TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Mark Shafran 201753IVSB

An Analysis of Session- and JWT-based Authentication Methods: a Comparative Study with Secure Implementation Examples

Bachelor's thesis

Supervisor: René Pihlak
MSc

Tallinn 2023

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Mark Shafran 201753IVSB

Seansi- ja JWT-põhiste autentimismeetodite analüüs: võrdlev uuring turvalise rakendamise näidetega

Bakalaureusetöö

Juhendaja: René Pihlak
MSc

Tallinn 2023

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Mark Shafran

15.05.2023

Abstract

The first line of defence against assaults is usually authentication. Additionally, authentication is frequently the most important barrier of defence because once a threat has gotten past authentication, the application will typically treat them as a valid user. According to OWASP Top Ten [1] 2021, the most serious vulnerability vector, with average incident rate of 3.81%, is broken access control. This includes different issues connected to cookies, as well as session and JWT (JSON Web Token) invalidations. In the previous study of OWASP in 2017, the broken access control was the fifth most severe vulnerability. Thus, investigating topics related to broken access control are of utmost importance.

This thesis examines two most notable methods of creating an authentication on the web application. The paper offers a comprehensive security risk assessment of two most noticeable in-house solutions of handing authentication on web-based applications: session-based and JWT-based authentications. It also provides the NodeJS back-end code, written by me, that implements the assessed security workarounds. The paper aims to ascertain what are the differences between using each of the solutions, and which of the methods are best suited for authentication forms in different circumstances.

The analysis concludes by providing valuable information about expected challenges, potential complications, and security risks to be aware of, as well as sample authentication implementations for developers and security experts should they decide to adopt and further work with one of the technologies examined in this paper.

This thesis is written in English and is 57 pages long, including 7 chapters, 12 figures and 13 tables.

List of abbreviations and terms

API	Application Programming Interface
Brute-force attack	A malicious party uses trial-and-error approach to crack sensitive data
Cookie	A piece of information which is stored on a computer device and consists of information related to what a web browser is required to remember
CSRF attack	Cross-site request forgery attack, a malicious party tricks a user into executing an unwanted action on a web application, by exploiting the trust relationship between the client and the application
DoS attack	Denial of service attack, a malicious party floods the target system to cause the system to crash
Guessing attack	A malicious party tries to guess or predict the value through statistical analysis techniques rather than brute-force techniques
HTTPS	Hypertext Transfer Protocol Secure
HTML	Hypertext Markup Language
ID	Identifier
IP	Internet Protocol
JWT	JSON Web Token
MITM attack	Man-in-the-middle attack, a malicious party intercepts communication between two parties, such as a client and a server
Preimage attack	A malicious party tries to find a message that matches a given hash output
RFC	Request for Comments
SCS	Secure Cookie Sessions
TLS	Transport Layer Security
URL	Uniform Resource Locator
XSS attack	Cross Site Scripting attack, a malicious party injects malicious code, such as JavaScript, into a webpage to steal sensitive information or perform other malicious actions.

Table of contents

List of figures

List of tables

1 Introduction

Authentication is a security measure that ensures that only authorized individuals or entities are granted access to resources, systems, or information [2]. Modern web applications often require an online authentication since it allows companies to guarantee the security and privacy of their customers. However, it might be difficult to choose the best authentication system from the wide range of options available and securely implement it. My thesis focuses on two in-house authentication techniques that are often implemented: session-based authentication and JWT-based (JSON Web Token-based) authentication. To store both authentication methods, I use cookies as a client-side storage mechanism and cookie implementation is also discussed in scope of this thesis. Overall, I investigate the security concerns of both authentication approaches and offer suggestions for developers and security experts wishing to design and deploy safe authentication systems.

The security of cookies, session- and JWT-based authentication methods is crucial, as, according to OWASP Top Ten 2021 [1], it is the most serious vulnerability vector. This includes different issues connected to cookies, as well as session and JWT invalidations. Moreover, the third most important security risk is “injection attacks”, which includes list of cookie related attacks. Furthermore, session related vulnerability vectors are additionally discussed in scope of “identification and data integrity failures”, which is another category in the list of most common security risks of 2021. All of these categories are of utmost importance and possible solutions of these attack vectors are discussed in this thesis.

In the second part of this thesis, I made exhaustive literature review of client-side storage mechanisms and authentication methods, to provide a comprehensive analysis of these methods. I evaluated different client-side storage options such as cookies, local storage, session storage, and IndexedDB, and compared their possible attack vectors such as MITM (man-in-the-middle), XSS (cross-site scripting), and CSRF (cross-site request forgery). Based on my findings, I chose cookies as the client-storage technology to be

used in the solutions discussed later in this thesis. I also reviewed the most common discussed authentication methods, including session-, token-, and third-party based authentication, and evaluated their pros and cons. In this thesis, I mainly focus on internal solutions, particularly session-based authentication and JWT-based authentication.

In the third part, I explain methodology I use. This includes scoring methods and methodology to select the best solution.

In first section of fourth part, I have investigated the security issues connected to each of these authentication strategies. Especially, I assess the technical characteristics and security concerns of each security workaround method proposed for JWT-based authentication implementation. As the result, I provide suggestions for developers and security specialists who want to develop secure authentication systems. I specifically propose two novel options that deal with the security issues caused by JWT-based authentication implementation. In the second section of fourth part, I offer sample implementation solutions, which could be useful for developers to employ in upcoming projects. Finally, in the third section of fourth part, I compare the results of my work.

In the fifth and sixth parts, I discuss the results and present key findings from my study. I also suggest ideas for future work and provide a broader perspective on the significance and implications of my work for the field of study.

2 Literature review

Authentication is frequently used as the initial layer of defence against cyberattacks. Once a malicious party has passed authentication, web application will often regard them as a trusted user, so authentication is frequently the most crucial line of defence. Whenever user is authenticated to the system, there is a need to store authentication-related data. According to [3], client-side storage is one of the options widely used for that. This technique refers to the storage of user data on the client's device rather than the server or servers. Client-side storage grown popularity among developers because of its capacity to lessen server load and improve user experience. Cookies, local storage, session storage, and IndexedDB are most widely used examples of client-side storage methods [3]. Each of the above-mentioned client-storage mechanisms has its own pros and cons. For example, [4] cookies are small text files that are stored on the client's device and sent to the server with each request. Cookies were not built as a secure client-side storage option at first, as they are sent from the client to the server as plain text, which brought a MITM attack vector. By default, they are also vulnerable to XSS and CSRF attacks. However, nowadays, both MITM, XSS and CSRF attack vectors could be mitigated. To mitigate MITM, cookies should be only sent via encrypted format. Thus, the secure channel between parties should be established [5]. To do so, `Secure` attribute shall be used, which will only allow setting a cookie using HTTPS (hypertext transfer protocol secure). HTTPS uses TLS (transport layer security) protocol, which secures communication between client and server by encrypting it with asymmetric public key infrastructure [6]. XSS attack could be implemented by malicious JavaScript code, which may steal the cookie. However, adding `HttpOnly` attribute to a cookie makes it inaccessible to the JavaScript API, so mitigating XSS. Cookies were also vulnerable to cross-site request forgery (CSRF) attacks, but there are several workaround ways proposed [4]. By the information available on [7], [8, p. 714] and [9][section 8.8.2] CSRF could be mitigated either setting `SameSite` attribute to `Strict` or making a double submit cookie, mentioned in [10]. Overall, cookies are often used to store sessions in them. In contrast, local storage is widely used for storing token-based authentication [11]. Local storage is a key-value storage mechanism that allows web applications to store data in the client's browser. However, it brings security concerns, as this client-storage method is vulnerable to CSRF [12] and XSS [13] attack vectors. Another storage technique is session storage, which is

similar to local storage, but data is deleted when the current session ends. Session storage is rarely used for the authentication as it could not be considered as user friendly – authentication is needed on every new tab or window [14]. It is also vulnerable to the same attacks as local storage and will bring an additional load on server. There is also a newer technology for client-side storage gaining popularity: IndexedDB. IndexedDB is a non-relational NoSQL database that allows web applications to store large amounts of data on the client's device. Currently, the technology is also affected by XSS attack, since the IndexedDB stores data in the unencrypted state [15].

The next table compares all the above-mentioned local storage methods in terms of possible vulnerability vectors. Each attack vector impact weight is similar and mitigating the attack is graded as one point.

Table 1. Solved attack vectors comparison of different local storage methods.

Attack vector	**Cookies**	**Local Storage**	**Session Storage**	**IndexedDB**
XSS	✓*	✓	✓	✓
MITM	✓†	✓	✓	✓
CSRF	✓‡	✗	✗	✗
Final Grade	3.0 / 3.0	2.0 / 3.0	2.0 / 3.0	2.0 / 3.0

As it was mentioned above, cookies were not built as a secure method at first. However, as it is mentioned in Table 1 above, all the investigated attack vectors (XSS, CSRF and MITM) could be mitigated by it. To alleviate XSS (*), `HttpOnly` attribute in cookie should be used. The MITM (†) can be fixed using `Secure` attribute. To work around CSRF (‡), there are different ways proposed. According to [7], [8, p. 714] and [9][section 8.8.2], it could be mitigated with `SameSite` or using double submit cookie, which is proposed by [10].

It could be also interesting for the reader of this thesis to see overall comparison of different local storage mechanisms as well. It may happen that cookies are not suitable for due to storage size or data that could be stored. The next sources were used for the data comparison: [16], [17], [18] and [19].

Table 2. Comparison of local storage methods.

Characteristic	Cookies	Local Storage	Session Storage	IndexedDB
Maximal storage size	4KB	5MB	5MB	50% of available disk space[*]
Stored datatypes	String data	String data, JS objects[†]	Strings data, JS objects[†]	Complex data[‡]
Volatile and temporary storage	✗	✗	✓	✓
Limited to current browsing context	✗	✗	✓	✗

Maximal storage size is the maximal storage size of the storage mechanism. As it can be seen from the Table 2, the storage size of cookies is limited to 4KB, which could be insufficient for some use-cases. For the IndexedDB (*), the storage size depends on browser. According to [16], [17], [18] and [19], for most of the widely used browsers, maximal storage size is limited to 50% of available disk size.

Stored datatypes provide the list of data types that could be stored in the storage method. (†) Even though both local and session storage are only able to store string key-value pairs, it is also possible to store JavaScript objects by using the `JSON.stringify() method`. This method converts the object to a string and stores it as string. Afterwards, `JSON.parse() method` is used to convert an object string back to an object. Complex data types (‡) are objects, arrays, and binary data.

Volatile and temporary storage indicates client-storage mechanisms that store data in non-persistent manner. This type of data may be lost after a certain period of time or when the user closes the browser.

Limited to current browsing context – is used to represent storage methods, which is limited to only one browser tab or window. If a user opens multiple tabs or windows of the same website, each of them will have its own storage. This could lead to inconsistent authentication states and bring bad user-experience.

As the main idea of this thesis work is to provide secure authentication method, cookies are the most reasonable option based on above comparison (Table 1) as they are the only client-side storage method that has the highest grade of three points. Therefore, they will be used while implementing storage for authentication information on client's browser.

Authentication-based methods are used to ensure that users are who they claim to be when accessing web applications. The most frequently used options highlighted by [11], [20], [21], and [22] are session-based, token-based and third-party access-based authentication.

Each of the aforementioned techniques offers benefits of its own. Session-based authentication is regarded as secure [5], in case properly used. In case of misconfiguration, it could be affected by session-fixation, brute-force and guessing attack vectors [24]. Additionally, session authentication brings additional server load, since it is stateful and requires database call on each user request. On the other hand, an authentication token could be stateless. Authentication tokens are divided into two different types: physical token and web token. Physical token requires an additional device or software from the clients, so it is hard to deploy them for the web-application available for external clients. For this reason, physical tokens are left outside of the scope of this thesis work. In regard to web-token solutions, the most widely discussed is JWT (JSON Web Token), which is usually used as an alternative to session-based authentication. JWT could be used as a stateless token, which means it is to be trusted by default [25]. Due to JWT being stateless, tokens could be better in scope of scalability and performance [26] [27]. However, token-based authentication introduces security risks as it is impossible to revoke token from the system [28]. To mitigate this issue, different solutions are proposed, for example [11], [28], [29] and [30].

