# Security analysis of JSON web tokens- Attack scenarios and countermeasures

Project Report

Name: Hafizur Rahman Anik
Student ID: 20210769

Aalborg University
Electronics and IT


**Electronics and IT**
Aalborg University
http://www.aau.dk

**Title:**
Security analysis of JSON web tokens-Attack scenarios and countermeasures

**Theme:**
Masters Thesis

**Project Period:**
Fall Semester 2024

**Project Group:**

**Participant(s):**
Hafizur Rahman Anik

**Supervisor(s):**
Henning Olesen

**Page Numbers:** 85

**Date of Completion:**
January 4, 2024

**Abstract:**

JSON Web Token (JWT) is a very popular open standard for transmitting information securely between parties as a JSON object. They are being used in common data flows in many modern technologies such as OAuth 2.0, DPoP-JWTs, OpenID Connect, Selective Disclosure (SD-JWTs), and as alternatives to session tokens. Regardless of all their uses and high popularity, JWT has its own flaws as well. This project will focus on a deeper systematic analysis of the uses of JWTs in different data flows to get an overall assessment of how secure they are. The research will proceed further with finding out commonalities and differences among them, Security objectives provided by the JWT with the current attack scenarios on those data flows that disrupts those objectives to be fulfilled with their countermeasures and some recommendations for best practices as well.

# Contents

# Preface

Aalborg University, January 4, 2024

Hafizur Rahman Anik
hanik21@student.aau.dk

# Chapter 1

# Introduction

In digital realm it is very important to ensure the security of web applications through proper user authentication and authorization. Where the authentication refers to verify a user's identity and authorization is the procedure to verify that the user has the necessary permissions to access an application resources or perform specific actions. In a web application at first the authentication takes place and then authorization is performed.

## 1.1 JWT

Now to perform both authentication and authorization one of the popular method is using JSON Web Token (JWT). JSON Web Token, or JWT ("jot") is a very popular compact, standard and URL-safe means for representing and safely transferring claims between two parties as a JSON object [59]. Now couple of things needs to be elaborated to achieve more precise understanding about JWT. Such as JSON or JavaScript Object Notation which is both human and machine readable lightweight and text-based data interchange format. Despite the name JavaScript Object Notation and its derivation from the ECMAScript programming language [26], it is actually language-independent which is used for the serialization of structured data and exchanging it between a server and web applications over a network [91] [31]. JSON consists of four primitive data types which are strings, numbers, booleans, null and two structured data types which are objects and arrays. Some of the uses of JSON are– storing user-generated data such as submitting form on a website, transferring data between systems such as sending email address via JSON format to validation service API to check the validity or verify the address.[37]. In addition, the claim is referred to the information about an user or a subject and some additional data. The representation of a claim has a claim Name and a claim Value. JWT Claims Set refers to a JSON object that contains the claims conveyed by the JWT [91] [106]. More details about JWT and claims can be found in chapter 3.

JWTs as a JSON object can be used in authentication, authorization and for information exchange as well. The claims are represented in a compact way for space constrained environments such as HTTP Authorization headers and URI query parameters. Some of the reasons for their popularity are the relatively small size and compact transmission format, which avoids querying a database more than once to verify an entity [77]. Also JWT plays an important role to ensure the authenticity of the sender and to ensure that the message has not been tempered on its way via digital signature and encryption. The basic structure of JWT are as below:

JSON Web Tokens consist of header, payload and signature. The header typically consists of two parts– name of the signing algorithm for example HMAC SHA256 or RSA and type of the token which is jwt. payload contains the claims. The third part which is the signature is the combination of Base64 encoded header, payload and a secrete [32]. figure 1.1. Now the Base64 encoding [36] is the representation of binary data in an American Standard Code for Information Interchange (ASCII) string format which allows to transport binary over protocols that can not accept binary data and required printable text [13].

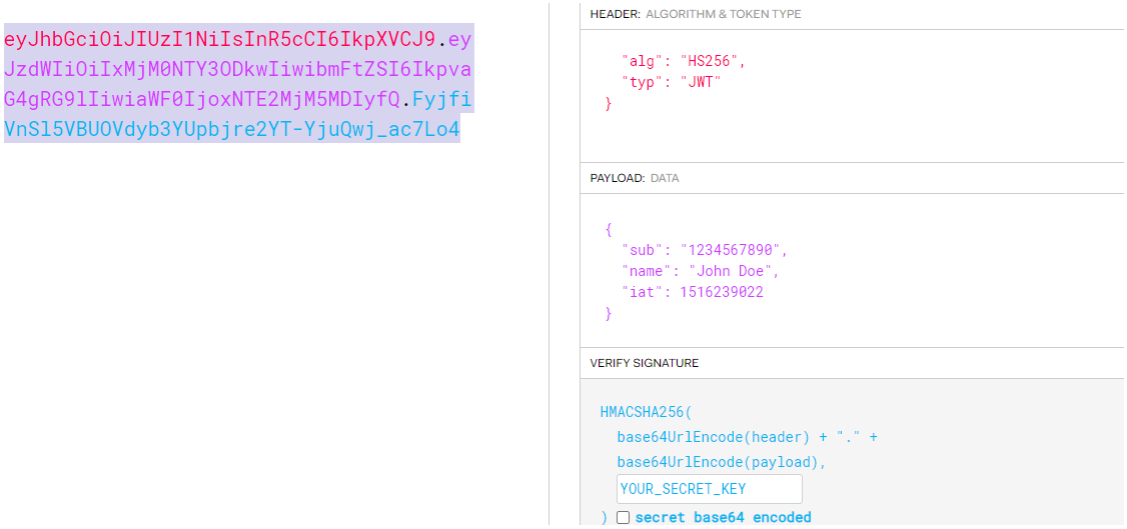


**Figure 1.1:** Different parts of JWT [40]

Due to its simple deployment as a security token format it is being widely used in different protocols and applications such as OAuth 2.0, OpenID Connect, Single Sign-on (SSO), verifiable credentials and Selective Disclosure (SD-JWTs) and as alternatives to session tokens.

Regardless of all uses and high popularity of JWT, it has its own vulnerability or weaknesses [107]. Since JWT has been used for safely transferring claims between two parties, it needs to fulfill some security objectives during their imple-

mentations to ensure that nobody can intercept and modify it at the mid point of the transmission.

## 1.2 Security objectives:

Before co-relating those security objectives with the use of JWT, it is essential to know some basic information about them.

1. **Confidentiality:** It refers to the assurance for keeping the sensitive information private via providing authorized access on personal information [97]. Encryption can be used to keep the data confidential. If sensitive information is used in JWT claims then it can provide the confidentiality of those claims by using JSON web encryption (JWE) standard. More about JWE can be found in 2

2. **integrity:** It ensures the protection of data from tampering or manipulation during transmission over a network. Signed JWT can provide the integrity of the claims or message inside it to ensure that it has not been tampered on its way. More about JSON web signature (JWS) can be found in 1.

3. **non-repudiation:** It refers to ensure that the transmission occurred via agreeing by both senders and the receiver so that no parties can deny having processed the data later on. In this case sender needs to send data with the proof of delivery and proof of the sender's identity needs to be provided to the receiver as well so that the receiver can not deny it [64] [8]. It can be done by using of digital certificates and public key cryptography to sign transactions, messages, and documents [80]. More about digital certificate and signature can be found in section B.3. Point to be noted that in terms of JWT which is signed with a symmetric signature or shared secrete can not ensure non-repudiation since if the key is compromised then anyone can create a new JWT with that secrete and can impersonate himself as an authentic sender.

4. **Authenticity:** Authenticity ensures that the message is coming from an authentic source and it is not being altered along the way via assuring or verifying that the person at the other end is really what it claims to be. It can be achieved by digital signature [47]. JWT can ensure the authenticity of the sender if it is signed with using public/private key scheme 4.

## 1.3 Some common attacks

Some common attacks that disrupts these security objectives are as below:

- **Man-in-the-middle attack (MITM):** Man-in-the-middle attack (MITM) occurs when the attacker or an unauthorized intruder intercepts a communication or often even altering the exchanged information between two parties. The captured information can be a session cookie, also changing an amount of money transaction, private discussions and other sensitive information can also be possible [46]. Avoiding public wifi and using a secure VPN to ensure all information is encrypted and cannot be viewed and implementing multi-factor authentication for business purposes can reduce the risk of MITM attack [45]. Now if JWT is not encrypted or not uses TLS during transmission, it might be vulnerable by MITM attack.

- **Session hijacking:** Session hijacking is an impersonation attack where the attacker successfully manage to exploit web session control mechanism by stealing session token to achieve unauthorized access to the Web Server [83]. Some common ways to compromise or steal the session token are–

  1. session sniffing where the attacker uses the sniffing tools to capture the 'session ID' which is a valid token session.
  2. if an attacker sends a malicious link to the victim which includes malicious JavaScript and if the victim clicks the link it can sends the cookie value of the current session to the attacker known as Cross-site script attack (XSS) [83].
  3. MITM attack.

  Prevention can be done by ensuring that all server communication is completely encrypted or protecting the communication with VPN during working in public spaces cause public wifi can be very dangerous [84].

- **Cross Site Request Forgery (CSRF):** It happens when the attacker causes the victim user who is already authenticated in a web application to perform an action unintentionally by tricking victim's browser into sending a request from different browser. Performing an undesired function on the victim's behalf can cause changing the email address, making a funds transfer. But if the victim is an administrator's account then the attacker can take the full control over the entire web application. Normally any browser requests automatically include the user's session cookie, IP address etc. But if the user already authenticated to the site then the server can not distinguish in between forged or legitimate request [42]. Suppose, if the application depends only on session cookies to identify the user then the attacker can send an HTTP request to the vulnerable web site to change the user's email address and if the victim were in logged in state in the same website. The browser will automatically include the session cookie in the request and will

take it as a normal legitimate request. Some prevention machanism are using CSRF tokens and SameSite cookies. [14]. Point to be noted that JWT can prevent both XSS and CSRF by implementing it with HttpOnly cookie [75] [63]. Now in brief, HTTPOnly is just an additional flag that needs to be added in a Set-Cookie HTTP response header which helps to reduce or mitigate the risk by preventing the attacker to get or access the protected cookie by XSS or CSRF attack [29] [51]. Short-lived JWTs can also be helpful. JWT that are not stored as cookies, can also prevent CSRF [73]. Some other prevention procedure can be found in OWASP Cheat Sheet Series [15].

- **Server-side request forgery (SSRF):** SSRF can take place when the server somehow failed to validate the client supplied URL that allows the attacker to force the the server-side application to send a modified request to an unintended location. It can happen even if the protection is provided by a firewall, VPN, or another type of network access control list (ACL) [81] [82].

  Some of the SSRF attacks will be described in section 4.7 with their countermeasures.

  Some more attacks can be found on OWASP Top Ten [71].

Now it is essential to know that security for JWT depends on how tokens are being used and validated. Although JWTs contain a cryptographic signature, it does not make them completely secure, and a number of attack scenarios have been identified. Unless following good practices, JWTs can be vulnerable to cyber attacks. Since JWT can be used in authentication, session management, authorization and information exchange there are some attacks on JWT which can cause a disruption to fulfill the above mentioned security objectives. Such as token hijacking can be possible where the attacker can get the access of the token and send the modified token to the server to bypass authentication and access controls by impersonating another user who has already been authenticated which is violating both integrity and confidentiality of data[41]. If the JWT contain sensitive information without encryption, it can cause data breach of this confidential information to unintended parties. [107]. JWT can also be vulnerable by CSRF and XSS attack if it is stored inside cookies [73].

It is therefore essential to analyse why, how and for which purposes JWT are being used on those modern technologies and the possible attacks and countermeasures that have been studied so far to find out exactly where those security implications are needed. In chapter 4 we will explain in details how and for which purposes JWT are being used nowadays on those technologies which leads us in chapter 5 to find some commonalities in between them in terms of uses of JWT and have some practical demonstrations of attacks on JWT and analyse how uses of JWT can be vulnerable for each of those technologies with particular types of attacks including some recommendations for best practices and better security.

## 1.4 Problem Formulation

This project will focus on a deeper systematic analysis of the use of JWTs in different scenarios to get an overall assessment of how secure they are. Based on primarily literature studies, the research will proceed with finding out commonalities and differences among them, the current attack scenarios and their countermeasures.

So the main goal of this project is to find–

***what are the main threats or vulnerability of different uses of JWT in different data flows in between client-server communications and IAM, and what are the best practices of using JWT for these purposes?***

We can consider this problem statement as our main goal which can be subdivided further in order to narrow down the project scope as below:

- What are the commonalities and differences in terms of using JWT in different data flows?
- What are the common types of use of JWTs and which security objectives need to be fulfilled?
- What are the most important types of attacks on JWTs can take place when it has been for these purposes and how can they be prevented or minimized?
- What are the recommendations for best practices when using JWTs in order to fulfill the security objectives?

So, in brief the expected outcome of this thesis paper would be comparing the different uses of JWT and recommend some best practices of using JWT on them.

# Chapter 2

# Methodology

This chapter has been dedicated to explain the scientific method used throughout the project to conduct the research. Among many research methodology it is important to choose the right one so that it can lead us in the right direction throughout our project work and can help us to fulfill our proposed goals. Some of the important research methodology are as below–

1. **Descriptive research:** This kind of research related with survey and fact-finding investigation where the researcher has no direct control over it as it is just an explanation of the set of circumstances which is happening or has happened before.

2. **Analytical research:** In this scenario researchers can research facts, information, data which is already available and analyse them at the end to make some hypothesis.

3. **Applied research:** It refers to proceed the research in such a way which can lead the researcher to find the solution for any problem facing by the individual or organizations. For example- how to wiped out hate crime from a society.

4. **Fundamental research:** It is based on Formulating a theory and generalize them such as generalizing human behavior.

5. **Quantitative research:** It uses statistical, mathematical or computational techniques for systematic experimental analysis on something that can be counted.

6. **Qualitative research:** It involves upon in depth analysis on non-numerical data.

7. **Survey research:** it is about to collect a huge amount of real-time data by questionnaires and interviews for research purposes. [2] [72]

If we observe our project topic including the problem formulation and project scope then we can say that analysis research types of methodology can be the best option for systematic analysis on security perspective of JWT. The reason to choose this research methodology for this report is we needed to gain in depth knowledge in regards JWT, its uses in different data flows including their security perspective which could not be possible without collecting authentic research papers and analyse them carefully to find the security vulnerability for each of them and analyse their countermeasures.

