NTNU

Norwegian University of
Science and Technology

Specification-based security analysis of REST APIs

Petter Iversen

Master of Science in Informatics
Submission date: June 2018
Supervisor: Jingyue Li, IDI
Co-supervisor: Edvard Karlsen, Kantega
Bjarte Østvold, Norwegian Computing Center

Norwegian University of Science and Technology
Department of Computer Science

I want to dedicate a special thanks to my two external supervisors, Bjarte M. Østvold and Edvard K. Karlsen, as well as my internal supervisor at NTNU, Jingyue Li —without your excellent guidance, commitment and encouragement, this thesis could not have been the same.

Problem description

In January 2014, news outlets reported that the usernames and passwords of 4.6 million Snapchat users had leaked, when intruders exploited critical security holes in Snapchat's "non-public" back-end API. While such catastrophic incidents are uncommon, security bugs that could allow critical data leaks are regularly discovered in all kinds of web APIs. Most such bugs are simple to fix once discovered, and also possibly to spot given basic knowledge of web security. However, lack of knowledge or failure to allocate developer resources is a challenge.

Many web APIs are described using machine-readable modelling languages such as the OpenAPI Specification (formerly the Swagger Specification) and RESTful API Modeling Language. A specification written in such a language should be an interesting starting point for conducting automated security testing of an API.

The goal of this project is to design and evaluate techniques for conducting such testing, emphasising low developer effort techniques that can be deployed by most organisations.

Abstract

In the modern Internet era, web applications are typically driven by web services (WS). Web services are accessible on the Internet through their application programming interfaces (APIs). Due to the continuous exposure on the Internet, and being accessible for anyone, security testing is an increasingly important part of serious software development. Manual security testing is, however, an expensive and time-consuming activity.

Automated security analyses that do not require developers to specify individual test cases could reduce the entry barrier to get developers started with security testing. It would also help avoid large upfront costs for the development teams.

In this thesis, I introduce a set of such automated security analyses, a set of Representational State Transfer (REST) related security testing techniques, a minimalist API modelling language that the analyses use to generate test cases, and finally a proof-of-concept tool that implements and validates all of my other contributions.

An important focus in the thesis has been to keep programmer effort modest, i.e. limiting the required programmer input and required security related knowledge to the minimum sensible level, while still being able to find relevant and security crucial vulnerabilities in real-world applications.

Sammendrag

I den moderne Internettæra er web applikasjoner typisk drevet av webtjenester (WS). Webtjenester er tilgjengelige på Internett gjennom deres programmeringsgrensesnitt (API). På grunn av deres evige eksponering på Internett, samt det faktum at de er åpent tilgjengelige, har sikkerhetstesting blitt en viktigere og viktigere del av seriøs programvareutvikling. Manuell sikkerhetstesting er dog en dyr og tidskrevende aktivitet.

Automatiserte sikkerhetsanalyser hvor utviklere ikke spesifiserer individuelle testtilfeller kan redusere inngangsbarrieren til sikkerhetstesting. Det vil også kunne hjelpe utviklerteam å unngå store forhåndskostnader knyttet til sikkerhetstesting.

I denne oppgaven introduserer jeg et sett av slike automatiske sikkerhetsanalyser, et sett med Representational State Transfer (REST)-relaterte sikkerhetstestingsteknikker, et minimalistisk modelleringsspråk for APIer, samt et enkelt verkty som implementerer og validerer alle mine andre bidrag.

Et viktig fokus gjennom hele oppgaven har vært å holde programmererens innsats beskjeden, altså å minimere krav til input og sikkerhetsrelaterte forkunnskaper til et minimalt fornuftig nivå, men fortsatt klare å finne relevante og sikkerhetskritiske svakheter fra den virkelige verden.

Table of Contents

Chapter 1

Introduction

Manual security testing of web service APIs is an expensive and time-intensive, though necessary and important part of serious software development. Despite security being of utmost importance, it might be neglected for various reasons. Even when security is a priority for developers, it is possible that there exist one or more security vulnerabilities in a system. Automating parts of the security testing process allows software development teams to integrate automated security testing as a part of their automated testing process, reducing the chance of introducing testable security related software vulnerabilities in concurrent software builds.

As modern web services are designed with interoperability in mind, and due to the fact that practically every industry is moving to the Internet, web servers are more exposed to attackers than ever before. Large companies such as Facebook (2 billion monthly active users), Spotify (70 million monthly active, paying, users) and YouTube (1,5 billion monthly users) all have open, well documented, Representational State Transfer (REST) based APIs [1, 2, 3]. This increases the need for focus on security as all the different endpoints, inputs and authentication logic is publicly available for any attacker to look at.

According to recent reports security vulnerabilities are up 30% for Q1 2017 compared to Q1 2016 [4]. One could argue that the increased adoption of modern web services, Software as a Service (SaaS) and public APIs has played a role in these numbers. Modern web architectures, such as REST, come with their own caveat of security issues—and it is very easy for developers without formal security training to introduce security vulnerabilities. Because REST is so well defined, it is easy for attackers to look for vulnerabilities, as they do not need to know the implementation of the system in order to attack it through the REST facade.

With small and medium businesses adapting the same technologies, they might not have the resources available to provide security on the same level as the technological giants—

making them vulnerable to attacks. After all, software security is a matter of effort versus reward and if the effort is low the reward need not be as high.

1.1 Introductory example

The following example requires some knowledge of HTTP and REST. Readers which are unfamiliar with these concepts are suggested to take a look at section 2.1 before returning to the example.

Consider an HTTP GET request with the following query parameters:

```
userId=123
accessToken=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

Developers with security knowledge should be alerted when seeing such a request. What happens if the user ID parameter is changed? Is it possible to simply change a URL query parameter in order to retrieve another user's data? Is it possible to read any sensitive data?

This example might seem trivial, and that is because it is. This example is actually showcasing a vulnerability discovered in T-Mobile's API, exposing sensitive data for million of T-Mobile customers. Software security vulnerabilities rarely only occur once, and it turns out that this kind of security vulnerability is not unique to T-Mobile. A similar vulnerability was recently used to hack Tinder through the Facebook product Account Kit, as described by Prakash in his online article [5]. The gist of the vulnerability, as described by Prakash, is that Account Kit did not verify that the provided query parameters matched the actual target values, which allowed attackers to specify their own phone number to gain control over the victim's account. Both of these vulnerabilities have been patched since their discovery.

I will return to the T-Mobile data breach example in section 7.3.1, where I use this example to validate parts of my contributions.

1.2 Audience

The thesis introduces a set of techniques and analyses which can be used in order to conduct automated as well as manual security testing of web services. An important background for these techniques and analyses is the understanding of fundamental REST and hypertext transfer protocol (HTTP), as well as knowledge regarding authentication and authorisation—here in the form of JSON web token (JWT). Chapter 2 should provide the reader with sufficient knowledge in these areas.

The thesis is particularly useful for anyone who is developing REST based web services without being trained or educated within software security. The resulting analyses can be used by software developers who want to ensure that common security flaws are noticed

as soon as they are introduced—and ensure that when a security issue is found it will not be repeated.

That being said, the vulnerabilities my contributed analyses are able to find are well known to security professionals. This means that security professionals should not expect to learn about new security vulnerabilities. The automated model based approach should, however, be of interest to security professionals, as it allows for interesting regression testing approaches with regards to security testing, as well as ways of conducting automated security testing without specifying test cases.

1.3 Research hypothesis

Software development professionals like patterns. Software is designed, structured and implemented using a plethora of patterns, both at the code, architectural and infrastructural levels. RESTful[1] web services are prime examples of the usage of standards to develop uniform APIs for a web services. Though undeniably useful in the form of maintainability, discoverability and other constraints Fielding defines REST to emphasise [6], this comes with the heightened risk of introducing known security vulnerabilities into the code base, if precautions are not taken. Throughout this thesis I will be laying the foundation to ultimately argue for the following research hypothesis:

Automated specification-based security analysis can be used to find critical security vulnerabilities in large real-world applications, while keeping programmer effort modest.

Modern web services consists of a set of *specifications* and *constraints*. Whether these specifications and constraints are explicitly formulated e.g. through an official specification, or only exists as a set of requirements, they should be testable. A goal of this thesis is to find some of these testable properties and suggest a minimalist, formal, specification as well as lightweight automated test suite for them.

1.3.1 Research questions

In order to answer the research hypothesis, I will inspect the following research questions. While studying the research questions, my philosophy will be to keep programmer effort modest, as well as finding “low hanging fruits”, i.e. commonly seen security vulnerabilities that are easy to fix (and exploit) once discovered.

1. Is it possible to statically analyse a REST API specification in order to find common REST related security vulnerabilities?
2. Is it possible to use the same API specification to generate a set of dynamic tests that finds vulnerabilities in the system’s runtime?

[1] `http://www.oracle.com/technetwork/articles/javase/index-137171.html`

3. Is it possible to automatically test for common JWT vulnerabilities?

All of these research questions will be investigated, and should be found true, in the context of the final statement of the research hypothesis—keeping programmer effort modest.

1.4 Contributions

The paper *Writing Good Software Engineering Research Papers* by Mary Shaw [7] proposes many guidelines as how to best present research related to software engineering. According to Shaw, the results of software engineering related research typically falls into one or more out of seven distinct categories. The two categories relevant for my contributions are defined by Shaw as:

Qualitative or descriptive model: *Structure or taxonomy for a problem area; architectural style, framework, or design pattern; non-formal domain analysis, well-grounded checklists, well-argued informal generalizations, guidance for integrating other results, well-organized interesting observations.*

Tool or notation: *Implemented tool that embodies a technique; formal language to support a technique or model (should have a calculus, semantics, or other basis for computing or doing inference).*

As such, I will categorize the main contributions of this thesis, and rank them in order from highest to lowest priority:

1. A descriptive model—In chapter 3, I gather and describe some commonly seen security vulnerabilities. The analyses and techniques I then proceed to introduce in chapter 4 and 5 are used to discover these vulnerability patterns in an automated manner. The descriptive model consist of:

   (a) An analytic model—Two specification based analyses. The analyses are motivated by well known security vulnerabilities, which are listed in OWASP Top 10 [8] and the OWASP REST Cheat Sheet [9]. The analyses consist of both *dynamic analyses*, i.e. analyses that are executed upon a system at runtime, and *static analyses*, i.e. analyses that do not depend on a system runtime to discover security vulnerabilities.

   (b) A set of techniques—In chapters 4 and 5 I introduce several techniques which utilises RESTful constraints and best practices to conduct automated web security testing.

2. A notation—The *API model* which is introduced in chapter 4 is a light weight REST API specification language with focus on web service security properties.

3. A tool—In order to validate the findings in this thesis I have developed a small,

Kotlin[2] based, tool that implements the analyses and techniques and is able to find or suggest the various vulnerability patterns I discuss. The implementation of this tool is described in chapter 6 and used to validate my contributions in chapter 7.

[2] https://kotlinlang.org/

Chapter 2

Background

The aim of this chapter is to cover the background knowledge required in order to better understand the contributions of the thesis. The most important concepts described here are, in order:

1. REST and the HTTP in general, section 2.1.
2. General web service related software security concepts, sections 2.2 and 2.3.
3. Concepts related to authentication and authorisation, section 2.5.
4. JWT, section 2.6.

2.1 Representational State Transfer (REST)

The concept of REST was introduced by Roy Thomas Fielding in his doctoral dissertation in 2001 [6]. In the dissertation, Fielding lays out a set of constraints which suggests to utilize the HTTP protocol as originally intended.

REST is the most important aspect to grasp in order to extract the essence of what this thesis is offering, thus I will dedicate a fair share of explanation to this concept and how it relates to, and can be utilised to enhance, software security.

2.1.1 Key concepts

The following sections describes basic key concepts related to REST and HTTP.

URIs and URLs

While the difference between a Unique Resource Identifier (URI) and a Unique Resource Locator (URL) is subtle, I think it is worth pointing out. URIs are a superset of URLs. For a URI to qualify as a URL, it needs to provide information regarding what protocol is used to access the resource. Consider the following URIs:

Table 2.1: Sample URIs

Cleartext	URI	URL
https://api.myapp.com/users	Yes	Yes
ftp://api.myapp.com/users	Yes	Yes
api.myapp.com/users	Yes	**No**

As this thesis is focusing on HTTP-based REST APIs, I will explicitly be using URLs to define access to resources. I will also be using relative URIs in order to not have to explicitly list the entire URL every time I am referring to a *resource*. This means that the URL `https://api.chatapp.com/users/:userId`[1] and the URI `/users/:userId` should be treated equally within the context of this thesis. Make note of the preceding slash ('/'), identifying our relative URI.

REST Resources

Richardson and Ruby [10] defines a REST resource as anything important enough for the user of your API to care about. Here are some examples of resources and proposed URIs:

- A user: `/users/:userId`
- A list of users: `/users`
- A user's contacts: `/users/:userId/contacts`

REST resources are always accessed through URIs.

HTTP status codes

Many HTTP status codes have been defined in order to properly express various response statuses a HTTP-request might induce[2]. HTTP status codes consist of three digits and fall into one out of five categories, each category defined by the first digit of the response code. The five categories are:

1. 1xx Informational

[1] Here ':userId' is a URI parameter. URI parameters are explained later in this chapter in the section **URI parameter**

[2] A full list of the official HTTP status codes can be found in RFC 7231 [11].

2. 2xx Success
3. 3xx Redirect
4. 4xx Client error
5. 5xx Server error

Note that different systems might use the different response codes for the same purpose—there is no explicit standard that enforces strict adherence. Facebook, for example, sends a 200 response code to any request, as discussed by S. Marx at Dropbox [12]. This is apparently to make reverse engineering harder.

By using HTTP verbs, REST and HTTP status codes as originally intended, the task of conducting automatic security testing becomes a lot easier, both conceptually and practically.

HTTP methods

While there are eight official HTTP methods [11], API tools like Postman[3] allow users to select from a list of 15. This is a common theme in the field of HTTP; while there are defined standards, people are able to implement their own version of e.g. HTTP methods or HTTP response codes. I am going to be focusing on just two out of the eight official ones, namely the GET and POST method.

The HTTP GET method indicates that the client wants to retrieve whatever resource is identified by the request-URI, while the POST method indicates that the client wants to create a new resource located within the collection[4] defined by the request URI.

Query parameter

The *query* part of a URI is the part after the first question mark ('?'), terminated by a number sign ('#'), as defined per RFC 3986 [13]. Query parameters are key-value pairs contained within the query part, each pair separated by an ampersand ('&'). Consider the URI `/users?username=Petter&age=24`. This URI contains two query parameters, *username* and *age*. The value of the username parameter is Petter, and the value of the age parameter is 24. We are able to pass as many variables as we want to through query parameters. We do not need to terminate the query part when it is at the end of the URI.

Query parameters can be used for anything, but should be limited to query related issues, such as pagination, ordering or filtering of the query.

[3] Postman is an HTTP inspection tool that allows developers to send HTTP requests and specify and inspect properties of the request such as headers and request bodies. Available at https://www.getpostman.com/

[4] A collection in this context is typically just a list of resources, e.g. all the users in the system or a given user's chat rooms.

URI parameter

Some variables are more relevant than others. URI parameters are variables which are part of the URI itself. Say we want to retrieve a user identified with the id 42, we could do so through the URI `/users/42`.

In order to denote a variable URI parameter, I use the convention of appending a colon (':') to the variable name. For the user ID example, the previous URI becomes `/users/:userId` when utilizing the variable naming convention.

It is fully acceptable to chain URI parameters, e.g. `/users/:userId/chats/:chatId`, which should look up a chat resource in the context of a user resource. It is also possible to combine the usage of URI and query parameters, e.g. `/users/:userId/chats?minimumChatParticipants=3`—which should return all chats for the given user where the participant count is greater than or equal to three.

URI vs Query parameters

Note that there is no absolute requirements as to how the URIs are composed. The URI `/users/42` and the URI `/users?userId=42` could both link to the same resource, depending on the implementation. It is, however, common to think of REST resources as catalogues, and thus the concept of chaining a collection and a collection entry as shown in the URI `/users/:userId` will be used in the context of this thesis.

2.1.2 Examination of a sample REST API

REST has become a de facto standard for modern web API architecture. In fact, over 80% of ProgrammableWeb's APIs are categorized as REST APIs [14].

Even though Fielding's dissertation introducing REST [6] proposes a strict set of constraints, most REST APIs do not follow these adherently. There is, however, typically many commonalities between REST APIs. An important, and widely adopted, aspect of most REST APIs is to group and traverse data by chaining collections. Another one is to use the HTTP verbs as they were initially indented to be used—to describe *actions*. Consider the following example where the HTTP method POST is executed on the URI `/users`, rather than the HTTP GET method on the URI `/users?action=create`.

Table 2.2 lists several actions to access and mutate data in an imaginary API. In addition to information contained in the paths shown in the table, it is important to note that REST APIs typically operate with a request and response body. It is within the body of a request the essence of a REST operation lies, either be it storing a POST request's body to the database, or retrieving a resource from a GET's response body. Let us examine some of the rows:

ID	Method	Path	Result
M1	POST	/users	A user is created
M2	POST	/login	Retrieve a token that can be used at later requests in order to identify the authenticated user.
M3	GET	/users?username=u	Search the user collection by username
M4	POST	/chats	Create a new chat
M5	POST	/chats/:id/users	Invite a user or a list of users to a chat
M6	POST	/chats/:id/owners	Add a new user to the owner list of the chat
M7	GET	/users/:userId	Get a collection containing the chats of the given user

Table 2.2: REST API example

M1

M1 is a standard POST method, without any attributes to its path. Sending a POST request to the */users* path will create a new user, given that the request body matches the constraints[5] on the server.

M2

As REST in no way defines how to deal with authentication, the action of signing in has no good RESTful representation—as REST is *resource*-based and not *action*-based. A common way of handling login in a REST context is to create an endpoint where a client can POST a user's credentials in exchange for an access token or a session id, which is then attached to subsequent requests.

M3

M3 allows us to search the user collection. We do so by providing a *query parameter*, which the server will use to filter the response to match the query. The exact implementation of the search may vary from API to API, some will match exactly while others will match on a regular expression. The important thing to make not of here is that query parameters should only be used to execute a query, and not pass additional required information such as access tokens to the server.

[5] By constraints I mean business rules, such as an email field containing a valid email address that is not already used within the system.

2.2 Web service security

Although a fair understanding of REST is required to understand the context of the analysis described in this thesis, web service security is the main focus of the thesis. I will use this section to introduce security related concepts relevant to this thesis.

2.2.1 The Open Web Application Security Project (OWASP)

"*The Open Web Application Security Project (OWASP) is a worldwide not-for-profit charitable organisation focused on improving the security of software*" [15]. OWASP's top ten project [8] describes the top ten most critical web application vulnerabilities. The project team includes a variety of security experts from around the world. OWASP will be used as a source in order to determine which security bugs to focus on in this thesis.

OWASP Cheat Sheets

The OWASP Cheat Sheet Series was created to provide a concise collection of high value information on specific application security topics [16]. In particular, the REST Security Cheat Sheet [9] will be an important factor when deciding which bugs to focus on in this thesis.

2.2.2 Common Weakness Enumeration (CWE)

CWE is, according to their about page [17], a formal list of known software vulnerabilities. The purpose of the initiative is to serve as a common language for software vulnerabilities, serve as a frame of reference for security testing tools and to become a standard for prevention and mitigation of such vulnerabilities.

In the CWE registry[6] developers are able to perform a search on weaknesses by keyword. Each vulnerability is documented with information such as a description, a list of applicable platforms (or deemed platform independent), list of common consequences, demonstrative examples and more.

2.2.3 Cross-site request forgery (CSRF)

A CSRF weakness arises when the API is relying solely on cookies stored in the browser to authorize requests. Attackers could be able to exploit this vulnerability by creating a malicious site, which they send the target user to. The malicious site will, without the user's knowledge, send a malicious request to the exposed API, altering data without

[6] `https://cwe.mitre.org/`

the user's knowledge. CSRF vulnerabilities[7] are typically mitigated through the usage of CSRF-tokens.

2.3 Software security

In order to understand why the security testing approach selected for my contributions was chosen, the reader must familiarise themselves with some basic software security related concepts. This section will introduce those relevant concepts and terms.

2.3.1 Black-box testing

Black-box testing is not a concept that is unique to software security. In general, black-box testing is a testing approach in which the tester does not have any insight into the inner workings of an application.

With regards to software security, the general definition is still sound. A typical scenario in security testing is to have a penetration tester attempt to hack the system as a black box, i.e. without any knowledge of the inner workings of the system.

2.3.2 Hacking

A security hacker is someone who seeks to breach defenses and exploit vulnerabilities in a system or network. There are many possible motivations for a hacker to do this, including:

- Economical gain
- As a challenge, to learn or prove themselves
- To find and help patch vulnerabilities

Hackers without malicious intents, e.g. to find vulnerabilities in their own system or on behalf of the system creators, are typically referred to *white hat hackers*. Hackers with clear malicious intents are referred to as *black hat hackers*, and hackers which does not hack on the behalf of personal gains, but also not in agreement with the system creators are referred to as grey hat hackers.

2.3.3 Cryptographic hashing

Hashing is the process of mapping data of arbitrary length into data of fixed length. Stinson describes cryptographic hashing in chapter 4 of his book; *Cryptography: Theory and*

[7] `https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet`

Practice, third edition [18, p.119-155]. Cryptographic hashing adds some constraints to a regular hashing function:

1. **Deterministic**. The same input always results in the same hash.
2. **Collision free**. It should be very hard, ideally impossible, to find two different inputs that map to the same output.
3. **One-way function**. Given the output, it should be very hard, ideally impossible, to compute the input.

2.3.4 Dictionary attack

In software security, a *dictionary attack* is a hacking strategy that is commonly used to brute force a solution to a cryptographic problem, e.g. to find a signing key or a user's password. By storing a large dictionary with commonly used passwords hashed, it is possible to reverse engineer a hash into a password, if the password was hashed without a salt[8]. Password cracking is not a focus of this thesis, but the term dictionary attack will be used for analyses which iterates over a predefined set of known malpractices, in order to look for security vulnerabilities.

2.3.5 The seven security touchpoints

Building secure software is hard, and fixing insecure software is expensive. In order to reduce the chance of introducing software security vulnerabilities while developing a system, McGraw defines seven software security touch points, as depicted in figure 2.1. The figure shows how each of the security touch points relates to various software artifacts. In their following descriptions, the software security touchpoints are listed as McGraw rank their importance [19].

1. Static analysis and code review
2. Risk analysis
3. Penetration testing
4. Security testing
5. Abuse cases
6. Security requirements
7. Security operations

[8] A salt is, in the context of cryptography, some random data which is added as additional input to a one-way function. The purpose of the salt is to ensure that two equal inputs never hashes to the same value.

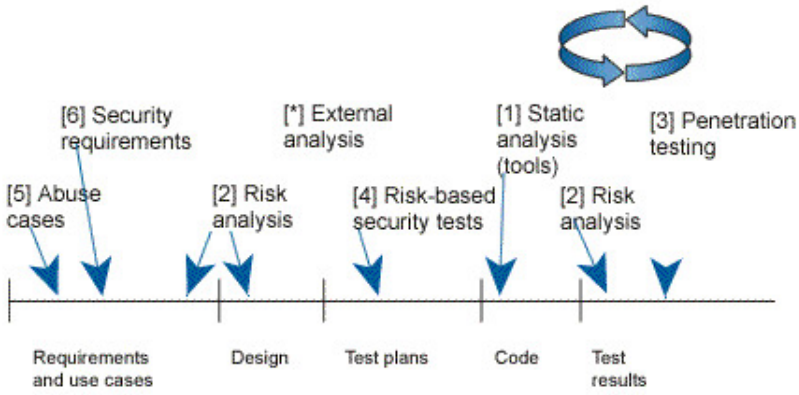

Figure 2.1: Software security touch points

Bonus touchpoint

Finally, McGraw mentions a bonus touchpoint which he describes the following way: "*External analysis (outside the design team) is often a necessity when it comes to security. Software security touchpoints are best applied by people not involved in the original design and implementation of the system.*" [19].

2.4 Web services

If you search the Internet for a definition of the term web service, you will find many different definitions. I will summarise what I think is the most important commonly mentioned features in order to define what I mean when using the term web service in this thesis in the following paragraph.

A web service provides a standardised way for software applications to communicate with each other over the internet, e.g. over HTTP or HTTPS. What separates web services from other Internet based applications is that web services focus on application-to-application communication, rather than application-to-human communication. A common example of this is that a web service does not focus on providing data in a human readable format, such as HTML, but rather focuses on computer readable formats such as JSON or XML.

In the earlier days of the Internet, the various layers of a web application would typically be very tightly coupled—websites would be rendered on the server and the data used to render the web pages would not be available through other means than actually parsing the

HTML—and this was fine (although, one could argue, a bit *smelly*[9])! Figure 2.2 shows a very simplistic web service serving static web pages.

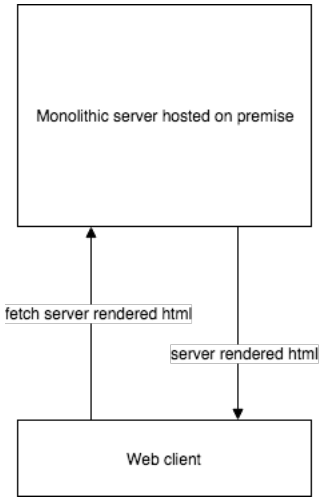

Figure 2.2: Old monolithic web application

[9]Martin Fowler defines a *code smell* as a surface indication that there is a deeper problem in the system [20].

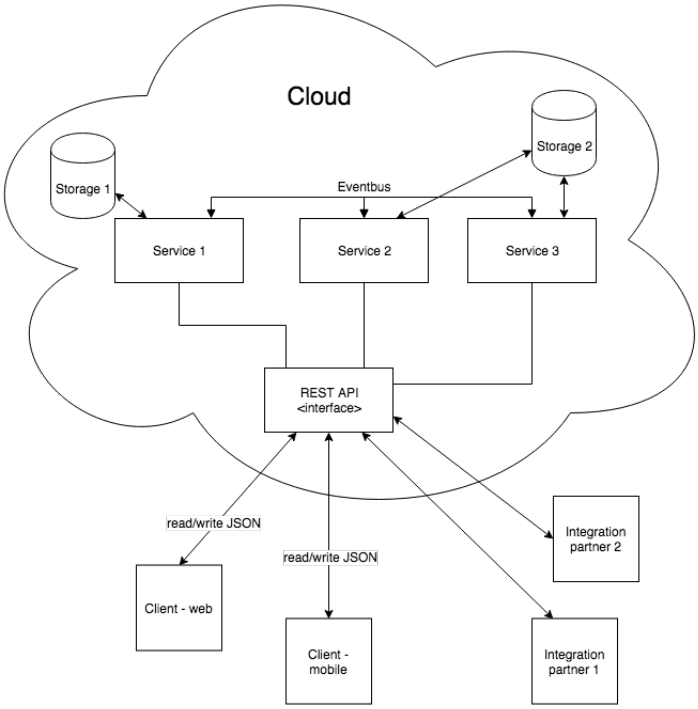

Figure 2.3: Modern web service

As technologies developed and the need to access data used in web sites through other means—such as mobile applications or through integration—increased, fundamental architectural changes as to how one built web applications had to be made. As application to application communication became a larger and more prominent business requirement, standards for this had to be developed. It was no longer feasible to only expose the application functionality through HTML, as it made application to application communication very tedious and too complex. REST, which I discussed in section 2.1, is an example of an architecture which focuses on the structure of the data, and not how the data is displayed. We call such applications web services[10]. Figure 2.3 shows a modern approach to building web services. The service is running in a cloud system, and exposes all functionality through a REST API. Not only is all functionality exposed, it is out there in the open well documented, so that anyone can integrate with the service at any time, which adds strict security requirements.

[10]SOAP and RPC are examples of other service oriented architectures

2.5 Authentication and authorisation

Though a subsection of the software security umbrella, authentication and authorisation are so important concepts to this thesis that I dedicate a separate section to explain them.

2.5.1 Authentication and web services

With the transition to web services, and REST architectures in particular, an increasing amount of the *state* of an application is moved to the client-side. This includes keeping track of whether or not a user is signed in. Ideally, following RESTful principles, the web service stores no state—this includes login state. Removing state from our web services also comes with an added benefit of increased scalability. If there is no state on the service, we are able to spin up another instance of the service without having to care about synchronising state between our replicated services. The following sections describes the two most popular ways to conduct authentication and authorisation today.

2.5.2 Session-based authentication

session-based authentication is an umbrella term describing implementations that let the server identify a user over multiple HTTP requests[11]. The way sessions typically are implemented is by storing relevant information to a user's session in a database, and providing the client with a session ID that correlates to this information. When a client signs in, it receives the session ID, which is attached to each subsequent request, typically through the usage of cookies. A problem with this approach is that it often is tied closely to the browser implementation, making it hard or sometimes impossible to use sessions for mobile applications without a custom implementation. Another issue with session-based authentication is that the server relies on the database in order to validate sessions—which in turn can make it challenging to handle authentication in large scale distributed servers.

2.5.3 Token-based authentication

token-based authentication attempts to deal with some of the issues seen in session-based authentication. token-based authentication moves the responsibility of keeping track of session related state to the client side. As token-based authentication standards typically utilizes the HTTP *Authentication* header, they are also well suited for usage on mobile platforms—as this is not a feature tied to browsers[12]. With asymmetric signing, token-based authentication works very well with distributed systems and micro services, where session-based authentication systems would have to replicate the session state among every

[11] As HTTP itself is stateless, measures need to be taken to maintain login state, so the user does not need to sign in for every request.

[12] Whereas sessions typically use cookies, which does not work the same way for native mobile applications as in browsers.

instance in the distributed system. During the last few years token-based authentication has gained much traction due to the aforementioned features. token-based authentication aims for a stateless approach to authentication. This goes hand in hand with the stateless constraint of REST.

2.6 JSON Web Token (JWT)

JWT is an open industry standard for representing claims securely between two parties. The JWT spec is defined in RFC7519 [21]. In order to get a better understanding of what a JWT is, let's examine a JWT:

Header	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
Payload	eyJpZCI6IjEyMyIsIm5hbWUiOiJQZXR0ZXIiLCJleHAiOjE1OTA5NjI0MDAwMDAsImFkbWluIjp0cnVlfQ.
Signature	CAk9y_vzGlZ4hlMlRK8on1Qz9Jm_XuFGgZyU3ZWn_r0

All JWTs consists of three parts, each part Base64 encoded and separated by a dot.

2.6.1 Header

The first part of the JWT is the *header*. The header of a JWT typically[13] consists of two parts: the *type* of token and the *hashing algorithm* used to sign the token. It is worth noting that JWTs can be unsigned, i.e. the header alg field set to *none*.

