Security Threats in a RESTful API

The Confusing World of Keys

Oscar Lifh

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona Sweden

1 ABSTRACT

Context. With the huge growth of popularity within client side programming, RESTful API's popularity grows with it. Security within a RESTful API is of huge importance to secure the API from uninvited guests. What techniques are frequently used to secure API's, and why?

Objectives. In this study, we investigate which techniques of API security that is used frequently today and consider some examples of distributed systems and how they work with security. Including some articles discussing the important requirements of error handling, access control and format checking.

Results. We address principles for handling errors with HTTP status codes and the main methods of protection against interception by "bad users" and misuse by "good users". This results in a list of principles regarding HTTP status codes, access control and format checking.

Conclusions. We conclude that many well distributed systems are using similar techniques to address the security issue of RESTful APIs. Yet there is no specific standard defined for API security, which leads to a lot of confusion. From these well distributed systems we take knowledge and addresses the techniques and why we need the techniques provided.

Keywords: REST, security, Access Control, API Key

2 CONTENTS

3 INTRODUCTION

3.1 Background

In the year of 2000, a man named Roy Thomas Fielding gave rise to a concept that was going to change the architecture of websites. Fielding described the "Representational State Transfer" (REST) in his Ph.D. dissertation [6], as the architectural style of the World Wide Web. REST advocates that web applications should regress to HTTP requests and to how it originally was envisioned. This means dividing the system into client and services and to use GET, PUT, POST and DELETE. The benefit of regressing to these requests is the "Service-oriented Architecture" (SOA) [7] that the system achieves. With this style of architecture, systems do not have to be integrated with strong dependencies. Instead, the systems are divided into a client side and a service side.

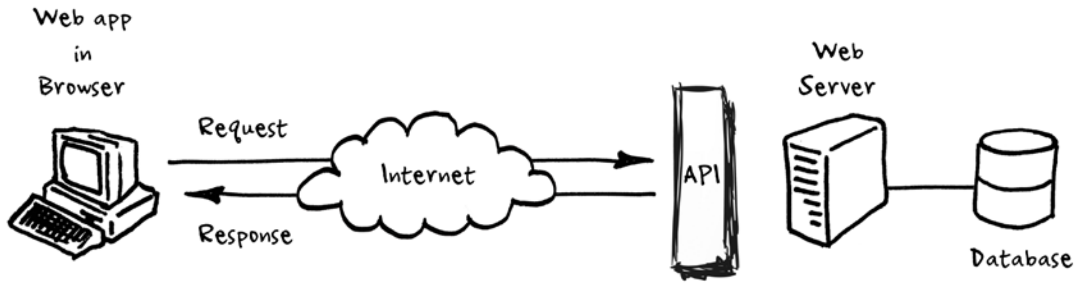


Figure 3-1. The web as a client-server application framework

To communicate between the client side and the service side (back-end), a so called RESTful API can be used. RESTful API's has two main responsibilities, listen for requests sent by the client side, and to respond these requests with appropriate data. This communication chain uses the original HTTP requests listed above.

Because the RESTful API's often is used with public access, chances of misuse and the risk of interception is huge within them. Therefore, every developer must be aware of the risks they are encountering while using this as a part of their system.

3.2 Purpose

The market is rapidly adapting to the modern technologies of frontend frameworks in focus. Adapting in such a fast phase that RESTful services often loses priority. Developers tend to miss out on the knowledge of the vulnerabilities that comes with the API's, with possible cause of end-users breaking the API with misuse or interception by hackers. The purpose of this study is to address the main security problems with RESTful API's and to specify some standard routines to prevent weaknesses in them.

3.3 Disposition

In this study, questions listed in the part "Research Questions" is discussed thoroughly. The study starts off with a background of REST and how RESTful API's work. An important part to understand the problem this study is facing, and the questions being investigated.

The study then considers understanding the concept of RESTful API's, diving into the method used for researching, and a review of the literature. This leads up to analysis and discussion where information from sources is being explained. Results are then being presented and leads to a conclusion which describes some methods of safely protecting your RESTful API's. Links to the references are to be found in the last section of the study.