To fulfill our goal we needed to choose couple of platforms like Google scholar and IEEE to find some project related research paper. Also Request for Comments (RFC) documents were played a vital role in this project to gather some really valuable and authentic scientific information about JWT and its different uses. It is produced by the Internet Engineering Task Force (IETF) which deals with different aspects of computer networking [78]. Moreover, As our project is based on security aspects of JWT the Open Worldwide Application Security Project (OWASP) helps us to know about some common vulnerabilities related with web application in a more understandable way. It is a nonprofit organization which works to enhance the security of software [1].

In addition, the keyword has been used throughout the project to find the related research papers are JWT, JWT rfc, JWT handbook, JWT vs session token, Attacks on JWT, Security on JWT rfc, DPOP rfc, Oauth rfc, Oauth 2.1 rfc, mTLS rfc, SD-JWT rfc, Verifiable credential, Single sign on(SSO) rfc, OpenID Connect (OIDC), OIDC handbook. Most of the cases through those above mentioned keywords it can lead us directly to some authentic and trustworthy web portal including the rfc documents about related topics.

# Chapter 3

# JWT in details

We already know what is JWT. Now it is time to dive deep to know more about the basic structure of JWT and how it works and can be used. As mentioned in chapter 1 that JWTs has three different parts header, payload, and the signature/encryption data. The header and the payload contains JSON objects of a certain structure. The signature part depends on the algorithm used for signing or encryption. JWTs can be encoded in a compact representation where three Base64-URL encoded [36] strings separated by dots which can be easily passed in HTML and HTTP environments figure 1.1 [73]. The signature can be produced with a secrete which is known to both the sender and receiver or with a private key which only known by the sender [33] [59]. JWTs can be either signed or encrypted or can be both.

## 3.1 The Header

Every JWT contains a header known as JOSE header as well. The encrypted JWT header contains type of the token and name of the signing algorithm. If we refer to figure 3.1 then the example represents that the object is a JWT, and the JWT is a JWS which used HMAC SHA-256 algorithm [59] [88]. But the unencrypted header only contains alg claim where the value of the claim must be 'none' [73].


**Figure 3.1:** Different parts of JWT header [32]

### 3.1.1 The Payload

The payload is a JSON object which contains the claims. Claims can be user data and also can be some other additional data that are defined in JWT spec. No claims are mandatory but if any claims are not understood by the implementation then it should be ignored [73].

There are three types of claims: registered, public, and private claims.

#### Registered claims

There are seven reserved claims mentioned in the JWT specification figure 3.2 which is registered with the Internet Assigned Numbers Authority (IANA) [39] to allow interoperability with third-party applications. These claims are not are not mandatory but recommended [38].

- `iss` (issuer): Issuer of the JWT
- `sub` (subject): Subject of the JWT (the user)
- `aud` (audience): Recipient for which the JWT is intended
- `exp` (expiration time): Time after which the JWT expires
- `nbf` (not before time): Time before which the JWT must not be accepted for processing
- `iat` (issued at time): Time at which the JWT was issued; can be used to determine age of the JWT
- `jti` (JWT ID): Unique identifier; can be used to prevent the JWT from being replayed (allows a token to be used only once)

**Figure 3.2:** Registered claims [38]

#### Public claims

These claims can be defined at will by those using JWTs. To prevent collision it either needs to be registered in the IANA JSON Web Token Registry or be a public name which contains a collision resistant name through namespacing and ensure that you are in control of the namespace you use [59]. In a more general way we can refer it as a custom claims which might contain name and email. For example we can check IANA JSON web token claims Registry [39] to search for public claims registered by OpenID Connect (OIDC): auth_time, acr and nonce [38].

**Private claims**

We can also call these types of claims as a custom claims but the main difference is it is not collision resistant like public claim, so it should be used with caution. Both producer and consumer of a JWT may agreed upon using these claims to share information between them [59].

Point to be noted, Although a signed token is protected against tampering, the data inside payload and header can still be readable by anyone. So it is recommended not to put any secrete information in the header or payload unless it is encrypted or it is only transmitted using TLS which ensures endpoint authentication [32] [59].

### 3.1.2 Signature

Signature can be created by combining the encoded header, the encoded payload, a secret, the algorithm specified in the header and signing them together figure 3.3. The signature ensures the authenticity of the message that it wasn't changed along the way and signing with a private key ensures the authenticity of the user to prove that the sender of the JWT is who it says it is [32].

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

**Figure 3.3:** Example of the signature using the HMAC SHA256 algorithm. [32]

## 3.2 The JSON Object Signing and Encryption group (JOSE)

The JSON Object Signing and Encryption group (JOSE) [17] provides the standardize mechanism for integrity protection via signature and MAC or encryption. The JSON Web Token (JWT), JSON Web Signature (JWS), JSON Web Encryption (JWE), JSON Web Key (JWK) and JSON Web Algorithm (JWA) all are the parts of JOSE [73].

1. **JSON Web Signature (JWS):** JWS represents or defines that the data contained in the JWT is secured with digital signatures or Message Authentication Codes (MACs) using JSON-based data structures [58] [60]. It is important to know that signature does not prevent reading the content by other parties which can be solved by the encryption. The procedure to check the signature of a JWT is known as validation or validating a token. A JWT can be

considered valid even if there is no signature such as if the header has the alg claim set as 'none'. Thus using the none algorithm should be avoided. Moreover, even if the JWT has the valid signature, it may be considered invalid for other reasons such as according to the 'exp claim' if it becomes expired. Stripping the signature is a common types of attack against signed JWT. According to the JWS spec there are several types of signing algorithm available and a single type of algorithm needs to be supported by all conforming implementations [73]. The main three recommended algorithms are–

- HMAC using SHA-256, called HS256.
- RSASSA PKCS1 v1.5 using SHA-256, called RS256.
- ECDSA using P-256 and SHA-256, called ES256.

There are two types of scheme used for JWS: HMAC based shared secret which stands for HMAC SHA) and Public key pair scheme (either RSA or ECDSA keys). Caution needs to be taken when using a shared secret, as anyone with the secret can create or forge new JWTs. In fact, it is recommended to use public key pair instead to validate a JWT from an untrusted client (web-page, mobile app, etc.) [19].

2. **JSON Web Encryption (JWE):** JWE represents encrypted JWT using JSON-based data structures [61] [56]. While JSON Web Signature (JWS) is used for JWT validation, JSON Web Encryption (JWE) is used to create unreadable encrypted tokens. Like JWS, JWE also provide two schemes– JWE essentially provides two schemes: a shared secret scheme, and a public/private-key scheme.

   For a shared secret scheme, all parties needs to know the shared secrete that means whoever have this shared secrete can both encrypt and decrypt the information which is analogous with JWS shared secrete scheme where whoever obtain the shared secerete can both verify and generate the signed tokens.

   On the other hand, it works differently for public/private-key scheme. While in JWS the party holding the private key can sign and verify the token and public key holder can only verify them, in JWE the holder of the private key can both encrypt and decrypt the token but holder of the public key can only encrypt the token. So the main difference is in terms of JWE the public key owner can generate or introduce new data but for JWS the public key holder can only verify data but can not generate or introduce any data. In a more simpler way for JWS the data flow can only from private-key holders to public-key holders but for JWE the data flow is totally opposite figure 3.4. Since JWT can be encrypted using receiver's public key, it can only ensure

the confidentiality of the data. So from security perspective JWE does not provide the same guarantee as JWS, therefore can not replace the role of JWS in token exchange, rather they are complementary when public/private key schemes are being used [73].

| | JWS | JWE |
|---|---|---|
| Producer | Private-key | Public-key |
| Consumer | Public-key | Private-key |

**Figure 3.4:** Comparison between JWS and JWE. [73]

Point to be noted, if both signing and encryption are required for nested JWT then it is recommended to sign the message first and then encrypt the result although it can be done in any order. But by following the recommended way it can provide the privacy for the signer and prevent the signature stripping attack as well. Also following the opposite procedure such as signatures over encrypted text sometimes considered as invalid by many jurisdictions [59].

3. **JSON Web Keys (JWK):** JSON Web Key (JWK) is a JavaScript Object Notation (JSON) data structure which represents a set of public keys [53] [54]. This specification mainly deals with the unified representations for all keys used for signatures and encryption which is supported in the JWA spec. This unified representation allows easy sharing and keeps the keys independent from the complications of other key exchange formats. Keys are specified in different header claims where JWKs can be found under the jwk claim figure 3.5. On the other hand, the jku claim represents a set of keys stored under a URL. Both of these claims follows the JWK format [73].


```
{
    "kty":"EC",
    "crv":"P-256",
    "x":"MKBCTNIcKUSDii11ySs3526iDZ8AiTo7Tu6KPAqv7D4",
    "y":"4Etl6SRW2YiLUrN5vfvVHuhp7x8PxltmWWlbbM4IFyM",
    "d":"870MB6gfuTJ4HtUnUvYMyJpr5eUZNP4Bk43bVdj3eAE",
    "use":"enc",
    "kid":"1"
}
```


**Figure 3.5:** Example of JWK claims. [73]

The figure 3.5 represents the key is an Elliptic Curve [25] key which is used

with the P-256 Elliptic Curve and its x and y coordinates are the base64url-encoded values. kid is the key identifier [53]. Point to be noted, we should not use the same key for both encryption and signing, even though it provides compatible algorithms for both

4. **JSON Web Algorithms (JWA):** This specification registers cryptographic algorithms and identifiers to be used with the JSON Web Signature (JWS), JSON Web Encryption (JWE), and JSON Web Key (JWK) specifications [52].

| "alg" Param Value | Digital Signature or MAC Algorithm | Implementation Requirements |
|---|---|---|
| HS256 | HMAC using SHA-256 | Required |
| HS384 | HMAC using SHA-384 | Optional |
| HS512 | HMAC using SHA-512 | Optional |
| RS256 | RSASSA-PKCS1-v1_5 using SHA-256 | Recommended |
| RS384 | RSASSA-PKCS1-v1_5 using SHA-384 | Optional |
| RS512 | RSASSA-PKCS1-v1_5 using SHA-512 | Optional |
| ES256 | ECDSA using P-256 and SHA-256 | Recommended+ |
| ES384 | ECDSA using P-384 and SHA-384 | Optional |
| ES512 | ECDSA using P-521 and SHA-512 | Optional |
| PS256 | RSASSA-PSS using SHA-256 and MGF1 with SHA-256 | Optional |
| PS384 | RSASSA-PSS using SHA-384 and MGF1 with SHA-384 | Optional |
| PS512 | RSASSA-PSS using SHA-512 and MGF1 with SHA-512 | Optional |
| none | No digital signature or MAC performed | Optional |

**Figure 3.6:** Algorithm used for Digital Signatures and MACs. [52]

The figure 3.6 shows the algorithm used for Digital Signatures and MACs defined in the JWA specification. Now if we look the implementation requirements in figure 3.6 then we can notice HS256 is required and RS256, ES256 are recommended for implementations. There are couple of things we can discuss regarding these suggestions. At first we can take a look and compare in between HS256 and RS256. Both of them can be used to check the integrity or authenticity of JWT but RS256 ensures non-repudiation as well which provides the surety that the sender created the signature. It also more secure and allows to rotate the keys without re-deploying the application with a new one if the keys are being compromised whereas in terms of HS256 the re-deployment of an application is needed. But there are some scenarios as well where the HS256 suits the most such as if the application don't support RS256 and if the application generates large number of requests because HS256 is more efficient than RS256 [34]. On the other hand, ES256 prevents side channel attack where the attacker can measure the length of time of a signing operation [98].

Since we are talking about JWA we can look at another algorithm based attack called 'none' algorithm attack which is practically demonstrated in appendix A.1.2. The reason to mention it here particularly because this attack can be possible easily since many JWT libraries still support this none algorithm in their specification. To avoid none algorithm attack it is recommended to use php-jwt library [65] where the package does not support none algorithm or empty signature. Another way is using TLS because it provides end-to-end cryptographic protection. Using none algorithm can be acceptable in this case since JWT is cryptographically protected via TLS. [107]

# Chapter 4

# Different Uses Of JWT

We already explained JWT and its basic structure in previous chapter, now in this chapter we will describe about different data flows where the JWT has its involvement in the authentication, authorization procedure and information exchange.

## 4.1 JWT as an alternative to session tokens

### 4.1.1 The traditional session authentication

Before the origin of JWT we had the traditional session token for user authentication which works by generating a random string. Unlike JWT, after authentication with a username and password server generates a session token and stores that token with user's info in database. Upon each subsequent request by the user the token has been included in the HTTP auth header, and then validated by checking the database on each request.[77]

According to figure 4.1 The explanation of the data flow are as below:

1. The user sends username and password to the server to authenticate himself.
   (a) After authenticating user server generates a session token.
   (b) And stores the token with user's info in the database.
2. Server then returns a cookie containing session id to the user.
3. As long as the user remain logged in, the cookie will be sent to the server in every subsequent request . The server then compare the session id stored on the cookie against the session information stored in the database to verify user's identity and deny or approve the request. [77]

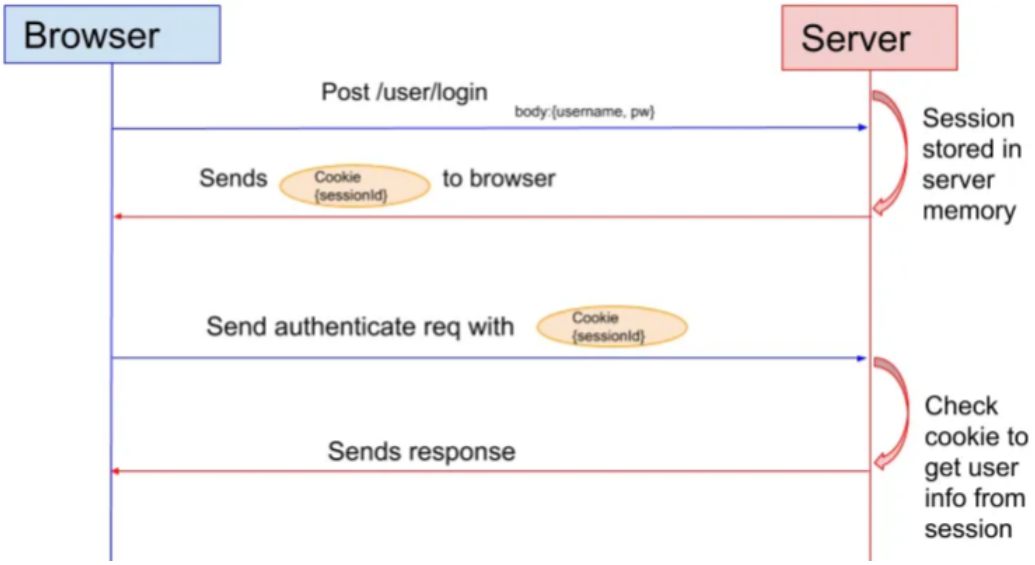


**Figure 4.1:** Workflow of session authentication [10]

### 4.1.2 Limitations

The following limitations have been observed in [79] [77] [85] [4]

1. In each user request it needs to hit the database every time to validate the user which is fine for small or medium scale apps but it is not suitable to handle large volume of requests in a very short time. it actually slows down overall response time.