```
1 {
2   "alg": "HS256",
3   "typ": "JWT"
4 }
```

Listing 2.1: Header decoded

There are many online tools[14] and libraries one can use when working with JWTs. Here the header of the sample JWT is decoded, and we can see that the *alg* and *typ* fields are set to *HS256* and *JWT* respectively. The HS256 algorithm is a synchronous hashing algorithm, and our token type is JWT. Alternatives to the JWT types are JSON Webtoken Encrypted (JWE) or JSON Web Token Secure (JWS), but they are not a focus in this thesis.

2.6.2 Payload

The second part of the JWT, the *payload*, consist of a set of claims. Each claim contains some information, typically related to the user which the token is valid for. If we decode

[13] As the JWT standard is very lenient, JWTs can be implemented in a myriad of ways and abide the JWT spec.
[14] `https://jwt.io/#debugger` is an excellent tool for online JWT inspection and debugging.

the payload of our example token, we get the following:

```
{
  "id": "123",
  "name": "Petter",
  "exp": 1590962400000,
  "admin": true
}
```

Listing 2.2: Payload decoded

Note that the payload of a JWT is just encoded plain text. While JWTs are good at persisting integrity, they do not enforce confidentiality out of the box. Developers should think twice before passing sensitive information in JWTs—one might mistake the Base64 encoding for encryption.

2.6.3 Signature

The final piece of the JWT puzzle is the signature. This is the part that ensures the JWT's integrity. When the JWT is created, the header and the payload are Base64-encoded separately, concatenated with a dot('.') as separation and hashed with a cryptographic hash function using a secret key. If an asymmetric hashing algorithm is used, anyone with the public key is able to verify the token's integrity, i.e. that the token has not been tampered with after the provider signed it. Only authorities with knowledge of the secret key is able to issue new tokens. When using symmetric hashing, only authorities with knowledge of the secret key is able to verify the token's integrity and sign new tokens.

In order to verify the JWT, the process conducted during signing is repeated. If the new signature matches the signature provided with the token, the content of the JWT is verified and we can be sure that the token has not been tampered with (except, of course, if the private key is compromised).

2.6.4 Registered claims

The JWT RFC presents a list of registered claims [21]. These claims are by no means necessary, but rather a suggestive set of useful interoperable claims. The analysis implementation in this thesis requires that the JWT under test is using the following registered claims:

- The *exp*, or expiration time, claim identifies the time that a JWT must no longer be accepted.
- The *iat*, or issued at, claim identifies the time that a JWT was issued.

2.6.5 Tying it all together

Here I demonstrate how we can tie this together, and authenticate and authorise requests. Consider the pseudo code in listing 2.3.

```
1  SIGNING_SECRET = 'secret';
2
3  verifyIsAuthenticatedAndAdmin(request, response):
4    token = request.headers['Authorization']
5    if token is None:
6      // missing authorization header for authorized endpoint
7      response.status(401).send()
8    else
9      validatedToken = jwtLibrary.validate(token, SIGNING_SECRET)
10     validatedUserType = validatedToken.userType
11     if validatedUserType is UserType.ADMIN:
12       // We ensure user has admin rights without doing a DB lookup
13     else if validatedUserType is UserType.USER
14       response.status(403).send()
15 }
```

Listing 2.3: Verifying a user's admin status by examining a JWT

Notice the weak signing secret defined in the first line of listing 2.3. Signing secrets should **always** be randomly generated cryptographically secure keys. It is in other words possible for developers to introduce a large security weakness by choosing insecure signing secrets, e.g. through blindly following an online guide on how to implement JWT and just copy-pasting the trivial example secret.

In line four, we retrieve the Authorization header from the HTTP request. The reason for passing JWTs in the Authorization header rather than using cookies is to avoid CSRF issues (see 2.2.3 for more info on CSRF), which can be a huge problem for REST APIs. If an access token is not passed, the request should be rejected with a 401 status code.

2.7 REST API modelling

REST API modelling is the process of formalising the structure and expected behaviour of a REST API, for example through documentation. Public actors with public REST APIs such as Facebook [1], Spotify [2] and YouTube [3] all have well documented REST APIs, which makes it easy for developers to integrate their software with their service.

There are multiple API specification languages available, such as Swagger [22], API Blueprint [23] and RESTful API Modelling Language [24] (RAML). In order to conduct the automated tests, we need some information about the API we are testing. The bare minimum of information required to be able to conduct the automated test should be structured in an API model.

It is important to understand that modern REST APIs which conform to common REST best practices typically will share many architectural features. These features includes usage of decentralised authentication and authorisation, standardised usage of HTTP verbs

and response codes as well as proper usage of query parameters and URI parameters. All of these properties can be formalised in an API model.

Another feature of REST is that data should be transferred in the HTTP response body. This is typically done through a JSON-structure, but this is not mandatory, and also nothing the API model should depend on. What the API model is reliant upon, however, is the fact that these response bodies should be possible to model with key-value pairs, e.g. how listing 2.5 is modelling the user response of listing 2.4.