4 RESEARCH QUESTIONS

This study will focus on the following listed questions, analyzing and discussing them thoroughly in the fourth section ("Analysis and Discussion") of the study. The questions investigated where written to cover the necessary parts needed to securely communicate with a RESTful API.

Q1. What is the best principles of handling error states in a RESTful API?
End-users need response on actions they have taken in the system. Without a noticeable response, one often gets frustrated and starts redoing the action repeatedly. With repeated misuse of the system, it could be interrupted, or in a worst-case crash. To prevent this we try to find the best principles to handle these error messages, with readability and understanding in mind.

Q2. What is the main types of interceptions that the RESTful API is vulnerable to, and how to protect the API from them?
There is always a risk of a bad-user trying to break into your system and hurt your business. This part focuses on presenting a way to protect your API from unintended use and from possible interception of critical data.

Q3. What is the main types of misuses that may be caused by "good users", and how do we prevent them from happening?
Even though the end-user may use the system correctly, it would still be vulnerable to misuse. Just because the end-user thinks he/she is doing the correct action, doesn't necessarily mean he/she really does it. How do we prevent errors from "good users"? This focuses on presenting a way to protect the API from the most common risks

5 METHOD

In this section, several methods used when writing this study is explained. Focusing on method for reviewing literature, analysis and writing. The purpose of this section is to explain the process of writing so that the reader understands the procedure.

5.1 Literature Review

REST was first introduced in Roy Thomas Fielding's dissertation [6] in the year of 2000. All history reviewed in this study is therefore pointing back towards Roy Thomas Fielding's explanation of it.

Literature about RESTful API principles and the security within them became a widely-discussed topic much later. Articles reviewed in this study is therefore no older than from 2010 with the reason of low knowledge in this area before the 10's.

While reviewing literature and articles, the popularity of the writer was to consider. Most writers of the mentioned texts have several articles in the same topic, and is considered as a valid source by reviewers. Others point their information towards more reliable sources. Which has led me to primary sources.

5.2 Analysis

To be able to analyze the gathered sources, at least two articles on the same topic is compared. The reason for this is to have several aspects on the topic and to be able to compare them and hopefully conclude it into something useful.

5.3 Writing

To write this study in the correct format, skrivguiden [10] was of significant help. With few exercises in writing, skrivguiden offers useful information of the content that should be present in the different sections of a study.

This study uses IEEE Citation Reference system to refer to sources. The reference system was discussed in class and concluded with several students and course coordinator. We can use whichever reference system that we prefer. But the course coordinator prefers the IEEE system over other.

This document follows the proposed study template given in the course PA1452 at BTH autumn term of 2017.

6 Analysis and Discussion

This section analyzes the content of the literature and discusses the different approaches they have. With analysis of the Research Questions we gather information from the Literature and compiles it to later come up with a Conclusion.

6.1 Principles of Error Handling

Protecting an API often causes restriction on the client-side. These restrictions are designed to protect the system from breaking, but also to assist the end-user with information in real-time. Notifying the end-user about the situation he/she is facing is handled with HTTP status codes. HTTP status codes is the key communication chain between developers and end-users. The purpose of having a status code is the possibility to share critical information with both the system and the end-user. Therefore, content of error-codes is of huge importance. But what principles should a developer really consider while designing a "good" response message?

6.1.1 HTTP Status Codes

To really answer this question, one must understand the basics of http status codes. The format of HTTP status codes is a three-digit code, with the first digit describing the category type. These status codes are divided into five categories, described in the table below:

The Five Categories of Status Codes	
1XX – Informational	Received request, and the process is continuing.
2XX – Success	Action successfully received, understood and accepted.
3XX – Redirection	Further action must be taken to complete request
4XX – Client Error	Problem on the client-side, often because of incorrect syntax.
5XX – Server Error	Problem on the server-side, failing to fulfill a valid request.

Figure 6-1. The Five Categories of Status Codes

Each of these categories contain a bunch of standardized status codes used widely. But the question we should ask is if it really is necessary to support all these error codes. Guy Levin writes in his article [8] that the only states we really need to consider in a RESTful API is:

- Everything worked as expected (200 – OK)
- The application did something wrong (400 – Bad Request)
- The API did something wrong (500 – Internal Server **Error**)

Levin also mentions three distributed systems and their list of error codes being used. Distributed systems use no more than 8-10 status-codes. With systems like Netflix using 9 status codes, we could for sure say that there is no need to implement that many status-codes. The technique Levin recommends is therefore to start with the three codes listed above and if needed, expend the list of error codes.