Another authentication method that is often discussed is to implement third party access-based authentication. Third-party based authentication shifts responsibility for authentication to external vendors, reducing the organization's security responsibilities for the authentication implemented on their web application. In case of a security breach within the organization, there is less data to be leaked, as the affected organization needs to store much less client data on their side. Furthermore, using third-party based authentication reduces the number of passwords that are needed to be managed by the client, so reduces the risk of clients using unsafe or similar passwords. It could also improve the overall web-application experience from the client perspective. However,

shifting responsibility to the external vendor means that authentication service is out of control of the organization. The organization must trust that the external vendor will properly manage the accounts. In case there will be an issue with authentication service on vendors side, the authentication on the web-application will be inaccessible. Furthermore, if a potential client has no account created on the external vendors' side, it will be impossible to access the organizations web-application service.

There are differing opinions and potential security risks of authentication methods discussed in various sources, for example [5], [11], [28], [29], [30] and [31]. This makes it challenging to determine which approach or approaches should be used. It is necessary to carefully analyse and compare the different strategies, particularly for token-based authentication, such as JWT, which has multiple workarounds addressing different security concerns suggested above in the literature review. Furthermore, despite an exhaustive search, no existing solutions were found that fully implement either session-based or JWT-based authentication while adequately addressing all above-mentioned security concerns. Addressing this research gap, pre-existing solutions are used as a starting point and subsequently modified to ensure proposed solution robustness in address to security concerns. Additionally, I provide a novel practical solution for each of the two methods and evaluate their security according to the methodology discussed in the next part.

3 Proposed Methodology

In order to securely implement different authentication methods, in scope of literature review, I considered various client-side storage options, including cookies, local storage, session storage, and IndexedDB, and compared the possible attack vectors associated with each, such as XSS, MITM and CSRF. The less attack vectors available, the better security is of the option and each of vectors has similar impact weight (see Table 3).

Table 3. Attack vectors impact weight.

Attack vector	**Score**
XSS	1.0 point
MITM	1.0 point
CSRF	1.0 point

In the literature review, I analysed which of authentication methods were, so called, in-house and did not require an additional device or software from client's side. I narrowed the scope of this thesis to only two most used in-house solutions: session-based and JWT-based authentication solutions. Through my literature review, I identified the most common security issues associated with session-based and JWT-based authentication that are analysed in the next part of thesis. To fulfil that, I went through variety of journal articles, conference papers and RFC (Request for Comments) standards, as well as online web-sources: official developer websites, blogs, forums.

I provide a deep analysis of the security issues identified during the literature review and discuss important considerations when implementing these solutions (in part 4.1). There are three similar issues considered for both session and JWT-based solutions. These shortcomings are revocation on: 1) user logout, 2) password change and 3) role change. They could be grouped as **revocation problems**. In all three cases, the previous session or token shall be considered invalid and revoked. There are also specific problems for each solution. For session-based implementation, these are: 1) **brute-force** and **guessing** and 2) **session fixation**. For JWT-based solution: 1) **preimage** and 2) **token-prediction** problems. Brute-force, guessing attack and preimage attack vectors are all related to cryptographic attacks even though meaning different attacks. In this thesis, both brute-

forcing attack for sessions and preimage attack for JSON Web Tokens are used for an attacker to create a forged session/token that the server would accept as legitimate. Due to that, they are compared on the same row in Table 4. In scope of this thesis, token prediction term for JWT is similar to session fixation for sessions in scope of attack vector and final workaround, so they are compared in one row as well. Keeping in mind these problems, I bring up suggestions on safe session-based implementation. Also, I provide a detailed comparison of available security workarounds for JWT-based authentication. I compare them in scope of the number of problems solved: the more solved – the better. The problems compared are logout, user password change, user role change, preimage attack vector and token prediction. Each problem is graded equally and gives one point if solved. As attack vectors for session-based and JWT-based could be compared, one table for both solutions is created (see Table 4).

Table 4. Issue workaround comparison impact weight.

Issues of session-based authentication	**Issues of JWT-based authentication**	**Score**
Revocation on client logout		1.0 point
Revocation on client password change		1.0 point
Revocation on client role change		1.0 point
Brute-force and guessing attack vectors	Preimage attack vector	1.0 point
Session fixation attack vector	Token prediction attack vector	1.0 point

Additionally, I compare technical characteristics of different solutions (see Table 5), where architectural complexity, invalidation latency, acquisition frequency and estimated scalability are compared.

Architectural complexity. This is used to describe how many parts there are in a system, how they interact, and how much infrastructure support is necessary. In general, more safety precautions are required to ensure the system's robustness the more complicated a solution's architecture is.

Invalidation latency. This refers to the delay between the moment an intent to invalidate a token is made and the actual point in time when it becomes unusable for accessing a protected resource.

Acquisition frequency. This refers to the rate at which new token acquisitions and invalidations occur for remaining clients due to the method in use. This is only applicable to methods that revoke several tokens at once, for example methods using group secret.

Estimated scalability. This refers to the system's ability to grow and adapt as the number of clients increases, with regard to log-out handling. The primary consideration for scalability is how the system handles the rising number of revocation events that come with increased client numbers. The proposed scalability is an estimation based on calculations done [28] and therefore the data may vary in real-life implementations. Due to that, the estimated scalability does not affect the final grade and the estimated points for it are mentioned separately in the brackets.

The lower architectural complexity, invalidation latency and the higher acquisition frequency and estimated scalability, the better. As invalidation latency is critical for security of system, everything but instant gives minus one point.

Table 5. Technical comparison of solutions impact weight.

Technical Characteristics	**Grading**			
Gradation	Poor	Passing	Good	Excellent
Architectural complexity	Very High	High	Medium	Low
Invalidation latency	Non-Instant	-	-	Instant
Acquisition frequency	Very low	Low	Variable or Medium	N/A* or High
Estimated Scalability	Very bad	Bad	Linear or Medium	Good
Score	-1.0 point	0.0 points	0.5 points	1.0 point

*N/A – not applicable.

Additionally, I propose two novel ways to enhance security and improve server load for JWT-based authentication.

Next, I develop a sample back-end code in NodeJS that demonstrates the implementation of both session-based and JWT-based authentication methods (part 4.2). This code consists of login and logout pages, and an authentication-required verification page. The security issues mentioned in earlier parts are addressed and implemented in the sample code. To develop similar database tables, I also offer sample database queries with NodeJS. Libraries I use for that: `express`, `body-parser` and `cookie-parser` to simplify the building of web application authentication process. Specific libraries used are `Argon2` for password hashing, `MySQL` for database and `Crypto` library for generating random universally unique identifiers and random bytes for the JWT secret. Additionally, `jsonwebtoken` package is used for JWT related operations.

Afterwards, I compare my own developed solutions in scope of problems that are solved (part 4.3): 1) logout, 2) user password change, 3) user role change, 4) session fixation and token prediction attacks, and 5) preimage and brute force attacks. At first, both session-based and JWT-based solution are compared in scope of the number of problems solved. The bigger number of solved problems, the better (see Table 4). Next, both my session-based and JWT-based implementations are compared in scope of technical characteristics correspondingly to Table 5.

Using this methodology, I sum up my findings and provide developers and security specialists with a comprehensive understanding of the security considerations associated with different authentication technologies, to help them create more secure and reliable online authentication systems.

4 Analysis of implementations

As per the literature review (part 2), I wrote that determining which security strategy or strategies should be implemented is a challenging task, especially for token-based authentication like JWT, which has various workarounds suggested to address security concerns. Therefore, it is essential to carefully analyse and compare different strategies to ensure the most secure approach is implemented.

4.1 Analysis of theoretical implementations

All the authentication methods that are discussed in this thesis require a place to store some information that would be later on exchanged between client and server. As it was discussed in part 2 (literature review) and compared in the Table 1 the most secure and suitable from widely used methods is cookies storage. What is more, cookies are recommended as a storage mechanism for session-based authentication by different official sources, for example [32, pp. 38-39] and [33]. Therefore, both Session and JWT implementations will use cookies to store required information in.

4.1.1 Cookies

As the cookie is stored in the browser in plain text and it should not contain any personal information in it. In case of stateful technologies, such as session-based authentication, a cookie should only contain an identifier that was given by the server due to its storing information in plain text, which could be edited by malicious party. Cookie may contain additional information for the JWT, since JSON Web Token is signed by the server, so the information can be trusted in the later stage.

The `Set-Cookie` header is inserted by the server on the response to create a new cookie. There may be one or multiple `Set-Cookie` headers in a single server response depending on the need. Each cookie should have `cookie-name=cookie-value`, which should be unique [34, p. 45].

According to developer.mozilla.org [35], there are seven cookie attributes to work with. Older RFC variants, such as [33] and [36], mentioned less attributes. Currently available

attributes are: `Expires`, `Max-Age`, `Secure`, `HttpOnly`, `Domain`, `Path`, `SameSite`. Some of these attributes are used for different purpose, others are interchangeable. According to [35], these attributes could be divided into three groups: lifetime, access restriction and scope definition attributes.

The shorter lifetime of cookie is, the less chance for a malicious party to perform session hijacking and session fixation attack vectors. Both attacks are connected to getting access to the client's session identifier. Therefore, either `Expires` or `Max-Age` attribute should be set. The `Expires` attribute requires a date when a cookie should expire and is relative to the date on client's browser, rather than the servers. `Max-Age` requires a period of time (in seconds) after what the cookie should expire.

Access restriction attributes are used to restrict access to cookie from unforeseen parties or scripts. Cookie could be affected by the MITM (man-in-the-middle) attack due to being sent from the client's browser to the server. To mitigate the issue, there is a need of encrypted connection between client and server, so HTTPS shall be used. The `Secure` attribute blocks the possibility of using the unsecured HTTP connection apart from localhost case. It also means that insecure sites cannot set the `Secure` attribute to true. Cookie could be also affected to XSS (cross-site scripting) attack. As an example, the unintended JavaScript could try to access the cookie using `Document.cookie` API (Application Programming Interface). To mitigate this issue, `HttpOnly` attribute could be used [37]. Both `Secure` and `HttpOnly` should be used together to enhance the security of a cookie and mitigate MITM and XSS attack vectors.

Scope attributes are used to define the URLs (Uniform Resource Locators) where the cookie should be sent to. There are three attributes that are used to define scope: `Domain`, `Path`, `SameSite`. For cookie to be prone from CSRF attack, `SameSite` attribute should be set to `Strict`, so to restrict cookie to a first party. Additionally, I recommend undeclaring `Domain` attribute, since it will be automatically set to issuing host, excluding the subdomains of it. `Path` attribute could be used in specific use-cases but cannot be used to mitigate any attack vectors. If it is impossible to set `SameSite` attribute to `Strict`, it is needed to adopt a session management system that relies on two cookies [9][section 8.8.2], since, according to [9][section 5.5.7.1], setting the `SameSite` to `Lax` will only partially stop CSRF attack vector.

Concluding the discussion of different cookie attributes, I made the next example of a cookie prone to MITM, XSS and CSRF attack vectors:

```
Set-Cookie: id=01ce5983-6746-4123-b282-d12b2f42a112*†; Max-Age=120*; Secure;
HttpOnly; SameSite=Strict
```

* – cookie value of id and max-age will vary depending on implementation.

† – the id must not be constant; this is explained in the following section.

It should be mentioned that all of the above mentioned attributes are currently supported by all major browsers: `SameSite=Strict` [38], `HttpOnly` [39], `Max-Age` [40], `Secure` – every compliant to RFC6265 browser [33]. Implementing a cookie with such attributes sufficiently lowers malicious actors possible attack vectors and is sufficient to store authentication related data inside it. In the current thesis, I use `Set-Cookie` with flags mentioned in the example above to store authentication information in scope of both session-based and JWT-based authentication methods. In the next part of this thesis, I have in depth analysis of session-based authentication, and I assume the use of cookie for storing HTTP session related information on the client-side storage.

4.1.2 Session-based authentication

An HTTP session is a file that consists of different user-based information. The information that may be needed to be held can vary depending on the use-cases of the web-application. For example, language preference, application settings or, in our case, user's identity information, which is then used for the authentication process. As it was mentioned in the literature review part, this information is usually stored on the client-side storage.

In the previous section, I proposed the use of cookies as a client-side storage mechanism. In this part of thesis, you can get an exhaustive overview of session-based authentication security risks and an advice for secure session implementation. To begin with, it is essential to understand the process of how client receives the session. The process is followed by the general HTTP authentication framework also called as the challenge and

response flow. According to [41] and [42], the process could be described in the following steps:

First, the client requests access to a protected resource on the server by sending an HTTP request. The server responds with a 401 Unauthorized status code along with a WWW-Authenticate response header. The response should consist at least one authentication scheme that is supported by the server.

After the client received the 401 Unauthorized response and the WWW-Authenticate header from the server, it chooses one of the authentication scheme options provided by the server. The client sends a new request to the server with the Authorization header and authentication credentials included. Authorization header contains the authentication scheme and the user's credentials encoded in specified format.