2. Since,cookies typically works on a single domain or subdomains it can be problematic if APIs requests are sent to different domains.

3. session is statefull, that means it holds every information of every client accessed to server, which overloads the server and its management can be a big challenge.

4. Cookies are relatively more vulnerable to Cross-Site Request Forgery (XSRF or CSRF) attacks where the user can be redirected to a hostile website and the attacker can exploit cookies with some JS and sends malicious requests to the server.

### 4.1.3 JWT as an alternative

In fact, one of the alternatives to resolve the above mentioned drawbacks of session authentication is JWT. It can be used to completely eradicate extra database lookup

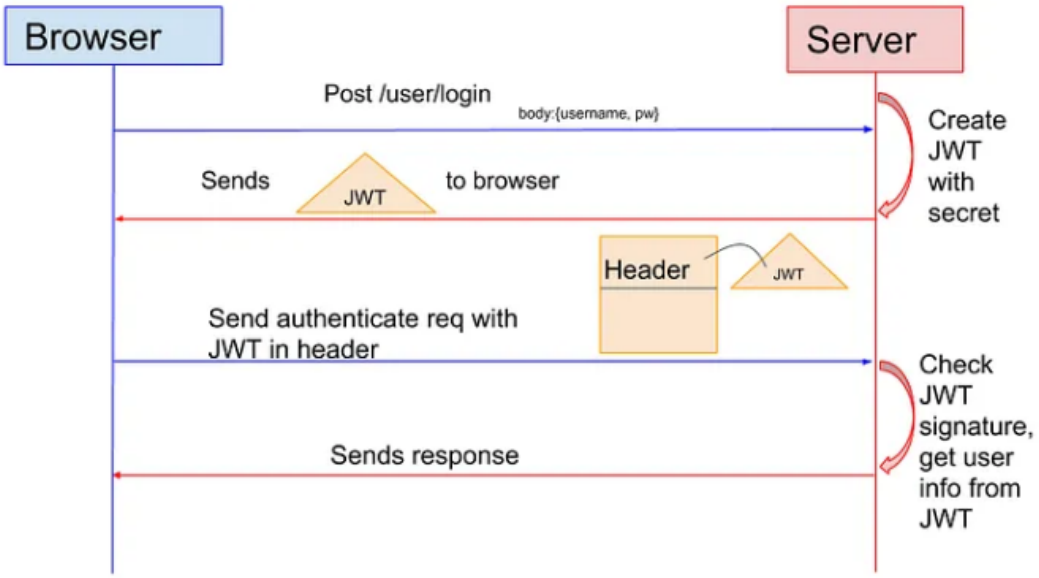


**Figure 4.2:** Workflow of JWT authentication [28]

when used as a session because it includes user information inside the session token which is signed with a secrete that only knows by the server. Even if multiple requests are sent to multiple services the same token can be used for authentication. The token-based technique is completely stateless , it doesn't hold any information about the user at all, because the token is stored in the client side and sends it upon each request directly from client to server. CSRF attacks can also be handled by JWT. But even if JWT stored inside cookies it can be vulnerable with CSRF attack.It can be solved by implementing JWT on HttpOnly cookie. Because HTTP-only cookies cannot be accessed by client-side scripts, they are not accessible via JavaScript or other client-side programming languages.they are only sent to the server with each HTTP request. [77] [75] [63] [85] [4]

According to figure 4.2 The explanation of the JWT authentication data flow are as below:

1. The user sends username and password to the server to authenticate himself.
   (a) The server authenticates the user via querying the database and after authentication it generates the token using user's info and secrete and sends the signed token to the user.
   (b) The signed token can be then stored on the client-side either in the Session storage, Local storage or Cookie Storage.

2. upon each user request it sends the token alongside the request
3. After receiving JWT token, the server validate the signed part with secret and retrieve the user information from the payload. In this way it sort out the extra DB call issue. [77] [75]

### 4.1.4 Limitations

The following limitations have been observed in [77] [75] [41] [49]

1. It can not be revoked or ivalidate or update as it has a particular expiration time to expire itself. The complication here is if someone get the access of the token they can easily continue their malicious activity until it expires. Even if someone logged out the attacker can still have the possession of the token and continue to access until its expiration.
2. If server admin wants to stop the malicious activity by blocking the user he can not do that for the same reason. Changing users details also not possible since it is stored on the users machine.
3. If a lot of data is encoded within the JWT, it can create a significant amount of overhead with every HTTP request.
4. man-in-the-middle attack is possible, since JWT are often not encrypted. token stealing, header parameter injections, JWT authentication bypass, brute-forcing secret keys are also possible.
5. JWT library has some serious vulnerability issues if not implemented properly. Such as attacks on algorithm, breaches of the weak secrete key.

## 4.2 Observations:

Since we already explained some pros and cons for using both of the traditional session token and JWT as an alternative to session tokens, now it is time to have some short conclusion or observations regarding using them.

Although both of them have some vulnerabilities in their perspective area of uses, if we consider the performance then we can say JWT can enable faster authorization and interchangeability with external apps compared with traditional session token. Also some of the attacks on JWT can be manageable by proper implementations and with some modern way of using JWT such as using DPoP-JWTs with Oauth or mTLS explained in section 4.4 and 4.6.

We can think about combining them together as well where a long lived session token can be used to retrieve a new JWT which has a short life span. To do that the servers needs to ensure that the underlying session is still active before passing

back a new JWT. The revocation of access will take place within whatever token age is set for the JWT, If the user logs out.

## 4.3 OAuth 2.0 Authorization Framework

OAuth 2.0 framework, or in another way "Open Authorization" enables a website or third party application to obtain limited access or resources from other web apps on behalf of a user. The steps of basic oauth 2.0 flow are as below:

(A.) Client sends authorization request to the authorization server via using resource owner's user agent with client identifier and redirection URI. figure 4.3

(B.) The authorization server authenticates the user or resource owner and ensure whether the resource owner accepts or denies the access request done by the client. After that AS sends authorization grant to the client using the same redirection URI provided in step A as a response. Here, authorization grant is a credential which represents resource owners authorization to access its protected resources. It is also used to obtain an access token by the client. resource owner's authorization has been expressed by one of four types of grant— authorization code, client credentials, implicit and resource owner's password credential. Which grant type will be chosen is depends on the method used by the client and on the types supported by AS.

(C.) Client requests an access token to the authorization server with presenting the authorization grant.

(D.) Authorization server issues an access token after authenticate the client and checking the validity of the authorization grant.

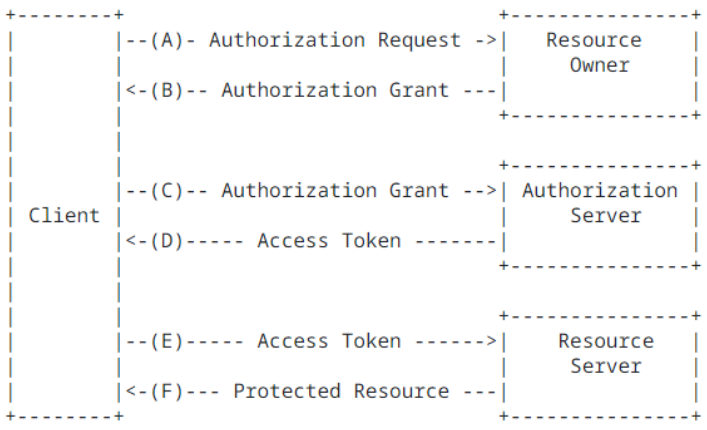


**Figure 4.3:** Oauth protocol flow. [22]

(E.) Client sends the access token to the resource server to achieve protected resources.

(F.) If the provided access token is valid then in return resource server allows to access the protected resources [22]. figure 4.3.

Point to be noted that we can see the client can achieve authorization grant from the resource owner in the step no A and B in figure 4.3 but the preferred method is to use the authorization server as an intermediary in terms of obtaining authorization grant shown in figure 4.4.

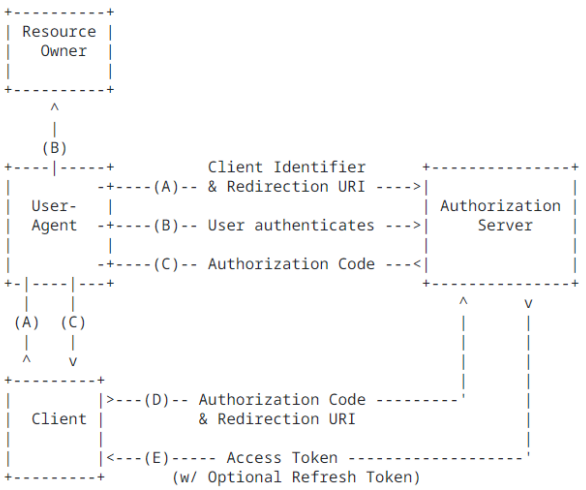


**Figure 4.4:** Oauth Authorization Code Flow. [22]

### 4.3.1 Relevant security objectives

Before explaining the relevant security objectives, we need to recall back the data flow to recognize exactly on which steps JWT has its involvement. First of all, oauth 2 specification did not make any token format mandatory, which means it can be any as usual token or JWTs can be used as OAuth 2.0 access Token [100].

1. In step C from figure 4.3 oauth can use a new types of authorization grant which can uses a JWT Bearer Token [55] to request an access token [5]. Point to be noted, in OAuth 2.0 specification the JWT Bearer token as authorization grant type and authorization code grants are two different things where the JWT bearer token authorization grant types is not a common types of grant and it can be used in some special cases but authorization code grants can be used in any apps or integrations figure 4.5 [99].

2. In step D from the figure 4.3, AS can issued an access token in the form of JWT which is consumed by the client or relying party. Proper client authentication plays a vital role here for issuing and supplying the token by AS otherwise client impersonation attack can take place. Also token leakage and

token substitution attack can also take place in this step. Both of them is described in section 4.3.2.

3. And in step E from the figure 4.3 when the same client wants to access any resources on behalf of a user, it again needs to use that very same JWT to get the access of the resources. If it uses JWT then the resource server can parse and validate the token directly without any involvement of AS. In addition, Proper validation of the issuer is needed here to avoid Cross-JWT confusion explained in section 4.3.2.

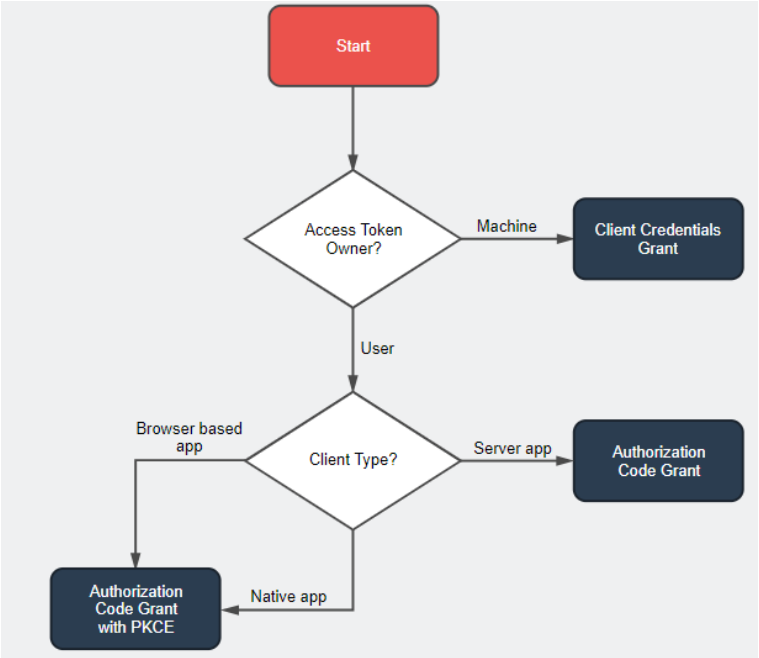


**Figure 4.5:** Oauth 2 grant. [103]

### 4.3.2 Important attacks:

Some important attacks on Oauth that can cause to disturb these security objectives are as below–

1. **Client impersonation:** The AS should not allow the client to register an arbitrary client id value because In step no D in figure 4.3, if the AS uses this arbitrary client id as 'sub' value for token issuance then any malicious client can select the 'sub' of a high-privilege resource owner to confuse resource server which is dependent on 'sub' value during token introspection if resource server does not carry out any additional check. It can cause both the

leakage of confidential resources of user and failure of proper authentication [100].

It can be prevented by AS through restricting the client to influence their client id or sub value or any other Claim. If it is not possible then AS must provide other means for RS to differentiate in between access token authorized by a resource owner and by the client itself [94].

2. **Cross-JWT confusion:** It is a specific type of substitution attack where the JWT token have been used for another purpose rather than its intended purpose [107]. It can hampers confidentiality if the same JWT is used to obtain different resources rather than the intended resources it issued for.

   For the countermeasure in step E from figure 4.3 after receiving JWT from client proper validation of the issuer is needed. If a same issuer can issue a JWT for the purpose of using it by more than one client or applications then it must needs to use distinct identifier as an 'aud' claim value to ensure that the same issuer issued the JWT for distinct resources [100] [107]. Also if the JWT contains an "iss" (issuer) claim, application must need to confirm that cryptographic keys used in the JWT are belongs to the issuer otherwise it must reject the JWT. Such as– in OIDC it uses "jwks_uri" value which is a "https" URL for retrieving issuer's keys as a JWK Set [107].