With security in mind it is a good idea to already consider adding a few status-codes. The reason for this is the implementation of authorization later in the study. With authorization, we use 401 and 403, two different types of restrictions. We also add the 401 just to give a hint to the developer that the route to the call may be incorrect for some reason.

- Authorization required (401 – Unauthorized)
- Valid request, but refused (403 – Forbidden)

- Resource not found (404 – Not Found)

6.1.2 Content of Error Code

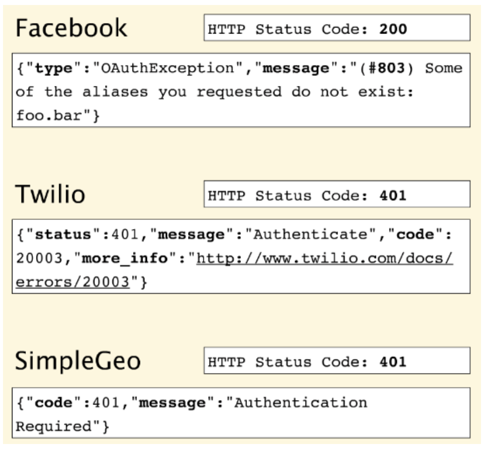
With the HTTP status codes defined, we now have an error-code for the computer. Developers and end-users may be able to interpret them as well, with varying results. Kristopher Sandoval writes in his article [3] about *three basic criteria* to be helpful both for the computer, developer and end-user. **The three criteria are as follows:**

- An HTTP Status Code
- An Internal Reference ID
- Human Readable Message

As we already discussed the HTTP status codes, we jump straight into the second criteria. Internal Reference ID is a custom identifier separated from the status codes explained earlier. This has more to do with the application than a HTTP error and it is therefore created by the developer. The purpose of this ID is to point the end-user and developer to a specific problem or functionality in the system itself, giving a hint of the problem that has occurred.

These error codes are as mentioned pointing towards a specific problem. To explain the problem within the server better, a good practice is to add a human readable message. The message is used to explain the error code itself. This could also be used as an error message for the client-side towards the end-user.

As we see in Figure 6-2, all three of these contributed systems uses a HTTP Status Code, but they also include some sort of human readable information, in form of a JSON string. They are for sure not following the same standard, but the principle of a helpful answer is present in all of them.


Facebook

```
HTTP Status Code: 200
{"type":"OAuthException","message":"(#803) Some of the aliases you requested do not exist: foo.bar"}
```

Twilio

```
HTTP Status Code: 401
{"status":401,"message":"Authenticate","code":20003,"more_info":"http://www.twilio.com/docs/errors/20003"}
```

SimpleGeo

```
HTTP Status Code: 401
{"code":401,"message":"Authentication Required"}
```

Figure 6-2. REST examples

A description of the error can sometimes be difficult to formulate with few words. Therefore, some distributed API's include a link in the response JSON object, that is describing the error thoroughly. This is of great service towards the customers using the system. Quick and easy answer to the issue they are facing. In some cases, a description on how to solve the problem is also included, which could be of huge value.

Huge applications like Facebook handles all errors by themselves. Even though you get a 200 OK from the API as seen in Figure 6-2. There is still a chance of errors occurring. The reason for this is that the developers has more control over error-messages and can easier find the cause of the problem. With a server seen as a black-box for the developer, a dictionary is a very useful practice.

6.2 Principles of Security

6.2.1 Why Security?

When designing the architecture of a RESTful API, one should take security to consideration. But why is this important? By describing some threats introduced in the article RESTful API Security [11], we hopefully open your eyes to securing your RESTful API. As you are going through this section of the study, you will learn the necessity of security. How you should design your architecture to make it secure whilst it still is as functional as you first thought.

6.2.1.1 Data Interception

A RESTful API is the communication chain between the client and the server as we explained in the Introduction section. This means that all vulnerable information is sent through it. Therefore, it is of huge importance that the data is protected so that access to information is restricted for users that does not meet the requirements.

