presented fuzz tester and the results it produced, to be useful for the developers of MCP to increase the security of the system. The virtual lab environment that we designed will make it easier to deploy the fuzz tester in a controlled environment and generate consistent results.

In conclusion, the main contributions of our work are the following:

- We have provided a detailed risk assessment of the MCP Identity Registry.
- We have provided a virtual lab environment using Docker in which the MCP Identity Registry can be run for testing purposes.
- We have created a fuzzer framework, RESTFuzzer, that can be used to create fuzzers for REST APIs. It uses the Kitty fuzzing framework as the backend. We used this framework to build a fuzzer for the identity registry.
- Using our fuzzer, we found some possible issues in the identity registry which should be investigated further.

References

[1] Greenberg, Andy. The Untold story of NotPetya, The Most Devastating Cyberattack in History. *Wired*, August 2018. urlhttps://www.wired.com/story/notpetya-cyberattack-ukraine-russia-code-crashed-the-world/.

[2] Fruhlinger, Josh. What is Stuxnet, who created it and how does it work? *CSO Online*, August 2017. `https://www.csoonline.com/article/3218104/what-is-stuxnet-who-created-it-and-how-does-it-work.html` accessed August 28, 2019.

[3] Liljedahl, John Henning and Ventegodt, Ole and Dietrich, Ove W. Søfart, i Den Store Danske. `http://denstoredanske.dk/index.php?sideId=168804`, December 2014. Accessed March 12, 2019.

[4] Danish Shipping. Facts and Figures. `https://www.danishshipping.dk/en/flash-analyser/download/Publications_Model_Publication/31/danish-shipping-facts-and-figures-juni-2018.pdf`, June 2018. Accessed March 14, 2019.

[5] Danish Shipping. Facts and Figures. `https://www.danishshipping.dk/en/flash-analyser/download/Publications_Model_Publication/27/danish-shipping-facts-and-figures-web.pdf`, December 2017. Accessed March 14, 2019.

[6] Wagner, I. The world's leading container ship operators as of August 26, 2019, based on number of owned and chartered ships. https://www.statista.com/statistics/197643/total-number-of-ships-of-worldwide-leading-container-ship-operators-in-2011/, August 2019. Accessed August 30, 2019.

[7] PwC. The Digital Transformation of Shipping. `https://www.pwc.no/no/publikasjoner/shipping/The-Digital-Transformation-of-Shipping_HE-NO.pdf`, 2017. Accessed May 2, 2019.

[8] Danish Defence Intelligence Service. Threat Assessment: The Cyber Threat Against the Danish Maritime Sector, 1st version. `https://fe-ddis.dk/cfcs/publikationer/Documents/Cyber_threat_against_danish_maritime_sector.pdf`, January 2019. Accessed March 14, 2019.

[9] Danish Maritime Authority. Cyber and Information Security Strategy for the Maritime Sector. `https://www.dma.dk/Documents/Publikationer/Cyber%20and%20Information%20Security%20Strategy%20for%20the%20Maritime%20Sector.pdf`, 2019. Accessed March 14, 2019.

[10] The Maritime Cloud Development Forum. Maritime Cloud conceptual model. `https://maritimeconnectivity.net/docs/IALA%20Input%20-%20Maritime%20Cloud%20conceptual%20model.pdf`, 2017. Accessed May 16, 2019.

[11] EfficienSea2. Maritime Connectivity Platform. `https://efficiensea2.org/solution/maritime-connectivity-platform/`, 2019. Accessed August 14, 2019.

[12] EfficienSea2. MCP Identity Platform. `https://developers.maritimeconnectivity.net/identity/index.html`, January 2019. Accessed August 26, 2019.

[13] Swagger. What is OpenAPI? `https://swagger.io/docs/specification/about/`, 2019. Accessed August 15, 2019.

[14] Sutton, Michael and Greene, Adam and Amini, Pedram. *Fuzzing: Brute Force Vulnerability Discovery.* Pearson Education, 2007.

[15] Stallings, William and Brown, Lawrie. *Computer Security: Principles and Practice.* Pearson Education, Fourth (global) edition, 2018.

[16] OWASP Foundation. OWASP Top 10 - 2017: The Ten Most Critical Web Application Security Risks. `https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf`, 2017. Accessed August 22, 2019.