3. **Access token leakage:** In figure 4.3 the access token leakage can take place in step D. If the token is sent by AS to the client through a URI fragment of the redirect URI without making the communication or endpoint secure, the token can be leaked by the returned URI and the attacker can get the access of confidential data from resource server using that leaked token.

   To prevent this issue AS must ensure the confidentiality of the response by using TLS for the communication in between client and the server [96]. Using DPoP-JWTs with oauth can also be an useful countermeasure for this types of attack if it is used instead of the typical bearer token because it uses DPoP proof JWT with token request which helps the AS to verify that the client possesses the private key and after verifying the client in return the AS provide public key bound access token. DPoP-JWTs in details is explained in section 4.4

4. **Token Substitution:** This vulnerability can take place on an application which depends on OAuth protected service API to get identity data of a user for login the users. In a nutshell, for a client resource server API can be treated as an "identity" API because after getting access token by oauth server, client look up for an identifier inside Identity API or resource server and uses this identifier to find out internal user account data. If the client find those data then it assumes that the user has been authenticated. Now

the attack scenario can take place when the attacker trapped the victim to logging into a malicious app and uses the same identity provider to issue an access token. Attacker then achieve this access token through the malicious app. After manipulating the authorization response, the attacker then uses or substitute his identity bound access token to get the access of the victims confidential or private resources.

The countermeasures can be using OIDC [50] for user login by the client because audience restrictions on clients are supported there [96]. Point to be noted that 'aud' claim helps to identify the intended recipients of JWT.

## 4.4 DPoP-JWTs

DPoP is an OAuth 2.0 security extension which helps to prevent attackers from using leaked or stolen access tokens by providing sender-constraining access tokens. That means during the issuance of access token it binds the access token to a public part of a client's key pair and client needs to prove the possession of the private key to use the token thus used in contrast to the typical bearer token. The value of the DPoP header is a JSON Web Token (JWT) that enables to bind issued tokens to the public part of a client's key pair. In a traditional mechanism, API access can be allowed if the client application can provide a valid token but in DPoP it does an additional check to prove whether the client application is the same one that the access token has been issued to. BTW DPoP mechanism is not a client authentication procedure. The use of DPoP is primarily for applications on a user's device which do not use client authentication. [9]


**Figure 4.6:** Basic DPoP Flow [9]

According to figure 4.6 the basic DPoP flows are as below–

(A.) Before token request there are some steps that needs to be mentioned. The process starts with Generating public and private key pair. Then preparing JSON including the generated public key and a payload which will be used as the header of the JWT. Then generating signature using private key for the created data. After that DPoP proof JWT has been produced by packing them into a JWT. Now it is ready for sending request for access token (and potentially a refresh token) to the token endpoint.

(B.) Token endpoint then Extract the DPoP proof JWT from the token request. Verify the signature using the public key included in the DPoP proof JWT that ensures the client application has the private key which is corresponds with the public key. Token endpoint then generate the access token and bind public key to the access token and sends it to the client application. If refresh token is issued then it also bound to the public key.

(C.) Client generates fresh proof and send access token with newly created DPoP proof JWT to the resource server.

(D.) The resource server then check whether the public key included in the DPoP proof JWT matches the one that is bound to the access token. If it matches then it grants the access. [9] [35]

## 4.5 Relevant security objectives:

If we look figure 4.6 then we can see in every steps JWT has its involvement except step D.

1. In step A, Client needs to proof its authenticity to AS. For this purpose, client sends the token request to the AS with DPoP proof JWT through which the AS can ensure the authenticity of the client by ensuring that the client possesses the private key.

   In addition, It needs to be ensured during implementation that only asymmetric digital signature algorithms for example ES256 are used for signing DPoP proofs JWT which considered as more secure and recommended as per JSON Web Algorithms (JWA) specification 4.

   Now if the attacker exfiltrate the authorization code through Authorization Code Rebinding Attack explained in section 4.5.1, then it can sends the token request in this step with his own created DPoP key and DPoP proofs.

2. In step B, AS needs to check the authenticity of the client and must verify that the typ field is dpop+jwt in the headers of the JWTs otherwise it must not accept it. The AS then sends the sender constrained access token binding

it with the client's public key which prevents the attacker to use the stolen token.

Moreover, in this step, DPoP proof JWT replay detection and prevention can be done by AS through sending an error with 'dpop-nonce' header in response to ensure the freshness of the DPoP proof JWT Figure 4.7 4.8. Details about DPoP proof replay attack is explained in section 4.5.1


**Figure 4.7:** Refreshing DPoP Flow with nonce challenge. [11]

In addition, an authorization server may also issue access token which is not DPoP bound with a value of Bearer in the token type parameter of the access token response. In this case client must discard the response if it wants to maintain the application security. otherwise,it may continue as in a regular OAuth interaction. Moreover, a different token type value than DPoP can not ensure the security protection provided by DPoP. [11] [9]

3. In step C, client needs to produce a fresh DPoP proof JWT and sends it

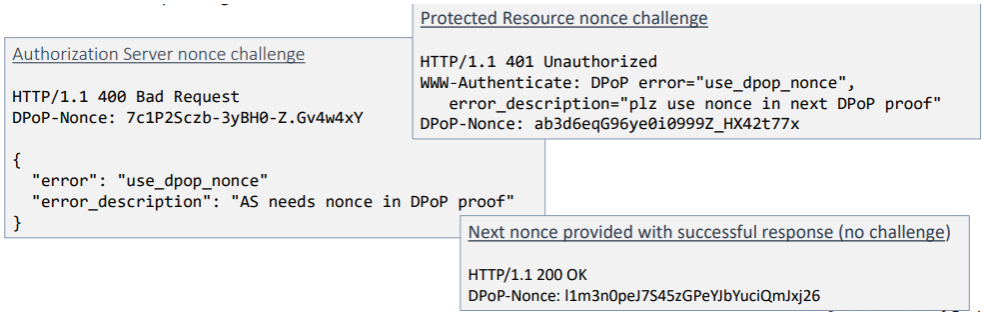


**Figure 4.8:** Example of server provided nonce challenges. [35]

alongside with the public key bound access token to the RS so that the RS can compare the public key in between them to ensure the authenticity of the client.

Moreover, in this step client can proof the authenticity of DPoP proof JWT by sending the fresh DPoP proof JWT which also been used to prevent DPoP proof JWT replay attack [9]. Figure 4.9.

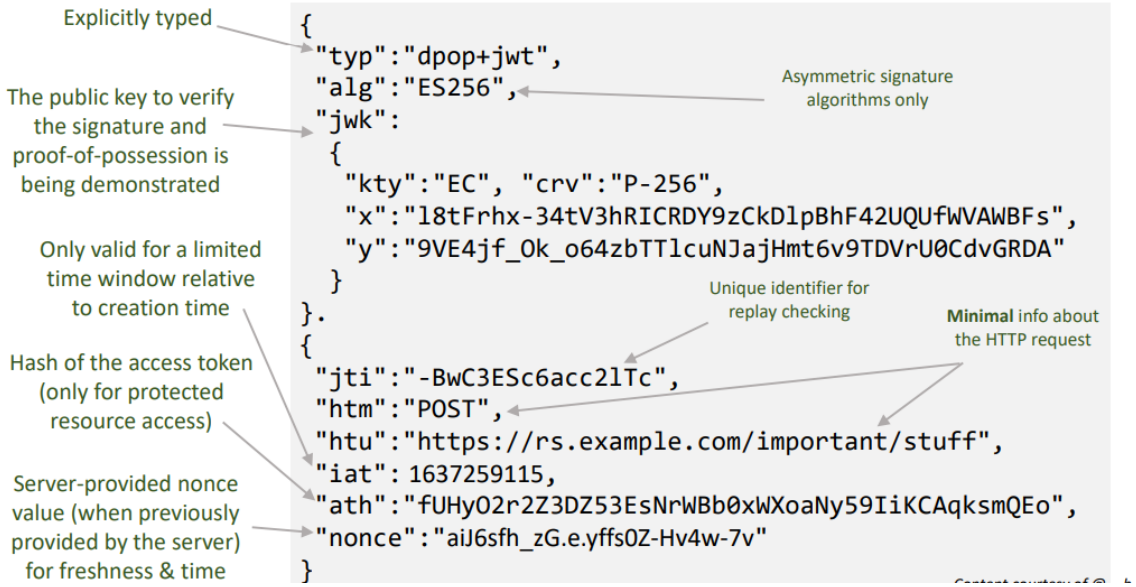


**Figure 4.9:** Anatomy of a DPoP Proof JWT. [35]

point to be noted that before providing the protected resources the RS may also supply 'nonce' as a response for an additional security check which can be used to prevent DPoP Proof Pre-computation Attack. Figure 4.14 4.15.

Both of the attacks are explained in section 4.5.1.

### 4.5.1 Important attacks:

1. **Authorization Code Rebinding Attack:** Now even before the token request DPoP flow can be brought backward to create an end to end connection or binding of the entire authorization flow. figure 4.10.


**Figure 4.10:** Authorization Code Binding to DPoP Key. [35]

In a traditional public client authorization request it uses Proof Key for Code Exchange (PKCE). But even though using PKCE and having authorization codes one-time-use practice, there are often a time window during when the replay attack can be possible. Attacker can also use XSS to obtain the authorization code and PKCE parameters. [68].

In brief, after authorization request when authorization server responds with authorization code, the attacker can get access of this code where the HTTP messages containing them are logged such as- proxy or Web server. figure 4.11.

The solution can be including dpop_jkt Parameter in the authorization code request figure 4.13. The value of dpop_jkt Parameter is DPoP jwk thumbprint. Now in return, AS binds the authorization code with DPoP JWK.

So even if the attacker exfiltrate authorization code from log file and sends his own created DPoP key and DPoP proofs as a token request in step A, from figure 4.6 to the AS, the AS can compares JWK Thumbprint of the proof-of-possession public key in the DPoP proof with the dpop_jkt parameter value in the authorization request. It reject the request if it does not match. figure 4.12

In brief, JWK thumbprint is a method to compute a hash value over a JSON Web Key (JWK) to identify which fields in a JWK are used in the hash computation so that it can be used for identifying the key represented by the JWK. [62].

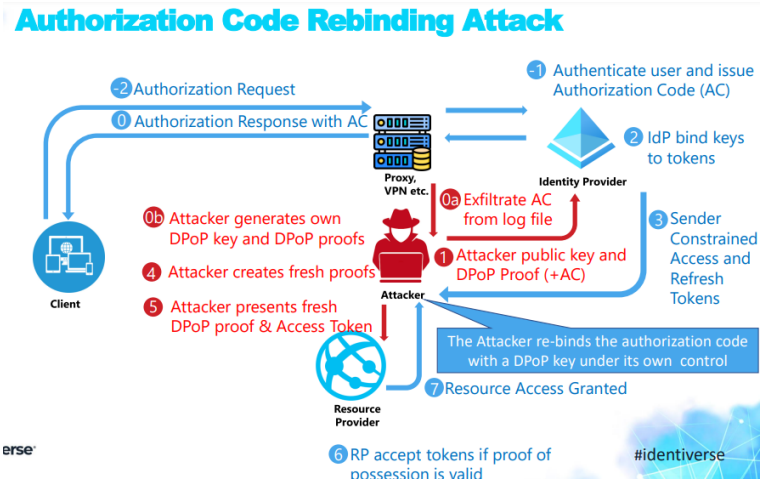


**Figure 4.11:** Authorization Code Rebinding Attack. [35]

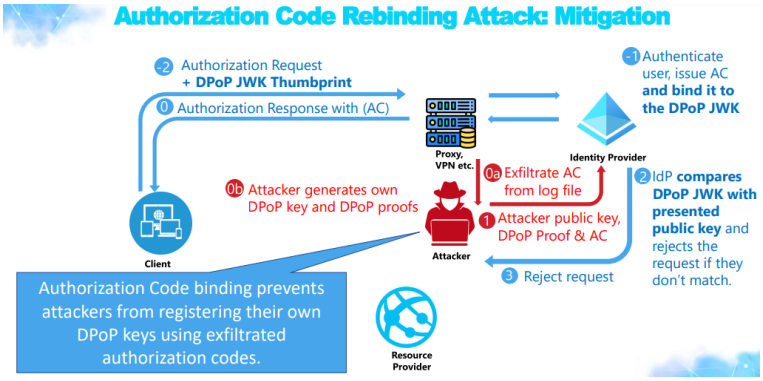


**Figure 4.12:** Authorization Code Rebinding Attack Mitigation. [35]

2. **DPoP Proof Pre-computation attack:** The attack can take place in step no C, from figure 4.6. An attacker who is a legitimate user of the client can pre-compute DPoP proofs by using the iat value in the DPoP proof to be signed by the proof-of-possession key figure 4.14. For example, the attacker can pre-generate or pre-compute DPoP proofs on one computer holding the proof-of-possession key upon which they were generated and copy them to another machine which does not possess the key for using in future. Then the attacker can use that pre-computed DPoP proof with exfiltrated token as a request to the resource server.

   The prevention can be done by using resource server provided 'nonce' which is not predictable by the attacker figure 4.15.