6.2.1.2 DOS Attacks

Denial of Service (DOS) [14] is a widely-spread method used by attackers worldwide. The principle of DOS is to use a bunch of connections (slaves) to point massive load towards a victim. As seen in Figure 6-3

If the system is available to everyone, it is also running a huge risk of being attacked with huge amounts of data. This hurts the functionality, and sometimes kills the API totally. Making it impossible to send requests to it. Therefore, avoiding requests from uninvited users is very important.

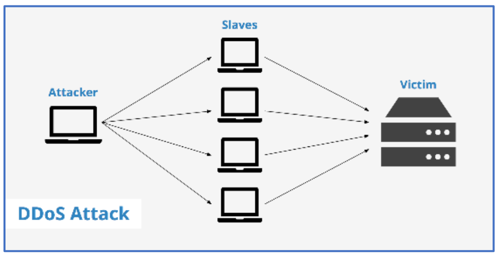


Figure 6-3. DDoS Attack

6.2.1.3 Farming

Farming is the principle of scraping a RESTful API to be able to work with data from someone else's API. Preventing farming is important so that no unnecessary data is being transmitted. If we're not protecting ourselves against farming, chances of an unnecessarily overloaded API stand upon us.

With these possible threats in your mind, we will now dive into the principles of protecting an API against these. With a few steps, you can safely work with your API as you'd want to, in a safe way!

6.2.2 Authentication VS Authorization

But before we introduce the key principles of security in a RESTful API, it is good to know the difference between Authentication and Authorization. Nowadays, many people mistake these concepts with each other causing a lot of confusion while trying to understand security of API's.

Travis Lindsay explains in his article [4] that "Authentication is, quite simply, verification of who or what someone is". This is basically the way you identify yourself with a username and password, in the most standard cases. Authorization on the other hand, happens after you have been authenticated in a system. It is more like a check in the system that evaluates if the given user should be allowed to use a specific part of the system. For example, an Admin is authorized for a bunch of functions that a regular user does not have access to.

6.2.3 HTTPS

Most of the threats described earlier are threats that we can be dealing with using a few methods. Just protecting the API from untrusted connections, will help us a long way. As of every security principle when it comes to the web, it is very important to use HTTPS instead of the simple HTTP

HTTPS or *Hyper Text Transfer Protocol Secure* is a more secure protocol to use instead of the normal HTTP. The difference in these two is mainly that HTTP sends messages in cleartext compared to HTTPS encrypted messages. Advantage of an encrypted message is that interception of the system is very hard. Not only does this prevent eavesdroppers. It also ensures that the website you are visiting is the correctly certified website for the specific business. The company that issued the certification vouches that the website you are connected to is a part of their organization and not a scam.

However, Chris Hoffman mentions in his Article [16] about HTTPS that even though HTTPS can seem to be safe from phishers and scammers, it is not. He says that some phishing websites have noticed that people just check for the HTTPS and assumes it is safe, even though anyone can certify a website with HTTPS. Certifying a website using HTTPS means that the user in charge of the website gets a root certificate that can verify the validity of the website.

HTTPS was originally developed to handle login with passwords only, but has later become a standard for every website, to protect data overall. This is a great start of protecting the data being sent with the API. Both personal data and protection of API keys for example.

6.2.4 Access Control

According to OWASP [2] it is of huge importance to perform access control on each endpoint of a nonpublic API. This is handled with the two concepts of API keys and JWT.

API Keys is the beginning of the modern API Security. In fact, many people still believe this is the way to securely handle an API. But an API key working alone is not any fancier than a revocable, non-expiring, bearer-access token. Travis spencer compares an API key to a normal padlock in his presentation [18] from 2015. The API Key is obviously presented as the key, and the padlock is presented as the API. The provided key will most likely be able to access the API as it should. But the problem comes if the key is being passed to another user, which also would be able to unlock the padlock. The API provider has no real chance of really evaluate if the person really is the one who really was meant to gain access to the system. Making it vulnerable to access from unreliable sources, if they only get the hands on the key.