The server receives the request with the Authorization header, decodes the credentials, and verifies them against the user's credentials stored on the server. In case credentials are valid, the server authorizes the login, saves a session in the database, and returns a cookie containing the session ID (identifier) to the user. Otherwise, server responds with 401 Unauthorized status code and the authentication process repeats.

The illustration of the process could be also seen in the Figure 1.

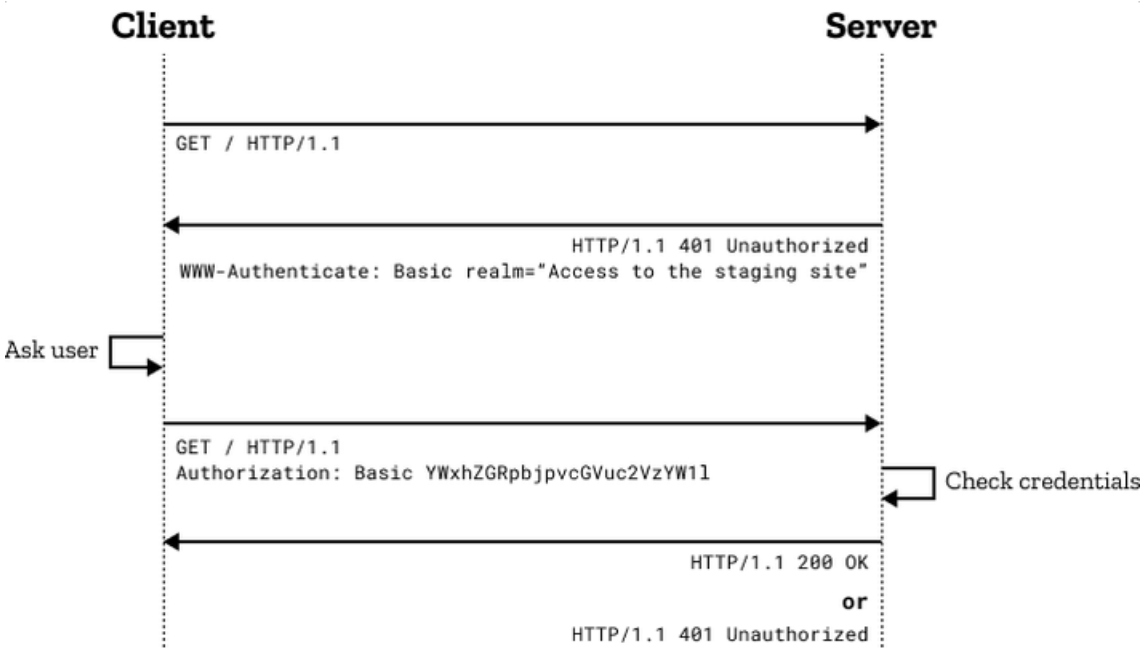

Figure 1. Example illustration of a HTTP session cookie establishment [41].

In the Figure 1 the “Basic” authentication scheme is used. In the illustration, it sends the credentials encoded but not encrypted. This is completely insecure and the secure SSL/TLS connection should be used instead HTTP [24].

There are several different attack vectors that could be used with session-based authentication methods, such as guessing attack (also known as session ID prediction), brute force and session fixation. Each of the attack vectors is analysed and solution is provided below in this part.

After the client is authenticated, the unique session identifier will be issued to the user. Server uses session ID as a unique key to search for client’s information in the database. According to OWASP [24] it is recommended to use identifiers that are at least 128 bits (16 bytes) length to prevent **brute-force attacks** on the session ID. The session identifier must only contain an identifier and no personal information should be stored in it. To mitigate **guessing attacks**, where a malicious actor tries to predict a valid session ID using statistical analysis, the session ID should be sufficiently unpredictable. OWASP advice to use a reliable Cryptographically Secure Pseudorandom Number Generator (CSPRNG) to achieve the sufficient level of randomness. The session ID value should provide a minimum of 64 bits of entropy in case of Pseudo-Random Number Generator (PRNG).

As an HTTP is a stateless system, it does not store any information in itself. It makes the response faster but brings a problem of storing the session credentials. To solve the issue, according to [33] and [24], cookies are commonly used. I have discussed the main advantages over other client-storage methods in paragraph 2 and secure implementation of cookies in paragraph 4.1.1. The main attribute for any cookie is its identifier. As mentioned before in this paragraph, the random session ID is used by the server to identify the session on the later requests and should never include any sensitive information. In addition to a random session ID, some implementations also include a secret key or seed value in the session generation process to further enhance the security to the session ID. An example solution would be to generate a session ID by combining a unique user ID

with a random number generated by the server, and then hashing the result using a secret key. However, this may bring additional load on the server.

In case of using sessions for the authentication, there is a need of keeping the session valid only for a particular time. As the common practice, the validity of session is lengthened each time the request is received from a client. Whenever it exceeds the time of expiration, the session becomes invalid and cannot be used anymore. It is crucial for the ID to be unique to prevent any duplication. Therefore, the random session ID generated must not already exist within the current set of session IDs. Whenever the site authenticates the user, it shall regenerate and resend the session cookie. This brings up more load to the server, as the database shall be used on every request. However, it is crucial to prevent **session fixation attacks**, where a threat actor could potentially use a user's session.

Overall, according to [5], HTTP sessions are widely accepted and used worldwide and have been tested in various environments for usage, management, and security. Furthermore, any new vulnerabilities discovered are promptly addressed. In case of following all the above-mentioned recommendations, it is possible to safely use session-based authentication. However, as I pointed out in part 2 of this thesis, there are alternatives to consider. One of them, which is often discussed as a session-based replacement, is token-based authentication. It is thought to be better in scope of scalability and performance [26]. I discuss token-based authentication in more details during the next section of this thesis.

4.1.3 Token-based authentication

Token-based authentication is usually advertised as the session-based authentication replacement. The token-based authentication is brought up as a session-based authentication alternative due to better scalability and performance [26]. Token-based authentication could also bring second factor for authentication. An authentication token could be divided into two different types: a physical token and a web token.

A physical token is a palpable device, which contains the user's information in it. It stores a secret key inside a physical device, which brings another layer of protection. It could be

also divided into two elements: hard tokens and soft tokens. Hard token requires an additional device, such as a smart card or a USB (Universal Serial Bus) dongle. Soft token is an easier implementation as it only requires a mobile phone or a computer to send the encrypted code from via authorized application or SMS.

A web token on the other side is a fully digital process, where the initial token is given in the similar way to session-based authentication. The client sends the user credentials, the server verifies them and generates a digital signature which is then sent back to the requestor (client) machine. The most widely mentioned implementation is known as a JSON Web Token (JWT), a standard for creating digitally signed tokens.

Concluding from the above, physical token implementation requires additional device or software from the clients, so it is harder to deploy this solution. As it was mentioned in the part 2 of this thesis, this solution is left outside of the scope of this thesis work.

For the web token, JSON Web Token will be discussed as the most widely used one.

JWT could be divided into two different sub-categories: stateless JWT and stateful JWT. The main advantages of stateless JSON Web Token could contain all the needed public data inside it encoded with developer chosen method. As the token is signed by the server, it could be trusted by default [25]. This reduces the amount of database calls. It is also thought to be easier to horizontally scale for stateless JWT [26].

On the other side, Stateful JWT is similar to Session authentication as it only contains a reference or an ID. This means that the amount of database calls will be same to session-based authentication and the advantage of scaling is lost. Due to that, stateful JWT is left out of scope in this thesis work. Stateless JSON Web Token implementation and security considerations is discussed in the next section of this work.

Stateless JSON Web Token implementation

When considering the implementation of JWT tokens as an authentication method, several advantages and disadvantages should be considered. One significant advantage of stateless JSON Web Tokens is that they are signed by the server, so could be trusted. Thus, JWT could reduce the count of calls created to the database. However, tokens are

not centrally managed and stored on the client side, which brings new challenges when a token needs to be revoked. Different sources address several issues connected to token-authentication. These are logout, user update and token prediction problems.

Logout problem – the validity of token is verified by the contents and ability of server to verify the signature. However, it is impossible for server to tell apart if the token is valid or invalid due to verifying the contextual information and the data that is stored inside the token.

User update problem – in case user changes the role or password, the previous token should be invalidated immediately. Otherwise, it could be possible to use previous JWT to access website contents with the previous privileges. The problem is divided into two subgroups: **user role change** and **user password change problems**. This is done due to some solutions providing workaround to only one or another part of the problem.

Preimage attack vector – the author of [30] highlights the potential for preimage attacks, I believe this issue is subject to debate. This is because JWTs can and should be used with commonly accepted algorithms that are secure against preimage attacks [43]. Although a successful preimage attack on the hash function alone would not reveal any confidential information, using a weak hash function in generating JWT signatures could potentially compromise the security of the system by allowing an attacker to create a forged JWT that the server would accept as legitimate. To mitigate this issue, I will follow secure to preimage attack vector hashing algorithms.

Token prediction problem – similarly to session fixation attack, tokens could be stolen by a malicious party and further used for malicious activities. This attack vector was slightly highlighted by [30]. This attack vector is similar to fixation attack and is mitigated in similar manner. To mitigate it, the token should be recreated on every request.

There are different workaround methods discussed to overcome these three problems: [11], [28], [29] and [30]. Each of these workarounds is focused on different problems and use different methods to solve them. The next workaround solutions are provided:

- Short-lived token [11], [28]
- Blacklisting [11], [28]

- Changing the plain JWT secret [28], [29]
- Changing the token on each client request [30]
- Changing the group/individual JWT secret [28], [29]

Each of these solutions has different pros and cons and solve different security vulnerabilities connected to JWT-based authentication. In the below part of this section, I am analysing and comparing them to each other.

Short-lived tokens are the tokens that have an expiry date included in them. The implementation described in [11, pp. 775-777] suggests the utilization of short-lived tokens to protect them from being misused by attackers who may gain access to them through attacks like CSRF. The limited lifespan of these tokens ensures that any potential attacker would have only a brief window to exploit them. This can partly solve the logout problem, since the tokens lifetime should be short. However, according to [28, p. 3], using this solution token revocation cannot be instantaneous. What is more, it will either "…provide significant performance overhead or a drastic drop in user experience", as the user will need to repeatedly log in. The proposed in [28] solution mentions short-lived tokens as an additional enhancement for access tokens: "Optionally, for increased security, it [JWT] may also contain an expiration date.". The solution does not provide any workarounds for the token prediction issue.

Blacklisting – keeping all invalidated tokens in a centralized database and verify JWT upon each request. This method brings back a centralized place of storage and increases database usage and was implemented by the [11] as well as discussed in [28]. JSON Web Tokens may have a cumulative effect, since tokens are stateless, meaning that once a token is issued, it contains all the information needed to verify the authenticity of the request it belongs to. To mitigate this issue, expiration date should be set on the token itself and there is a need of a job that will clear expired tokens from database. Even with that, there is a possibility of DoS (denial-of-service) attack vector. The malicious attacker could potentially fill database with revoked tokens. It could be fixed by limiting the number of token revocations that a single client could actuate with bringing timeouts after some number of revocations. From advantages of this method, blacklisting brings instant revocation of the token, so the logout issue is solved. User update and token prediction problems are not solved by implementing blacklisting.

Changing the plain JWT secret is another way to invalidate JWTs. The scope of this method is to change secret that the server/ servers used to generate JSON Web Tokens. Using changing of plain token secret, only one secret is used for all tokens signed. This approach does not require a centralized place of storage and invalidates all the tokens instantaneously. However, it can lead to increased load on the systems, which may impact the initial scalability of JWTs. As a result, it is not recommended for large-scale use cases due to non-linear load functions [28]. This method was not used in proposed solutions by any of investigated works.

Changing the group/individual JWT secret – this is a proposed enhancement for "Changing the plain JWT secret" method. Different papers propose to use more than one secrets for generating JSON Web Tokens. [28] suggests dividing users on different groups, where each group will have different JWT secret. This solution will provide instantaneous token revocation while retaining scalability of JSON Web Tokens. This solution proposes to have access tokens and refresh tokens. Access tokens should contain user public information inside that will be used later and they are used for authentication on the web application. Access tokens have small validity time. On the other side, refresh tokens are used for updating the access token and have the long validity time. It is also mentioned that for the refresh tokens is not necessarily a JWT. One of possible issues that provide [28] is that the system is vulnerable for DoS attacks. Although it is possible to make it harder for attackers to carry out token revocation attacks by grouping clients, a small number of clients can still cause a large number of such events. Similarly to blacklisting DoS attack vector, one possible way to address this issue is to impose restrictions on the number of token revocations that a single client can initiate and introduce timeouts after a certain number of revocations. To further understand the architecture of solution provided by [28], please refer to the Figure 2.