3. **DPoP Proof Replay:** DPoP proof replay can be possible in step no B figure 4.6 if the attacker somehow can get hold of a DPoP proof JWT.

```
GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=xyz
    &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
    &code_challenge=E9Melhoa2OwvFrEMTJguCHaoeK1t8URWbuGJSstw-cM
    &code_challenge_method=S256
    &dpop_jkt=NzbLsXh8uDCcd-6MNwXF4W_7noWXFZAfHkxZsRGC9Xs HTTP/1.1
Host: server.example.com
```

**Figure 4.13:** Authorization Request using the dpop_jkt Parameter. [9]

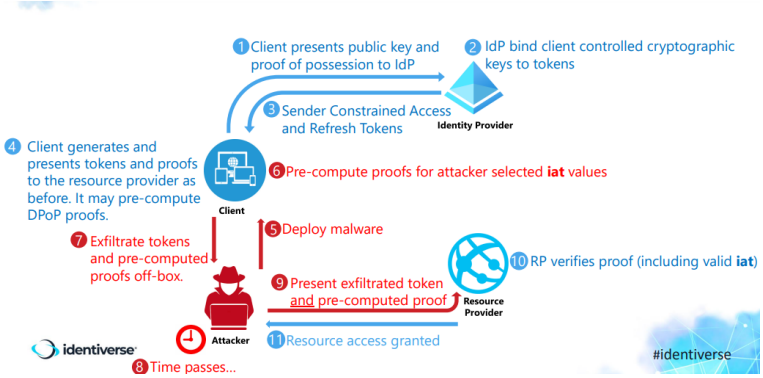


**Figure 4.14:** DPoP Proof Pre-computation Attack. [35]

Prevention can be done by AS if after verifying DPoP header AS can provide an additional check by sending an error with 'dpop-nonce' header in response to ensure the freshness of the DPoP proof JWT. If it is the case, then the client must needs to send request again including 'nonce' and 'jti' values in the JWT payload. Here 'jti' is an unique identifier for the DPoP proof JWT.

In addition, if the attacker tries to replay that DPoP proof JWT to get the access of the resources from protected RS in step no C, figure 4.6 then it can also be prevented by using fresh DPoP proof JWT by the client which contains an additional claims which is 'ath'. It is a hash of the associated access token. In this way, Uses of a same proof with multiple different access token values across different requests can be prevented. Point to be noted, to check ath only is not enough to prevent replay of the DPoP proof. It is also important to check 'htm', 'iat' and 'htu' parameter as well. [9]. Figure 4.9

## 4.6 mTLS–better security option than DPoP

Before jumping on explaining mTLS it is necessary to know the basic rules of TLS to verify identities which is explained in Appendix B. TLS is fine For everyday as usual use to keep private data secure and encrypted as it crosses the various networks and to verify the server's identity. But it is not enough to prove the identity for the client as anyone can connect to the server and initiate a secure connection,

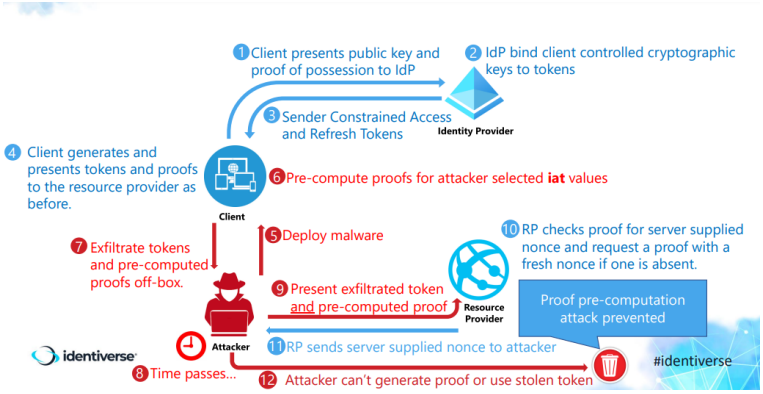


**Figure 4.15:** DPoP Proof Pre-computation attack Mitigation: Server Nonces. [35]

even if the client is not authorized to do so. Here, to avoid this situation an extra security implementation is needed. Whereas in TLS only the Server presents the certificate issued by a Certificate Authority (CA) in mTLS the Client also presents a client certificate issued by the CA to establish its identity and a mutual trust in between them. Since, it is an extension of TLS, the TLS connection in between client and the authorization server needs to be re-established with mutual-TLS X.509 certificate authentication.

The basic mTLS flow is shown in figure 4.16

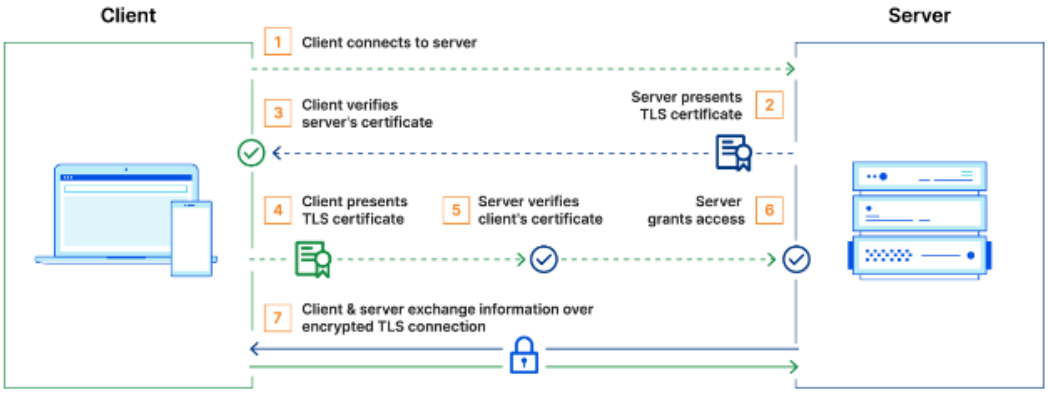


**Figure 4.16:** mTLS flow. [102]

### Client Certificate-Bound Access Tokens

Some important observation in between DPoP and mTLS are as below:

In DPoP, authorization server can bind DPoP with issued access token such

a way so that it can be accessed or verified by the protected resource. In mTLS it bind the issued access token with the certificate hash directly using JWT Certificate Thumbprint or which enables mutual TLS to serve purely as a proof-of-possession mechanism. In this case, the access token itself can not be used to make decision whether to request mTLS or not, to use mTLS or certificate-bound access tokens the resource server must have the knowledge that mTLS is to be used for all of its resource accesses. In mTLS protected resource-access flow, client request upon getting protected resources must be done over a mutually authenticated TLS connection using the same certificate that was used at the token endpoint using mtLS. To ensure that same certificate has been used, the protected resource server must obtain the client certificate used mTLS from TLS implementation layer to compare it with the certificate bind with access token. If they do not match then resource access request must be rejected with an HTTP 401 status and the invalid token error code.

### JWT Certificate Thumbprint Confirmation Method

Here JSON Web Tokens (JWT) is used to represent access token. The hash of a certificate also represented by a JWT where the payload contains x5t#S256 for the X.509 Certificate SHA-256 Thumbprint confirmation method [57]. In JWT claim set the X.509 certificate thumbprint confirmation method can be found inside a cnf claim under "x5t#S256" header or confirmation method member which contains the hash of the client certificate to which the access token is bound. [7]. figure 4.17

```
{
  "iss": "https://server.example.com",
  "sub": "ty.webb@example.com",
  "exp": 1493726400,
  "nbf": 1493722800,
  "cnf":{
    "x5t#S256": "bwcK0esc3ACC3DB2Y5_lESsXE8o9ltc05O89jdN-dg2"
  }
}
```

**Figure 4.17:** JWT Claims Set with an X.509 Certificate Thumbprint Confirmation Method. [7]

here the hash of the certificate carries the meta information to the protected resource in a token introspection response. The protected resource then compares this certificate hash with the hash of the client certificate used for mutual-TLS authentication and rejects or accepts the resource access on the basis of the comparing result.figure4.18

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "active": true,
  "iss": "https://server.example.com",
  "sub": "ty.webb@example.com",
  "exp": 1493726400,
  "nbf": 1493722800,
  "cnf":{
    "x5t#S256": "bwcK0esc3ACC3DB2Y5_lESsXE8o9ltc05O89jdN-dg2"
  }
}
```

**Figure 4.18:** Introspection response for certificate bound access token. [7]

### Security aspects

Since mTLS is based on zero trust architecture, it can give high level security. Where it can prevent modifying communication, spoofing web server, credential leakage, Brute force attack, Malicious API requests [102]. Even though its high security assurance, in digital realm nothing is hundred percent secure. As implicitly granted access assuming all the origination is from the authentic client, it can be dangerous in case of misconfigured server where due to inappropriate it might not ask certificate from the client. This opens the possibility for the attacker to inject the request and the request can be served silently unauthenticated.

Some more security aspects are as below which needs to be considered seriously –

1. **X.509 Certificate Spoofing attack:** When the attacker uses certificate with the same subject (DN or SAN) but issued by a different CA whom the authorization server trusts, then this kinds of certificate spoofing attack can be possible. Here point to be noted, X.509 certificates are traditionally used for mTLS OAuth client authentication. It does that by validating certificate chain and comparing a single subject distinguished name (DN) or a single subject alternative name (SAN). For each client only one subject name value is used. In TLS handshake besides validating client's possession of the private key, it also validate the corresponding certificate chain which contains an ordered list of all certificates and Certificate Authority (CA) Certificates which used to verify the trustworthiness of the sender and the CA. In terms of validating (SAN) it checks if the single expected subject which is used to register a client is matched with the subject information in the certificate. For comparing the certificate's subject DN to the client's registered DN the distinguishedNameMatch rule from [89] are needed. To avoid this threat,the authorization server should accept only some limited trustworthy CAs which meets its security requirements in terms of issuing certificate policy. To vali-

date the certificate chain the client and the server must agreed upon a set of trust anchors. Without consider these precautions it might open the possibility of certificate spoofing attacks.

2. **Parsing and Validation of X.509 Certificate:** Without parsing and validating certificate properly which is a complex procedure and due to implementation mistakes, many security vulnerabilities may take place. [7]

BTW both mTLS and DPoP are designed to provide sender constraint and ensure the legitimacy of the token sender but DPoP is recommended to use only if there is no mTLS or oauth token binding are available. [95]. The reason behind it can be many. some reasons are as below–

1. DPoP is a newer method compared to mTLS, that means it may not have the same level of widespread acceptance. mTLS was already adopted by FAPI 1.0 Advanced [66] and is widely used in Open banking UK where DPoP is relatively new.

2. mTLS provide strong security guarantees since it relies on Public Key Infrastructure (PKI) and certificate-based authentication.

   Even though mTLS has widespread adoption and strong security, the situation might change cause DPoP does not rely on PKI infrastructure, which makes it easier to implement compared to mTLS, it also provide Application-layer security with asymmetric cryptography and lightweight JSON Web Tokens (JWTs). Both public and confidential clients can use it. Most importantly, in the new version of FAPI 2.0 Security Profile allows both mTLS or DPoP to be used. [20]

## 4.7 OpenID Connect (OIDC)

OIDC is an authentication protocol which supports different applications to authenticate and verify the identities of their users in a secure way. The apps that uses OIDC rely on identity provider for secure authentication and verify the identities of their users. Point to be noted, identity provider is not employed inside OIDC, the OIDC protocol uses authorization server to authenticate end users.

In a more simpler way we can say, if we use an application which provide their services only to the authenticated users, instead of using users credentials the app can use OIDC to redirect the authentication request to an identity provider (for example– Google, Microsoft) to prove their identities. After authenticating the user via identity provider, the app can allow the authenticated user to access their services. How those identity providers maintain the authentication procedure of the users in the first place is not something that OIDC needs to be concerned about.

The main role of OIDC is to ensure the secure interaction in between apps and the identity provider to authenticate users. [43]

### 4.7.1 Important terms to know

- **Client registration:** In OIDC client needs to register itself first at the Authorization Server so that it can recognize the registered client whenever it requests something from the authorization server on behalf of the user. To do that new client needs to send an HTTP post message to the Client Registration Endpoint of authorization server with any Client Metadata parameters. Client specify the metadata for itself. The authorization server then assigns a unique client ID, optionally client secrete and associates client's metadata with this issued Client Id. The authorization server can reject any of the Client's requested field values and substitute them with suitable values. If it happens then authorization server must includes those fields with its response. The response from authorization server may contain a Registration Access Token which can be used by the client to perform any subsequent operations afterwards. figure 4.19 [67]

```
HTTP/1.1 201 Created
Content-Type: application/json
Cache-Control: no-store
Pragma: no-cache

{
 "client_id": "s6BhdRkqt3",
 "client_secret":
   "ZJYCqe3GGRvdrudKyZS0XhGv_Z45DuKhCUk0gBR1vZk",
 "client_secret_expires_at": 1577858400,
 "registration_access_token":
   "this.is.an.access.token.value.ffx83",
 "registration_client_uri":
   "https://server.example.com/connect/register?client_id=s6BhdRkqt3",
 "token_endpoint_auth_method":
   "client_secret_basic",
 "application_type": "web",
 "redirect_uris":
   ["https://client.example.org/callback",
    "https://client.example.org/callback2"],
 "client_name": "My Example",
 "client_name#ja-Jpan-JP":
   "クライアント名",
 "logo_uri": "https://client.example.org/logo.png",
 "subject_type": "pairwise",
 "sector_identifier_uri":
   "https://other.example.net/file_of_redirect_uris.json",
 "jwks_uri": "https://client.example.org/my_public_keys.jwks",
 "userinfo_encrypted_response_alg": "RSA1_5",
 "userinfo_encrypted_response_enc": "A128CBC-HS256",
 "contacts": ["ve7jtb@example.org", "mary@example.org"],
 "request_uris":
   ["https://client.example.org/rf.txt
     #qpXaRLh_n93TTR9F252ValdatUQvQiJi5BDub2BeznA"]
}
```

**Figure 4.19:** example of OIDC client registration request to the authorization server client registration endpoint. [67]

- **ID token:** ID token is a JSON Web Tokens (JWT). it contains:
  - unique user identifier.

- issuing authority that means the AS URI.
- The intended audience which is the client application.
- Issue and expiration times.
- How the user was authenticated for example either via password or multi-factor-authentication.

ID tokens are used for enabling user access to client-hosted resources, that means it is an identity and authentication assertion issued by the AS and consumed by the client. In order to provide the authenticity and integrity of the token, the OP or OpenID provider or identity provider is responsible for signing it using JSON Web Signature (JWS). figure 4.20


**Figure 4.20:** Example of OIDC ID token. [69]

- **Access token:** On the other hand, access tokens are used for enabling user access to a resources hosted by a resource server. Which is an identity and authorization assertion issued by the AS and consumed by the RS.

### 4.7.2 How OIDC Works

1. When end user wants to get access of resources on the client, the user requests to start authentication procedure via the client. Client then prepares authentication request with desired parameters or scopes which contains the information the client wants to access on behalf of the user.

2. The client redirects user to the identity provider which is the authorization server in this case. The identity provider or OP or OpenID Provider redirects to the user with some artifacts where it might ask which particular identity provider the user wants to use to prove his identity or to authenticate himself

(such as - Google or Microsoft). This user choice used by the app or relying party to issue the request to the identity provider to complete user authentication. OP or open ID provider then authenticate end-user and obtain end user consent or authorization. figure 4.21

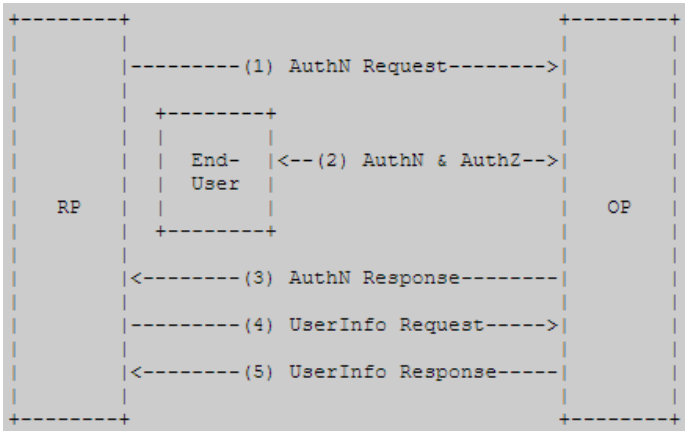


**Figure 4.21:** Basic OIDC flow. [50]

3. The token endpoint of OP or OpenID provider then responds with an ID token and an access token. Client then validates the token and recovers the End-User's Subject Identifier from the token.

4. If client needs some more information about enduser then it can send HTTPS "GET" request to the user info endpoint including access token in the authorization header.