Figure 6-4. Padlock

To prevent this vulnerability of the system, Padlock talks about OAuth2, an industry-standard protocol for authorization. He explains it as a protocol of protocols, a base for other specifications. OAuth adresses some important requirements that API Keys does not fulfill. Some examples of those are delegated access, no password sharing and revocation of access. Overall, this provides more control for the resource owner. Involved in the “dance” of OAuth is four actors:

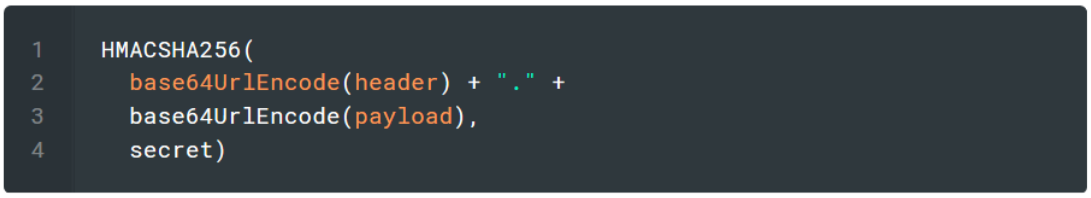
1. Resource owner
2. Client
3. authorization server
4. resource server.

These four roles all have unique roles and is each very important to the concept of OAuth. But is OAuth2 enough security to call your API “safe and secure”. According to David Blevins [12] it is not enough. But this has less to do with the security than expected.

David compares the OAuth2 method to a HTTP Session that only lives in a specific scope. The problem described in his presentation is that request load on the authorization server and resource server is way too large and may hurt the system in case of an attack from the outside. Instead, David introduces another method called JWT that combined with OAuth would make the load on the server less heavy.

JWT or JSON Web Tokens [17] is a self-contained solution to safely transmit information between client and server side as a JSON object. Self-contained means that it contains all information about the user, and does not have to search the database for each individual call to the API. This decreases the load on the server by a huge amount, and makes the authentication process much faster.