Individual JWT secret technique was advised by [29]. The technique is to combine a secret key with a hash value of user's password, to produce a dynamic secret key. In case the client notices a malicious activity and changes the password, the old token will become unverifiable. According to author, it will eliminate the need for blacklisting database. However, from my point of view, [29] does not provide any method of logout and the token will be revoked only after client changes the password. Thus, there could be time after the malicious party received the token and before the client changes password, to misuse the token and gain unauthorized access to the user's account.

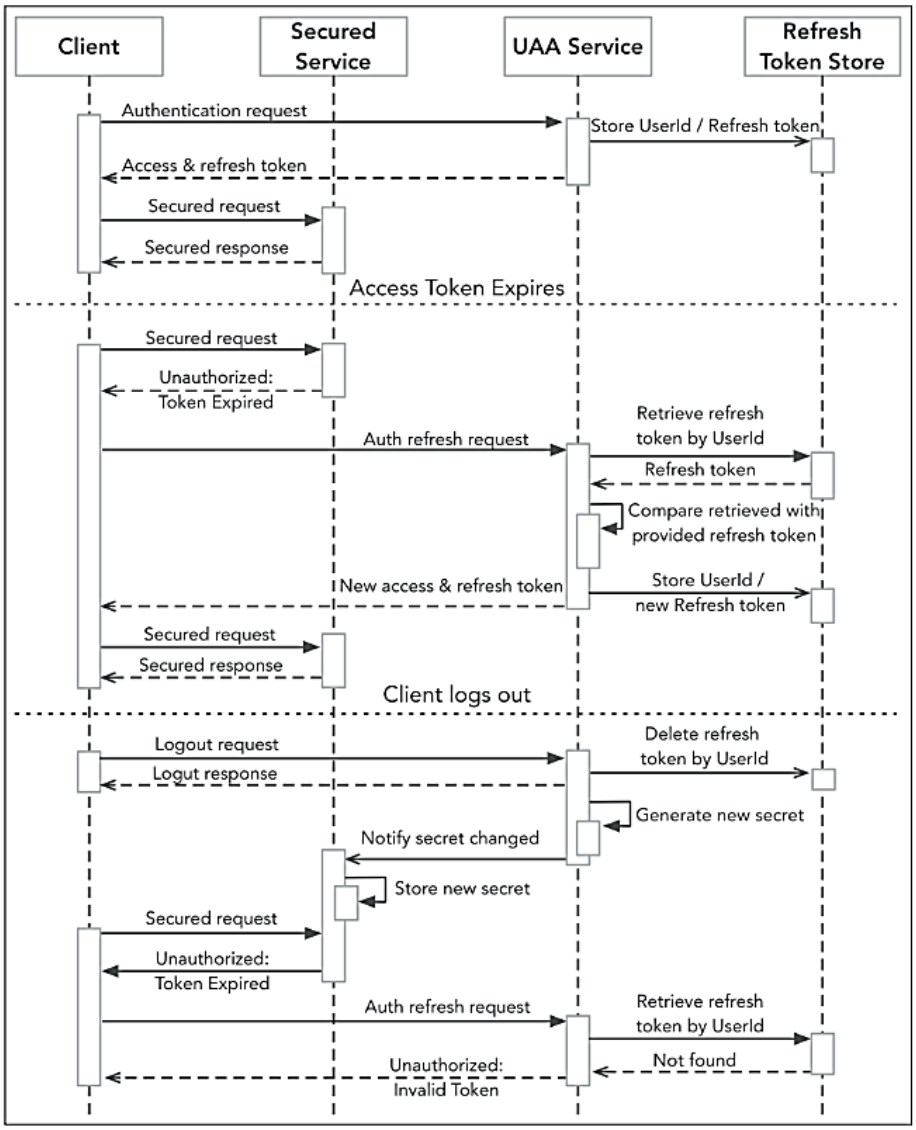

Figure 2. Architecture of JWT solution 1 [28].

Changing the token on each client request – this method, suggested in [30], involves creating a unique random-access token using a signature calculated with the client request time, server response time, and a random integer added to the payload of token data. This method should not be confused with secret change, as the secret used for all the tokens is left intact – one secret for all the users, no revocation of secret is happening. The proposed approach solves the predictability issue with JWTs, as the token will have a unique data inside it on each client request. It also addresses the problem of malicious actors or clients retaining privileges from the previous request, since the: "On new token generation recent user role will be picked to avoid further secured resource access". To fulfil this changing the token on each client request, this method requires additional database, as there is a need of storing the next information: "…email from credentials, role as 1, SRT [Server Response Time] and Rand value from system against JWT". If the role of the user is

changed, role status should be changed to 0 in the database. According to the author of papers: “information of valid role can be retrieved the with time complexity of `O(1)`.” Author of [28] brings similar complexity of `O(1)` for blacklisting, which, according to him: “introduces a considerable performance overhead for each request”. However, in contrast to blacklisting, the additional information added by this method, does not require to be stored forever, so there is no cumulative effect.

According to the verify process step 8: “If the JWT and `JWT is not matched or role is 0 server will reject the request and move to step1. If role is 1, the server will generate `JWT using `CRT [same as CRT in request, it will change in next request from same client], SRT, and the rand value.” Whereas step 1 is a new login http request from the client. Therefore, in case of role change or JWT mismatch, the affected user shall re-log in to the server. Considering the information above, it seems that it will generate additional server load, since each request will require either token regeneration or start of authentication process from step 1. Besides that, this approach does not provide any solution for logging out.

The full authentication process could be seen on the Figure 3 or read on [30, p. 3]

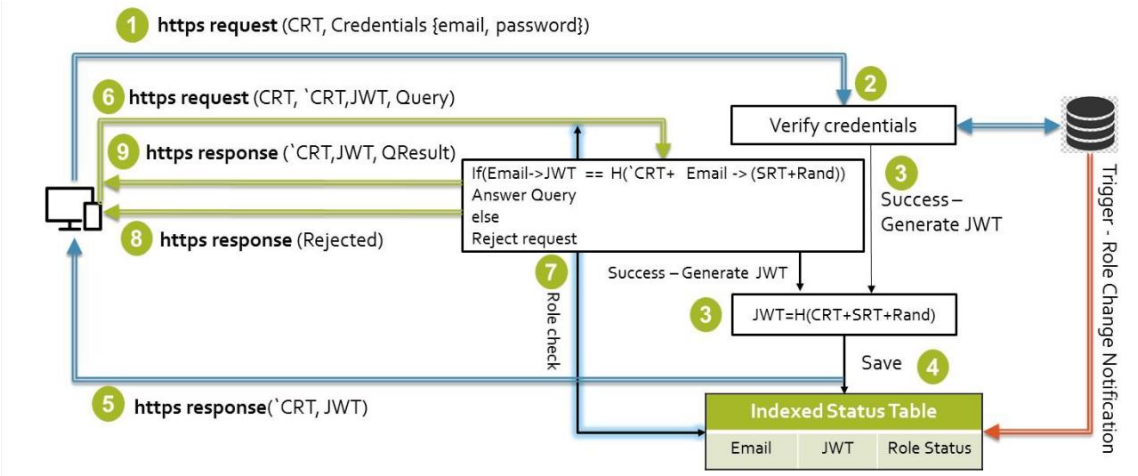

Figure 3. Architecture of JWT solution 2 [30].

Summarizing, while the token recreation per user request approach can solve the user role change or token predictability, when it comes to user logout, it has limitations there. Moreover, this method provides additional overhead in terms of database and overall server usage. What is more, the initial scalability of stateless token is to be questions, as such token shall be recreated on each request.

To provide an overview and compare different proposed solutions, three tables were built. In the first table, I am comparing papers to solutions that each of the methods propose. In scope of secret change only "group/individual JWT secrets" are discussed. The plain JWT secrets are left out from the table, as no solutions propose to use this solution.

Table 6. Methods used by each of investigated JWT solution.

Method proposed by	**[11]**	**[28]**	**[29]**	**[30]**
Short-lived tokens	✓	✗	✗	✗
Blacklisting	✓	✗	✗	✗
Secret change	✗	✓	✓	✗
Token change	✗	✗	✗	✓

Using short-lived tokens together with blacklisting may solve the issue. However, according to [28], it will increase load on the systems and negatively impact the initial scalability of JWTs. In the above analyse of token change methodology proposed by [30], I have distinguished an increased load and scalability issues as well. However, Table 6 does not impact the final grade of proposed methods, as load and scalability are further investigated in Table 8 and Table 9.

The next table summarizes the issues that are solved by each of the proposed solution.

Table 7. Issues solved by each of investigated JWT solution.

Issues solved by	**[11]**	**[28]**	**[29]**	**[30]**
Revocation on client logout	✓	✓	✗	✗
Revocation on client password change	✓	✓	✓	✓
Revocation on client role change	✓	✓	✗	✓
Preimage attack vector	✓	✓	✓	✓
Token prediction attack vector	✗	✗	✗	✓
Final Grade	4.0 / 5.0	4.0 / 5.0	2.0 / 5.0	4.0 / 5.0

Additionally, I created a table based on the information presented in [28] (Table 1. Comparison of different JWT revoking methods). The table compares architectural complexity, invalidation latency, acquisition frequency and scalability of proposed workaround methods.

Table 8. Technical comparison of investigated JWT revoking methods.

Technical Characteristics	**Short lived**	**Blacklist**	**Secret change**	**Token change***	**Group secret change**	**Individual secret change**
Architectural complexity	Low	High	Low	High†	Medium	Medium‡
Invalidation latency	Non-Instant	Instant	Instant	Instant	Instant	Instant
Acquisition frequency	High	N/A	High	N/A	Variable	N/A
Estimated scalability	Linear	Bad	Very Bad	Bad†	Linear	Linear‡

The data provided by the [28] is a baseline of this table. I am evaluating new solutions ([29] and [30]) by the methodology proposed by [28]. The solutions are token change on each request (written as token change in the table*) and individual secret change. (†) *Token change on each request* solution is similar to blacklist in terms of compared data, as both require database checking operation on each client request. Both provide a shared, centralized place for storing some information. Therefore, the architectural complexity and estimated scalability are set to same as blacklist. (‡) Due to implementation similarities in *individual secret change* and *group secret change*, the architectural complexity and estimated scalability for *individual secret change* stays on the same level as for the group secret change method. The next table (see Table 9) gives the final overview of each workaround technical comparison and their final technical grade.

Table 9. Technical comparison of investigated JWT workarounds.

Technical Characteristics	**[11]**	**[28]**	**[29]**	**[30]**
Architectural complexity	High	Medium	Medium	High
Invalidation latency	Instant	Instant	Instant	Instant
Acquisition frequency	High	Variable	N/A	N/A
Grade	2.0 / 3.0	2.0 / 3.0	2.5 / 3.0	2.0 / 3.0
Estimated scalability	Bad	Linear	Linear	Bad
Additional Grade	0.0 / 1.0	0.5 / 1.0	0.5 / 1.0	0.0 / 1.0
Final Grade	2.0 / 4.0	2.5 / 4.0	3.0 / 4.0	2.0 / 4.0

From the analysed data, it is possible to draw conclusion. Assessing the Table 6, solutions proposed by [28] and [29] seem to be more efficient in scope of system load and scalability. However, as it was already mentioned, this is not measured, as Table 8 evaluates the scalability as well. Evaluating the Table 7, the solution proposed by [29] seems to be the least secure one with only two points. Other solutions have four points each. To evaluate the data of Table 9, I should mention that solution proposed by [11] combines short lived tokens and blacklisting. In combination, the solution proposed by [11] will have similar characteristics to blacklisting solution, as the architectural complexity staying on the worse level – high, invalidation latency becoming instant, acquisition frequency becomes N/A and scalability of such solution is bad. Workaround proposed by [28] implements group secret change, [29] – individual secret change and [30] – token change on each request. Therefore, in the technical comparison [11] receives 2.0 points, [28] – 2.5 (= 2.0+0.5) points, [29] – 3.0 (= 2.5+0.5) points and [30] – 2.0 points.