```
{
  "username": "petter",
  "password": "1234",
  "id": 123
}
```

Listing 2.4: User response

```
{
  "fields": [
      "username",
      "password",
      "id"
    ]
}
```

Listing 2.5: Simple user model

For a REST API that satisfies these best practices, it is should be possible to formalize a simple API model, which describes the various endpoints and the entities that relate to each endpoint. This model could then be used to conduct a set of dynamic security analyses, inspecting the API at run time, in essence performing a light-weight external analysis (penetration test, see 2.3.5).

The API model could also be used as a foundation for a static model-based analysis. This analysis must not be confused with static code analysis, as the model based approach will be source code agnostic. The analysis will simply analyze patterns in an attempt to identify commonly known security vulnerabilities in REST APIs. The API model will be formally introduced in section 4.1.

Chapter 3

Analysis and vulnerabilities

The focus area of this thesis is to use commonly known patterns related to REST architecture, and the related authentication and authorisation process, in order to conduct an automated security analysis of a system. The intent of this chapter is to introduce two analysis technique categories, and then define a set of security vulnerabilities which one should be able to find using various implementations of said analysis.

3.1 Analysis techniques

This section aims to identify some common techniques which can be used to discover security vulnerabilities in a web service. Each technique will be explained, and it will be discussed as to which extent each technique could have false positive or false negative results.

Before continuing, there are two important distinctions I want to show and discuss. The first distinction is the one between *static* and *dynamic* specification based analyses, and the second one is the distinction between *dictionary* and *algorithmic* analysis approaches.

3.1.1 Static and dynamic analysis

Figure 3.1 depicts the conceptual difference of the static and dynamic specification based analyses. A green box indicates which part of the stack the analysis interacts with, while a red box indicates that the part was never executed. The figure is split into three layers.

The vulnerability

The bottom layer is the actual vulnerability itself. The vulnerability might be a cross site scripting, SQL injection or any other web service related security vulnerability.

The code

The code might introduce a security vulnerability. The only way to know whether or not a vulnerability is introduced is to have knowledge of this vulnerability and discovering it during code review or a similar processes, or by using third-party frameworks to detect vulnerabilities.

The API

The only way for outsiders know how to interact with the application code is through the API. The API is the foundation for both the static and dynamic analysis.

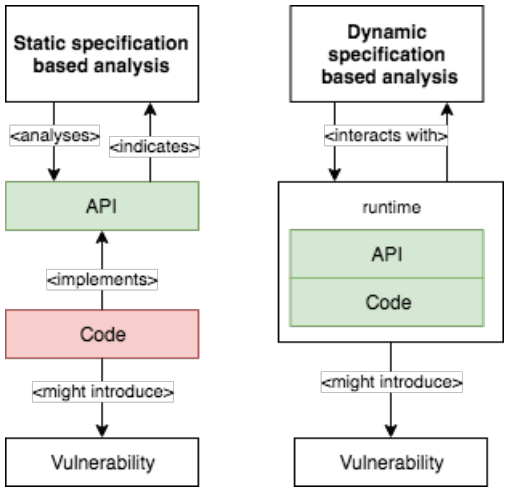

Figure 3.1: Static and dynamic analysis

Static analysis

As the static analysis never interacts with the system under test, the only way it can produce test results is by analysing the API model itself. This means that the underlying code is never executed, which often makes it hard to be 100% sure that an indicated vulnerability actually is present in the source code (section 3.1.2 describes this in more detail). From the figure it is apparent that the static analysis never interacts with the source code.

Dynamic analysis

The dynamic analysis on the other hand, interacts directly with the system under test. It does this through a series of requests. For each request the response is analysed in order to determine what actually happened on the server and if this action violated any security constraint.

3.1.2 Static analysis categories

As mentioned, there are two static analysis categories: the algorithmic and the dictionary-based. Figure 3.2 depicts the two approaches. The main takeaway from this figure should be the artifacts required for each of the analysis approaches. While the dictionary approach needs a dictionary in addition to the specification in order to execute, the algorithmic approach requires only the specification.

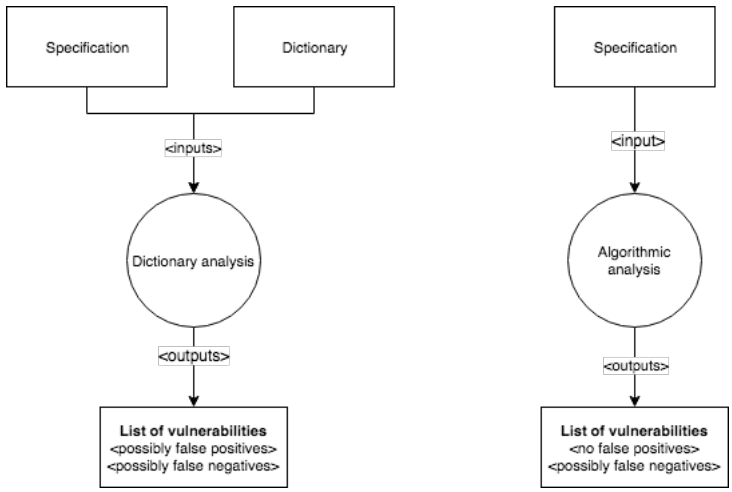

Figure 3.2: Dictionary- and algorithmic analysis

3.1.3 ST1 - Dictionary-based analysis

Although the dictionary-based analysis technique is inspired by the dictionary attack described in section 2.3.4, the terms must not be confused. While the dictionary attack focuses on brute forcing cryptographic problems, the dictionary-based analysis applies a similar approach to brute force possible vulnerabilities. The dictionary-based analyses consists of a high-level analysis of the API model, which I introduce in section 4.1. It does this by iterating over a set of known vulnerabilities, comparing various API traits to these known vulnerabilities to suggest possible vulnerabilities in the API. Due to the fact that this approach does not use any form of simulation to mimic the server side source

code behaviour, the result of the analysis itself can yield false negatives. The analysis will, however, produce a set of warnings where indications of software vulnerabilities are made apparent to the web service developer. The developer must then, manually or through usage of other tools, check whether or not the warning actually reveals a security vulnerability, or if it is a false positive.

Some security issues relate to a set of *symptoms*. An example of such a symptom is suspicious usage of query parameters, e.g. where an access token and a user id both are passed through the query parameters at the same time. This could be an indication of a security vulnerability (a symptom) where users are able to manipulate the user id query parameter in order to obtain information regarding other users, which the API developer did not intend for. This symptom could produce false positives, e.g. in cases where users are supposed to be able to query the API for other users' information based on their user id.

The goal of an analysis implementing this technique should be to warn developers about potential, commonly seen, dubious security related practices.

```
1 dictionary = [
2     ...
3 ]
4
5 analyseDictionary(subject):
6   for dictionaryEntry in dictionary:
7     if isBug(subject, dictionaryEntry) and is not dictionaryEntry.
          whiteListed
8       # Notify the developer that there might be a security issue
```

Listing 3.1: Dictionary-based analysis

Consider listing 3.1. The pseudo-code is very similar to the one shown in listing 3.2—the main difference is that the dictionary-based technique is less precise, and thus we are not able to confidently deem the result a security vulnerability without conducting a closer review, e.g. through manual penetration testing or source review. As such, the developer utilising the tool should be able to specify a 'white list' property for each of the subjects under test, e.g. a query parameter or JWT body field, in order to declare that a code smell is indeed just that, and no security bug.

3.1.4 ST 2 - Algorithmic analysis

The algorithmic analyses leverage the fact that there are times when one does not need to inspect the source code implementations of a feature in order to determine whether a vulnerability has been introduced. Instead of inspecting the source code, we can examine an artifact produced by the system such as JWT to find security vulnerabilities. Due to the JWT standard, the test suite itself is able to simulate the server side implementation code. A test implementation with this approach is able to produce *sound* results, as there should be no difference between the test running in isolation and a dynamic analysis interacting with the system.

Some security issues relate to a set of strict specifications that must not be violated. Examples of this could be that a JWT must not have a longer expiration time than one hour, or that a JWT must be signed with a cryptographically secure key.

By storing known violations of such constraints, such as a set of known commonly utilised insecure cryptographic keys, and creating a test that iterates over this set (the *Dictionary*), we can ensure that previously discovered security vulnerabilities are not introduced in our system. Even better, it should be possible to share these dictionaries among any system, ensuring a collaborative approach to automated security testing.

Example usages of this technique is for finding Weak Cryptographic Keys (section 3.2.1) or potential Bypassing of Access Control vulnerabilities (section 3.2.2).

```
knownIssues = [
    ...
]

analyseDictionary(subject):
  for knownIssue in knownIssues:
    if isVulnerability(subject, knownVulnerability)
      # Notify the developer that there is a security issue
```

Listing 3.2: Algorithmic analysis

Consider listing 3.2. We start off by defining the dictionary, *knownIssues* (in a real world application this should be stored in a file, database or loaded from a Web API). This dictionary could, for example, consist of compromised cryptographic keys found through mining public GitHub repositories or in online articles explaining how to implement something where a cryptographic key is used. A concrete implementation of this example will be shown in section 4.3.

The analysis itself is straightforward. We iterate through the known issues and for each known issue we conduct some sort of test to determine whether or not the subject is susceptible to the issue.

3.1.5 Dictionary-based analysis and false positives

When having a black-box approach to security testing, it can be hard or even impossible to distinguish between something that looks like a vulnerability and something that actually is a vulnerability. Consider the example shown in 3.2.2-*implied security vulnerabilities*. From the outside it is impossible to tell the difference between the erroneous authorisation implementation shown in listing 3.4, and the correct implementation shown in listing 3.3. If the API is indeed implemented as in listing 3.3, a dictionary-based analysis of the scenario will yield a false positive.

```
getUser(request, response):
  authorization = request.headers['Authorization']
  if authorization is none:
    return response.status = 401

  userId = request.query['userId']
```

```
7   validatedToken = validateToken(authorization)
8   if not isAuthorized or not userId === validatedToken.userId
9     return response.status = 403
10
11  else:
12    return response.body = users.find(userId)
```

Listing 3.3: Erroneous authorisation

In addition to the possible false positives, any vulnerability analysis conducted in this thesis comes with the possibility of introducing a false negative. In practice this means that just because an analysis is unable to find a security vulnerability, security vulnerabilities could still be present in the system. This is not a concept unique to this thesis, in fact any security testing approach can never guarantee that vulnerabilities does not exist.

3.2 Vulnerability patterns

The goal of this section is to present the various vulnerability patterns covered in this thesis. A vulnerability pattern is a repeating security vulnerability, which I aim to find an automated security test for. The vulnerability patterns listed are based of the OWASP Top 10 initiative [25] and the OWASP REST cheat sheet [9].

Figure 3.3 shows an overview of how this chapter relates to the analyses introduced in chapter 4 and 5.

3.2.1 B1 - JWT misuse

The OWASP Rest Security Cheat Sheet (described in 2.2.1) lists JWT as one of its focus areas. According to the cheat sheet, there seems to be a convergence towards using JWT as the format for security tokens. As the JWT implementation introduces a single point of failure with regards to authentication and authorisation, it is very important to get it right, security wise.

Section A5 of OWASP Top 10 2017 explains how exploitation of broken access control is a common skill among attackers. Analysis Tools and security testing tools can detect the absence of access control, but cannot verify if it is functional when it is present [26].

This section describes some security pitfalls developers must avoid when implementing JWT-based authentication mechanisms in their web services.

JWT1 - Insecure cryptographic key

JTWs are typically signed using a symmetric or asymmetric hashing function, as described in section 2.6.3. As anyone with the secret key is able to issue new JWTs (which in effect compromises the integrity of the entire system), it is very important that the secret key is

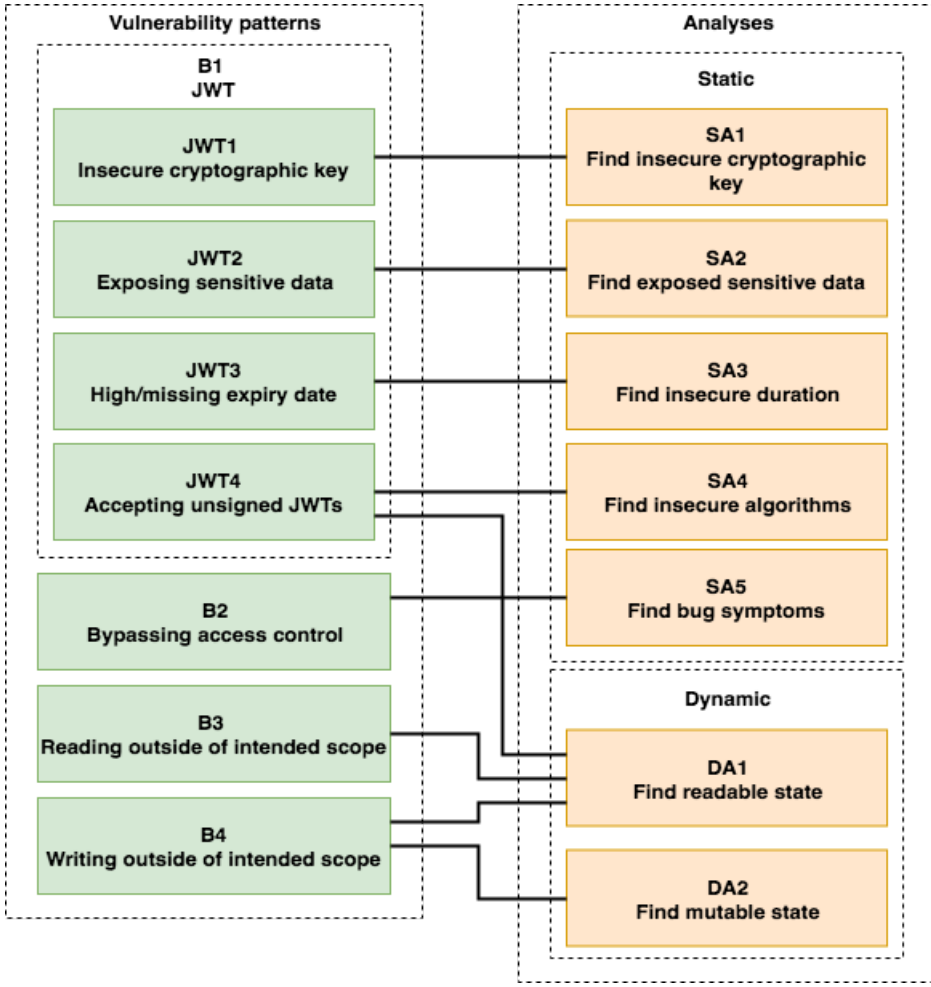

Figure 3.3: Vulnerability patterns and related analyses

indeed secret. Common pitfalls here includes an uneducated developer blindly following a guide online on how to implement JWT, and simply copying and pasting the secret used in the guide. Another pitfall is that developers unknowingly check their secret in to source control systems for open source projects.

JWT2 - Exposing sensitive data

As the payload of a JWT is simply base64-encoded JSON, it is not a suitable place to store sensitive data, such as a users email, phone etc. I have seen cases where entire user-objects have been inserted into the payload directly from the database without any form of sanitation.

While one may argue that JWT exposing sensitive data is not a security concern, as if someone has access to a JWT they are able to access all of the user's information in the system, it should still be considered bad practice to expose sensitive information in the JWT. The reason for this is impact mitigation—imagine that there is a security breach where attackers are able to retrieve a list of expired JWTs. The JWTs themselves can no longer be used, but they expose sensitive data as clear text.

JWT3 - High or missing expiration date

JWT is often used as a form of stateless authentication. Due to this, the server might not be able to invalidate access tokens (this would require server-side state keeping track of invalidated tokens—which invalidates the point of stateless authentication). If this is the case, having a long or non-existing expiration date is a security weakness. It is important to note that this also applies to the refresh-tokens, as they too should expire eventually (though not nearly as fast as access-tokens). The Rest Security Cheat Sheet explicitly states that the claim expiration ('exp') should be verified for JWTs. To take this one step further, it is possible to combine the expiration and issued at claims, to determine the duration of a JWT.

JWT4 - Unsecured JWT

The JWT specification supports unsecured JWTs. This is in general considered bad practice, and should only be used when the developer truly intends the JWT to be unsecured. In a blog post Auth0 reveals issues in many common JWT libraries [27]. This vulnerability caused libraries to accept tokens signed with the none algorithm as valid tokens, even though the developers did not intend for this. It is thus important to make developers aware that they are using unsecured JWTs (if they are), as this might not be intentional. Unsigned JWTs can be created by anyone[1], as they contain no signature—and are thus not secure at all.

[1] In a custom implementation the token might contain some encrypted value that is decrypted in order to verify the token, but this seems like malpractice compared to just using signed JWTs...

3.2.2 B2 - Bypassing access control

A common issue related to broken access control is the bypassing of it [26]. There are many known ways this common vulnerability might manifest itself. It can be discovered e.g. through modifying the URL, modifying application state or modifying HTML (which all are instances of client side manipulation).

Query parameters

Modifying query parameters is a subclass of modifying the URL. The security issues related to misuse of query parameters can be split into two categories:

1. Direct security vulnerabilities
2. Implied security vulnerabilities

Direct security vulnerabilities

This category is straight-forward: URL query parameters are simply not safe [28]. Even though the URL parameters are encoded during HTTPS transmission, there are still a multitude of issues related to passing sensitive data through them:

- They get saved in the browser history
- They will be visible in logs
- Users could share the URL without knowing they are sharing sensitive information
- They are visible to browser extensions

Implied security vulnerabilities

The second issue relating to query parameters is that passing information such as user IDs through them often is a code smell[2]. If something used to identify the user on the server side is passed through a query parameter, an attacker may only need to change this query parameter in order to obtain the authentication context of another user.

Consider a GET request to the endpoint `/users?userId=123`. What would you expect this request to return? It is very likely that this request returns the user object related to the ID passed in the query parameter. Even novice attackers will notice this, and attempt to change the query parameter to see if they can gain access to another user's private information.

On its own, this might not be a security issue. If the developers of the endpoint properly implemented access control, the endpoint would check if the request is authorised, and then

[2]Martin Fowler on code smell: `https://martinfowler.com/bliki/CodeSmell.html`

check if the authentication context is indeed related to the userId we are trying to fetch. However, this is easy to get wrong. Consider the pseudocode shown in listing 3.4