JWT consists of three key parts - header, payload and signature. These three are divided within three different JSON objects. The header normally consists of the token type and the algorithm used for hashing. Payload is the metadata that is interesting to authenticate a user. It could be username, role, ids and more. It also contains claims about the entity in question. The last part is the signature.



```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

Figure 6-5. Design of Signature

The signature is created by encoding the header, payload and a secret with the algorithm specified in the header. It is later used to verify the end-user to be the identifier it says it is in the payload part, and ensure that no changes have been made to the token.

As specific data about the user can be sent through the JWT, the end-user can hold their information needed to authenticate themselves. Making it much easier to just authenticate the user with a single signature validation, validating that the signature is known and that its has not changed. Reducing the overall load on the authorization server. This is what David compares to a HTTP Cookie in his presentation [12].

With this technique, we have a way of safely locking our API, with the possibility to recognize the user with the minimum load on the server. But is this really the full solution to the problem we are facing, is it safe now? Many developers would probably be more than happy with this solution.

But we are still facing a threat with all this protection. We have dealt with threats from outside, but what if the hacker reaches our internal servers. The Target Hackers [19] is a great example of hackers gaining access from the inside. They received the login information while working at the company Target, giving them access to the whole enterprise server. How could we possibly protect our internal servers from attacks like these. Well, it is simple if the methods explained above already exist in the system. David Blevins covers this section in his presentation [12] as well. He says that the solution to protection of the internal part is simply to send the users JWT inside the Internal system as well. Hopefully a user-role is saved to the end-users JWT and authorization of the privileges he/she has is the way to go. This prevents unauthorized workers from accessing the whole system, making them restricted to their section only.

6.2.5 Format Checking

Another important part According to the OWASP [2] is to validate the data being sent to the API. The reason for this has more to do with the REST services functionality than protection from "bad users". If incorrect data would be sent to a REST service, chances of responses failing is huge. Worst case scenario with incorrect data coming through is the possibilities of total breakdown. Developers always aims for 100% uptime, because failure in systems are very expensive.

The first important part of format checking is to validate the content type being sent. According to OWASP the requests shall match the intended content-type in the header. Otherwise chances of misinterpretation at consumer side is huge, and could also lead to code injections being a threat. The most common content type today is the "application/json", but is in some cases combined with multiple response types if necessary, such as "application/xml".

The second important part is input validation. Input validation is performed to ensure that properly formatted data is being sent to the service. The difference between validating content-type and input validation is that content-type is more like the protocol or language it is being written in, and validation of input is more like spellchecking of sentences. OWASP recommends the developer to not trust input parameters or objects from anyone, meaning that every request shall be handled as a possible threat.

To prevent untrusted requests, limiting the possibilities of incorrectly formatted data is key to success. OWASP has a list of principles to limit formatting of responses.

6.2.5.1 Strong Types

As a developer, having control over the types of answers we achieve from end-users is important. A good start is to set strong types like Booleans, numbers, floats or date objects. Making it easier to control what comes in, and protecting against misuse of the API.

6.2.5.2 Length and Range Validation

Validating range and length is important to prevent the end-user from doing some sort of code injection. If the end-user would be able to write several lines of code in input fields, it would have been a much larger chance for the developer to miss out on some sort of injection style.

6.2.5.3 Request Size Limit

We have already talked about load on the server by addressing the issue with "bad" users causing load on the server-side. With this prevented, we can still manage load problems with unnecessary large sizes of requests. By limiting the request size, we prevent overloading the API. Using the Status Code 413 Request Entity Too Large to address the issue.

6.2.5.4 Constrain Using Regular Expressions

Regular Expressions or "RegExp" is a great method for validating string inputs. If the program needs a string input, we can evaluate the content of it, and remove unwanted special characters for example.

7 RESULTS

7.1 Principles of handling error states

Several principles of error-handling are presented in section 6.1 of this study. One of the key principles in handling error states is to really consider the amount of error states being used by the system. An Example being presented in section 6.1.1 shows that well distributed companies aims for a small amount of eight to ten (8-10) error-codes used in their entire system.

Section 6.1.1 also addresses the technique used for error-handling to start with as few error-codes as possible, expanding the system if necessary. The following error codes is a great start to handling errors in a RESTful API.

- Everything worked as expected (200 – OK)
- The application did something wrong (400 – Bad Request)