```
{
   "sub"            : "alice",
   "name"           : "Alice Adams",
   "given_name"     : "Alice",
   "family_name"    : "Adams",
   "email"          : "alice@wonderland.net",
   "email_verified" : true,
   "phone_number"   : "+359 (99) 88200305",
   "profile"        : "https://c2id.com/users/alice",
   "ldap_groups"    : [ "audit", "admin" ]
}
```

**Figure 4.22:** Example of user info in JSON format. [69]

5. User info endpoint of OP responds with user info as a JSON document asked by the client or RP when fetched via HTTP. figure 4.22 [50] [104]

### 4.7.3 Different Uses of OIDC

Most interestingly OIDC can be used in three different scenarios–

1. Without asking the user to create new user profile or account with their credentials, the application can take the advantage of OIDC to reuse their accounts on an identity provider to authenticate themselves. It can make sign up procedure much more smoother. It also make it easier to acquire more personal information about users.

2. By using OIDC the app can connect to a single hub that works for multiple identity provider, instead of communicating with multiple providers separately. The connection of OIDC with SSO and Federated Identity Management (FIM) is explained in appendix B.5

3. Thirdly it can be used as a proxy for other protocols like SAML.

### 4.7.4 SAML (Security Assertion Markup Language) vs OIDC

Now another important question is both SAML (Security Assertion Markup Language) [6] and OIDC uses digitally signed token to carry users personal data and both can be used for the same purposes, then why using OIDC. First of all, SAML does not support SSO for mobile devices or provide API access, it can only be used to access browser-based applications. Also SAML is XML based which can create an issue during signature verification if two elements listed in a different order, also another reason is XML format can be very heavy where there is a low-speed connection and also just for authenticating an end-user with some of their attributes. In addition, from security perspective using JWT format referred to as JOSE for signing and encryption provide more compact security tokens than XML used in SAML. Small security tokens are also suitable for addressing the URL and header size constraints in mobile environments. [43]

### 4.7.5 Relevant security objectives

As we know about the data flow of OIDC and how it works, now it is essential to point out exactly where the JWT plays its role in OIDC to fulfill the security objectives.

1. if we look into figure 4.21 we can see in the first step the relying party sends the authentication request to the OpenID provider or OP. Now OIDC can use JWT to sign and optionally encrypt the authentication requests to provide the confidentiality of the request. To do this it uses some authorization request parameters such as request_uri or request parameters which are represented as JWTs. SSRF via request_uri parameter can take place if the request_uri

is not pre-registered by the client during registration procedure which can allow a malicious client to use any arbitrary request_uri and impersonate itself as a benign client. It might cause leaking of sensitive data, such as authorization credentials. Disclosure of server response can also take place in this step which can be prevented by authenticating the client properly by AS. Details about this kind of attacks is explained in section 4.7.6

2. In step no 3, The authentication response by OpenID provider contains an ID token which is a JWT figure 4.20. The integrity of the token can be hampered and leakage of confidential data can take place If it is been compromised by the attacker. if a malicious "jwks_uri" has been used by the attacker during new client registration then the attacker can achieve an authorization code for any user and can obtain the token from this step by using that code which is explained in section 4.7.6.

3. In step 5 the user info response can also be signed and/or encrypted using JWT [50]. Again as the additional user info as per figure 4.22 needs to be transferred in this step in the form of JWT from OpenID provider to relying party, the integrity of the JWT and confidentiality of the data inside it needs to be maintained.

### 4.7.6 Important attacks

1. **SSRF via request_uri parameter**

   In OIDC new clients needs to register itself to the Client Registration Endpoint which is an OAuth 2.0 Protected Resource. In this endpoint new client registration can be requested. request_uri is a part of a client metadata which is sent during registration request. Clients may pre-register request_uri by the RP for use at the OP. OPs can cache and pre-validate the request parameters at Registration time, that means they don't need to be retrieved at request time. The RP can use this request_uri to provide a URL that contains a JWT with the request information. figure 4.23.

   To avoid SSRF attack the client must need to serve this uri beforehand during registration so that server can whitelist those URLS and don't allow any arbitrary request_uri. [50] [90]

2. **SSRF during dynamic client registration**

   If we look figure 4.19 which is an example of a POST request send by the client to the registration endpoint for registering itself with the OpenID provider. One of the value is "jwks_uri" which is passed in via URL links and can be potentially vulnerable to SSRF attack. Also most of the servers do not resolve