[17] Newman, Lily Hay. Github Survived the Biggest DDoS Attack Ever Recorded. *Wired*, March 2018. `https://www.wired.com/story/github-ddos-memcached/` accessed August 27, 2019.

[18] Meyer, David. Nokia: Yes, we decrypt your HTTPS data, but don't worry about it. January 2013. `https://gigaom.com/2013/01/10/nokia-yes-we-decrypt-your-https-data-but-dont-worry-about-it/` accessed September 1, 2019.

[19] Atlidakis, Vaggelis and Godefroid, Patrice and Polishchuk, Marina. REST-ler: Automatic Intelligent REST API Fuzzing. *arXiv preprint arXiv:1806.09739*, 2018.

Appendix A

A1 Setup on a DigitalOcean VPS

We tested both the setup of Docker and running the code on a DigitalOcean "standard" 1 vCPU, 3 GB droplet. It ran Ubuntu 19.04 64-bit with kernel 5.0.0-20-generic. The following steps will start the MCP Identity Registry and required services as well as start the fuzzer. All steps are assumed to be run as root. It can be a good idea to start tmux first, as we will need to start the Docker containers and run the program simultaneously.

Edit `/etc/hosts` and the Kitty source code as described in Section 4.3 and Section 5.4, respectively. Run the following commands to set everything up:

```
apt update
apt upgrade -y
apt install -y git docker docker-compose python3-pip tmux
pip3 install kittyfuzzer

git clone https://github.com/satvug/thesis.git

reboot

cd thesis/docker

docker-compose up
```

It takes a bit of time for `docker-compose up` to finish. You'll know it has finished when the last message says "`docker_keycloak_startup_1 exited with code 0`". We can now, in a different window, run the fuzzer:

```
cd ~/thesis/src
python3 idreg_fuzzer.py
```

A2 Example of Swagger path and schema entries

Listing A2.1: Example of Swagger path entries

```
"paths": {
  .
  .
  .
  "/oidc/api/org/apply": {
    "post": {
      "tags": [
        "organization-controller"
      ],
      "summary": "applyOrganization",
      "operationId": "applyOrganizationUsingPOST",
      "consumes": [
        "application/json"
      ],
      "produces": [
        "application/json;charset=UTF-8"
      ],
      "parameters": [
        {
          "in": "body",
          "name": "input",
          "description": "input",
          "required": true,
          "schema": {
            "$ref": "#/definitions/Organization"
          }
        }
      ],
      "responses": {
        "200": {
          "description": "OK",
          "schema": {
            "$ref": "#/definitions/Organization"
          }
        },
        "201": {
          "description": "Created"
        },
        "401": {
          "description": "Unauthorized"
        },
        "403": {
          "description": "Forbidden"
        },
        "404": {
          "description": "Not Found"
          }
        }
      }
    },
    .
    .
    .
```

Listing A2.2: Example of Swagger schema entries

```
  "definitions": {
    .
    .
    .
    "Organization": {
      "type": "object",
      "required": [
        "address",
        "country",
        "email",
        "mrn",
        "name",
        "url"
      ],
      "properties": {
        "address": {
          "type": "string"
        },
        "certificates": {
          "type": "array",
          "description": "Cannot be created/updated by
              editing in the model. Use the dedicate
            create and revoke calls.",
          "items": {
            "$ref": "#/definitions/Certificate"
          }
        },
        "country": {
          "type": "string"
        },
        "createdAt": {
          "type": "string",
          "format": "date-time"
        },
        "email": {
          "type": "string"
        },
        "federationType": {
          "type": "string",
          "description": "Type of identity federation used by organization",
          "readOnly": true,
          "enum": [
            "test-idp",
            "own-idp",
            "external-idp"
          ]
        },
        "id": {
          "type": "integer",
          "format": "int64"
        },
        "identityProviderAttributes": {
          "type": "array",
          "items": {
            "$ref": "#/definitions/IdentityProviderAttribute"
          }
        },
        "mrn": {
          "type": "string",
          "description": "The Maritime Resource Name"
        },
        "name": {
          "type": "string",
          "description": "The name of the organization"
        },
        "updatedAt": {
          "type": "string",
          "format": "date-time"
        },
        "url": {
          "type": "string"
        }
      },
      "title": "Organization"
    },
    .
    .
    .
```