Table 10. Final grades of investigated JWT workarounds

Grades	**[11]**	**[28]**	**[29]**	**[30]**
Issues solved	4.0 / 5.0	4.0 / 5.0	2.0 / 5.0	4.0 / 5.0
Technical Characteristics	2.0 / 4.0	2.5 / 4.0	3.0 / 4.0	2.0 / 4.0
Final Grade	6.0 / 9.0	6.5 / 9.0	5.0 / 9.0	6.0 / 9.0

In summary, in case we count the estimated scalability, the implementation proposed by [28] appears to be the most secure and suitable for further project developments, with the highest grade of 6.5 points out of 9.0 points. However, to further enhance acquisition frequency of group secret change method, my proposal is to adopt secret generation logic which was discussed in the [29] solution. This means, to use group secrets as proposed in [28], but using hash of the user's password to it. This would improve the acquisition frequency and increase the complexity of the secret while still requiring a valid password hash from the database for a malicious party to sign a new token even if the group secret is leaked. This workaround will be used in a practical implementation part and is referred as *my proposed solution 1* in next parts of this thesis.

Another potential solution that I would like to propose is created on a basis of [30], as it has similar grade to [28] but exhibits comparatively lower estimated scalability, I propose to combine it with [29]. It is possible to implement logout leaving one secret for all the users as proposed in [29]. To accomplish this, a dynamic variable unique to each user could be introduced and stored in the database. This variable would be changed every time a user logs out, and its secure generation could be achieved using the techniques discussed in [30], such as combining the Client Request Time and Server Response Time with a randomly generated value. This value shall be then used while generating the secret. The secret could then be generated using this dynamic variable and, for added security, the hash of the user's password. After the user logout, the value shall be changed. JWT itself could contain an IP address of the client to verify the authenticity of requestor by verifying that the IP address has not changed. The IP verification may also help with token prediction issue. However, the current version of this solution has a user-experience flaw connected to a potential client logging in with more than one device. Either several

records per person shall be created in the database or there is a need of limiting connections to only one machine login at a time per account. There is also a need of investigation if this use-case provides better scalability and/ or lower server load of server/ servers. In conclusion, the above-described workaround solution is outside the scope of this thesis and is not the subject of a thorough investigation. Instead, it is recommended as a possible direction for future investigation. This solution is referred as *my proposed solution 2* in next parts of this thesis.

While the focus of this thesis work is to analyse security aspect and create a sample implementation of in-house solutions: session-based authentication and JWT-based authentication, in the next part I provide a brief discussion of third-party based authentication to offer a more comprehensive overview of authentication topic and bring up ideas for further development.

4.1.4 Third-party access-based authentication

Third-party access-based authentication is another method that could be used for authentication purposes. Even though third-party based authentication is not an in-house solution and left out of scope of this thesis work, I want to provide a brief overview of the technology and provide some recommendations for further investigation.

Third-party access is a process, when, the organization creating the web-application authentication, relies on external vendors account database. Both pros and cons of third-party based authentication were brought up during the part 2 of this thesis. Relying on this comparison, I would like to emphasise that the organization should carefully evaluate the security and reliability of the external vendor before deciding to use their authentication service. They should also have a contingency plan in case the vendor experiences an outage or security breach. Finally, the organization should consider providing alternative authentication methods for users who cannot or do not wish to use external vendor authentication.

One of the widely discussed technologies of third-party based authentication is OpenID/ OpenID Connect [20], [21]. It is a decentralized open-source community project, which is supported by Google, Facebook, Yahoo!, Microsoft, AOL, MySpace and others [44].

While OpenID was the first protocol to provide decentralized single sign-on (SSO) functionality, OpenID brought additional security and functionality to the standard. This includes support of OAuth 2.0 authorization flows, user information sharing, and management of sessions. I would like to highlight OpenID Connect implementation, as it has some similarities to token-based authentication. It uses an identifier token to authenticate end-users, which is represented as a JSON Web Token (JWT) according to [45] (paragraph 2. ID Token). Cookies are also mentioned as a possible storage mechanism. Additionally, paragraph 16.18 of [45] states that "The Authorization Server SHOULD provide a mechanism for the End-User to revoke Access Tokens and Refresh Tokens granted to a Client". This suggests that OpenID Connect may help mitigating issues discussed in section 4.1.3 of this thesis work by providing a mechanism for users to revoke access tokens and refresh tokens, rather than requiring custom implementation by the organization. Given these advantages, it may be worth exploring different OpenID Connect solutions to see if there are any additional security measures that could be implemented in a JWT-based authentication solution.

Summarizing, OpenID Connect uses an identifier token to authenticate end-users and provides a mechanism for users to revoke access tokens and refresh tokens. While it could be a potential solution for authentication, it may not be suitable for all potential users as they may not have an external account. As I mentioned above, only in-house solutions are in the focus of this thesis and third-party based authentication is not discussed in the next part: analysis of practical implementations.

4.2 Analysis of Practical implementations

In scope of this thesis, session-based and token-based authentication were discussed, including their potential vulnerabilities and, for session-based authentication and JWT-based authentication, analysis of theoretical implementations was done. I have also introduced OpenID and OpenID Connect as third-party authentication methods that shift the responsibility for authentication to external vendors, thereby reducing the organization's security responsibilities.

In this part, practical implementations of session-based and token-based authentication methods are further discussed. From the token-based authentication method, JSON Web Token was chosen for the implementation (4.1.3). It was also mentioned in parts 2 and

4.1.4 of this thesis that third-party based authentication is left out of scope of this thesis. Both session- and JWT-based authentications are to be stored in a cookie, as discussed in 4.1.1.

Despite an exhaustive search, no existing solutions were found that fully implemented methods outlined in the part 4.1 (Analysis of theoretical implementations) while adequately addressing all security concerns. To address this gap, pre-existing solutions will be used as a starting point and subsequently modified to ensure their robustness in addressing security concerns mentioned in this thesis work.

The resultant codebase includes back-end functionality for user login, logout pages, as well as a verification page that is only accessible to authorized users.

As base solutions for both session-based authentication and JWT-based authentication, several solutions such as [46], [47], [48] and [49]. The code that I developed is publicly available on GitHub: [50].

As a matter of personal preference, NodeJS was selected as the implementation platform for the practical section of this study. This decision was based on NodeJS's compatibility with the requirements of the project, particularly its ability to address any security concerns that may arise. Both solutions require some data to be stored in the database. My decision to use MySQL was influenced by, according to [51] and [52], its widespread popularity and my own prior experience with it.

Both session-based and JWT-based solutions need to store hashed passwords in database. I have chosen to use Argon2 hashing algorithm, which won Password Hashing Competition 2015 [53] and is recommended as a hashing algorithm by [54].

There was also a need to generate universally unique identifier (UUID) in both solutions. There are different modules that could generate UUID available for NodeJS, two widely used are `crypto.randomUUID` and `uuid.v4`. According to [55] and [56], `crypto.randomUUID` is better than `uuid.v4` in scope of speed, even though both are based on similar RFC standard. Therefore, `crypto.randomUUID` was chosen for UUID generation.

The final structure of the developed project consists of three folders: jwt-auth, postman and session-auth. Folders jwt-auth and session-auth contain my developed code for JWT-

based and session-based authentication methods accordingly. I have tried to retain a similar code structure and style, so it would be easier to understand and compare both solutions. Due to developing only back-end code in scope of this thesis work, all the testing was done via postman application. For the ease of verification, I have added postman folder that consists so called "postman collections" – preconfigured JSON files, which could be used to test all the implemented functionality. It should be noted that on the POSTMAN screenshots in the next sections, the cookies will not have the Secure attribute set to true. This is due to conducting all tests on a local machine, where setting the `Secure` attribute to `true` will not work. However, in the production environment, the `Secure` attribute should be set to true. What is more, `SameSite` attribute is not visible in POSTMAN even though it is set to `Strict`.

In the next two sections of this part, I provide an in-depth discussion of each of two aforementioned implementations.

4.2.1 Session-based authentication

The analyzation of session-based authentication done in scope of part 4.1.1 proposes that session-based authentication is widely used worldwide and has enough security by default. It is needed to store some information about sessions on the server side. Therefore, I have created a simple database, which consists of two tables: users and sessions Figure 4.

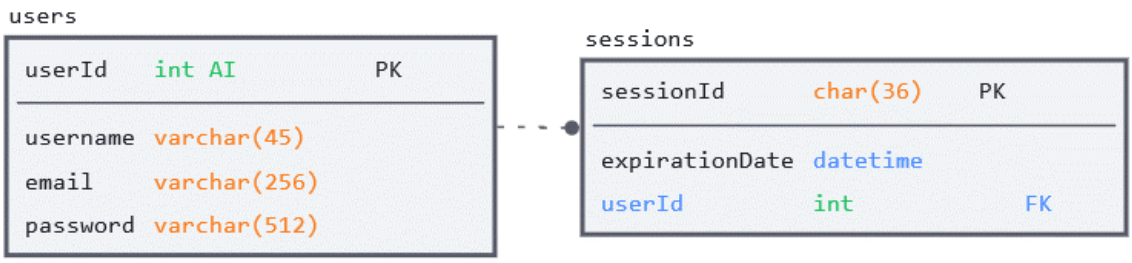

Figure 4. Database architecture for Session-based authentication.

Table users consists of: `userId`, `username`, `email` and `password`. `userId` is a primary key, `username` and `email` fields are unique and `password` field consists of hash of the password. The sessions table consists of: `sessionId`, `expirationDate` and foreign key to users table (`userId`). To make sessions secure, `sessionId` is generated using `crypto.randomUUID();` which is, according to [57], compliant to OWASP recommendation [24] of having a session ID at least 128 bits (16 bytes). Additionally, `sessionId` is set to be a unique value, so no duplication is possible. To do

so, custom function `generateRandomUUID()` verifies that the `sessionId` is not yet in the database. In case it is, the function tries to regenerate the session for ten times. If the function fails to generate a valid `sessionId` for ten times in a row, then it returns Internal Server Error 500. It is also required to regenerate and resent the session cookie on each request to prevent session fixation attacks. To resolve that, the `refreshHandler` is used as a callback function and should be used on every authenticated client request. A showcase of the `refreshHandler` function is the "`/welcome`" page, which is handled by `welcomeHandler`. The handler verifies that the user is authenticated and refreshes client's session by calling the `refreshHandler`. In case of success, a "Welcome `${userId}`!" message is returned to user (refer to Figure 6). The initial login to the web application could be seen on the Figure 5. As a data sent to the server, I have inserted valid credentials into the body. The credentials are then verified with the corresponding ones from the database. In case they are the same, the `loginHandler` issues a new cookie `sessionToken` with a `sessionId` inside. Additionally, it stores the information in the `sessions` table in the database.

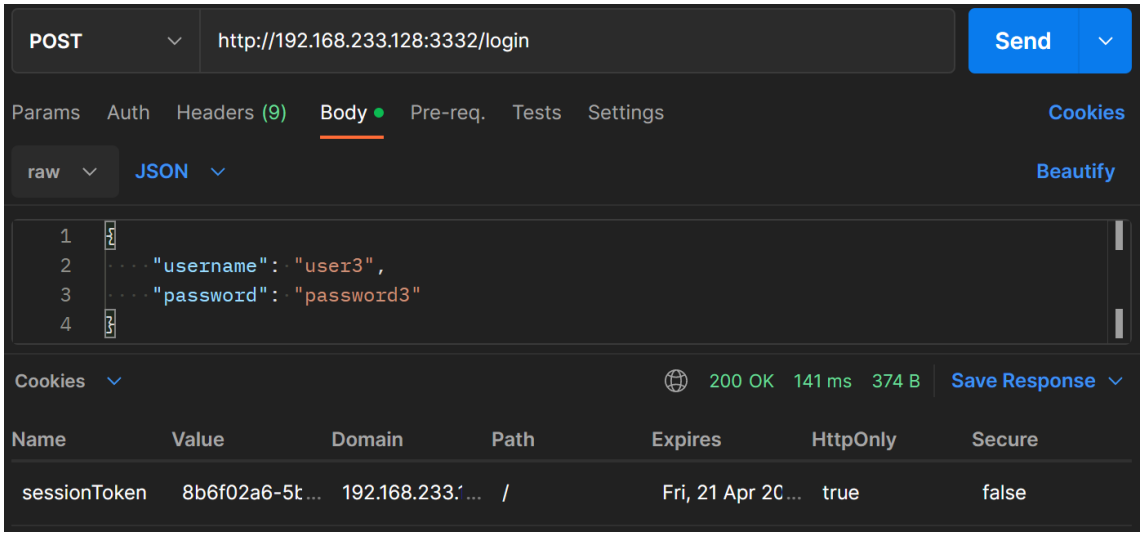

Figure 5. Login page in session-based solution via POSTMAN.