- The API did something wrong (500 – Internal Server Error)

But with security in mind, we can already think of reasons to expand this error-state base. When considering expanding the system, a well thought reason behind it should be presented. The reason is just so that the developer is confident in the decision and knows the reason of expanding. Adding the following three error-codes with the reason of creating an authentication based RESTful API is a reason good enough.

- Authorization required (401 – Unauthorized)
- Valid request, but refused (403 – Forbidden)
- Resource not found (404 – Not Found)

The reason for trying to limit the number of error-codes is obvious when presented. The HTTP error codes should handle the connection between clinic and server. With the only outcomes of connected successfully, failed to connect, or connected but failed to execute. Internal errors on the server side should instead be presented with an internal error handling system. Examples in Figure 6-2 shows how three well established companies handles errors internally. With a JSON string sent back, presenting an internal error is much easier. It should consist of:

- Internal error code (have some system to it)
- human readable error message (short and descriptive)
- link to help and solutions (used in best cases)

With well-designed error-handling, users get a better response message resulting in less unnecessary calls to the API. Error-handling can provide against unnecessary load on the API or in worst case API overload. Companies should put down hours into designing it.

7.2 Main types of interception and protection against it?

This is the biggest concern regarding security within RESTful API's. As most RESTful API's out there are public we need some sort of validation to delimit who can access it. Without delimiting the access points to the API, chances of interception, farming and DDOS attacks are huge.

7.2.1 Threats

Interception of the system may occur if the API requests are not encrypted. If somebody where to wire-tap the connection to the API whilst a user is trying to connect, their credentials are standing a huge risk of getting taken by the hacker. This could possibly give the hacker access to the whole enterprise and all secrets that is has, with devastating consequences.

As Mention in section 6.2.1 **Farming** is the method of scraping a RESTful API to be able to work with data from someone else's API. Farming of the system may also occur if the requests are not encrypted. Though farming has more to do with uninvited use of the system and is therefore handled with authentication. Farming can cause a lot of unintended load which could lead to slow loading times for the users really intended to use the system. Another threat mention in section 6.2.1 is the widely used **DOS** (Denial of Services) attack. The principle of DOS is simply to overload the system with way too many requests at the same time. This is hard to prevent, but with a superior design you can minimize the risks of unintended requests causing damage.

7.2.2 Protecting Against It

7.2.2.1 HTTPS

To begin with, HTTPS is a MUST to be able to create the access control at a later stage. HTTPS or "Hyper Text Transfer Protocol Secure" is an alternative protocol to the normal HTTP, with the difference of encrypted messages. As Mentioned in section 6.2.3 it was first used to handle logging in on websites. But has later become a standard for almost every serious enterprise websites. The reason for this is simply that HTTPS uses encrypted messages. By having encrypted messages, we achieve a way to send "secrets" to the API. Secrets that may contain login information, but in this case API keys. This makes it possible to authenticate users individually.

7.2.2.2 Access Control

Performing access control on each endpoint of the system is important to authenticate the users connecting to the system. API keys are not the way to go because it works like a real-life padlock. The key can be sent to several devices and gain the same access as the one intended to have the key. Causing a problem in authentication of the user, with the system having no feasible way of identifying the identification of the end-user. To solve this, a combination of OAuth2 framework and JWT is used.

OAUTH2 is a framework for authorization. As mention in section 6.2.4 OAuth adresses some important requirements that API Keys does not fulfill. Some examples mentioned are delegated access, no password sharing and revocation of access. With these functionalities, the API has a way of identifiying the end-user safely without a threat of the identification being sent further. This may sound great but is still not enough to safely handle access control. Also mentioned in 6.2.4 is the problem with authentication request that has to be sent over and over again to validate the information to the authorization server. While OAuth is compared to a HTTP session that has a life length that has to be renewed, including JWT makes it comparable to a HTTP cookie with all necessary information inside of it.

JWT or "JSON WEB TOKEN" is a self-contained solution to safely transmit information between client and server side as a JSON object JWT is used to save information for authenticating the user on the client side instead of the server side, taking away a sizable chunk of load from the authorization server. Making requests only first time and whilst renewing of tokens is needed.

7.3 Main types of misuse and protection against it?

The most common types of misuse by normal users is input of incorrect data. This is handled with several methods of format checking. Presented in section 6.2.5 is a list of principles from OWASP that explains the main types of format checking. Including topics about strong types, length/range validation, request size limit and regular expressions. These should be working together to prevent unnecessary big data or incorrect data to pass through to the API. If not handled correctly chances of overloading the API, injections and errors are much larger.