```
https://server.example.com/authorize?
  response_type=code%20id_token
  &client_id=s6BhdRkqt3
  &request_uri=https%3A%2F%2Fclient.example.org%2Frequest.jwt
  %23GkurKxf5T0Y-mnPFCHqWOMiZi4VS138cQO_V7PZHAdM
  &state=af0ifjsldkj&nonce=n-0S6_WzA2Mj
  &scope=openid
```

**Figure 4.23:** Example of authorization request made by the client to the authorization endpoint using req_uri. [50]

those URLS immediately, they just save those parameters for later use during Oauth authorization flow.

**jwks_uri:** This refers to the client's set of JSON Web Key [JWK] document. During the use of JWT for client authentication the server needs those key sets to validate signed request made by the client to the token endpoint. SSRF attack may take place through this parameter by sending a malicious "jwks_uri" during new client registration which can allow to obtain an authorization code for any user. [90].

The countermeasures can be done by making the client registration procedure much more stricter so that client can not register itself without confirming "jwks_uri" and the AS needs to whitelists those "jwks_uri" so that no malicious "jwks_uri" can be accepted by the AS later on.

3. **Disclosure of server response**

   As the server response might contain the sensitive information about client, it can be vulnerable with different types of attack if it is been disclosed to the attacker. The disclosure can be prevented by using the client_id and client_secret to authenticate the client by AS. Also the response needs to be sent over TLS. In addition a signed and encrypted JWT can also be used to protect the confidentiality and integrity of the response.

   Since OIDC works on top of the the OAuth 2.0 protocol, some of the attacks explained in section 4.3 can also be applied on OIDC as well.

## 4.8 Selective Disclosure JWT (SD-JWT)

JSON Web Token (JWT) has its many uses. Representing a user's identity is one of the common use case of a signed JWT. If a person wants to disclose some of his verifiable information to a specific verifier, he typically uses one-time JWT which contains those claims. Due to increasing number of uses of this kinds of JWT the "Holder" of the JWT or the user needs to create a signed JWT once and then used

it number of times. Here selectively disclosure JWT or SD-JWT takes the control where it creates a subset of those claims depending on the verifier to ensure minimum disclosure and prevent Verifiers from obtaining unnecessary or irrelevant claims. SD-JWT also can provide the protection against undetected modification as the claims can be hidden, but cryptographically secure [21].

### 4.8.1 Verifiable Credentials

One of the example of above mentioned multi-use JWT is a verifiable credential where the authorship of the claims about a subject can be verified cryptographically. In a more simpler way, we can compare verifiable credentials with physical wallet where we can hold one or more physical identity cards. In digital world an app on a mobile phone can act as a digital wallet. Verifiable credentials are like the identity cards into the wallet app. A verifiable credential can be presented to a web service, same as a loyalty card that we need to present during making purchases at a grocery store.

**Important terms to know**

1. **Claim:** A claim is an assertion or statement about a subject.
2. **Holder:** Holder is the possessor of one or more verifiable credentials of a subject. It stores these credentials in credential repositories.
3. **Issuer:** The role of issuer is asserting claims about one or more subjects, generating verifiable credential from these claims and transmitting them to a holder.
4. **verifiable presentation:** It is a cryptographic method where the authorship of the data can be trusted after cryptographic verification process. It derived from one or more verifiable credentials which can be referred as a subset of one's persona. It packed in such a way that the authorship of the data is verifiable or trusted. We can co-relate this with zero-knowledge proofs where without disclosing the actual value an entity can prove to another entity that they know a certain value.
5. **Verifier:** It receives one or more verifiable credentials, optionally inside a verifiable presentation for processing. It can be co-relate with relying party. [101]

**Relationship in between Issuer-Holder-Verifier**

1. In general issuer generate verifiable credentials from claims and issues it to the holder.

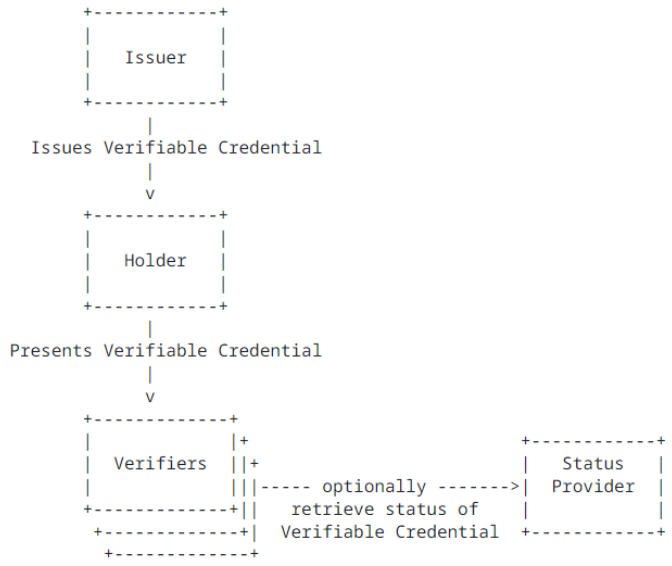


**Figure 4.24:** Issuer-Holder-Verifier Model. [70]

2. Holder then presents the verifiable credential to the verifiers.

3. Verifiers then verifies the authenticity of the data inside verifiable credential which is a cryptographically signed statements or claims about a Subject. Also it optionally asks the holder to proof possession of a cryptographic key mentioned in the credential to ensure that they are the actual holder of the Verifiable Credential.

4. The role of status provider here is optional where revocation information about a verifiable credential are kept and it is being used to retrieve those informations or the current status of a credential.

   This common specification can be used both with SD-JWT based verifiable credential and without selective disclosable claims as well. figure 4.24 [70]

### 4.8.2 Issuance of SD-JWT

**Some important terms to know**

1. **Digest hash:** Digest hash or Digest/Hash function generates a digital summary of information known as message digest which provides a digital identifier for a digital document. It is Base64 encoded that represents binary data in an American Standard Code for Information Interchange (ASCII) string format. [24]

2. **Random salt:** Random salting refers to adding random data into the input

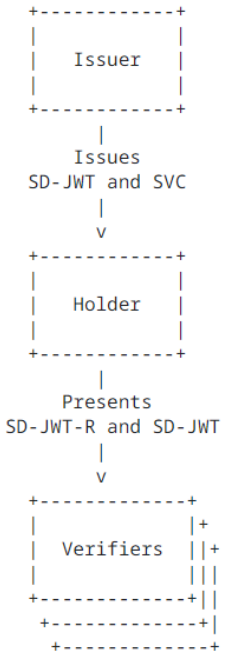


**Figure 4.25:** Issuance of SD-JWT with presentation flow. [21]

of a hash function to generate a completely unique output even though the inputs are the same which strengthen the data against different attack vectors. such as hash table or brute force attack. [3]

3. **SD-JWT Salt/Value Container (SVC):** It is the container of the mapping between salt value for each claim and raw claim values contained in the SD-JWT which is a JSON object created by the issuer.

4. **SD-JWT Release (SD-JWT-R):** It is a subset of the claim values of an SD-JWT created by the holder. It is a JWT which contains those informations in a verifiable way.

5. **Holder binding:** It is the ability of the holder to prove its possession of SD-JWT. It does that by proving its control over a same private key during issuance and presentation. When issuer signed a SD-JWT, it contains or refers a public key which matches with the private key of the holder. [21]

### Explanation of the SD-JWT Issuance and Presentation Flow

1. The first step is to generate SD-JWT by the issuer which contains hash digests over the claim values including unique random salts and other metadata. It is

a digitally signed document using the issuer's private key. The issuer sends this SD-JWT with SVC.

2. In the next step holder creates a JWT known as SD-JWT-RELEASE which contains a subset of the SD-JWT claim values in a manner so that it can be disclosed to the verifier and the verifier can varify the information. Holder then sends that SD-JWT-RELEASE alongside with the SD-JWT to the verifier. Optionally the holder binding can also be added. To do that holder needs to sign SD-JWT-RELEASE with its private key which corresponds to holder's public key.

3. Vrifier then checks if the hash digest hash matches the one under the given claim name in SD-JWT for each claim in SD-JWT-RELEASE. Which means it validate the claim values by calculating the hashes of those values received in SD-JWT-R and comparing them with the hashes in the SD-JWT. figure 4.25 [21]

## 4.9 Relevant security objectives

Now couple of security measures needs to be taken to make the SD-JWT flow secure or fulfilling the security objectives [21].

1. If we follow figure 4.25 where the flow of sd-jwt has been explained with Issuer-Holder-Verifier model then we can see in the first flow Issuer needs to issue sd-jwt and sends it to the holder. Here issuer must need to sign the SD-JWT with its private key to ensure the integrity of the claims so that the attacker can not modify the claims.

   Now after receiving the SD-JWT and SVC from the issuer, the holder should check or verify the bindings in between them by checking that all the claims mentioned in SVC are present in the SD-JWT and the hashes of the claims in the SVC are matches with those in the SD-JWT.

2. In the next step verification needs to be done by the verifier when it Receives SD-JWT and SD-JWT-R from the holder.

   At first, ensure if the holder binding needs to be checked (as it is optional) to prove that the SD-JWT contains the public key corresponds with the private key of the holder. It ensures the authenticity of the SD-JWT.

   To ensure the integrity of the SD-JWT that it has not been tampered on its way since its issuance by the issuer, the verifier must need to check the signature of the SD-JWT to ensure that the signing key belongs to this issuer. In this way it validated the issuer as well. The SD-jWT must be rejected, if the signature can not be verified.

The verifier needs to Check the validity of SD-JWT by using nbf, iat, and exp claims.

Validating the SD-JWT-R is also required by ensuring both SD-JWT-R and SD-JWT has been signed using the same key and by checking that SD-JWT-R was created for this particular verifier using a nonce and aud field inside SD-JWT-R.

Claims in the SD-JWT Release are also needs to be verified to ensure that all the claims inside SD-JWT are present in SD-JWT-R in a verifiable way.

### 4.9.1 Security aspects of SD-JWT

1. **SD-JWT signing must be mandatory**

   Issuer must protect the integrity of the issued claims by signing the SD-JWT with its private key to avoid the modification of the claims such as changing email attribute to take the possession of the victim's account or adding claims or attributes to produce a fake academic qualification. On the other hand, verifier must check the SD-JWT signature to ensure it is not been tempered since the issuance by the issuer. It also must reject the SD-JWT if it is not possible to varify it by the verifier.

2. **Ensuring Uniqueness of the salt**

   To maintain the security principle of the model it is necessary to ensure that salt must be cryptographically random, unique, long enough and has high entropy which must be at least 128 bits so that it can not be guessed by the attacker.

3. **hash function choice**

   The hash algorithms MD2, MD4, MD5, RIPEMD-160, and SHA-1 must not be used since they have some fundamental weaknesses. Also the hash function needs to be collision resistant where it is hard to find 2 inputs with the same hashed output. [21] [93]

   Furthermore, Using those insecure hashes can make the data vulnerable to collision attacks where the attacker can produce two similar hashes for two different pieces of data which can be used to substitute one piece of data for another silently without being detected. It can cause sensitive data disclosure, dictionary attacks to break weak or outdated hashes. [30]

# Chapter 5

# Analysis

As we already described in details about JWT, its uses in different data flows, the relative security objectives which needs to be fulfilled on those flows and also some important attacks, now it is time to draw some reflections upon our overall findings.

In this chapter, we will try to summarise our overall findings and give some overviews which can lead us to fulfill our thesis goal mentioned in section 1.4.

## 5.1 Some similarities and differences amongst the data flows in terms of using JWT

In table 5.1 we can see different uses of JWT in different data flows. It has total 7 columns which represents the different uses of JWT and it has total 5 rows which represents the name of the different data flows. If any row or column space filled with red then it means that the particular uses of JWT is not applicable for that particular data flow. But if any column space or rows is filled with green color including check marks then it means that the particular uses of JWT is applicable for that particular data flow.

Now if we look for the similarities and differentiation among the data flows regarding different uses of JWT from table 5.1 then we can observe JWT as an access token can be used in four of the data flows whcih is Oauth, DPoP-JWTs, OIDC, mTLS. Since access token is used for requesting resources from the RS (Resource server), the above mentioned four data flows uses that same access token as JWT from Authorization server to request resources from the RS. Now in terms of token request, only oauth uses JWT as an authorization grant in some special cases since it is not a common types of grant. The common types of grant used in oauth 2.0 can be found here [103]. DPoP-JWTs can also be used in token request but none of the other flows such as OIDC, mTLS uses JWT for token request. Among the data

flows mentioned in table 5.1 only OIDC uses JWT for authentication request and as an ID token.

## 5.2 Uses of JWT in different data flows

Here if we look at table 5.1 then we can observe the different types of uses of JWT in different data flows which is explained in chapter 4.

| Name | Authentication request | Authorization grant | Token request | Access token | ID token | Resource request | Information transmission |
|---|---|---|---|---|---|---|---|
| OAuth | | ✓ | ✓ | ✓ | | ✓ | |
| DPoP-JWTs | | | ✓ | ✓ | | ✓ | |
| OIDC | ✓ | | | ✓ | ✓ | ✓ | |
| mTLS | | | | ✓ | | ✓ | |
| SD-JWT | | | | | | | ✓ |

**Table 5.1:** Uses of JWT in different data flows

### 5.2.1 Usage of JWT as authorization grant and token request

JWT can be used as an authorization grant for Oauth which is used by the client to request an access token from AS and through this access token the user can grant the limited access of his resources to another entity without exposing his credentials. So, JWT plays a very important role in this case because through this request JWT conveyed the client secrete that can be used to authenticate the client by AS. Now if this grant somehow intercepted by the attacker specially if the authorization flow happens over a non TLS or vulnerable TLS connection or if a user's router has been compromised then it can make the whole flow vulnerable because if we observe the security objectives which needs to be fulfilled by the JWT as an authorization grant then we can notice that the grant is used to ensure the authenticity of the client and also to provide the non-repudiation so that the client can not deny the request sent by it later on. Also if the grant can be leaked then the attacker can obtain the access token using that same grant from AS which disrupts the integrity of the token and using this token the attacker can also gain access of

the confidential data of the user. So, four of the security objectives explained in section 1.2 can be hampered if the attacker gain the possession of this grant.

Now we can try to co-relate this problem with some of the other data flows discussed in chapter 4 and try to overlook if we can find any uses of JWT in this similar point where we need to use authorization grant or code for token request to avoid this grant leakage by the attacker.

One solution can be using oauth 2.1. Although, it is not finalized yet but we can discuss it as a future usage and countermeasures. The reason to have oauth 2.1 in this discussion because oauth 2.1 specification made the usage of PKCE mandatory for all oauth 2.1 clients which can solve this issue by providing the protection against authorization code exfiltration during its issuance by AS. It was primarily designed to protect public clients or native apps from authorization code exfiltration attacks in oauth 2. But in reality all clients are vulnerable to authorization code injection attacks which can be solved by PKCE using the authorization code flow.

Moreover, even if we use PKCE and one-time authorization code usage practice, there are often a time window where the attacker can obtain the authorization code and PKCE parameters and do the replay attack.

So, lets see if we can find some other data flows from the table 5.1 that uses JWT for token request, can give more strong security in this case. We can see from the table that DPoP-JWTs also can be used with oauth in an HTTP header for the token request. If we recall the usage of DPoP-JWTs with oauth flow from section 4.4 we can find that during the authorization code request DPoP jwk thumbprint as a dpop_jkt Parameter can be included. So, even if the attacker get the code and manipulate it using his own created DPoP key and DPoP proofs for token request, the AS can compare the JWK thumbprint value from DPoP proof JWT with the dpop_jkt Parameter. So, in this scenario if it does not match with attackers created DPoP proofs then it rejects the request thus prevents the access token stealing using the exfiltrated authorization grant or code.

Point to be noted, in both cases where we can use only Oauth with PKCE or Oauth with DPoP-JWTs might not be able to prevent the authorization code exfiltration or leakage but even if the attacker obtain the authorization code, the attacker can not obtain the access token using that same authorization code if we use DPoP-JWTs with Oauth.

That means the four security objectives can be ensured if we use DPoP-JWTs with oauth because in this way client authentication by AS can be more secure since DPoP proof JWT can provide client's proof of possession of private key. In the same way it can establish non-repudiation as well. In addition, by preventing leakage of the access token it can ensure the integrity of the token as well. But one security aspect needs to be considered here which is, even if we use DPoP-JWTs, a legitimate user can still use the token for some malicious purposes which can be

the reason to obtain the confidential data or resources by the attacker. The details about this kind of vulnerability will be discussed in the next sub-section.

### 5.2.2 Usage of JWT as an access token and resource request

In previous sub-section we explained what security objectives need to be maintained during using JWT as a token request, how it can be vulnerable and disrupts those objectives and how we can make the secure usage of JWT in a token request. Now here in this sub-section we will keep the continuation of the previous sub-section where we will discuss after achieving the token securely what security complications can take place regarding using JWT as an access token.

JWT as an access token is one of the most important usage of JWT in terms of providing application security since it is related to obtaining resources from the resource server. Ensuring the integrity of the token is very important here to safeguard the leaking of confidential data from the resource server.

Now when AS issues an access token, it issues the refresh token as well which is used to generate a new access token. Normally the life span of a refresh token is much more higher than the access token that is why it needs more attention and care to secure it. Otherwise if the attacker get it then he can make a new access token with it and the resources protected by that access token are no longer safe anymore.

The countermeasures to detect refresh token replay attack can be done by ensuring that it must be either sender-constrained by using mTLS, DPoP or one-time use where it rotate the new refresh token with the older one in every access token refresh response. In this way even if the attacker get the refresh token and the same refresh token being used by both attacker and legitimate user, then one of them will show invalidated refresh token. Although the server will not be able to find the attacker using invalidated refresh token, It at least can withdraw the active refresh token and also the access authorization grant associated with it so that the legitimate client needs to obtain a fresh authorization grant.

In addition, some security complications may take place during using the JWT as an access token to gain the access of resources from the RS. Such as, using the same JWT for another purposes rather than its intended purpose and access token manipulation by sending attacker bound access token to the RS. Another security concern regarding access token is if any legitimate user of a client turned into an attacker or a malicious user then he can use the access token for malicious purposes to achieve the protected resources from RS.

To avoid above mentioned scenarios the RS needs to take some extra precautions so that it can ensure the client authenticity, token integrity, non-repudiation and preventing confidential data leakage from RS by the attackers.

One of the measures can be using DPoP-bound access token or certificate bound

access token with mTLS so that the RS can easily check the authenticity of the client and ensure non-repudiation. If client authentication can be ensured properly by using mTLS or DpoP then we can ensure the token integrity as well. Thus, prevents the confidential data leakage. Some additional countermeasures can be using resource server supplied 'nonce' which can not be guessed by the attacker. Also to prevent the usage of same JWT for retrieving different resources rather than its intended resource we can use OIDC because it uses "jwks_uri", restricting 'aud' claim to be pre-registered by the client and adding 'iss' claim. All of them ensures that the same issuer issued the JWT for distinct resources.

### 5.2.3 Usage of JWT as an authentication request

JWT can also be used as an authentication request in OIDC as per table 5.1 where it needs to maintain the confidentiality of the request. Here the relying party sends the request to the OpenID provider (OP). So, client's authenticity and non-repudiation needs to be ensured here. Disclosure of this request can reveal the sensitive information about client such as client id and client secrete.