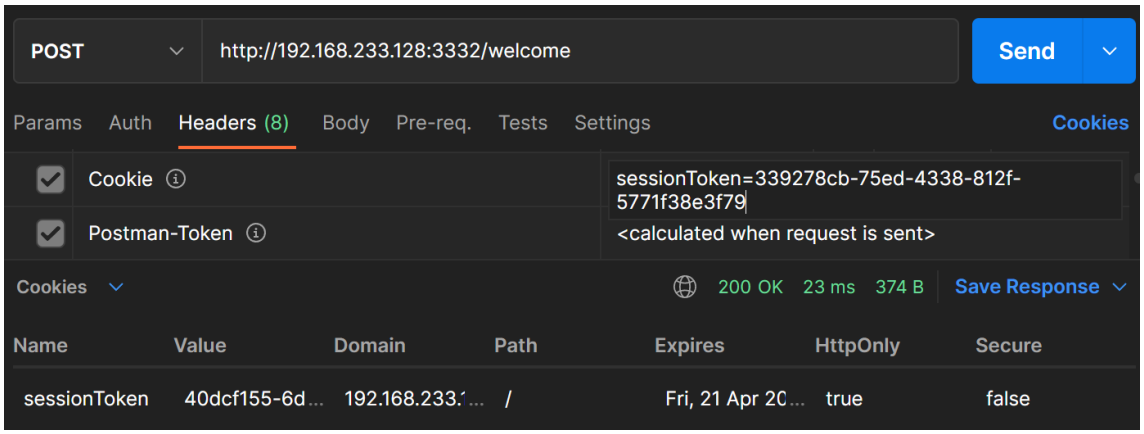

Figure 6. Welcome page in session-based solution via POSTMAN.

To log out from the web application, the client should go to "`/logout`" page as could be seen on Figure 7. The "`/logout`" page then calls `logoutHandler`, which searches for the `sessionId` provided as a cookie in the `sessions` table. If it is found, the `sessionId` gets revoked from the database and the cookie is replaced with a blank one, by setting "`Set-Cookie: sessionToken=; Path=/; Expires=Thu, 01 Jan 1970 00:00:00 GMT`".

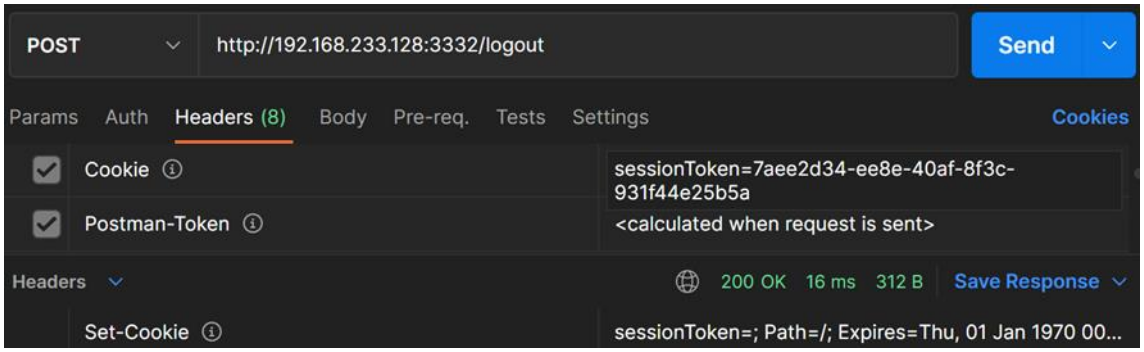

Figure 7. Logout page in session-based solution via POSTMAN.

Summarizing, my practical session-based solution implements all the security concerns discussed in paragraph 4.1.2. All the technical tests that I conducted during the development were successful and I was unable to perform any successful attacks from the discussed attack vectors. As an example of handled attacks, I tried to change data of cookie storing the session. Additionally, I conducted tests, which confirm that the session is verified and regenerated on every user request. Moreover, I affirmed that it is highly unlikely to brute-force the `sessionId` in the session lifetime, as the session is generated according to OWASP recommendations. In the next part, I discuss my JWT-based solution.

4.2.2 JWT-based authentication

In scope of part 4.1.3 I have analysed different proposed workaround architectures and made a decision to base my solution JWT-based authentication on architecture proposed by [28] with some additional implementations discussed in [29]. The architecture of authentication service could be seen on Figure 2.

The database created for the JWT-based authentication could be seen from Figure 8. There are three tables in total: `users`, `userGroups` and `refreshTokens`. `users` table is similar to discussed in session-based authentication apart from additional foreign key to `userGroups` table (`groupId`). `userGroups` table consists of `groupId` and `secret`. As it was proposed by [28], each user will be assigned to one of groups from `userGroups`. Each of the groups should have its own secret, which is regenerated whenever one of group users logs off.

To fulfil these needs, the function `generateTokenSecret();` is used, which returns `crypto.randomBytes(64).toString('hex');`. This method is mentioned as secure by [55]. The returned string is then combined with client's password hash and is used as a secret in the generation of `accessToken` process. The `accessToken` is a JSON Web Token, which is stored in a cookie and is set to be valid for five minutes. To lower the database calls, `accessToken` consists of client's `username` and `groupId`. Afterwards, `accessToken` could be refreshed using `refreshToken`, which is a cookie with randomly generated UUID. The UUID is stored in `refreshToken` table and is connected to user account to verify client and give a new `accessToken`. Additionally, this table contains `expirationDate` row, which is verified whenever user tries to refresh token. In case the date is expired, the server removes it from database.

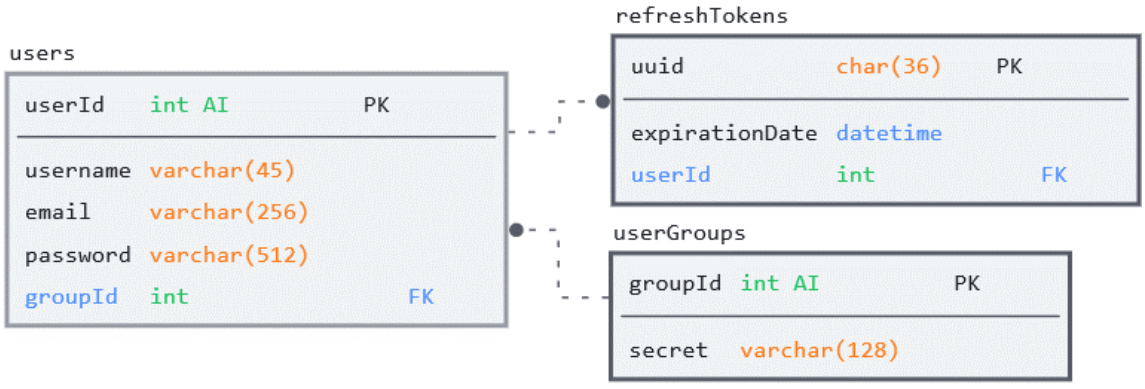

Figure 8. Database architecture for JWT-based authentication.

On the Figure 9 you could see the successful authentication process. To login, I have provided sample credentials stored in the `users` table. As the response, I have received two cookies: `accessToken` and `refreshToken`.

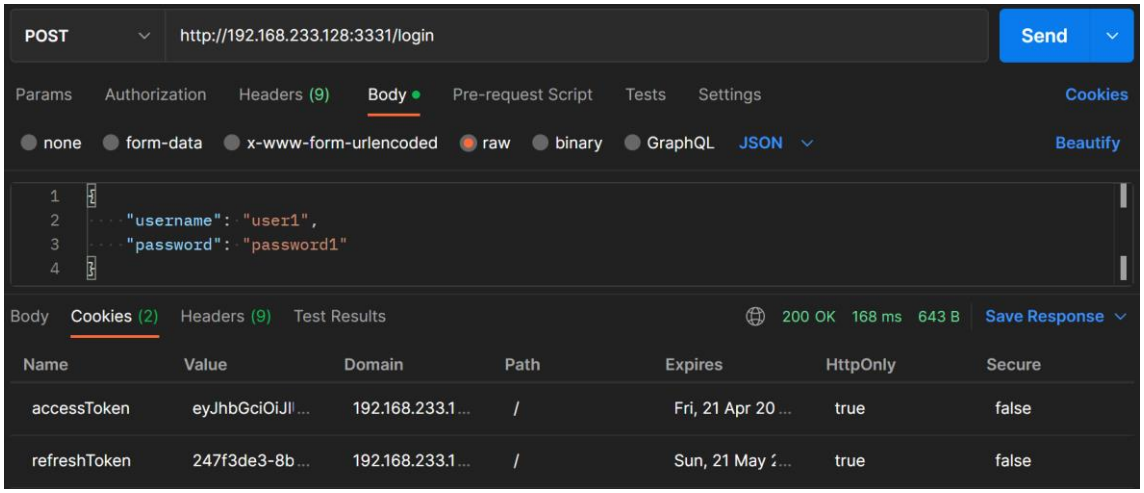

Figure 9. Login page in JWT-based solution via POSTMAN.

Next, on the Figure 10, I have successfully verified that the `accessToken` received on the login page works. In case "`/welcome`" page receives the correct `accessToken`, it responds with simple JSON object that contains username and groupId of the user. Otherwise, it responds with "403 Forbidden" error.

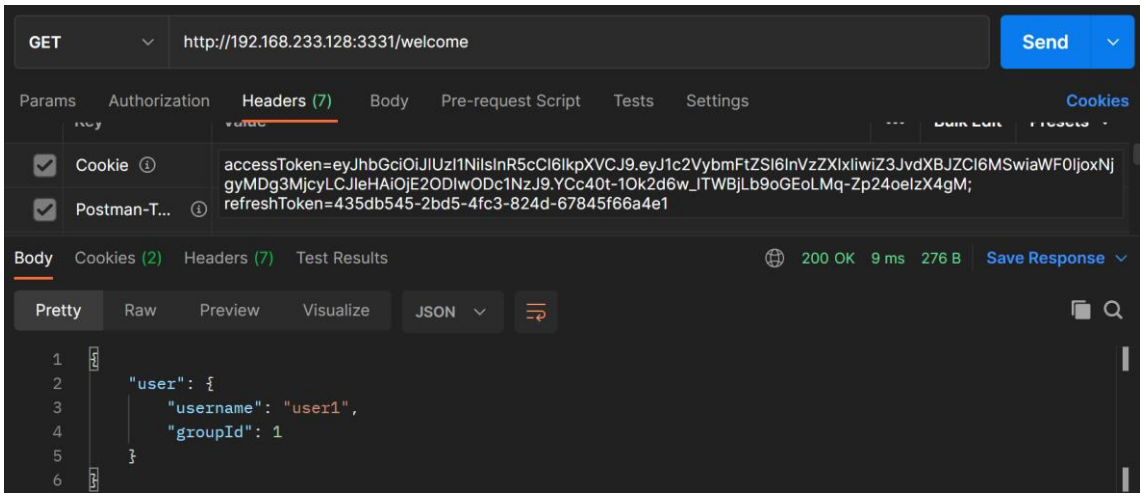

Figure 10. Private page in JWT-based solution via POSTMAN.

Afterwards, on the Figure 11, I have successfully refreshed the `accessToken` using `refreshToken`. In case "`/refresh`" page receives the correct `refreshToken`, it responds with two new cookies containing new `accessToken` and `refreshToken` (previous `refreshToken` is revoked). If the `refreshToken` cannot be verified, the server responds with "403 Forbidden" error.

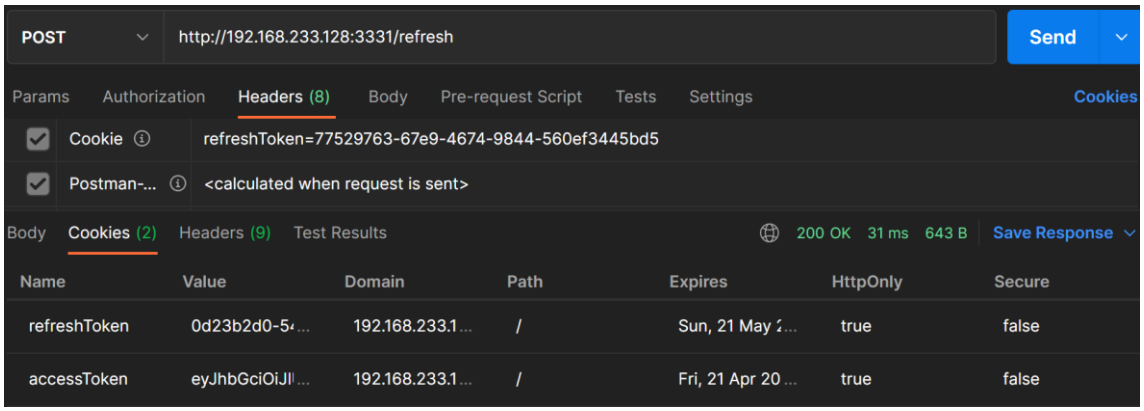

Figure 11. Refresh page in JWT-based solution via POSTMAN.