```
1  getUser(request, response):
2    authorization = request.headers['Authorization']
3    if authorization is none:
4      return response.status = 401
5    isValidToken = validateToken(authorization)
6    if not isValidToken:
7      return response.status = 401
8    else:
9      userId = request.query['userId']
10     return response.body = users.find(userId)
```

Listing 3.4: Erroneous authorisation

The function getUser defined in line one is hooked up to the URI `http://api.myapp.com/users/:userId`, and thus invoked whenever the endpoint is queried with an HTTP GET request.

In line two the authorisation header is retrieved, and if the request lacks the header a 401 'unauthorised' response is issued. The authorisation is validated in line three, by using a library function that validates tokens. If the validation fails, the token must be expired or some other token related condition must have failed, and a 401 status code is returned. If all the checks passes, the userId query parameter is retrieved in line 9, and the user related to the userId is returned in line 10.

There is a potential big security weakness in this example, which would cause all users to be able to retrieve any user's data, given their userId. The issue is that the endpoint does not use the user ID which is related to the authorisation context, but rather uses a user ID which is provided as a query parameter, which is very easy to change when using HTTP request or proxy tools.

Manipulating the client

Another security property of REST APIs is related to the separation of data access and how data is displayed. As web sites fetch data from an API dynamically, it is easy for inexperienced developers to forget that there are other ways to query data than through the user interface. An example here could be that we are fetching the list of a given user's friends to display somewhere on our page. We want to show a list of their names and profile pictures, so we query the endpoint `/users/:userId/friends` to retrieve it. Now imagine that the backend developer forgot to sanitise the user data exposed through the endpoint, resulting in the endpoint leaking sensitive information such as emails and phone numbers. By looking at the UI, we would never realise that this issue existed, but through the usage of a proxy tool or HTTP request tools such as Postman, anyone is able to query the endpoint directly, and retrieve all our friends' emails and phone numbers. Now imagine this being the list of *potential* friends instead—the possibilities when it comes to overexposure of data are limitless.

The lesson is that developers need to carefully white list what data they expose through their endpoints, and not rely on the UI for anything security related—it does not take much (if any) skill for an attacker to query endpoints directly!

3.2.3 Acting outside of intended scope

A typical theme with REST APIs is user scoping. In essence this means that as a developer, you need the users of your application to be able to access different resources, based on the context of their authorisation scope, e.g. users should be able to access all of their own data, but not the personal data of their friends (unless their friends explicitly shared this information). User scoping can be a hard task to do right, and failing to do it properly can cause vulnerabilities related to bypassing of access control and elevation of privilege, as explained in OWASP A5-Broken Access Control [26].

B3 - Reading outside of intended scope

The example above show how failure to restrict read capabilities can cause leakage of sensitive information. It can also cause other issues, e.g. users solving a multiple choice quiz being able to read the correct answer of each question. It is thus important to take measures to ensure that the read capabilities of each user level is as intended.

B4 - Writing outside of intended scope

An arguably more critical vulnerability arises when users are able to *write* to the API outside of their intended access scope. Regular users could be able to change vital information in the system, such as changing the quiz questions or saying they passed the quiz regardless of their score.

Chapter 4

Static specification-based security analysis

The goal of this chapter is to introduce a set of static security based analyses, as well as describing techniques that makes the analyses possible. I start off by describing a proposed way to model an API. This model is also going to be relevant for chapter 5.

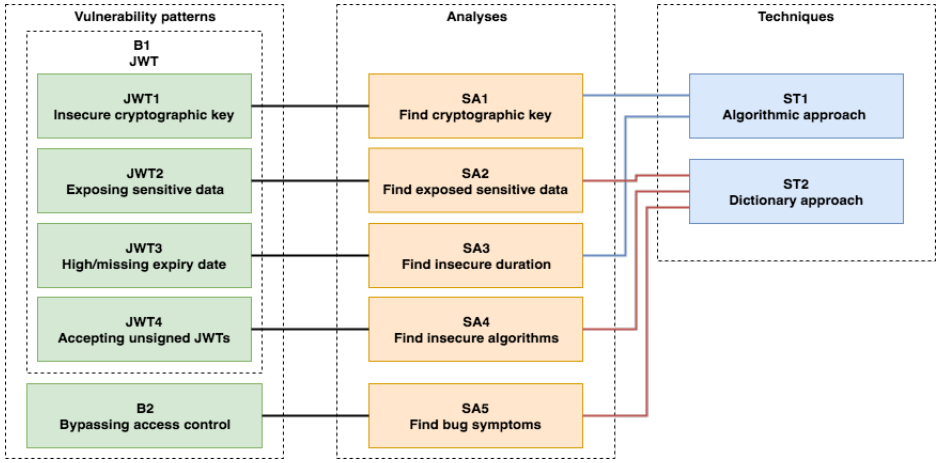

Figure 4.1: Static analysis overview

Figure 4.1 shows an overview of how how the concepts introduced in this chapter relate to the vulnerability patterns described in chapter 3. Each of the proposed analyses are based upon a more abstract technique. I will then be doing a closer review of the various issues explained in subsection 2.6 and 3.2.2—*manipulating the client*.

4.1 Introducing my API model

In order to avoid having to crawl[1] the API (or use similar techniques to map the various endpoints, endpoint fields, etc.), the analyses are reliant on an API model, provided by the developer of the API under test. This API model should be as simple yet expressive as possible. In addition to this, I did not want to end up bogged down by implementation details in RAML or Swagger, which is why I have chosen to introduce my own definition of an API model rather than using existing models[2]. The API model will serve as the foundation of the analyses proposed in this chapter and in chapter 5. Note that the API model expects the API to be implemented using REST standards, i.e. utilising HTTP verbs the way they were intended, having a cohesive URI-model where endpoints share a common root URL and proper usage of HTTP status codes. Figure 4.2 shows an overview of the API model structure, and the rest of this section describes each part that makes up the API model.

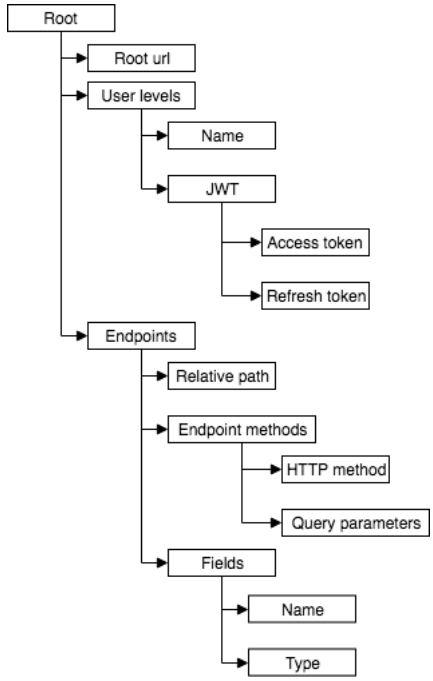

Figure 4.2: API model structure

[1] `https://www.owasp.org/index.php/Crawling_Code`

[2] It should, however, be possible to extend existing models with the properties introduced in this model, in order to allow developers who already model their API in a certain way to leverage the automated security testing proposed in this thesis.

4.1.1 Root URL

The root URL is the root entrance to the API. Typically it is going to look something like `https://api.myapp.com/v1`. For each endpoint, the root URL is going to be concatenated with the relative path to determine the resource URI.

4.1.2 User levels

In order to distinguish between the various access restrictions in an API, the term *user levels* is introduced. The user level field is a list of the various user levels of the application, which I will now describe.

Name

The user level is identified by the *name* field. The name field is there just to give some context to each user-level front end, i.e. it does not need to relate to anything on the server. It is simply there to give better output messages to the developer.

JWT

Each user level is associated with a valid JWT set, consisting of an access and a refresh token. See section 2.6 for more information regarding access and refresh tokens.

4.1.3 Endpoints

The endpoints field of the API model is a list with endpoints, each endpoint consisting of the following parts:

Relative path

Each endpoint is identified by its relative path. The relative path is concatenated with the root URL to determine the resource URI. For example would the root URL `https://api.myapp.com/v1` and the relative path *users* result in the endpoint-URI `https://api.myapp.com/v1/users`.

Endpoint methods

Each endpoint has a list of one or more endpoint methods. An endpoint method defines an action, and has a set of query parameters related to it.

The *HTTP method* of an endpoint should ideally be an enum of the HTTP methods defined in RFC 7231 [11], possibly extended to contain all known HTTP methods.

The *query parameters* is a set of query parameters which may be included when querying the given endpoint. Refer to section 2.1.1 for more information on query parameters.

Fields

The fields set related to an endpoint is used to describe the expected structure of the resource available through the endpoint. Each entry in the set is defined by its name, and needs to be described by a type enumeration. The type enumeration is important because the tool needs to know how to generate proper field data when creating the test suite.

4.2 Static analysis techniques

Here I describe the two static analysis categories. Each of the categories groups together similar analysis approaches.

4.2.1 ST1 - Algorithmic analyses

An algorithmic analysis tests a single security constraint of a security sensitive property. It is utilised when there is no need for a dynamic approach to security testing, e.g. when testing the duration of a JWT or checking a JWT's alg header. It is hard to generalise a notation for the algorithmic analysis techniques, as each implementation is custom tailored to the individual bug pattern—the only generic way to describe it is as a function that returns true if a security constraint is violated and false otherwise.

An important property for this technique is that it will not produce false positives. This is due to the fact that the server side behaviour is simulated because of the known standard and can we are able to understand what happens inside the black box without actually peeking inside it.

4.2.2 ST2 - Dictionary based analyses

The dictionary based analyses utilise a set of assorted values in order to attempt to determine whether or not a security vulnerability is present. These sets should consist of known security vulnerability issues. An example of such set is commonly seen misused HTTP query parameters, such as 'password' and 'token'. The dictionary should ideally accumulate over a longer period of time, incrementally becoming a solid source for common security malpractices.

The analysis iterates over the values of a given set, and for each value it makes a comparison with a security related property. This analysis is not precise enough to produce only true positives, which means there will be false positives, which are described in section 4.2.3.

Consider the set of known security bugs K, related to the security sensitive property S. A test, T is performed for all the values in K, and for each failed T, the result should be tracked in a list of the found bugs, B. A failed test indicates a possible or definitive security bug, depending on the context. Tests could be exclusive for a set, meaning that if one test fails, all other tests will pass. Having all tests pass does not mean there is no security vulnerability, just that the test suite was unable to find one.

4.2.3 Dictionary-based analyses and false positives

A repeating issue with the static analyses which falls under the dictionary-based analysis category is that they are capable of producing false positives. False positives occur when an analysis warns the developer about a security vulnerability that is in fact not a security vulnerability. This is related to the issue described in section 3.2.2—**Implied security vulnerabilities**. The essence of this issue can best be described with an example. Consider that the API exposes the endpoint `/users?userId=...` to retrieve data related to a given user. In general, this does not comply with REST best practices, as the userId in this scenario should be provided as a URI parameter, i.e. `/users/:userId`. Common vulnerabilities related to missing access restriction has been seen to implement this flawed URL pattern. However, just because this flawed pattern is used, there is no way of telling whether or not there actually exists a vulnerability in the system runtime, the only way to know this is by actually executing a series of requests and analysing their responses. This type of analysis is not within the scope of this thesis, and as such the static analysis will be capable of producing such false positives.

4.3 SA1 - JWT spoofing

The process of signing a JWT is a pure function[3]. This fact can be leveraged to specify an analysis that is able to determine whether a given JWT is signed using an insecure cryptographic key defined in a dictionary.

```
1 insecureCryptographicKeys=[
2   'secret',
3   'KMi23MneEmEPPoEW',
4   ...
5 ]
6 analyseInsecureCryptographicKey(verifiedJWT):
7   for insecureKey in insecureCryptographicKeys:
8     spoofedJWT = jwtLibrary.sign(verifiedJWT.header, verifiedJWT.payload,
          insecureKey)
```

[3] A pure function is a function where the return value is only determined by its input parameters, i.e. given the same input always will produce the same output.

```
    if spoofedJWT.signature == verifiedJWT.signature:
      # The jwt is signed using an insecure cryptographic key
```

Listing 4.1: JWT spoofing analysis

Listing 4.1 shows pseudocode for the proposed analysis. The function is passed a verified JWT from the API under test. Then, for each of the known insecure cryptographic keys, a new JWT is created by utilising a JWT library or an implementation of the JWT signing process. By using the JWT provided by the developer in the API model, we can ensure that the spoofed JWT was created with the exact same header and payload as the verified JWT. Due to this, an equality in the signatures between the spoofed and valid token implies that both tokens are signed with the same cryptographic key (or with keys that generate the same hash, which essentially is the same). The developer should be notified that their JWT is being signed using insecure cryptographic keys.

4.4 SA2 - Analysing JWT payload

As explained in section 3.2.1, JWT payloads are typically not encrypted - and should thus not contain sensitive information. This is dictionary based, and finds exposure of potential sensitive JWT payload fields.

4.4.1 Analysing JWT payload by field names

A proposed, simple, way of analysing JWT payloads is to use a set of known commonly misused payload field names, as shown in *commonlyExposedJWTPayloadFields* in listing 4.2.

```
commonlyExposedJWTPayloadFields=[
  'email',
  'password',
  'phone',
  'name',
  ...
]

analyseJWTPayload(jwt, whiteListedFields):
  for exposedField in commonlyExposedJWTPayloadFields:
    if hasField(jwt.payload, exposedField)
       and not hasField(whiteListedFields, exposedField):
      # The JWT is exposing a dubious field, warn the developer
```

Listing 4.2: Analysing JWT payload by field names

Listing 4.2 shows one approach for this type of testing. It relies on a set of known commonly exposed fields which are considered bad practice to pass in JWT payloads. This is a dictionary based approach, and thus unable to definitely guarantee that the result is a true positive. To compensate for this, developers should be able to define a set of white-listed fields to pass down to the analysis.

4.4.2 Analysing JWT payload by regular expression

Another way of analysing the JWT payload is to use regular expressions. The main benefit of this approach is that we no longer depend on the developers to use specific field names to detect possible misuse. The main problem with this approach is that while it is pretty straight forward to use regular expression to find e.g. email and phone numbers, it could be challenging to use it to find more ambiguous sensitive fields, such as passwords or names.

```
sensitiveDataRegexes=[
  emailRegex,
  phoneRegex,
  ...
]

analyseJWTPayload(jwt, whiteListedFields):
  for sensitiveDataRegex in sensitiveDataRegexes:
    if field.match(sensitiveDataRegex)
      # The JWT is exposing a dubious field, warn the developer
```

Listing 4.3: Analysing JWT payload by regular expression

4.5 SA3 - Analysing JWT duration

JWTs should expire within a reasonable time frame, as explained in section 3.2.1. Testing for high expiration includes finding the fields representing the time the token was issued and when the token expires. If either of these values are not found, the developer should be notified–as this is a possible security flaw. The issued time is subtracted from expiration date, in order to find actual expiration time. If this expiration time is above a given threshold, the developer is notified.

The following test should be able to find access tokens with expiration time over a certain threshold. The *dangerousExpirationTimeInHours*-parameter should be derived from industry best practices.

```
testTokenExpiration(token, dangerousExpirationTimeInHours):
  expiresAtMilliSeconds = token.body.exp
  if expiresAtMilliSeconds is None:
    # The token is not using the required expiration field name, or is
        missing expiration time
  issuedAtMilliSeconds = token.body.iat
  if issuedAtMilliSeconds is None:
    # The token is not using the required issued at field name, or is
        missing issued at time

  expiresInMilliSeconds = expiresAtMilliSeconds - issuedAtMilliSeconds
  expiresInHours = expiresInMilliSeconds / (60 * 60 * 1000)
  if(expiresInHours > dangerousExpirationTimeInHours):
    # Token expiration is too long, notify the developer!
```

Listing 4.4: Find expired token

This test assumes that the developers has used the JWT standard fields exp (expires) and iat (issued at) to describe when the JWT was created and when it will expires. It should be possible to extend this method to include any name for the fields, but as exp and iat is explicitly mentioned in the RFC I see no need for this feature.

4.6 SA4 - Unsecured JWT

I propose two ways of testing for unsecured JWT weakness; one relying on the API response, which is a dynamic analysis, and one relying on static analysis:

- Decode a provided JWT and check the alg header.
- Attempt to access restricted endpoints using a generated JWT with the alg: none header set.

Both of these tests should be performed, as one is related to configuration of the JWT, while the other is related to the server side authentication process implementation.

Testing for the alg header - static

The 'none' algorithm should only be used when the integrity of the JWT already has been verified [27], thus we warn developers when the algorithm is none as this is not a common use case while using JWT for authentication.

```
1 testAlgHeaderStatically(token):
2   if token.header.alg is 'none'
3     // alert the developer
```

Listing 4.5: Static inspection of alg header

Testing for the alg header - dynamic

In order to verify that the server is not accepting unsigned JWT's, a real request with a token with the alg header set to 'none' needs to be sent.