8 CONCLUSION

In this study, we have evaluated the process of securing a RESTful API. With several articles and information of approaches used by well distributed systems we have put together some interesting techniques that where found and investigated into. We've considered techniques like HTTP status codes, HTTPS, Access Control with OAuth2 and JWT and finally format checking. We have concluded that the optimal solution is not to have one of these techniques, but to combine them all into a framework working together.

To summarize this study, we found that:

In the first question regarding principles of handling error states, we found out that the optimal approach to error handling is to start with a few HTTP error codes, implementing an internal error handler within the API, with the purpose of describing the errors occurring in the backend to the end-user.

In the second question regarding main types of interceptions, we found out that the best solution to protect against interception and unintended use of the system is to use the combined technologies of OAuth2 and JWT. This makes the system able to identify the end user, with the possibility to save the authentication information in the JWT, so that the load on the authorization server is minimized.

Another important part that we introduced was the principle of securing the inside of the API. Without any security inside the enterprise system, it could be vulnerable to these concepts, by a human being just getting introduced to any valid connection details. By sending the signature through the backend as well, we can authorize the user and limit its access rights.

In the third question regarding main types of misuses, we found out a list of principles to protect against misuse from "good users", or users that is using the system correctly. We investigated in several types of format checking and why they are needed to ensure that the end-user only sends valid data. By implementing format checking in the form of Strong types, Length and Range validation, Request Size Limit and constrain data with regular expressions, we can limit the user to only push valid data.

9 REFERENCES

Most of the references in this study is articles found while browsing internet for valuable sources. As of today, most of the information found about web specific topics are to be found on the internet itself. However, the information about what REST and REST-API really is where found in the book REST API Design rulebook.

9.1 Sources

[1] REST API Design Rulebook
By: Mark Masse
Link: http://proquest.safaribooksonline.com.miman.bib.bth.se/9781449317904
ISBN: 978-1-4493-1050-9

[2] REST Security Cheat Sheet
By: Erlend Oftedal, Andrew van der Stock, Tony Hsu Hsiang Chih, Johan Peeters
Link: https://www.owasp.org/index.php/REST_Security_Cheat_Sheet
Assessed: 12-09-2017

[3] Best Practices for API Error Handling
By: Kristopher Sandoval
Link: https://nordicapis.com/best-practices-api-error-handling/
Assessed: 12-09-2017

[4] Authentication vs Authorization -- What's the Difference?
By: Travis Lindsay
Link: http://www.investorguide.com/article/16034/authentication-vs-authorization-d1503/
Assessed: 12-09-2017

[5] The Curious Case of API Security
By: Gunnar Peterson
Link:https://www.axway.com/sites/default/files/resources/whitepapers/axway_collateral_api_top_11_threats_en.pdf
Assessed: 12-09-2017

[6] Architectural Styles and the Design of Network-based Software Architectures
By: Roy Thomas Fielding
Link: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
Assessed: 13-09-2017

[7] Service-Oriented Architecture (SOA) Definition
By: Douglas K Barry
Link: http://www.service-architecture.com/articles/web-services/service-oriented_architecture_soa_definition.html
Assessed: 13-09-2017

[8] REST API Error Codes 101
By: Guy Levin
Link: http://blog.restcase.com/rest-api-error-codes-101/
Assessed: 22-09-2017

[9] RESTful API Design: what about errors?
By: Brian Mulloy
Link: https://apigee.com/about/blog/technology/restful-api-design-what-about-errors
Assessed: 25-09-2017

[10] Skrivguiden
By: Skrivguidens redaktion
Link: http://skrivguiden.se/
Assessed: 25-09-2017

[11] RESTful API Security
By: John Vester
Link: https://dzone.com/articles/restful-api-security
Assessed: 03-10-2017

[12] Deconstructing REST Security
By: David Blevins
Link: https://www.youtube.com/watch?v=9CJ_BAeOmW0
Assessed: 03-10-2017

[13] API Keys ≠ Security: Why API Keys Are Not Enough
By: Kristopher Sandoval
Link: https://nordicapis.com/why-api-keys-are-not-enough/
Assessed: 03-10-2017

[14] Denial of Service Attack
By: Margaret Rouse
Link: http://searchsecurity.techtarget.com/definition/denial-of-service
Assessed: 04-10-2017

[15] REST Security Cheat Sheet - Farming?
By: Jim Manico
Link: https://lists.owasp.org/pipermail/owasp-cheat-sheets/2015-August/000051.html
Assessed: 04-10-2017

[16] What is HTTPS, and Why Should I Care?
By: Chris Hoffman
Link: https://www.howtogeek.com/181767/htg-explains-what-is-https-and-why-should-i-care/
Assessed: 06-10-2017

[17] Introduction to JSON Web Tokens
By: jwt.io
Link: https://jwt.io/introduction/
Assessed 13-10-2017

[18] The Nuts and Bolts of API Security
By: Travis Spencer, Twobo Technologies
Link: https://www.youtube.com/watch?v=tj03NRM6SP8
Assessed 14-10-2017

[19] Target Hackers Broke in Via HVAC Company
By: Brian Krebs
Link: https://krebsonsecurity.com/2014/02/target-hackers-broke-in-via-hvac-company/
Assessed 15-10-2017

9.2 Figures

Figure 3-1. The web as a client-server application framework
Link: http://www.robert-drummond.com/2013/05/08/how-to-build-a-restful-web-api-on-a-raspberry-pi-in-javascript-2/

Figure 6-1. The Five Categories of Status Codes
Link: http://blog.restcase.com/rest-api-error-codes-101/

Figure 6-2. REST examples
Link: http://blog.restcase.com/content/images/2015/12/error-codes-rest-api.png

Figure 6-3. DDoS Attack
Link: https://www.keycdn.com/support/ddos-attack/

Figure 6-4. Padlock
Link: https://pixabay.com/sv/cybers%C3%A4kerhet-s%C3%A4kerhet-l%C3%A5s-1915626/

Figure 6-5. Design of Signature
Link: https://jwt.io/introduction/