The next Figure 12 provides a screenshot of "`/logout`" page. If there is an `accessToken` available, the code gets `groupId` from the token and changes the secret for the group. In scope of `refreshToken`, the `uuid` is revoked. Both cookies are replaced with "fake" ones – blank expired cookies.

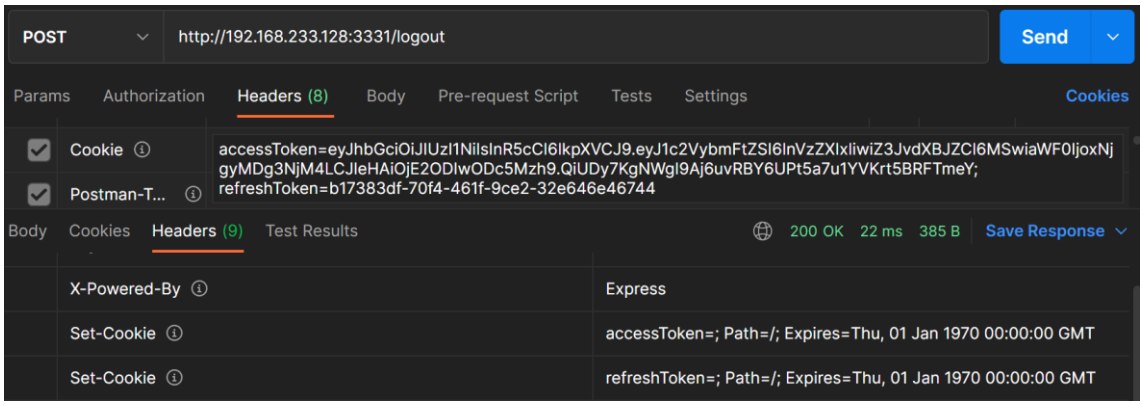

Figure 12. Logout page in JWT-based solution via POSTMAN.

Overall, my practical JWT-based solution implements four out of five security concerns discussed in paragraph 4.1.3. The only unimplemented attack mitigation vector is token prediction. Token prediction mitigation was left out of scope of this solution, as implementing it requires additional technical investigation and may be incompatible with the overall system design. Incorporating this attack vector mitigation does not undermine the core objective of maintaining a stateless state within the JWT-token. All the technical tests that I conducted during the development were successful. The only successful attack vector that I was able to implement from the discussed above is token prediction. As an example of attacks held, I manipulated the dates of cookies. Additionally, I deliberately modified information inside the JWT-token to check if my code verifies the JWT signature at each step, thus it is impossible to refresh or access any page with the

compromised/ fake token. Moreover, I conducted different revocation scenarios, testing possibility of bypassing the logics of revocation mechanism.

In the next part, I assess my technical implementations considering the security challenges covered in previous parts of this thesis.

4.3 Result Evaluation

In this section, I evaluate both my practical implementations according to security issues that were addressed in this thesis and technical comparison of my session-based and JWT-based implementations. I should highlight that for the JWT-based implementations, I brought up two possible solutions in scope of 4.1.3 *my proposed solution 1* and *my proposed solution 2*. As *my proposed solution 2* requires additional investigation and was not implemented in the scope of this thesis, it is left out of scope for result evaluation.

Table 11 compares issues that were discussed during this thesis. Each of issues has solved status evaluation, where ✓ – solved or ✗ – unsolved (see Table 4).

Table 11. Issues solved comparison of my implementations.

Issues solved by	**My session-based implementation**	**My JWT-based implementation**
Revocation on client logout	✓	✓
Revocation on client password change	✓	✓
Revocation on client role change	✓	✓
Preimage or brute force attack vectors	✓	✓
Session fixation or token prediction attack vectors	✓	✗
Final Grade	5.0 / 5.0	4.0 / 5.0

For my session-based implementation, all the brought-up issues were solved. Therefore, the final grade of solution is 5.0 out of 5.0. For my JWT-based implementation, all but token prediction issue were solved. Therefore, the overall grade is 4.0 out of 5.0. Token prediction problem should be further investigated in future works. It may be that token

prediction is solved by *my proposed solution 2*. It may also be that fixing the token prediction problem, the initial idea of stateless JWT will be lost.

It should be noted that during development phase, I held different security related tests. As an example of such tests, I conducted tests, where I manipulated the dates of cookies for both session- and token-based authentication. Additionally, I deliberately modified information inside the JWT-token to check if my code verifies the JWT signature at each step. Moreover, I held different refresh and revocation scenarios, testing possibility of both refreshing or revoking the session/ token.

The next Table 12 gives an overview of technical comparison of both my session-based and JWT-based implementations. This table is based on Table 4 from methodology part.

Table 12. Technical comparison of my implementations.

Technical Characteristics	**My session-based implementation**	**My JWT-based implementation**
Architectural complexity	Low	Medium
Invalidation latency	Instant	Instant
Acquisition frequency	N/A	Variable
Grade	3.0 / 3.0	2.0 / 3.0
Estimated scalability	Bad	Medium
Additional Grade	0.0 / 1.0	0.5 / 1.0
Final Grade	3.0 / 4.0	2.5 / 4.0

The architectural complexity for session-based authentication is set to low, as there is need of only one database with `sessionId` and its `expirationDate` to verify the user's authenticity. My JWT-based implementation has a medium architectural complexity as there is a need of managing two different type of tokens `accessToken` and `refreshToken`. Each of two tokens has different logics behind the verification process and requires specific database, so two database and more custom logic was required to have working solution. Both session- and JWT-based implementations have an instant invalidation latency, which means that after the `sessionId`/ `secret` is revoked, the `session` or `accessToken` is instantly invalidated across the system. There is no acquisition frequency for session-based authentication as each of the user's has unique `sessionId`. As the JWT-based solution has group secrets, the solution is left with

variable acquisition frequency, as it was in the Table 9 for solution provided by [28]. This will depend on the algorithm used by developers. Estimated scalability for session-based authentication is bad as the session should be regenerated and stored in the database on each client request to mitigate session fixation attack. For my JWT-based implementation token prediction attack is not solved, so it does not bring additional load to the system. However, the group secret revocation will require each group user to make an additional `accessToken` refresh request, which will increase the additional database load. Additionally, it is required to retrieve a client secret and verify the token on each request. Due to this, the estimated scalability of JWT-based solution is set to medium level.

Table 13. Final grades of my implementations.

Final evaluation of	**My session-based implementation**	**My JWT-based implementation**
Issues solved	5.0 / 5.0	4.0 / 5.0
Technical Characteristics	3.0 / 4.0	2.5 / 4.0
Final Grade	8.0 / 9.0	6.5 / 9.0

Overall, based on information from Table 13, my final session-based authentication solution has a grade 8.0 out of 9.0 points available. My JWT-based authentication implementation has a lower final of 6.5 out of 9.0 points. However, further investigation is needed to compare and verify the scalability of each solution.

While both solutions are functional, they have some aspects that were not implemented and could be investigated in future works. Firstly, both the session-based and JWT-based implementations have tables with a column called `expirationDate`. Specifically, the session-based implementation has a `sessions` table with this column, while the JWT-based implementation has a `refreshTokens` table with the same column. The `expirationDate` is verified in both solutions and cannot be used after the expiration. However, they are still stored in the database until requested. This can cause the database to become bloated over time. Therefore, an automatic service is needed to periodically remove expired rows from the above-mentioned tables. This will keep the database running smoothly. Additionally, it should be mentioned that the database for this example has been manually updated with all the essential data. This indicates that using the current solutions to register, edit, or delete users is not possible. What is more, the JWT-based

solution requires manually grouping every user into a certain `userGroup`, which is impractical in a production environment. Instead, in the JWT-based solution, a specific algorithm needs to be created to automatically assign users to their relevant groups.

Secondly, comprehensive testing was conducted throughout the development and implementation phases to validate the efficacy of the outlined security aspects. However, I recommend performing a practical verification of the developed code's security aspects, aiming to identify any potential security vulnerabilities or oversights that may have been overlooked. Furthermore, I suggest conducting a comprehensive comparison of scalability of each solution in practical scenarios.

Lastly, I propose to investigate *my proposed solution 2* as a possible direction for future JWT-based secure implementations.

In the next part of this thesis work, I am to discuss about received results for both my implementations and give advice for further works.

5 Discussion of Results

In above sections of this thesis, I examined the security of the session-based and JWT-based authentication techniques and created secure NodeJS implementations of each. In the part 4.1, I identified possible attack vectors for both session-based and JWT-based authentication techniques. Additionally, I provided recommendations on safe implementations of each above-mentioned method. I specifically suggested two novel ways for improving the security of JWT-based authentication in accordance with my analysis, which I believe can be used in a variety of projects.

In the part 4.2 of this thesis, I developed a sample code on NodeJS, which implements both session-based and JWT-based authentication methods with most security considerations identified in the part 4.1. This code includes login and logout pages, authentication logic, and a verification page that is only accessible to authenticated users. Both solutions implement cookies as a client-storage mechanism, which I identified to be the most secure from analysed client-storage methods and secure against XSS, CSRF and MITM attack vectors. Both solutions provide instantaneous session/token revocation and are prone to brute force and preimage attack vectors. Session-based authentication is also safe from session fixation. My JWT-based solution retained the main advantages of JWT-based solutions, such as scalability and better performance, while overcoming most vulnerable aspect of stateless nature: token revocation. The two drawbacks of such implementation are increased architectural complexity and possible token prediction attack vector. Possible directions for future work could include further investigation and analysis of third-party based authentication technologies; development and implementation of an automatic service to clear expired rows in database for both session-based and JWT-based implementations, to lighten the database and improve performance. Additionally, there is a need of developing an algorithm for JWT-based solution to automatically sort users into different groups, rather than relying on group manual sorting. Lastly, one of my JWT-based theoretical solutions (mentioned as *my proposed solution 2*) require further investigation, before practical implementation. This solution requires testing and evaluation to ensure its effectiveness and practicality in real-world scenarios.

Overall, my results showed that while both session-based and JWT-based authentication methods could be implemented securely using NodeJS, there are key differences in terms

of their implementations. Specifically, I found that session-based authentication required less development and infrastructure needs to ensure robustness of the system but may require more server resources and are not scalable well. On the other hand, JWT-based authentication is potentially better at scope of performance and scalability due to being stateless. However, all investigated security workaround solutions and both my proposed solutions required to have calls into database and partially abandon the stateless nature of JWT.

The results of my study have significance for developers and security specialists responsible for creating and implementing safe authentication systems. By providing a detailed analysis of the security of session-based and JWT-based authentication methods, and by offering recommendations and base solutions, I hope to contribute to a more secure and reliable online authentication ecosystem.

6 Conclusion

In conclusion, this thesis provides a comprehensive security analysis of four client-side storage mechanisms and two in-house authentication methods. Through an exhaustive literature review, I have compared and evaluated various client-side storage options and authentication methods, and identified the most common security issues associated with JWT-based authentication. The results suggest that cookies are the preferred client-storage technology, as they are capable of mitigating all analysed attack vectors. I have compared several authentication strategies and concentrated on the security issues related to session-based and JWT-based authentication solutions.

I have investigated the security concerns discovered during the literature review and gave a thorough analysis of them. Summarizing the findings of analysis, in the first section of the fourth chapter of this thesis, I have suggested strategies to mitigate found security issues. With regard to session-based and JWT-based authentication solutions, I have specifically identified the most frequent attack vectors of each and offered recommendations for secure session-based implementation, as well as evaluated current security workarounds for JWT-based authentication and provided two novel workarounds. These insights are of great value to web developers and security experts seeking to build secure authentication systems.

In the second section of the fourth part, I have demonstrated the implementation of both session-based and JWT-based authentication methods through sample code in NodeJS. I have addressed the security issues mentioned in earlier parts and provided sample database queries using specific libraries to simplify the building of web application authentication process. The sample code consists of four main handlers: login and logout handlers, authentication verification handler and refresh handler. This is a good base for professionals to start developing secure web application authentication.

This thesis has succeeded the overall objective, which was to contribute to the development of secure in-house authentication systems. This was done by giving developers and security experts an exhaustive understanding of the security issues related to session-based and JWT-based authentication solutions. The research described in this thesis will help developers and security experts improve the security of their authentication systems. Additionally, I provided suggestions for potential future research

areas to further the subject, such as in-depth comparison of third-party based authentication methods and development of grouping algorithm for JWT-based authentication method.