```
1 testAlgHeaderDynamically(token):
2   token = jwtLibrary.signToken('none') // Sign token with none algorithm
3   response = api.makeRequest(token, api.userDataEndpoint) //Request to
        access restricted endpoint
4   if response.code is ApiResponse.SUCCESS:
5     // notify developer
```

Listing 4.6: Dynamic inspection of alg header

4.7 SA5 - Analysing the API model

While static analysis of the API model does not reveal definitive security vulnerabilities, it can reveal *symptoms* of them. Through static analysis of the API model it should be possible to find these symptoms and warn developers about the possible security vulnerabilities.

The API model analysis consist of iterating through each endpoint in the model. For each endpoints, every query parameter is evaluated against a set of fields which are predetermined as potentially dangerous, very much similar to how the JWT payload is analysed in section 4.4.1.

Chapter 5

Dynamic specification-based security analysis

The purpose of this chapter is to introduce a set of dynamic security analyses, which can be utilised to discover security vulnerabilities in a system at runtime. The reader should be familiar with the API model as described in section 4.1 before continuing with this chapter.

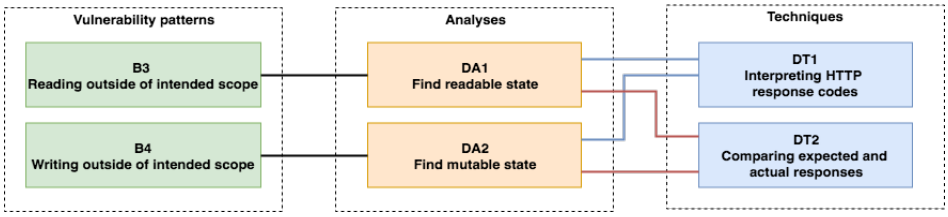

Figure 5.1: Dynamic analysis overview

Figure 5.1 shows an overview of how how the concepts introduced in this chapter relates to the bug patterns described in chapter 3. Each of the proposed analyses are based upon one or more abstract techniques, which I introduce in the technique section.

5.1 Dynamic analysis techniques

Dynamic security analysis inspects the system under test at runtime. This is done through the usage of defined HTTP standards, such as HTTP response codes, HTTP verbs and HTTP response bodies. It is thus required that the API under inspection is using these

HTTP and REST standards as proposed in various RFCs [11, 13] and as Fielding describes REST [6]. The analyses contributed in this thesis focuses on the HTTP GET and POST verbs, but it should be very possible to extend the analyses to also use other commonly used verbs such as PUT and DELETE.

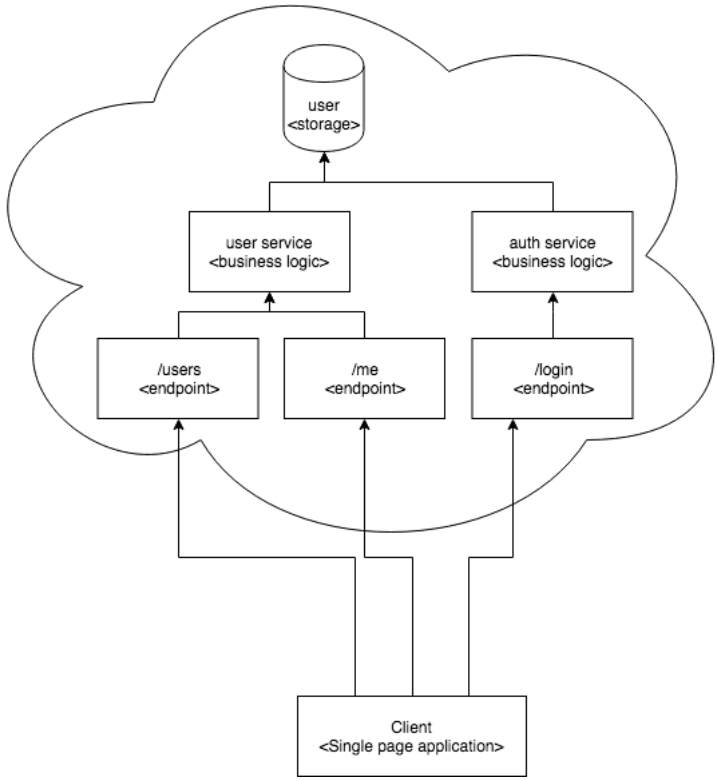

Figure 5.2: REST API overview

Before delving into the concrete techniques used to conduct the dynamic analyses, I want to introduce some example requests and responses which are commonly seen in REST APIs. I will inspect each example, and manually analyse it in order to show how we can use various properties of the requests and responses to conduct automated analysis. Figure 5.2 shows an overview of a very basic example REST API with three endpoints. The client, in this case a single page web application, is interacting with the various API endpoints which are exposed over HTTP. By using the proper HTTP verb in conjunction with a request body in the case of a POST request, the client is able to manipulate the server side state. Let us examine some of the possibilities this REST API provides a client side developer, and how it responds to various inputs.

Creating a user

In order to create a user, an HTTP request with the HTTP method POST is sent to the endpoint `/users`.

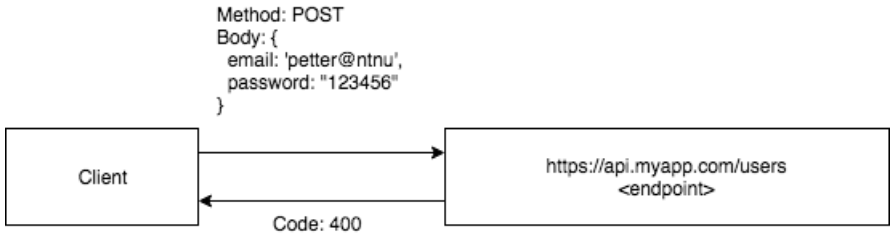

Figure 5.3: Invalid POST user request

Figure 5.3 shows a request which is sent with an invalid request body (note the invalid email syntax), and is thus rejected by the server. This rejection is shown with a 400—bad request—HTTP status code in the HTTP response. No further information than this is required by the client to know that there is a problem with the request content. When conducting automated testing we can leverage this fact, and if any response returns with a 400 bad request status code we know that something is wrong. Typically in this case the problem is going to be that the API model is improperly configured. Regardless of the cause, the developer should be notified so a manual revision of the request can be conducted.

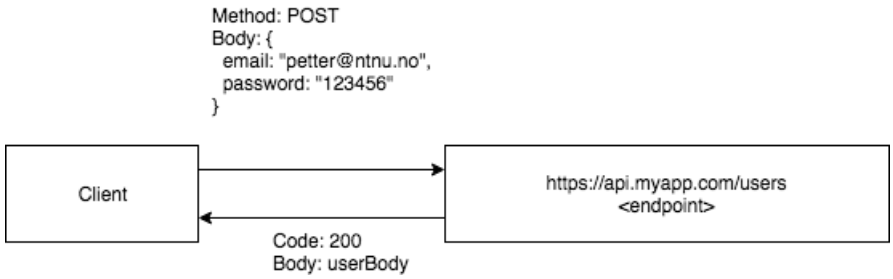

Figure 5.4: Valid POST user request

Figure 5.4 depicts a valid request, where the client has issued a new request with the fixed input. This time the server accepts the request, and thus a new user is created. This is indicated with a 200 HTTP response, and the newly created user resource is returned in the response body.

Note that the response now contains the newly created user resource. We can leverage the response body to conduct an analysis of the writable user scope, that is what properties a user is able to change on the server side when making a request to this endpoint. This is done by comparing the request body with the response body, and explained further in section 5.1.2.

Retrieving user data

To retrieve the user data of the currently authenticated user, a GET request can be issued to the endpoint `/me`.

Figure 5.3 depicts a request to the `/me` endpoint. The request lacks authorisation, and is thus rejected by the server, as access restriction is properly in place. If developers by some chance forgot to implement access restriction, or somehow removed it during development, the analysis could automatically notice this and notify the developers. This would be noticed by

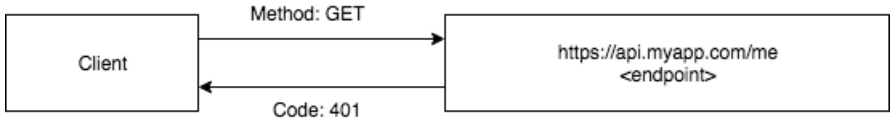

Figure 5.5: Invalid GET current user requests

In figure 5.7 the client has added the Authorization header and populated it with a valid JWT, using the Authorization Bearer Scheme[1]. This time, the server processes the request and returns the requested resource in the response body.

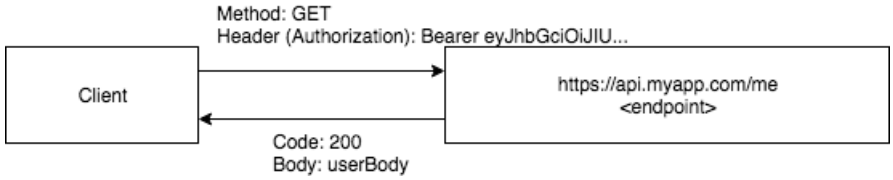

Figure 5.6: Valid GET current user request

Retrieving other users' data

The final example is a hacker trying to access another user's personal data. The hacker has reverse engineered the API, and attempts to retrieve the user data of the user with id 123. Figure 5.7 depicts the hacker's request.

Again, access restriction is properly implemented on the server side, and the server rejects the request. The notable difference between this scenario and the scenario in figure 5.5, is that this time a valid JWT is present in the Authorization header. Due to this, the server should not respond with the 401 status code. Instead, the response code is set to 403 which indicates that authorisation was provided, but insufficient to read the given resource.

[1] The Bearer Scheme is defined in RFC 6750 [29]

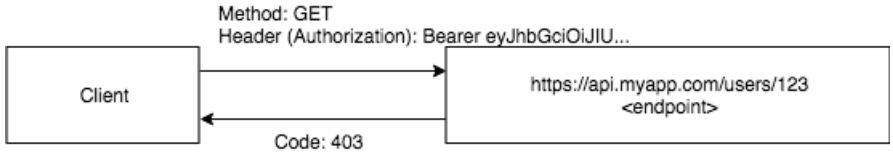

Figure 5.7: Invalid GET other user request

5.1.1 DT1 - Interpreting HTTP response codes

Both the readable and writable state analyses include analysing HTTP response codes. Solely by analysing the response code of a response, we are able to extract valuable information about how the execution of a request went. This, however, requires the API to properly use HTTP response codes. One might argue that strict usage of HTTP response codes makes the API easier to reverse engineer, and that would not be wrong. However, security by obscurity is never a solution and I would argue that it is more secure to have a deterministic testable API than an obscure, hard to test, API. A solution to this is issue is, however, to have a separate debug and production setting, which obfuscates HTTP response codes in the production environment. Another solution is to make the HTTP response codes in the tool configurable. Solving these problems is not a focus of this thesis.

As described in section 2.1.1, different systems might use different response codes to indicate the same result, hence table 5.1 lists the relevant HTTP status codes, and their interpreted meaning, in the context of this thesis.

Value	Status Code	Derived meaning
200-299	Success	The request was successful, we can assume that the user has access to the current endpoint
400	Bad Request	The server rejected the request due to missing required fields.
401	Unauthorized	The server rejected the request due to lacking authorisation.
403	Forbidden	The server rejected the request due to lacking permission level.
404	Not Found	The server was not able to locate the resource.
409	Conflict	The server rejected the request because there already exists an entity with the provided identifier.
500-599	Server error	Seems like we found an unaccounted scenario. Security wise this is never a good thing, and the developer should be notified.

Table 5.1: Interpreting HTTP response codes

5.1.2 DT2 - Comparing expected and actual responses

When comparing expected and actual HTTP responses, it is important to make a distinction between GET and POST requests.

A GET request should never change state on the server, hence the only way of knowing what to expect from such a request is by having it modelled, e.g. in an API model.

A POST request always creates a new entity on the server, hence the expected response is contained within the request itself.

The HTTP methods GET and POST directly correlate to read and write actions, respectively.

Comparing GET requests and responses

When reading from an API, there is typically a finite set of responses the server can return. The status of the response should be indicated with an appropriate HTTP code, as described in section 5.1.1. When the server does not respond with a success status code, we use table 5.1 to determine the reason for rejecting the request.

When the server does respond with a successful HTTP response code, it should also contain some data[2] in the response body. The content of the response body is what defines a user's *readable access scope*. The data contained within the response body from a given endpoint might vary with the user who is executing the request's authorization.

```
1 {
2   "username": "petter",
3   "password": "1234",
4   "id": 123
5 }
```

Listing 5.1: Sample response body

Consider the sample response body in listing 5.1. Assume this to be the response to the request GET `/users/123`. Would it be expected that one is able to retrieve a user's password through an API endpoint, regardless of authentication context? That seems like a terrible practice, even if the password is hashed and salted.

By defining an expected response body for a given endpoint it is possible to trigger a warning whenever something unexpected is returned, in order to ensure that no unexpected data is leaked from any API endpoint. The *Fields* property of the API model, described in 4.1.3, allows developers to specify a set of fields to expect a given endpoint to return.

```
1 {
2   // user endpoint
3   ...,
4   "fields": [
```

[2] It is common REST practice to return the resource in question, either the requested resource through a GET request or the newly created resource from a successful POST request.

```
    {
      "name": "username",
      "type": "string"
    },
    {
      "name": "password",
      "type": "string"
    },
    {
      "name": "id",
      "type": "number"
    }
  ]
}
```

Listing 5.2: Sample field definition

Listing 5.2 shows a simplified approach to defining fields for the `/users/:userId` endpoint. Now, we are able to issue a request to the endpoint, and compare the fields of the response body to the expected fields. If the response body contains fields not defined in the API model, something is off and the developer should be notified. Due to the fact that we define user levels in the API model (cf. section 4.7), we are able to run this analysis for each access level, producing a full log of what user group is able to access what information for each endpoint. Note that this thesis has a simplistic approach to the subject of access levels and access scope. Any non-trivial application will typically have complex authorisation graphs, and defining access levels as USER, ADMIN etc. will simply not suffice.

Comparing POST requests and responses

Similar to the previously explained analysis of GET requests, it is possible to conduct tests of what users are able to write through POST requests.

This technique utilises the practice of returning a newly created resource in the HTTP response body. By keeping a copy of the request body, and comparing each field of the response body to its respective field in the request body, we are able to determine which fields seems mutable and which ones are not.

```
{
  "username": "petter",
  "password": "1234",
  "isAdmin": "true",
  "id": "123"
}
```

Listing 5.3: Sample POST request body

```
{
  "username": "petter",
  "isAdmin": "false",
  "id": "24821",
```

```
5   "createdAt": 1525254977044
6 }
```

Listing 5.4: Sample POST response body

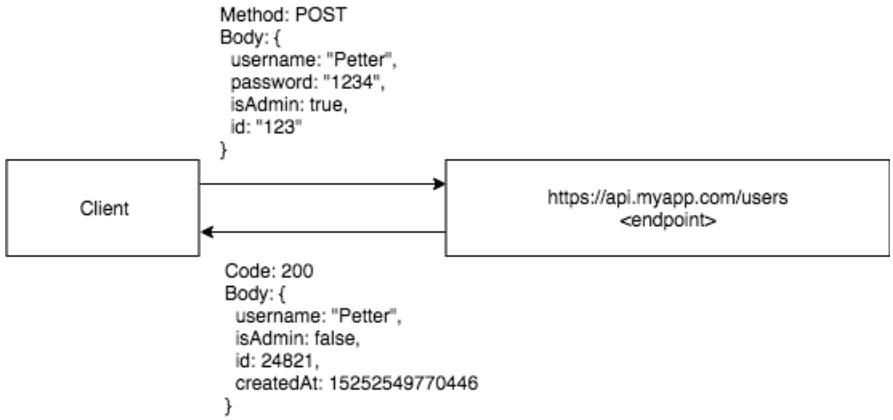

Figure 5.8: POST user request

Consider figure 5.8 which depicts the request needed to create a user. Listing 5.3 depicts an imagined POST request body for creating a user. After the server has processed the request, it returns the response depicted in listing 5.4. By iterating over each field in the response, and comparing each field to its respective request body value, it is possible to suggest which fields are mutable and which fields are not for the current user access level, which in this anyone, as anyone should be able to sign up.

Through a manual analysis here we are able to see that the *username* field seems mutable, while the *isAdmin* and *id* field seems immutable for the client, as they should be. An automated test could notify the developer of any mutable field for each endpoint.

The *password* field has disappeared completely, as it should. Fields modeled in the API model that do not appear in the response body should be labeled **not readable** by the analysis, so that developers can be sure that sensitive fields are not exposed.

Finally, notice the new field, *createdAt*, which was not included in the request. Fields not included in the request that appear in the response should also trigger a warning for the developer, as unmodeled fields are at risk of being mutable without intention.

5.2 DA1 - Readable state analysis

The readable state analysis makes usage of both the techniques described earlier in this chapter. It iterates over each endpoint defined in the API model. For each endpoint, it iterates over the user levels, and also attempts to make an unauthorized request.

$$\textbf{Total\_requests} = \textbf{\#Endpoints} * \textbf{\#User\_Levels}$$

This means that if an API has five endpoints and three user levels including unauthorized, a total of 15 requests has to be executed for the readable state analysis. For each of the requests, an analysis will be made. This analysis consists of two possible steps: interpreting HTTP response codes and, if successful, comparing expected and actual response.

The process of interpreting HTTP response codes is very generic, and needs no customization for the readable state analysis. This means that the first step of this analysis is to check the HTTP status code of the response. If the status code is not successful, there should be no response body–and step two cannot be conducted. A suitable test report should be produced, e.g: *GET `/users` NOT readable for user level **USER*** when the response code is Forbidden, or *GET `/userds` (note the mispelled endpoint) not found, are you sure the API model is configured properly?* when the response code is Not Found.

When a GET response is successful, it should contain the entity requested through the URI. The fields of the entity might depend on the user access level - e.g the user with id 123 might be able to get all user data by querying `/users/123`, while any other user might only be able to see the public profile of the user. Every field in the response is considered *readable* for the user executing the request.

```
1 analyseGetResponse(actualFields, userLevel):
2   for actualField in Fields:
3     # This field is readable for the current userLevel
```

Listing 5.5: Analysing HTTP response

5.3 DA2 - Mutable state analysis

The mutable state analysis is very similar to the readable state analysis. The main difference is related to how a success response is analysed.

When a POST response is successful, it should contain the entity submitted through the request. By comparing the fields of the response entity to the fields of the request entity, it is possible to assume which fields are mutable and which are not.

```
1 analyseGetResponse(requestFields, responseFields, userLevel):
2   for requestField in requestFields:
3     if requestField in responseFields:
4       if requestField is responseFields[requestField.label]:
5         # It is likely that the field is mutable for the given userLevel
6       else:
7         # It is unlikely that the field is mutable
```

Listing 5.6: Analysing mutable state

Key takeaways

Before ending this chapter, I want to summarise the most important concepts discussed. The key takeaways I want the reader to have after reading this chapter is:

- How predictable REST APIs become if they follow the proposed best practices and common standards, and how easy it becomes to conduct automated security testing of them.
- How much we are able to deduct regarding a HTTP request just by examining the HTTP response code when these are properly used.
- How we are able to analyse HTTP response bodies for successful HTTP requests in order to determine what the user is able to read and write to a given endpoint.

Chapter 6

Implementation of a proof of concept tool

This chapter describes how I have implemented the analyses described in chapters 4 and 5.

The tool is available on GitHub `https://github.com/pettermaster/kotlin-test-client`

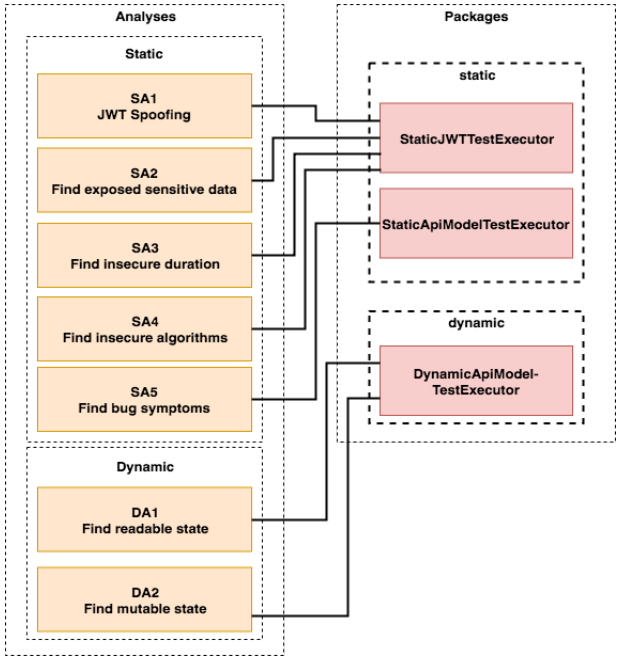

Figure 6.1: Chapter overview

6.1 Overview

The tool is written entirely in Kotlin. The implementation is tightly coupled to the proposed method of modelling an API described in 4.1. Kotlin's representation of algebraic data types, *sealed classes*, is excellent for modelling test results and HTTP responses, so they are utilised in the DSL implementation of the API model.

The API model is provided by the developer in JSON format, which is then parsed to the Kotlin specific representation.

6.2 Structure

The source code is located in the `main/kotlin` folder, separated in four sub packages.

6.2.1 Package api

The *api* package contains the API interface, which specifies how the dynamic test is interacting with the API. This makes it easier to mock the API.

The package also contains mock implementation of an API. This makes it easy to experiment with the tool locally, without having to set up an external API. When testing against a real API, simply replace the mock implementation with a real HTTP client which implements the API-interface.

6.2.2 Package domain

The domain package contains the Kotlin specific implementation of the API model. This is defined in the ApiModel file, which is able to represent the API model as described in 4.1.

It also defines DSLs for handling test results. These are specified in ApiModelTest and FieldModelTestResult.

6.2.3 Package dynamic

The *dynamic* package contains implementation of the dynamic analysis. The DynamicApiModelTestExecutor uses a random field generation method to generate the request body for POST requests.

6.2.4 Package static

The *static* package contains implementation of the static analysis.

ApiModelTestExecutor

This test executor is responsible for conducting static analysis of the API Model, i.e. an implementation of SA5 - Analysing the API model(4.7). The test is instantiated with the parsed ApiModel and a *queryParameterDictionary*, which is a set of query parameters which has been known to indicate security vulnerabilities.

StaticJWTTestExecutor

The StaticJWTTestExecutor file consists of a set of JWT test suites, which are all functions that implement the various JWT-related analysis. The JWT analysis is run by invoking *executeStaticJWTTest()*, passing a set of JWTs (an access and a refresh token) to test.

The *testTokenSecret()* function actually uses the signing secret exposed by the MockApiRepository as one of the entries in the *knownSecrets* dictionary. This is to showcase that the tool is able to spoof a JWT in the case that an insecure cryptographic key is used to sign the token.

Each of the JWT related analyses (SA1-SA4) are implemented as a function or a set of functions, as shown in table 6.1.

Analysis	**Implemented in**
SA1 - JWT spoofing 4.3	testTokenSecret()
SA2 - Analysing JWT payload 4.4	testTokenPayload()
SA3 - Analysing JWT duration 4.5	testAccessTokenSecret() testRefreshTokenSecret()
SA4 - Unsecured JWT 4.6	testUnsignedToken()

Table 6.1: JWT Analysis implementation

Chapter 7

Validation

This chapter aims to validate the results I've produced throughout this thesis. Different research approaches requires different forms of validation, as M. Shaw states [7, p. 6]. I would argue that the result of my research is three different components, listed in order of importance, grouped by the types of results defined by Shaw[7, p. 4]:

1. *Analytic model*—The Specification based analyses, as described in sections 4.3, 4.4, 4.5, 4.6, 5.2 and 5.3

2. *Procedure or technique*—The techniques described in sections 4.2.2, 4.2.1, 5.1.1 and 5.1.2

3. *Tool or notation*—The proof of concept tool implemented in Kotlin, which will be described in this chapter, as well as the API model described in section 4.7

The category groups are also summarised in figure 7.1.

7.1 Validating the analyses

Shaw states that when one introduces a new model, the power of the model must be made clear. Most of the ideas I introduce related to the analyses are vulnerabilities clearly defined in the OWASP Top 10 (broken authentication and broken access control) as well as the OWASP REST cheat sheet (JWT related vulnerabilities), which are considered state of the art when it comes to software security.

To validate that the analyses actually works I've implemented a proof of concept tool.

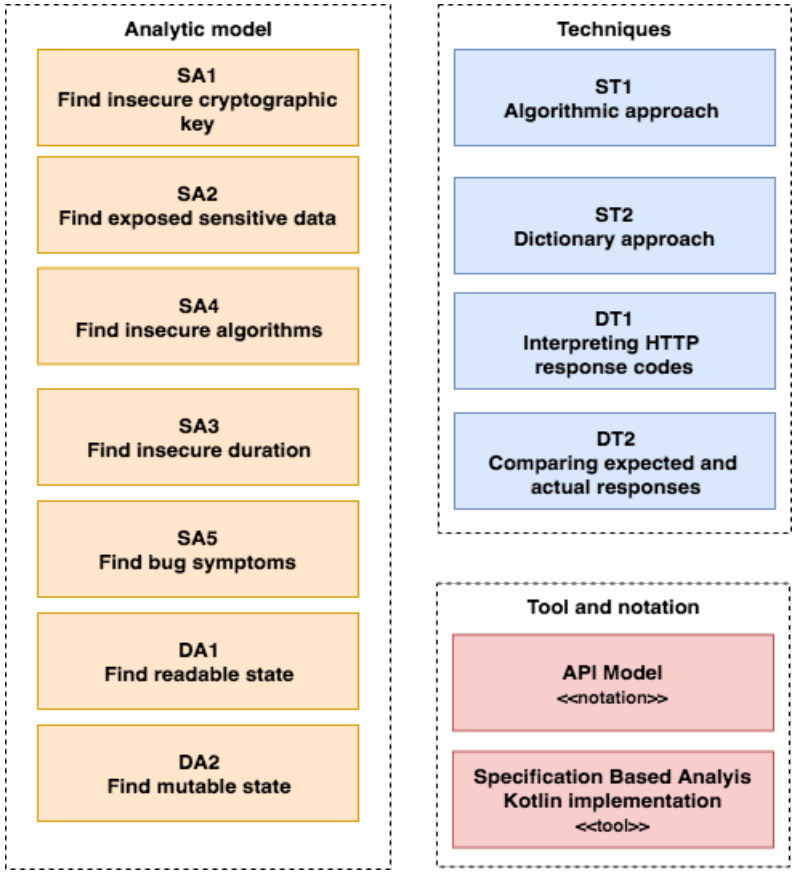

Figure 7.1: Thesis contributions

7.2 Validating the techniques

In order for the tool to work, the individual techniques must work and provide meaningful feedback. To validate that the techniques indeed work, I have used real world scenarios or artificial examples where I could not find real world scenarios, to validate that it is possible to use the techniques to find multiple security vulnerabilities.

7.3 Validating the tool and API model

As the tool and API model go hand in hand (that is, the implementation of the tool is heavily reliant on the API model notation) it is natural to validate them together. The following subsection each makes efforts to validate not only the tool and notation, but first and foremost each of the specification based analyses. For each of the analyses there is an

implemented algorithm, which utilises one or more techniques to conduct the Specification Based Analysis on an API specification which coheres to the notation introduced in section 4.7. Keep in mind that the tool itself is not an essential contribution of this thesis, but rather a mean of validating the efforts made with regards to the analytic model and techniques described earlier.

7.3.1 Finding vulnerability symptoms

This validation is based on a real world vulnerability found in T-Mobile's system, which is described in an online article[1]. According to the article, T-Mobile claims that they have no evidence that the vulnerability had affected any customer accounts. This does, of course, not mean that the vulnerability was not exploited. In October 2015 another vulnerability in the T-Mobile system led to a massive data breach, which affected around 15 million T-Mobile users. While the secondly mentioned vulnerability is not examined in this example, the impact it had shows how important it is to find these vulnerabilities before malicious hackers does.

The vulnerability is an example of the B2 - Bypassing access control vulnerability, described in section 3.2.2, and the tool utilises the analysis techniques described in section 4.2 to discover the potential vulnerability.

Detailed vulnerability description

In this security vulnerability, both an access token and a Mobile Station International Subscriber Directory Number (MSISDN) was passed as query parameters from the client to fetch some data related to the user executing the request. By changing the MSISDN, attackers were able to access sensitive data corresponding to any valid MSISDN known to the attacker. The attackers also demonstrated how easy it was to find someone's MSISDN.

The following steps describes how an attacker is able to exploit this vulnerability:

1. The endpoint `/user/lines` takes two query parameters: *access_token* and *MSISDN*. As we have seen earlier, this is not secure usage of query parameters, though no direct security vulnerability itself.

2. The attacker is able to find a target MSISDN through a Google search, and replaces the *MSISDN* query parameter with the new target MSISDN.

3. After submitting the query, the attacker is presented with sensitive user information related to the target MSISDN, such as the target's email.

[1] `https://arstechnica.com/information-technology/2017/10/t-mobile-website-bug-apparently-exploited-to-mine-sensitive-account-data/`

Using the tool to find this vulnerability

Note that the tool might produce false positive results for this type of vulnerability, it is only able to *suggest* that there might be a vulnerability present.

Setup

1. Add a new API specification by creating an empty JSON file.
2. For this validation, the rootUrl and userLevel fields are not relevant. They need, howvever, to be present so the program is able to parse the JSON.
3. Add an entry to the endpoint list.
4. For the newly added endpoint specify the relative path `/user/lines`.
5. Add a new endpointMethod.
6. Specify GET as HTTP method.
7. Add the following following entries to its query parameter array: ”access_token” and ”MSISDN”, as shown in listing 7.1.

```
{
  "rootUrl": "https://wsg.t-mobile.com/permissionManagement/v1",
  "userLevels": [],
  "endpoints": [
    {
      "relativePath": "user/lines",
      "endpointMethods": [
        {
          "httpMethod": "GET",
          "queryParameters": [
            "access_token",
            "msisdn"
          ]
        }
      ],
      "fields": []
    }
  ]
}
```

Listing 7.1: Simplified T-Mobile API specification

Execution

1. Create an ApiSpecification implementation by invoking the TestUtil function *parseApiModelFromPath* with the specification’s full path, as shown in listing 7.2.
2. Create a dictionary by specifying a set of strings. Make sure the set contains ”token” or ”access_token”, as shown in listing 7.3.

3. Invoke *executeTest* in ApiModelTestExecutor.

```
@BeforeClass
fun setup() {
  val apiSpecification = parseApiModelFromPath(...)
}
```

Listing 7.2: Specification implementation

```
val queryParameterDict = setOf(
      ...,
      "token"
    )
```

Listing 7.3: Dictionary implementation

As shown in the output of the tool (listing 7.4), the tool is able to warn developers about query parameter usage that seems insecure. Note that it should be fairly easy to change output method from a log in the console e.g. to a generated report.

```
user/lines
    GET
      Query parameter: access_token DANGEROUS (testSuite match: token)
      Query parameter: msisdn not dangerous
```

Listing 7.4: Test output

7.3.2 Finding weak cryptographic key

This validation is based on an fictive scenario where JWTs are signed using a weak cryptographic key, as described in section 3.2.1.

Detailed vulnerability description

In this imaginary scenario, the developer has used the key 'secret' to sign JWTs sissued by the API, and forgotten to change this weak key after deploying to production. This has introduced a big security vulnerability in the API, practically compromising the integrity of the entire authorisation protocol.

The following steps describes how an attacker is able to exploit this vulnerability:

1. The attacker discovers that JWTs are signed using a weak key by conducting a dictionary attack, as described in section 4.3
2. The attacker is now able to sign their own JWT, which in practice allows them to impersonate any person in the system.
3. The uses the fake JWT to authorise requests to otherwise access restricted resources, or could even get admin access if the same authorization is used for Admin related actions.

Setup

If you want to configure and experiment with the tool yourself, here are the step by step directions. If you just want to execute the test and see the output, see *Running the predefined test setup* at the end of this section.

1. Obtain a JWT with a known signature, e.g. from `https://jwt.io/`[2]. Make note of the signing key you choose, it will be needed later.
2. Copy the JWT into a new API Specification file (or use an existing one). Make sure that the specification contains all the required fields so the program is able to parse it.

Execution

1. Create a dictionary by specifying a set of strings. Make sure the set contains ”token” or ”access_token”.
2. Create a new instance of the StaticJWTTestExecutor and pass an instance of StaticJWTTestConfiguration. The StaticJWTTestConfiguration has default values, so only override the field cryptographicKeyDictionary to ensure that the list contains the previously selected signing key.
3. Invoke the *testTokenSecret* and pass a JWT

Running the predefined test setup

Invoke *JWT spoofing test* which is located in
`src/test/kotlin/validation/JWTSPoofingTest.kt`.

The output is shown in listing 7.5. The tool iterates over each *userLevel*-entry in the Specification, and tests the access and refresh token for each user level. If a token is compromised during the analysis the developer is warned, here in the form of a log to the console.

```
admin
  Weak access token secret
    Vulnerability description: Developers are notified if the JWT is
        signed using a known secret
    Error: The JWT is signed using the known secret 'secret'
  Weak refresh token secret
    Vulnerability description: Developers are notified if the JWT is
        signed using a known secret
    Error: The JWT is signed using the known secret 'secret'
user
  Weak access token secret
    Vulnerability description: Developers are notified if the JWT is
        signed using a known secret
```

[2]The implementation supports the algorithms defined in the JSON Web Algorithms specification [30]

```
11      Error: The JWT is signed using the known secret 'secret'
12    Weak refresh token secret
13      Vulnerability description: Developers are notified if the JWT is
            signed using a known secret
14      Error: The JWT is signed using the known secret 'secret'
```

Listing 7.5: JWT Spoofing test output

7.3.3 Finding exposed sensitive data

Detailed vulnerability description

As discussed in section 3.2.1, the JWT payload is just base64-encoded JSON data. As such, the JWT payload should not contain any sensitive data, as this might end up logged on the server, and is also visible to anyone viewing the JWT itself, even if the JWT is no longer valid.

Setup

If you want to configure and experiment with the tool yourself, here are step by step directions. If you just want to execute the test and see the output, see *Running the predefined test setup* at the end of this section.

1. Obtain a JWT, e.g. from `https://jwt.io/`. In the payload field on the right hand side you can specify custom payload entries as JSON. Add sensitive data fields, such as email. Make note of the label used to indicate the field, as it will be needed later.
2. Copy the JWT into a new API Specification file (or use an existing one). Make sure that the specification contains all the required fields so the program is able to parse it.

Execution

1. Create an ApiSpecification implementation by invoking the TestUtil function *parseApiModelFromPath* with the specification's full path.
2. Create a dictionary by specifying a set of strings. Make sure the set contains the labels specified in the JWT creation earlier.
3. Invoke *testTokenPayload* by passing the JWT and a descriptive name for the token[3]

[3]The token type description parameter is used to distinguish test results from tests where several types of tokens are tested at the same time, so if you're only testing one token type it does not really matter.

Running the predefined test setup

Invoke *Exposure of sensitive data test* in `src/test/kotlin/validation/ExposedSensitiveDataTest.kt`.

```
admin
  Dubious payload field
    Vulnerability description: JWTs are not encrypted, and should not contain sensitive information
    Error: access token contains the dubious field(s) 'email', ensure that this is intentional
  Dubious payload field
    Vulnerability description: JWTs are not encrypted, and should not contain sensitive information
    Error: refresh token contains the dubious field(s) 'email', ensure that this is intentional
user
  Dubious payload field
    Vulnerability description: JWTs are not encrypted, and should not contain sensitive information
    Error: access token contains the dubious field(s) 'email', ensure that this is intentional
  Dubious payload field
    Vulnerability description: JWTs are not encrypted, and should not contain sensitive information
    Error: refresh token contains the dubious field(s) 'email', ensure that this is intentional
```

Listing 7.6: Exposed sensitive data test output

7.3.4 Finding insecure duration

Detailed vulnerability description

As discussed in section 3.2.1, JWTs should expire within a reasonable time frame. In this example scenario, a developer has set the expiration time of the access token so high that it practically never expires.

Setup

If you want to configure and experiment with the tool yourself, here are step by step directions. If you just want to execute the test and see the output, see *Running the predefined test setup* at the end of this section.

1. Obtain a JWT, e.g. from `https://jwt.io/`. Make sure that the fields ”iat” and ”exp” are present in the payload to the right. Edit the ”exp” field so that it is over 1 hour for access tokens or over 60 days for refresh tokens (hover the field to see the actual date”.

2. Copy the JWT into a new API Specification file (or use an existing one). Make sure that the specification contains all the required fields so the program is able to parse it.

Execution

1. Create an ApiSpecification implementation by invoking the TestUtil function *parseApiModelFromPath* with the specification's full path.
2. Invoke *testAccessTokenExpiration* by passing the access or refresh token.

Running the predefined test setup

Invoke *Insecure expiration time* in `src/test/kotlin/validation/InsecureDurationTest.kt`.

```
admin
  High or missing access token expiration
    Vulnerability description: Developers are notified if JWT does not expire within a reasonable time frame
    Error: Access token expiration is 7916666 hours. Anything over 1 hours is considered bad practice.
  High or missing refresh token expiration
    Vulnerability description: Developers are notified if JWT does not expire within a reasonable time frame
    Error: Access tokens expiration date is 329861 days. Anything over 60 is considered bad practice.
user
  High or missing access token expiration
    Vulnerability description: Developers are notified if JWT does not expire within a reasonable time frame
    Error: Access token expiration is 7916666 hours. Anything over 1 hours is considered bad practice.
  High or missing refresh token expiration
    Vulnerability description: Developers are notified if JWT does not expire within a reasonable time frame
    Error: Access tokens expiration date is 329861 days. Anything over 60 is considered bad practice.
```

Listing 7.7: Insecure expiration time test output

7.3.5 Finding unsecured JWT

Detailed vulnerability description

As discussed in section 3.2.1 - *JWT4 - Unsecured JWT*, many JWT libraries had a very severe vulnerability where they would accept unsigned JWTs even if they were configured to use a specific signing algorithm. There is really no reason to use unsigned JWTs for authentication purposes, and developers needs to be notified if this is happening.

Setup

If you want to configure and experiment with the tool yourself, here are step by step directions. If you just want to execute the test and see the output, see *Running the predefined test setup* at the end of this section.

1. Obtain a JWT, e.g. from `https://jwt.io/`. Change the header value of the ”alg” field to ”none”.

Execution

1. Invoke *testUnsignedToken* by passing the earlier created JWT.

Running the predefined test setup

Invoke *Unsigned token* in
`src/test/kotlin/validation/UnsignedJWTTest.kt`.