7 References

[1] OWASP, “OWASP Top Ten,” 2021. [Online]. Available: https://owasp.org/Top10/. Accessed: Apr. 24, 2023.

[2] C. M. Gutierrez and W. Jeffrey, “FIPS PUB 200, Minimum Security Requirements for Federal,” National Institute of Standards and Technology, Gaithersburg, MD, March 2006.

[3] MDN contributors, “Client-side storage,” 21 March 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Client-side_storage. Accessed: Apr. 10, 2023.

[4] K. LaCroix, Y. L. Loo and Y. B. Choi, “Cookies and Sessions: A Study of What They Are, How They Work and How They Can Be Stolen,” in *2017 International Conference on Software Security and Assurance*, Virginia Beach, 2017.

[5] J. Hasan and A. M. Zeki, “Evaluation of web application session security,” in *2nd Smart Cities Symposium (SCS 2019)*, Kingdom of Bahrain, 2019.

[6] “What is HTTPS?,” [Online]. Available: https://www.cloudflare.com/learning/ssl/what-is-https/. Accessed: Apr. 18, 2023.

[7] A. Barth, C. Jackson and J. C. Mitchell, “Robust defenses for cross-site request forgery,” *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, p. 75–88, 2008, doi: 10.1145/1455770.1455782.

[8] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, T. Wan and N. Weaver, “Cookies lack integrity: real-world implications,” in *SEC'15: Proceedings of the 24th USENIX Conference on Security Symposium*, Washington, D.C., USA, 2015, doi: 10.5555/2831143.2831188.

[9] E. S. Bingler, E. M. West and E. J. Wilander, “Cookies: HTTP State Management Mechanism (draft),” 19 April 2023. [Online]. Available: https://httpwg.org/http-extensions/draft-ietf-httpbis-rfc6265bis.html. Accessed: Apr. 23, 2023.

[10] OWASP contributors, “Cross-Site Request Forgery Prevention Cheat Sheet,” OWASP, 3 January 2023. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html. Accessed: Mar. 23, 2023.

[11] Akanksha and A. Chaturvedi, “Comparison of Different Authentication Techniques and Steps to Implement Robust JWT Authentication,” in *7th International Conference on Communication and Electronics Systems (ICCES)*, 2022.

[12] K. S, “Cross Site Request Forgery (CSRF),” OWASP, [Online]. Available: https://owasp.org/www-community/attacks/csrf. Accessed: Apr. 11, 2023.

[13] K. S, “Cross Site Scripting (XSS),” OWASP, [Online]. Available: https://owasp.org/www-community/attacks/xss/. Accessed: Apr. 11, 2023.

[14] MDN contributors, “Window: sessionStorage property,” MDN, 10 February 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Window/sessionStorage. Accessed: Mar. 11, 2023.

[15] S. Kimak and J. Ellman, “The role of HTML5 IndexedDB, the past, present and future,” in *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, London, UK, 2015, doi: 10.1109/ICITST.2015.7412126.

[16] “Managing HTML5 Offline Storage,” 26 April 2018. [Online]. Available: https://developer.chrome.com/docs/apps/offline_storage/#types. Accessed: Mar. 14, 2023.

[17] sihui_liu@apple.com, “Changeset 237700 in webkit,” 1 November 2018. [Online]. Available: https://trac.webkit.org/changeset/237700/webkit/. Accessed: Mar. 14, 2023.

[18] MDN contributors, “Storage quotas and eviction criteria,” 8 April 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Browser_storage_limits_and_eviction_criteria. Accessed: Mar. 14, 2023.

[19] “Disc volume determines AppCache and IndexedDB limits,” 15 December 2016. [Online]. Available: https://learn.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/compatibility/mt732551(v=vs.85). Accessed: Mar. 14, 2023.

[20] V. Madurai, “Different ways to Authenticate a Web Application,” 5 February 2018. [Online]. Available: https://medium.com/@vivekmadurai/different-ways-to-authenticate-a-web-application-e8f3875c254a. Accessed: Apr. 14, 2023.

[21] A. Shaji, “Web Authentication Methods Compared,” 10 February 2023. [Online]. Available: https://testdriven.io/blog/web-authentication-methods/. Accessed: Feb. 20, 2023.

[22] “Web Authentication Methods Explained,” 17 February 2023. [Online]. Available: https://blog.risingstack.com/web-authentication-methods-explained/. Accessed: Feb. 20, 2023.

[23] “Session Management Cheat Sheet,” [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html. Accessed: Nov. 28, 2022.

[24] Auth0, “Introduction to JSON Web Tokens,” Okta, [Online]. Available: https://jwt.io/introduction. Accessed: Apr. 9, 2023.

[25] M. Calandra, “Why do we need the JSON Web Token (JWT) in the modern web?,” 6 September 2019. [Online]. Available: https://medium.com/swlh/why-do-we-need-the-json-web-token-jwt-in-the-modern-web-8490a7284482. Accessed: Apr. 10, 2023.

[26] D. Gałecki , “JWT authorization: How does it work for web applications?,” 16 November 2021. [Online]. Available: https://concisesoftware.com/blog/jwt-authorization-in-web-applications/. Accessed: Nov. 16, 2022.

[27] L. V. Jánoky, J. Levendovszky and P. Ekler, “An analysis on the revoking mechanisms for JSON Web Tokens,” in *International Journal of Distributed Sensor Networks*, 2018, doi: 10.1177/1550147718801535.

[28] P. Varalakshmi, G. B, V. S. P, D. T and S. K, “Improvising JSON Web Token Authentication in SDN,” in *2022 International Conference on Communication, Computing and Internet of Things (IC3IoT)*, Chennai, India, 2022, doi: 10.1109/IC3IOT53935.2022.9767873.

[29] S. Ahmed and Q. Mahmood, “An authentication based scheme for applications using JSON web token,” in *2019 22nd International Multitopic Conference (INMIC)*, Islamabad, Pakistan, 2019, doi: 10.1109/INMIC48123.2019.9022766.

[30] D. Fett, R. Küsters and G. Schmitz, “The Web SSO Standard OpenID Connect: In-depth Formal Security Analysis and Security Guidelines,” in *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, Santa Barbara, CA, USA, 2017, doi: 10.1109/CSF.2017.20.

[31] P. A. Grassi, J. L. Fenton, E. M. Newton, R. A. Perlner, A. R. Regenscheid, W. E. Burr and J. P. Richer, “Digital Identity Guidelines: Authentication and Lifecycle Management,” NIST Special Publication 800-63B, Gaithersburg, MD, June 2017, doi: 10.6028/NIST.SP.800-63b.

[32] A. Barth and U. Berkeley, “RFC 6265 - HTTP State Management Mechanism,” April 2011. [Online]. Available: https://tools.ietf.org/html/rfc6265. Accessed: Nov. 28, 2022.

[33] K. Drhová, “Authentication, authorization, and session,” 29 April 2018. [Online]. Available: https://dspace.cvut.cz/bitstream/handle/10467/76234/F8-DP-2018-Drhova-Klara-thesis.pdf. Accessed: Nov. 28, 2022.

[34] MDN contributors, “Using HTTP cookies,” 3 March 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies. Accessed: Mar. 3, 2023.

[35] S. Barbato, S. Dorigotti and T. Fossati, “SCS: KoanLogic's Secure Cookie Sessions for HTTP,” March 2013. [Online]. Available: https://tools.ietf.org/html/rfc6896. Accessed: Nov. 28, 2022.

[36] OWASP Foundation, “HttpOnly | OWASP Foundation,” [Online]. Available: https://owasp.org/www-community/HttpOnly. Accessed: Mar. 21, 2023.

[37] A. Deveria, “headers HTTP header: Set-Cookie: SameSite: SameSite=Strict,” [Online]. Available: https://caniuse.com/mdn-http_headers_set-cookie_samesite_strict. Accessed: Mar. 30, 2023.

[38] A. Deveria, “headers HTTP header: Set-Cookie: HttpOnly,” [Online]. Available: https://caniuse.com/mdn-http_headers_set-cookie_httponly. Accessed: Mar. 30, 2023.

[39] A. Deveria, “headers HTTP header: Set-Cookie: Max-Age,” [Online]. Available: https://caniuse.com/mdn-http_headers_set-cookie_max-age. Accessed: Mar. 30, 2023.

[40] MDN contributors, “HTTP authentication,” 3 March 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication. Accessed: Apr. 2, 2023.

[41] K. Kaur, “Explain HTTP authentication,” 27 March 2022. [Online]. Available: https://www.geeksforgeeks.org/explain-http-authentication/. Accessed: Mar. 14, 2023.

[42] M. Jones, J. Bradley and N. Sakimura, “RFC 7519 - JSON Web Token (JWT),” May 2015. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7519. Accessed: Apr. 15, 2023.

[43] “What is an OpenID?,” 6 December 2009. [Online]. Available: http://openid.net/get-an-openid/what-is-openid/. Accessed: Nov. 28, 2023.

[44] N. Sakimura, J. Bradley, M. B. Jones, B. d. Medeiros and C. Mortimore, “OpenID Connect Core 1.0 incorporating errata set 1,” The OpenID Foundation,

8 November 2014. [Online]. Available: https://openid.net/specs/openid-connect-core-1_0.html. Accessed: Apr. 3, 2023.

[45] E. Agbenyo, “NodeJs & Authentication with Cookies and Session (Part 2),” 28 August 2019. [Online]. Available: https://dev.to/edemagbenyo/nodejs-authentication-with-cookies-and-session-part-2-2752. Accessed: Apr. 3, 2023.

[46] Z. Aayush, “Session Cookies in Node.js,” 7 October 2021. [Online]. Available: https://www.geeksforgeeks.org/session-cookies-in-node-js/. Accessed: Apr. 3, 2023.

[47] F. Mendes, “Using Cookies with JWT in Node.js,” 27 May 2021. [Online]. Available: https://dev.to/franciscomendes10866/using-cookies-with-jwt-in-node-js-8fn. Accessed: Apr. 3, 2023.

[48] S. Kamani, “Session Cookie Authentication in Node.js (With Complete Examples),” 22 February 2022. [Online]. Available: https://www.sohamkamani.com/nodejs/session-cookie-authentication/. Accessed: Apr. 11, 2023.

[49] M. Shafran, “NodeJS Secure Authentication Implementations,” 15 April 2023. [Online]. Available: https://github.com/IrishIRL/nodejs-secure-authentication/. Accessed: Apr. 15, 2023.

[50] H. Kathuria, “The Most Popular Databases for 2022,” 11 January 2022. [Online]. Available: https://learnsql.com/blog/most-popular-databases-2022/. Accessed: Apr. 20, 2023.

[51] D. Kumar, “6 Best Databases To Use In 2023,” 9 January 2023. [Online]. Available: https://hevodata.com/learn/best-database/. Accessed: Apr. 19, 2023.

[52] “Password Hashing Competition,” 25 April 2019. [Online]. Available: https://www.password-hashing.net/. Accessed: Apr. 19, 2023.

[53] “Password Storage Cheat Sheet,” OWASP, [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html. Accessed: Apr. 19, 2023.

[54] J. Snell, “Introducing new crypto capabilities in Node.js,” 29 July 2021. [Online]. Available: https://www.nearform.com/blog/new-crypto-capabilities-in-node-js/. Accessed: Apr. 18, 2023.

[55] “Node.js uuid.v4 vs crypto.randomUUID. Which implementation is more cryptographically secure?,” 9 September 2022. [Online]. Available: https://crypto.stackexchange.com/questions/96019/node-js-uuid-v4-vs-crypto-randomuuid-which-implementation-is-more-cryptographic. Accessed: Apr. 18, 2023.

[56] “Web Cryptography API,” W3C Group, 1 November 2022. [Online]. Available: https://w3c.github.io/webcrypto/#Crypto-method-randomUUID. Accessed: Apr. 19, 2023.

[57] A. Deveria, “headers HTTP header: Set-Cookie: SameSite: Defaults to Lax,” [Online]. Available: https://caniuse.com/mdn-http_headers_set-cookie_samesite_lax_default. Accessed: Mar. 30, 2023.

[58] MDN contributors, “SameSite cookies - HTTP,” 3 March 2023. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite. Accessed: Apr. 2, 2023.

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Mark Shafran

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “An Analysis of Session- and JWT-Based Authentication Methods: A Comparative Study with Secure Implementation Examples”, supervised by René Pihlak
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

15.05.2023

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.