```
1 No unsigned token
2     Vulnerability testDescription: Secure JWTs must be signed using one of
          the known algorithms.
3     Error: JWT header field alg is none, token is not secure
```

Listing 7.8: Unsigned token test output

7.3.6 Finding missing access restriction for readable resources

This vulnerability validation is based on a real scenario from a project I’ve worked on. The scenario is altered to ensure privacy, though it preserves the essence of the problem found in the real system.

Detailed vulnerability description

Imagine a digital quiz application, where users are able to sign up for various quizzes. Any user participating in the quiz should answer a set of multiple choice questions. A client can access the `/quiz` endpoint in order to retrieve the quiz, including a list of questions for each quiz, and display them to a user. Unfortunately, the backend developer has forgotten to sanitise the output of the endpoint. Consider the question model shown in listing 7.9. Although the client application is not making use of the correctAnswerId in any way, the backend developer forgot to remove this (sensitive) information from the quiz object before showing it to regular users, allowing regular users to access this information through the usage of a HTTP proxy tool or simply by inspecting the HTTP requests in their browser. This means that any savvy user is able to know the correct answer to any question in the quiz simply by querying the server.

```
"questions": [
    {
      "text": "What is 2*2",
      "options": [
        {
          "id": 1,
          "text": "4",
          "isCorrect": true
        },
        {
          "id": 2,
          "text": "2",
          "isCorrect": false
        }
      ],

    }
  ]
```

Listing 7.9: Quiz endpoint return data

Setup

If you want to configure and experiment with the tool yourself, here are step by step directions. If you just want to execute the test and see the output, see *Running the predefined test setup* at the end of this section.

1. The first step of the setup is to make an implementation of the API interface. It should not matter whether this implementation relies on actual HTTP requests to a remote API or is just a local mock—the test executor relies on the interface and not an implementation.
2. Implement the *get* method in the new implementation so that it returns data for the relative URI `/quizzes`, e.g. as listing 7.10 shows.
3. Now configure the API specification, see 7.11. The important part here is that the endpoint property *relativePath* must match the when statement cases in the implementation (see how line 6 in listing 7.10 and 21 in 7.11 matches.
4. Finally, we need to implement the access restriction (or maybe we want to validate behaviour under its absence?). Configuring this step manually requires some knowledge of JWT. In `src/main/kotlin/api/MockBuggedAccessLevelApi.kt` the two methods *validateAccessToken* and *validateAdmin* should take care of this. Implement these to satisfy your authorisation requirements.

```
override fun get(relativePath: String, accessToken: String): ApiResponse {
        if(!validAccessToken(accessToken)) {
            return ApiResponse.Error(HttpMethod.GET, ResponseCode.
                UNAUTHORIZED, "Invalid token")
        }
        return when (relativePath) {
            "quizzes" -> ApiResponse.Success(HttpMethod.GET, Klaxon().
                toJsonString(quizzes))
```

```
            else -> ApiResponse.Error(HttpMethod.GET, ResponseCode.
                NOT_FOUND, "Endpoint not found in MockRepository")
        }
    }
```

Listing 7.10: Get implementation

```
{
  "rootUrl": "https://api.quizapp.com/v1",
  "userLevels": [
    {
      "name": "admin",
      "jwt": {
        "accessToken": ...,
        "refreshToken": ...
      }
    },
    {
      "name": "user",
      "jwt": {
        "accessToken": ...,
        "refreshToken": ...
      }
    }
  ],
  "endpoints": [
    {
      "relativePath": "quizzes",
      "endpointMethods": [
        {
          "httpMethod": "GET",
          "queryParameters": []
        }
      ],
      "fields": []
    }
  ]
}
```

Listing 7.11: Quiz API specification

Execution

1. Invoke *test readable fields* in **BuggedAccessScopeTest.kt**

Running the predefined test setup

Invoke *Test readable fields* in
`src/test/kotlin/validation/BuggedAccessScopeTest.kt`.

```
quizzes
  GET
```

```
3     admin
4       id is readable
5       name is readable
6       questions is readable
7     user
8       id is readable
9       name is readable
10      questions is readable
```

Listing 7.12: Readable resource test output

7.3.7 Finding missing access restriction for writable resources

This is another vulnerability based on a real scenario. Again the domain is altered to ensure privacy.

Detailed vulnerability description

Let's continue examining the quiz API described in section 7.3.6. For this validation, imagine that students are able to deliver assignments. Assignments are created on the server by POSTing on the `/assignments` endpoint. For this example, imagine that all the student has to do in order to answer an assignment is to submit some text. The system also needs a way to check the status of an assignment. For this scenario, the statuses NEW, PASSED and FAILED are introduced as an enum on the server.

Through intended usage of the server, i.e. through the user interface created by the developers, there is no way for a user to change this status. However, if a user is using an HTTP proxy tool, they are able to set this status to whatever they want it to be, due to lack of input sanitation on the server.

Setup

If you want to configure and experiment with the tool yourself, here are step by step directions. If you just want to execute the test and see the output, see *Running the predefined test setup* at the end of this section.

1. Make an implementation of the API interface, as described in section 7.3.6.
2. Similar to the read based implementation shown in 7.3.6, an endpoint has to be added. This time the relative URI should be `/assignments` and the HTTP method has to be POST. See listing 7.13.
3. Similar to the other configuration, a new endpoint must be added to the API specification JSON file. Set the *relativePath* attribute to assignments, as listing 7.14 shows. For this scenario it is also important to specify the fields, so that the tool knows how to generate the input for the given endpoint.

```
override fun post(relativePath: String, accessToken: String, requestBody:
    JsonObject): ApiResponse {
        if(!validateAccessToken(accessToken)) // return unauthorized
        return when (relativePath) {
            "assignments" -> {
                val jsonString = requestBody.toJsonString()
                val parsedAssignment = Klaxon().parse<Assignment>(
                    jsonString)!!
                val sanitizedAssignment = parsedAssignment.copy()
                ApiResponse.Success(HttpMethod.POST, Klaxon().toJsonString
                    (sanitizedAssignment))
            }
            else -> ApiResponse.Error(HttpMethod.POST, ResponseCode.
                NOT_FOUND, "Endpoint not found in MockRepository")
        }
    }
```

Listing 7.13: Post implementation

```
{
  "rootUrl": "https://api.quizapp.com/v1",
  "userLevels": [
    {
      "name": "admin",
      "jwt": {
        "accessToken": ...,
        "refreshToken": ...
      }
    },
    {
      "name": "user",
      "jwt": {
        "accessToken": ...,
        "refreshToken": ...
      }
    }
  ],
  "endpoints": [
    ...,
    {
      "relativePath": "assignments",
      "endpointMethods": [
        {
          "httpMethod": "POST",
          "queryParameters": []
        }
      ],
      "fields": [
        {
          "name": "text",
          "type": "string"
        },
        {
          "name": "assignmentStatus",
          "type": "number"
        }
      ]
```

```
39      }
40    ]
41  }
```

Listing 7.14: Quiz API specification

Execution

1. Create a new instance of the API Specification as shown in listing 7.2.
2. Create a new instance of *DynamicApiModelTestExecutor* by passing in the API specification and an instance of the API interface (see **setup** for further instructions regarding the API interface implementation).
3. Invoke *executeDynamicApiModelTest()* on the *DynamicApiModelTestExecutor* instance, which returns a list of test results. These test results could be iterated over, e.g. to log to the console or produce a written report.

Running the predefined setup

1. Invoke *test writable fields* in **BuggedAccessScopeTest.kt**

Chapter 8

Related work

In this chapter, I present relevant state of the art that is directly related to my contributions. I want to discuss the related work and how it compares to my work in the context of my research hypothesis, with focus on the following properties:

1. How easy is it to use?
2. How difficult it is to implement?
3. What vulnerabilities can it be used to discover?

Figure 8.1 depicts the related work discussed and how it relates to the contributions.

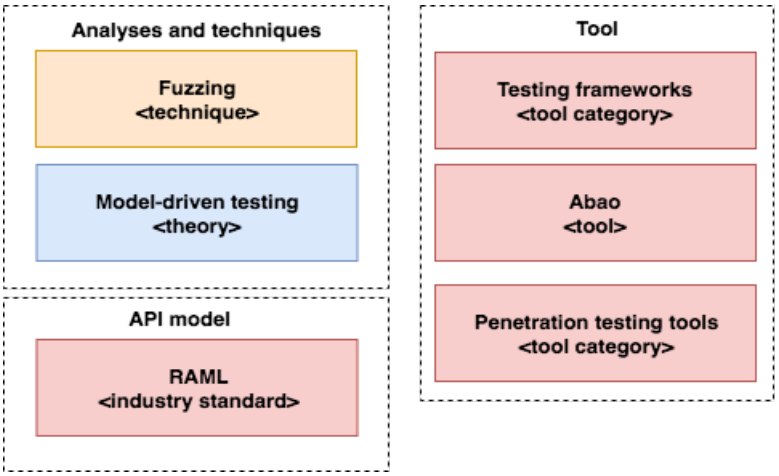

Figure 8.1: Chapter overview

8.1 Analyses and techniques

8.1.1 Fuzzing

Fuzzing is a black-box testing technique which essentially finds implementation vulnerabilities related to improper handling of unexpected inputs [31, 32]. According to OWASP [31], fuzzing typically finds vulnerabilities which cause buffer overflow or denial of service (which often are vulnerabilities that are tightly related to implementation languages and frameworks).

While the mutable state analysis (described in section 5.3) might seem fuzzing-like in that it uses random input data, there is an important distinction: The focus of the mutable state analysis is not to find language or framework-specific vulnerabilities, but rather to figure out whether or not there is a vulnerability related to missing or insufficient access control. However, if the random input causes the server to crash, i.e. return a 500 HTTP response or request time out, it should obviously trigger a warning.

Fuzzing is a technique that typically involves using a specialized tool, but developers should also be able to implement their own fuzzing library if they want to. As fuzzing is simply a test input generation technique, the developer must still specify test cases to successfully use it. Fuzzing is able to discover vulnerabilities the developer might not be aware of.

8.1.2 *Model-driven testing of RESTful APIs*

In 2015, T. Ferting and P. Braun wrote a paper with this title which investigates how to use an abstract model to generate test cases for RESTful APIs [33]. The process of generating test cases from a model is often referred to as model-driven testing (MDT). While Ferting and Braun's paper is not strictly security-oriented, it does present security as one of the testing categories it is able to produce test cases for.

An important difference between the approach Ferting and Braun takes and my approach is that my tool does not actually generate any code. Instead, the test cases are dynamically generated during the test execution. A benefit of my approach is that the user of the tool does not have to interact with, or be aware of, any generated code. A disadvantage of not generating code is that users of the tool are unable to customise or change the test cases should this be of interest to the developer. It was, however, never a priority or focus to produce a tool that generated code, as we want to keep programmer effort modest.

8.2 REST API modelling

Basic REST API modelling concepts are introduced in section 2.7. This section will focus on the state of the art in REST API modelling languages and tooling.

8.2.1 RAML

RAML is, according to the introduction on Github *"...a language for the definition of HTTP-based APIs that embody most or all of the principles of Representational State Transfer (REST)."* [34]. This description goes very well with my intents of utilising basic REST principle in order to conduct automated specification based security analysis.

When starting the work on this thesis, it quickly became apparent that it would be beneficial to use an API model as a foundation for the automated testing. Ferting and Braun [33] shows that it is feasible to generate tests from an API model. Both RAML, Swagger and API Blueprint were considered, and RAML was deemed the most relevant one, mostly due to how concise it is and how RAML is YAML based. YAML a super-set of JSON, which should be a familiar language to most web developers.

After some attempts at utilising RAML in the proof of concept tool, I realised that RAML seemed too detailed and thus not a good fit for what the thesis required—I would quickly become focused on RAML specific implementation details rather than the bigger thesis picture. After some quick experimentation I was able to come up with a very simplified API model which was a better fit for my needs, as it allowed me to focus on my contributions rather than being stuck with irrelevant RAML-specific implementation details.

8.3 The proof of concept tool

8.3.1 Testing frameworks

Ferting and Braun discuss many different testing frameworks in their related work section [33, p. 2]. From their paper, I was unable to find that Ferting and Braun contributed a testing tool themselves. They did, however, develop a test specification language, and in their experimental validation they mention that they have actually executed tests with their 'system'. In their summary section they mention how they were unable to provide penetration testing with their approach. Following is a summary of some relevant testing frameworks.

xUnit

xUnit is the collective name for several unit testing frameworks that derive their structure and functionality from Smalltalk's SUnit [35]. As the xUnit family's goal is to make it easier to write and execute unit tests manually, the family as a whole is not directly relevant to automated testing of web services. Unit tests also focus on testing very small building blocks of code, which is neither particularly relevant when we want to find vulnerabilities in web services—we are interested in the bigger, more abstract, picture.

SoapUI

Another tool Ferting and Braun discuss is SoapUI[1], which is a commercial API testing tool. SoapUI is designed to configure test cases for service oriented architectures, but due to the fact that it is configured through a GUI it is not feasible to use it for automated test case generation.

RestAssured

RestAssured[2] is a framework implemented in Java designed to make manual testing of RESTful APIs quicker and easier. Again, the problem with this framework is that it focuses on improving manual testing, which in essence is not related to automated test generation.

Test-The-Rest

Test-The-Rest (TTR) is described by S. Chakrabarti and P. Kumar in their paper *Test-the-REST: An Approach to Testing RESTful Web-Services* [36]. Chakrabarti and Kumar present a model driven approach to automatic testing of RESTful web services, which is indeed the same approach as I have taken in my work. They propose to use a XML-based specification to generate a test suite which is very behaviour focused, i.e. that a response is on the expected format and that all the possible combinations of query parameters are supported. While the researchers claim that their tool is built to be extensible and adaptable, I was unable to find ways to add security related constraints, such as authentication and authorisation, to their specification. The paper does not focus on security aspects. In addition, their proposed approach to test generation still requires the user of the tool to specify test cases. The test cases is then used to generate various possible combinations of the variable inputs, such as URI parameters and query parameters.

Summary

The xUnit family is very easy to get started with. Unfortunately, in the context of API security testing, it is a cumbersome and error prone approach to manually define test cases. It is suggested to use libraries or frameworks, e.g. RestAssured, on top of xUnit to conduct testing over HTTP in order to reduce overhead related to handling HTTP related properties such as asynchronous behaviour and parsing of response data. That being said, it should be possible to find all of the bugs described in this thesis through usage of any xUnit member, though it would require the developer to be knowledgeable in the field of software security as well as a fair share of manual effort for each test case.

[1] `https://www.soapui.org/`
[2] `https://github.com/rest-assured/rest-assured`

8.3.2 Abao

Abao, according to its Github README, is *"[...] a command-line tool for testing API documentation written in RAML format against its back-end implementation"* [37]. Abao is a relatively small tool, where the source code itself is 963 lines of code (3936 if we include the tests). Three of the features Abao promises is the verification of query parameters, HTTP request bodies and HTTP response bodies all are supported in service, i.e. at the system's runtime. Abao uses a RAML specification to generate HTTP requests and validates that the server behaves as described in the specification.

Abao uses similar techniques to the ones I discuss in chapter 5, though with no focus on security. Abao uses a RAML specification to generate a set of requests which are sent to the API. Each of the response properties, such as HTTP response codes and response bodies are then analysed, and compared with the properties specified in the RAML specification. This analysis ensures that the API is up to date with the RAML specification, i.e. behaves in the way the RAML specification states. My analysis takes this a step further, and find unintended security related behaviour as well, where Abao only is able to validate defined expected behaviour.

8.3.3 Penetration testing tools

A common problem with today's penetration testing tools is that they require knowledge of software security, and in general the more you know the better you are going to be at using the tool. This sets a high entry barrier for developers without formal security training. In this section I'll describe some of the most popular penetration testing tools in today's market.

Metasploit

Metasploit is, according to their web site [38], *"The worlds most used penetration testing framework"*—and with over 12 000 stars on Github[3] they have a solid argument. Metasploit is an open source penetration testing framework targeting software security experts. It helps the expert automate parts of the penetration testing process, with everything from brute force attacks to social engineering.

It is important to note that even though Metasploit promises automation, the tool still needs a lot of configuration from a security knowledgeable developer. It is only after configuring the tool the developer gains the benefits of continuous automation.

Core Impact

Core Impact is a commercial penetration testing tool. Due to the paywall, I was unable to dig any further than what is present on their web site [39] as well as available in online

[3] `https://github.com/rapid7/metasploit-framework`

reviews [40, 41].

Similarly to Metasploit, Core Impact seems to be focused on security professionals, and requires its users to specify individual test cases. Both of these properties adds to the minimum required programmer effort needed to use the tool.

Burp Suite

Though not classified as a penetration testing tool, Burp Suite [42] is a tool which is commonly used by penetration testers. The tool offers automation for penetration testing related activities such as information gathering and other penetration testing utilities such as proxying. Yet again we see a testing tool aimed at software security professionals.

Summary

While all of the penetration testing tools mentioned here provide automated security testing, they still require developers to specify individual test cases. This violates my constraint of 'keeping programmer effort modest' and requires that the developer is knowledgeable within software security.

Another key takeaway from the tools mentioned is that they all cost money. Burp Suite does have a free version, but it only promises 'essential manual tools', which does not satisfy the 'automated' constraint. When a smaller development team has to pay substantial amounts to be able to set up automated security testing, the chances of them doing it proportionally decreases with the cost of the product. Having an open-source alternative to these large commercial tools would greatly benefit smaller companies.

Chapter 9

Discussion and conclusion

The overarching goal of this chapter is to discuss my contributions. This includes a general overview of each contribution, as well as notes regarding limitations as well as suggested future work based on these limitations. Finally, I argue, based on the contributions and their validation, that I have found satisfactory answers to the research questions in section 1.3.1 and strong evidence to support the research hypothesis in section 1.3.

9.1 The analyses

Although the results produced by the analyses can be found manually today, it is the approach to obtaining those result that is the real novelty of my contributions. The two core features of this contribution is:

- Developers does not having to specify individual test cases to test for security vulnerabilities
- Developers using a tool based on the analyses do not need to be software security experts.

I believe that the proposed approach to automated security testing could help lower the entry barrier of security testing.

9.1.1 Future work

Interesting future work would be to investigate what other security vulnerabilities it is possible to find based on the API model or an extension of it. Here are some concrete ideas for interesting subjects:

- File uploading—The possibility to upload malicious files is a common security weakness, and could result in vulnerabilities with huge impact. As of such, I would suggest to investigate generation of test cases for this vulnerability.
- Only HTTPS traffic—Secure APIs must use only accept HTTPS communication. As of such, I suggest to explore possibilities around automatic request generation for non-HTTPS based communication.

9.2 The techniques

The techniques I contribute boils down to four key contributions:

1. The algorithmic analyses, section 4.2.1—An analysis category that is based on implementing a standard on the client side and using this to statically test for security vulnerabilities.
2. The dictionary based analyses, section 4.2.2—An analysis category that is based on defining a set of known vulnerabilities, e.g. a list of query parameters which are often misused in a way that leads to vulnerabilities.
3. Interpreting HTTP response codes, section 5.1.1—A technique which involves using HTTP status codes to inspect what happens on the server, e.g. finding endpoints without access restriction and possible server side implementation flaws (server returning 500), during program runtime.
4. Comparing expected and actual responses, section 5.1.2—A technique that inspects what fields each user-level is able to read from each endpoint.

9.2.1 Limitations

The dynamic analysis can produce both false positives and false negatives. This has to do with the fact that an endpoint might accept a certain set of values for a given field. This is best described as field enumeration. Imagine an endpoint creating a new user, POST `/users`. The response contains various fields, including the field *userLevel*. The userLevel field is enumerated, and it can have the values 0, 1 and 2—respectively meaning USER, MODERATOR or ADMINISTRATOR (Enumerated values are often passed as integers in JSON). Now imagine executing a POST request to `/users`, with the userLevel field set to 0. The server side implementation defaults this value to 0, to prevent users from signing up as administrators. However, the mutable state analysis is not able to distinguish between this scenario, and the scenario where the field is mutable. The analysis would present this as a **false positive mutable value**.

Similarly, an enumerated field might be mutable, but only as long as a valid enumerated value is passed. Imagine that admins are able to set other users' access levels by using the PUT verb om the `/users` endpoint. The request is issued with the userLevel value set to 42, but as we know the implementation only accepts 0, 1 and 2. An analysis of this request

is going to give a **false negative mutable value**. The field is indeed writable but the server is going to default to the existing value when an invalid value is provided.

The suggested future work to be made related to this limitation is in the implementation layer rather than in the technique itself; generating random input for each field would greatly reduce the chance of finding these false positive and false negative cases.

9.3 The API model

To utilise an API model to generate test cases is no novelty, as Ferting and Braun [33] as well as Chakrabrtin and Kumar [36] already have shown. However, the approach of adding a realistic authentication and authorisation scheme (where Ferting and Braun is using BasicAuth, and Chakrabrtin and Kumar does not mention authorisation) as well as focusing fully on the security aspect is what separates my contributions from the previously mentioned work.

9.3.1 Limitations and future work

Not using industry standard API model tools

One could argue that using an industry standard like RAML or Swagger would increase the usability of the tool on average. In the cases where the developer utilising the tool already has a RAML or Swagger specification it is obviously going to reduce effort, while having a custom, light weight, language is going to reduce effort for any developer which is not already familiar with RAML or Swagger.

Interesting further work on this issue is to work on the API model and make it more abstract, i.e. not rely on JWT . Ideally, it should be possible to make an analysis tool that supports multiple API modelling languages and then apply a model-to-model transformation, e.g. as described in the book Model-Driven Software Engineering in Practice [43, p. 107-123]. On an abstract level this should not be the biggest change, the API model would need to have the *userLevel*-concept reworked, so that various authentication implementations could be supported. This change would allow developers to contribute implementations for their favourite API model tool.

Supporting complex access graphs

The approach taken to model access control is very simplified—almost boolean. In many real world applications this is not the case. Typically a user's ability to access a given resource will be a more complex conditional statement than `if user.isAdmin`. In my early work I made some thought experiments around this issue, but it seemed like a very

complex[1] problem to solve, so complex that I rather decided to focus on the essence of specification-based security analysis—the analyses.

From my point of view, the largest problem related to this issue is related to how much manual effort the user of the API model is required to provide manually, either in the form of JWTs, or credentials such as usernames an passwords. As of such, it would be interesting to conduct a further study on this issue and whether it is possible to find a better approach to model access control for non-trivial applications.

9.4 Proof of concept tool

While the tool contributed is not the main focus of the thesis, it is important to state that it is also capable of conducting light weight automated penetration testing. This shows that it is possible to introduce automated penetration testing into the software development life cycle, which I briefly introduced in section 2.3.5. This also relates to the bonus software security touchpoint mentioned by McGraw, as he states that *"Software security touchpoints are best applied by people not involved in the original design"*, which automatic tools most definitely are not.

9.4.1 Limitations and future work

Note that any limitations related to the API model and analyses will extend to the proof of concept tool, as the tool itself is an implementation of the API model and analyses

Hard-coded implementations of interchangeable standards

In general, I would say that the most prominent limitations with the proof of concept tool is related to hard-coded dependencies on various standards. Ideally, a tool should be very extensible, which means that all of the implementation-specific details such as the ones towards JWT and JSON should be 'pluggable', i.e. not a direct dependency but abstract concepts such as *authorisation* and *data format*.

The proof of concept tool is hard-coded to rely on JSON based data structures. As most modern API's (e.g. Facebook [1], Spotify [2] and YouTube [3] mainly supports this data structure, I chose this approach in order to be able to focus on the high-priority contribution that is the analyses. However, unarguably, the tool should ideally support dynamic parsing of various data formats, e.g. through usage of the HTTP *Content-Type* header.

Similarly to the hard-coded data format limitation, the tool now utilises hard-coded authorisation in the form of JWT. While JWT is achieving broad adoption in the industry,

[1] An extreme example of complex authorisation is the Facebook API, where there is incredibly many combinatorial ways users can give other users and applications access to their data. Even in simple applications I have worked with it quickly becomes a business requirement for users to be able to explicitly grant or revoke the access rights to some resource for other users.

there are still many web services that use session-based authentication or other forms of token-based authentication.

Ideally these implementations should be abstracted away, and dynamically chosen depending on the API model. This obviously goes hand in hand with the API model as well, the model must also support various standards.

Generate input data

As of now the tool is very primitive in the regards of generating input data—it only supports Strings and Integers, and returns the same value every time for every type. The input data could be generated using a Fuzzing Tool such as QuickCheck[2].

9.5 Conclusion

9.5.1 The research questions

The research questions introduced in section 1.3 has guided the way through the theoretical parts, i.e. chapters 3, 4 and 5, and it is now time to conclude on these research questions. I will refer to them in the order they are listed in the introduction.

RQ1 - Static specification based analysis

This research question reads:

Is it possible to statically analyse a REST API specification in order to find common REST related security vulnerabilities?

Through the validation effort made in section 7.3.1, which builds upon the theory in sections 4.2 and 4.7, With regards to what I have shown in the validation, chapter 7, I can confidently say that this research question can be answered affirmatively—as the example showcased with the proof of concept tool is able to warn the developer about the looming potential vulnerability in the T-mobile production API.

RQ2 - Dynamic specification based analysis

This research question reads:

Is it possible to use the same API specification to generate a set of dynamic tests that finds vulnerabilities in the systems runtime?

[2] `https://github.com/pholser/junit-quickcheck`

Chapter 5.1 delves into this research question and suggests a way to utilise the same API model which was introduced earlier to generate a set of dynamic test cases which executes against the API during it's runtime. The validation sections 7.3.6 and 7.3.7 show how the tool is able to interact with an API during runtime, and find potential vulnerabilities related to read and write permissions.

This is, in my opinion, the most fragile part of the contributions in this thesis. The dynamic test is fairly hard coded, and also incomplete. While I would argue that the research question is proven true, I want to state that I think this research question is a bit vague, and maybe is covering too much.

RQ3 - Analysing JWT

This research question reads:

Is it possible to automatically test for common JWT vulnerabilities?

The research question itself is used as a base for section 3.2.1, and the implementation is discussed in section 4.3 through 4.6. All together I found four ways to analyse JWTs and test for possible vulnerabilities. This means that the research question can be answered in a positive manner.

9.5.2 The hypothesis

Throughout this thesis, I have been working towards the goal of verifying my hypothesis, as introduced in section 1.3:

Automated specification based security testing can be used to find critical security vulnerabilities in large real world applications, while keeping programmer effort modest.

In order to prove this hypothesis, I must validate the following partial steps:

1. Programmer effort must be modest.
2. Test generation must be automatic.
3. I must showcase that I am able to use the analyses, techniques and specification introduced in this thesis to find critical security vulnerabilities in real world applications.

The third statement is validated in chapter 7, so I am left to prove statement one and two:

Programmer effort must be modest

As shown in chapter 8, most of today's testing tools rely on the user to specify individual test cases. This work is not only tedious but also error prone, as manual effort often is.

By having the programmer only specify the API model, and nothing more—I am confident that programmer effort indeed could be categorised as modest compared to today's alternatives.

Test generation must be automatic

As shown in chapter 8, most of today's tools require the developer to specify individual test cases. In order for test generation to categorise as automatic within the context of this thesis, there should be no specification of test cases. As the tests which are executed in my proof of concept tool only rely on an API model, no individual test cases are specified.

The tests in my proof of concept tool are not only generated from a model, but the model does not include security related properties apart from authentication mechanisms (the user levels and the JWT), which are fundamental parts of any API and hard to avoid. I argue that this satisfies the *automated* constraint, as it allows developers without formal security training to test security constraints of their API and be confident that the API does not have common security vulnerabilities.

To summarise:

As I have been able to answer all of the research questions affirmatively, I conclude that the hypothesis as a whole is proven.

Bibliography

[1] Facebook, "Graph api reference," 2018, [Online; accessed 30-May-2018]. [Online]. Available: https://developers.facebook.com/docs/graph-api/reference

[2] Spotify, "Web api," 2018, [Online; accessed 30-May-2018]. [Online]. Available: https://beta.developer.spotify.com/documentation/web-api/

[3] YouTube, "Api reference," 2018, [Online; accessed 30-May-2018]. [Online]. Available: https://developers.google.com/youtube/v3/docs/

[4] R. Security, "29% increase in vulnerabilities already disclosed in 2017," 2017, [Online; accessed 07-May-2018]. [Online]. Available: https://www.riskbasedsecurity.com/2017/05/29-increase-in-vulnerabilities-already-disclosed-in-2017/

[5] A. Prakash, "How i hacked tinder accounts using facebooks account kit and earned $6,250 in bounties," 2018, [Online; accessed 03-June-2018]. [Online]. Available: https://medium.freecodecamp.org/hacking-tinder-accounts-using-facebook-accountkit-d5cc813340d1

[6] R. T. Fielding, "Dynamic architectural styles and the design of network-based software architectures," *SPEDoctoral dissertation*, 2000.

[7] M. Shaw, "Writing good software engineering research papers," in *25th International Conference on Software Engineering, 2003. Proceedings.*, May 2003, pp. 726–736.

[8] OWASP, "Category:owasp top ten project," 2017, [Online; accessed 16-April-2018]. [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[9] E. O. et. al., "REST Security Cheat Sheet," WWW, Article, November 2017. [Online]. Available: https://www.owasp.org/index.php/REST_Security_Cheat_Sheet

[10] L. Richardson and S. Ruby, *RESTful Web Services*. O'Reilly, 2007, p. 81.

[11] A. R. Fielding and J. Reschke, “Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content,” Internet Requests for Comments, RFC Editor, RFC 7231, June 2014. [Online]. Available: https://tools.ietf.org/html/rfc7231

[12] S. Marx, “How many http status codes should your api use?” 2015, [Online; accessed 03-June-2018]. [Online]. Available: https://blogs.dropbox.com/developers/2015/04/how-many-http-status-codes-should-your-api-use/

[13] R. F. et. al, “Uniform Resource Identifier (URI): Generic Syntax,” Internet Requests for Comments, RFC Editor, RFC 3986, January 2005. [Online]. Available: https://tools.ietf.org/html/rfc3986

[14] W. Santos, “Which api types and architectural styles are most used?” 2017, [Online; accessed 10-April-2018]. [Online]. Available: https://www.programmableweb.com/news/which-api-types-and-architectural-styles-are-most-used/research/2017/11/26

[15] OWASP, “Welcome to owasp,” 2018, [Online; accessed 04-April-2018]. [Online]. Available: https://www.owasp.org/index.php/Main_Page

[16] ——, “Owasp cheat sheet series,” 2018, [Online; accessed 26-April-2018]. [Online]. Available: https://www.owasp.org/index.php/OWASP_Cheat_Sheet_Series

[17] CWE, “About cwe,” 2018, [Online; accessed 07-June-2018]. [Online]. Available: https://cwe.mitre.org/about/

[18] D. R. Stinson, “Cryptography: Theory and practice, third edition,” Available at: http://www.ksom.res.in/files/RCCT-2014-III-CM/CryptographyTheoryandpractice(3ed).pdf, p. 119, 2014, accessed on 01.06.2018.

[19] G. McGraw, “The 7 touchpoints of secure software,” 2005, [Online; accessed 02-May-2018]. [Online]. Available: http://www.drdobbs.com/the-7-touchpoints-of-secure-software/184415391

[20] M. Fowler, “CodeSmell,” WWW, Article, February 2006. [Online]. Available: https://martinfowler.com/bliki/CodeSmell.html

[21] M. J. et. al., “JSON Web Token (JWT),” Internet Requests for Comments, RFC Editor, RFC 7519, May 2015. [Online]. Available: https://tools.ietf.org/html/rfc7519

[22] Swagger, “Swagger,” 2018, [Online; accessed 30-May-2018]. [Online]. Available: https://swagger.io/

[23] A. Blueprint, “Api blueprint,” 2018, [Online; accessed 30-May-2018]. [Online]. Available: https://apiblueprint.org/

[24] RAML, “Raml,” 2018, [Online; accessed 30-May-2018]. [Online]. Available: https://raml.org/

[25] OWASP-group, “Top 10-2017 Top 10,” WWW, Article, 2017. [Online]. Available: https://www.owasp.org/index.php/Top_10-2017_Top_10

[26] ——, “Top 10-2017 A5-Broken Access Control,” WWW, Article, 2017. [Online]. Available: https://www.owasp.org/index.php/Top_10-2017_A5-Broken_Access_Control

[27] T. McLean, “Critical vulnerabilities in json web token libraries,” 2015, [Online; accessed 03-April-2018]. [Online]. Available: https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/

[28] P. Mitton, “Never put secrets in urls and query parameters,” 2016, [Online; accessed 21-March-2018]. [Online]. Available: https://www.fullcontact.com/blog/never-put-secrets-urls-query-parameters/

[29] IETF, “The oauth 2.0 authorization framework: Bearer token usages,” 2012, [Online; accessed 03-June-2018]. [Online]. Available: https://tools.ietf.org/html/rfc6750

[30] J. W. Group, “Json web algorithms (jwa) draft-ietf-jose-json-web-algorithms-31,” 2015, [Online; accessed 14-May-2018]. [Online]. Available: https://tools.ietf.org/html/draft-ietf-jose-json-web-algorithms-31

[31] OWASP, “Fuzzing,” 2018, [Online; accessed 24-May-2018]. [Online]. Available: https://www.owasp.org/index.php/Fuzzing

[32] Wikipedia contributors, “Fuzzing — Wikipedia, the free encyclopedia,” 2018, [Online; accessed 24-May-2018]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Fuzzing&oldid=829613475

[33] T. Fertig and P. Braun, “Model-driven testing of restful apis,” in *Proceedings of the 24th International Conference on World Wide Web*, ser. WWW ’15 Companion. New York, NY, USA: ACM, 2015, pp. 1497–1502. [Online]. Available: http://doi.acm.org/10.1145/2740908.2743045

[34] raml org, “The restful api modeling language (raml) spec,” 2018, [Online; accessed 30-May-2018]. [Online]. Available: https://github.com/raml-org/raml-spec

[35] Wikipedia contributors, “Xunit — Wikipedia, the free encyclopedia,” 2017, [Online; accessed 26-May-2018]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=XUnit&oldid=807299841

[36] S. K. Chakrabarti and P. Kumar, “Test-the-rest: An approach to testing restful web-services,” in *2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, Nov 2009, pp. 302–308.

[37] Abao, “Abao,” 2018, [Online; accessed 24-May-2018]. [Online]. Available: https://github.com/cybertk/abao

[38] Metasploit, “Metasploit,” 2018, [Online; accessed 28-May-2018]. [Online]. Available: https://www.metasploit.com/

[39] C. Security, “Core impact,” 2018, [Online; accessed 30-May-2018]. [Online]. Available: https://www.coresecurity.com/core-impact

[40] H. R, “Three automated penetration testing tools for your arsenal,” 2011, [Online; accessed 28-May-2018]. [Online]. Available: https://www.computerweekly.com/tip/Three-automated-penetration-testing-tools-for-your-arsenal

[41] P. Stephenson, “Core impact professional,” 2013, [Online; accessed 28-May-2018]. [Online]. Available: https://www.scmagazine.com/core-impact-professional/review/6705/

[42] Portswigger, “Burp suite,” 2018, [Online; accessed 30-May-2018]. [Online]. Available: https://portswigger.net/burp

[43] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*. O’Reilly, 2012.