We can discuss some of the methods to ensure the security objectives mentioned above to make this authentication request secure. First of all, it can be signed and optionally encrypt the authentication request with JWT to ensure the confidentiality of the request. But it might not be ensure the safety always because a request_uri has been used to transfer this request which can be vulnerable with SSRF.

One countermeasure can be pre-register this request_uri during client registration so that any malicious client can not use any arbitrary request_uri to impersonate itself as a benign client. If not possible then another solution can be to use DPoP proof JWT which can ensure the client authenticity, non-repudiation and integrity of the request at the same time.

### 5.2.4 Usage of JWT as a way of information transmission

From the table 5.1 we can see that JWT can be used to convey the information of user's identity in the form of SD-JWT.

If we follow the flow of SD-JWT from the figure 4.25 then we can see that in this flow the issuer issues SD-JWT which contains some claims to represents a user's identity and sends it to the holder and the holder sends it to the verifiers whenever the user wants to disclose some of his verifiable information to a specific verifier.

Here in this overall flow, it is very important to maintain the integrity of those claims inside SD-JWT during its transmission from issuer to holder and from holder to the verifier. The most important part of this flow is to transfer the SD-JWT from holder to verifier because the verifier needs to check the integrity of this

SD-JWT to ensure that the confidentiality of those claims inside it are maintained and authenticity of the issuer and the holder also needs to be ensured.

To maintain all those above mentioned security objectives the verifier must need to verify the signature of SD-JWT to check the authenticity of the issuer that the signing key belongs to this issuer. Besides proving the authenticity of the issuer it also ensures the integrity of the SD-JWT since its issuance by the issuer. To check the validity of SD-JWT nbf, iat, and exp claims and by checking that the same key has been used to sign both SD-JWT-R and SD-JWT. By checking a nonce and aud field inside SD-JWT-R the verifier can ensure that this particular SD-JWT-R has been created for this particular verifier.

Now we can discuss some of the security aspects to focus on how above mentioned security checking can make the flow of SD-JWT secure. First of all, It ensures the integrity of the selectively disclosable claims in a way that a malicious holder can not modify any values from those claims because during the issuance of the SD-JWT to the Holder, the issuer includes a duplication of all hidden claims in cleartext besides issuer-signed JWT so that the verifier can verify that all disclosed claim values were part of the original issuer-signed JWT. In this way the verifier can detect if any modification happens by the malicious holder. Moreover, even if a verifier can turned into a malicious attacker, it can not obtain any claim value from an SD-JWT that was not revealed by the holder because the holder is in full control to decide which claims he wants to disclose or keep secrete. In addition, checking holders legitimacy by the verifier is also an important security concern which can be done by binding an SD-JWT to a Holder's public key. Although, This feature is optional but we think it can provide an extra security layer to prove the authenticity of the holder and the integrity of the claims as well during transmission the SD-JWT from holder to verifier if it can be mandatory.

## 5.3 Attacks on different data flows

In previous section we explained the different usage of JWT in different data flows and some issues which can be the reason against fulfilling the security objectives related with those usage of JWT. In this section we will focus on particular attacks which hampers the security objectives directly or indirectly provided by the JWT in different data flows.

Now if we look at the table no 5.5 5.3 and 5.4 then we can see the different attacks on different data flows including the disruption of security objectives occurred by them. Now we already explained in details about these attacks in section 4.3, 4.4 and 4.7. Now it is time just to explain how we categorized them in terms of hampering those objectives.

### 5.3.1 Attacks on Oauth

By following table 5.5 we can come up with the following explanations

Using arbitrary 'client id' as a 'sub' value by the AS during token issuance can be the reason of **client impersonation attack** because after issuing the token any malicious client can select the 'sub' of a high-privilege resource owner to confuse RS if it depends on the 'sub' value during token introspection. Client authenticity can not be ensured in this case since AS can take it as a benign client but ultimately it is a malicious client. In this way if it uses the token to achieve high privilege user's resources from RS then the confidentiality of data can not be guaranteed as well. The integrity of the token can also be questionable. Non-repudiation can also be hampered since the AS can not ensure the authenticity of the client thus, client can deny its actions later on.

| Name of the attacks | Authenticity | Confidentiality | Integrity | Non-repudiation |
|---|---|---|---|---|
| Client impersonation | ✓ | ✓ | ✓ | ✓ |
| Cross-JWT confusion | | ✓ | | |
| Access token leakage | | ✓ | ✓ | |
| Token Substitution | | ✓ | ✓ | |

**Table 5.2:** Attacks on Oauth which can be the reason to disrupt the security objectives provided by the JWT

**Cross-JWT confusion** happens if the same JWT has been used to obtain different resources rather than the intended resources it issued for, which causes the leakage of confidential data.

If **access token leakage** and **token substitution** happens then the integrity of the token is interrupted. And confidential data leakage can take place if the same token used to obtain sensitive or personal information from RS.

### 5.3.2 Attacks on DPoP-JWTs

The brief explanation of table 5.3 are as below.

**Authorization Code Rebinding Attack** is possible if the attacker somehow manage to get the authorization code and PKCE parameters. Now the complication of this kinds of attack can be vast. Because if the attacker obtain authorization code then he can issue an access token using this code from authorization server's token endpoint. The accessed token by the attacker can also be used to get sensitive or private information of the user from RS. The above mentioned actions by the attacker can make the whole DPoP flow vulnerable since it can make the authenticity of the client, integrity of the token and confidentiality of the private data questionable. Non-repudiation can not be maintained as well.

| Name of the attacks | Authenticity | Confidentiality | Integrity | Non-repudiation |
|---|---|---|---|---|
| Authorization Code Rebinding Attack | ✓ | ✓ | ✓ | ✓ |
| DPoP Proof Pre-computation attack | | ✓ | ✓ | |
| DPoP Proof Replay | ✓ | ✓ | ✓ | |

**Table 5.3:** Attacks on DPoP-JWTs which can be the reason to disrupt the security objectives provided by the JWT

If a legitimate user becomes a malicious user then **DPoP Proof Pre-computation attack** can take place. In this case the pre-computed DPoP proof with exfiltrated token are used to gain the access of resources from the RS by the attacker. If we follow the pattern of this attack then we can say it can hampers the integrity of the DPoP proof JWT and the access token which can be the reason of confidential data leakage as well from RS.

Due to lack of proper client authentication by the AS the **DPoP Proof Replay attack** can take place where the attacker can get hold of DPoP proof JWT. If this is the case then failure of client authentication, leakage of confidential data and

manipulation of DPoP proof JWT can be occurred.

### 5.3.3 Attacks on OIDC

Attacks on OIDC is showcased in table 5.4 including there security disruptions.

When client uses authorization request to the AS, it uses request URL which includes request_uri that contains a JWT. If the AS allows the client to use an arbitrary request_uri without preregistering and whitelisting it then **SSRF via request_uri parameter** can take place by the attacker. Authenticity of the client and integrity of the JWT used in this request can not be guaranteed if AS allows any arbitrary 'request_uri' to be used by the client. It also carries some sensitive and confidential information about the client and the user which can be leaked as well. Due to improper client authentication non-repudiation can be impeded.

| Name of the attacks | Authenticity | Confidentiality | Integrity | Non-repudiation |
|---|---|---|---|---|
| SSRF via request_uri | ✓ | ✓ | ✓ | ✓ |
| SSRF during client registration | ✓ | ✓ | ✓ | ✓ |
| Disclosure of server response | ✓ | | | ✓ |

**Table 5.4:** Attacks on OIDC which can be the reason to disrupt the security objectives provided by the JWT

Same scenario goes with the **SSRF during dynamic client registration** as well because it can happen by sending a malicious "jwks_uri" during new client registration by the attacker which can allow to obtain an authorization code for any user. By obtaining the authorization code can cause the obstruction of ensuring four of the security objectives that we explained before.

On the other hand, **disclosure of server response** can happen due to improper client authentication. In terms of security objectives, it disrupts authenticity of the client and non-repudiation.

## 5.4 Attacks that makes the different steps in different data flows vulnerable

We already explained different uses of JWT in different data flows and their security aspects in section 5.2 and explained different attacks separately for each data flows including their impact on particular security objectives in section 5.3. In this section we will try to focus on which direction of the communication in between client and server can be affected by those attacks.

From the table no 5.5 we can go through separately for each of the attack to have an overall overview.

If we focus on oauth first then the **client impersonation attack** can take place during communication from client to RS. Although the malicious client can only impersonate itself if the authorization server uses arbitrary client id as 'sub' value for token issuance in previous step in oauth flow figure 4.3 which open the door for the attacker to change the 'sub' value and gain the unauthorized access from the next step. So the main attack attempts can be done by the attacker during communication from client to RS, although it depends on the action of AS in previous step where the direction of the flow is from AS to client. In **Cross-JWT confusion** the attack flow is from client to RS when the attacker uses the JWT for different purposes rather than its intended uses. The **Access token leakage** and the **token substitution attack** can happen after token issuance by the AS. So the attack can take place during data flow from AS to client.

If we see the attacks on DPoP-JWTs then we can notice that the **authorization code rebinding attack** can take place if the attacker obtain the code during data transmission from AS to client when the AS issues the authorization code and sends it to the client. **DPoP proof pre-computation attack** happens when the legitimate user becomes a malicious user and pre-compute DPoP proof for future use with exfiltrated token to get the resources from RS. So the attack scenario can take place through transferring that pre-computed proof with the token from client to the RS. Although the **DPoP proof replay** can take place when the attacker can obtain the DPoP proof when the AS issues the proof and sends it to the client, the main impact of the attack can be noticed when the attacker uses the same proof with the token to achieve unauthorized access from RS. So the attacker uses the data flow from client to the RS to fulfill his malicious purposes.

In OIDC both **SSRF via request_uri** and **SSRF during client registration** can take place if the client becomes a malicious attacker. In the first case, the attacker sends arbitrary request_uri and in the next one malicious jwks_uri from client to the AS. The **disclosure of server response** can take place whenever the AS issues or response in returns of clients request.

Now If we look carefully then we can notice that most of the attacks we found for Oauth, DPoP-JWTs and OIDC happens during communication from authoriza-

tion server to client almost for 5 of the attacks out of 10. So, the precaution needs to be higher as well in this part of the flow because the authorization code and token is sent to the client from AS through this flow. The purpose of the whole flow can be broken down if the authorization code or token leakage takes place which can create a pathway for the attackers to generate some other types of attacks.

Another Observation can be if we look the attacks on normal oauth flow then we can find that most of the attacks can be resolved by adding some security extensions such as DPoP-JWTS, mTLS or protocols like OIDC with it.

Such as **client impersonation attack** can be prevented by using OIDC to restrict the client from using arbitrary client id during client registration. Same goes for **Cross-JWT confusion** where OIDC uses "jwks_uri to confirm that cryptographic keys used in the JWT are belongs to the issuer and it also ensure that a valid issuer issued the JWT for particular purposes. Restricted 'aud' claim in OIDC can also be used to find the intended audience or recipient of the particular JWT. It also helps to prevent the **token substitution attack**. **Access token leakage** issue can be solved by using DPoP-JWTs and mTLS since both of them provide sender-constrained access tokens.

On the other hand, the attacks on DPoP-JWTs can be resolved by having some extra precaution. Such as adding dpop_jkt Parameter in the authorization code request to prevent **authorization code rebinding attack** so that the AS can compares JWK Thumbprint of the proof-of-possession public key in the DPoP proof with the dpop_jkt parameter value in the authorization request. In addition, server provided 'nonce' can be used to prevent **DPoP-proof pre-computation attack** and **DPoP proof replay**.

Same goes with attacks on OIDC where some precaution needs to be taken during client registration so that the attacker can not use any arbitrary request_ur or malicious "jwks_uri" to initiate the attack flow.

## 5.5 The impact of using TLS on the mentioned attacks

On the basis of Oauth 2.0 specification the authorization server must use TLS in the oauth data flow [22]. Now it is important to focus or observe that how many attacks mentioned in table 5.6 can be prevented by just applying TLS. The details of TLS can be found in appendix B.

Btw, if we analyse the attacks mentioned in table 5.6 one by one then we can see the **client impersonation attack, Cross-JWT confusion, Token substitution, DPoP Proof Pre-computation attack, SSRF via request_uri and SSRF during client registration** can not be prevented just only by applying TLS. Because if we analyse the attack patterns for these attacks then we can see that they are not being initialized during the data flow takes place over TLS. The possibility of **client impersonation attack** being arose when the authorization server (AS) uses the arbitrary client id as

'sub' value during the token issuance. And the attacks being initiated if the client turned into a malicious client and changes this 'sub' value into a high-privilege resource owner to confuse resource server which is dependent on 'sub' value during token introspection. So, TLS has no contribution to prevent this kinds of attacks. In **Cross-JWT confusion** the JWT has been used rather than its intended purposes which can be prevented by proper validation of the issuer to ensure that the same issuer issued the JWT for distinct resource.

For **DPoP Proof Pre-computation attack** the legitimate user becomes a malicious user and pre-generate or pre-compute DPoP proof for future uses with exfiltrated token. The **SSRF via request_uri and SSRF during client registration** can be prevented if the authorization server make the client registration procedure much more stricter by confirming request_uri and jwks_uri so that the server can refuse the request if the attacker uses any arbitrary value in this uri.

On the other hand, **Access token leakage, Authorization Code Rebinding Attack, DPoP Proof Replay, Disclosure of server response** can be prevented by applying TLS since the attack pattern involves by stealing token, authorization code, DPop proof and server response where these leakage are possible during their transmission.

## 5.6 Some recommendations for the best practices

From our overall thesis work we can note down that JWT has its many uses in different data flows to ensure or fulfill the different security objectives such as integrity, confidentiality, authenticity and non-repudiation. One thing needs to mention that these security objectives can only be ensured by JWT if it is implemented and validated properly. Besides that using some security extensions such as DPoP-JWTs, mTLS or OIDC authentication protocol can enhance the security provided by JWT much more strongly. We already mentioned some of the countermeasures by using these security extensions after explaining each of the attacks in chapter 4 and chapter 5.

Here in this section we will try to give some recommendations of using these extensions in terms of improving the security of data flows which can ensure the fulfillment of the security objectives strongly and also regarding the widespread uses and adoption of these extensions.

Although JWE can provide the confidentiality and JWS can provide the integrity of the token, for ensuring the non-repudiation and authenticity strongly it might not be enough. On the other hand if somehow the attacker obtain the token then all of the security objectives mentioned above can be disrupted. In this case the security extensions like DPoP-JWTs and mTLS can make the difference by providing sender constrained access token which can also be a replacement of using bearer token since whoever obtain the bearer token can use it easily to get

the access of the resources. Now if we talk about which one is better regarding the current widespread uses and adoption then we can say although both mTLS and DPoP are designed to provide sender constraint token which ensures the legitimacy of the token sender, DPoP-JWTs are relatively new then mTLS. Also mTLS provide strong security guarantees since it provides certificate-based authentication. In terms of easy implementation DPoP-JWTs can be the best option since it does not rely on PKI infrastructure.

On the other hand, although the OAuth flow has made the uses of TLS mandatory, some attacks can still be possible which can be prevented by using DPoP-JWTs as an extra security layer.

Although the attacks which is related with token or authorization code hijacking can be prevented by using DPoP-JWTs, some other types of attacks which is connected with client registration can not be solved just by using DPoP-JWTs. In this case we need to use OIDC with DPoP-JWTs to make the client registration much more stricter. .

| Name of the attacks | OAuth | DPoP-JWTs | OIDC |
|---|---|---|---|
| Client impersonation | client-RS | | |
| Cross-JWT confusion | client-RS | | |
| Access token leakage | AS-client | | |
| Token Substitution | AS-client | | |
| Authorization Code Rebinding Attack | | AS-client | |
| DPoP Proof Precomputation attack | | client-RS | |
| DPoP Proof Replay | | client-RS | |
| SSRF via request_uri | | | client-AS |
| SSRF during client registration | | | client-AS |
| Disclosure of server response | | | AS-client |

**Table 5.5:** Attacks that makes the different steps in different data flows vulnerable.

| Name of the attacks | Can be prevented by applying TLS | Can not be prevented by applying TLS |
|---|---|---|
| Client impersonation | | ✓ |
| Cross-JWT confusion | | ✓ |
| Access token leakage | ✓ | |
| Token Substitution | | ✓ |
| Authorization Code Rebinding Attack | ✓ | |
| DPoP Proof Precomputation attack | | ✓ |
| DPoP Proof Replay | ✓ | |
| SSRF via request_uri | | ✓ |
| SSRF during client registration | | ✓ |
| Disclosure of server response | ✓ | |

**Table 5.6:** The impact of using TLS on the mentioned attacks

# Chapter 6

# Discussion

If we return back to the section 1.4 then we can see at the beginning of the thesis work we have been set some thesis goals. In this chapter we will discuss about how we accomplished the thesis goal step by step in details and try to discuss some future work as well.

To achieve the thesis goals we needed to set a proper methodology first What we did in chapter 2. Also it was necessary to have a broader discussion about the structure of JWT and how the JWT works in general to dive deep into different data flows that uses JWT in their flow for different purposes and understand them properly as well. For this purpose in chapter 3 we explained in details the structure of JWT and how the JWT provides the stan- dardize mechanism for integrity protection via signature and MAC or encryption. After that we started explaining the different uses of JWT in different data flows in chapter 4 including the security objectives fulfilled by JWT on those flows and the related important attacks as well that causes to disrupt those objectives with some of their countermeasures and recommendation for best practices. Now after all those details discussions about different data flows, it is time to summarise our overall findings and give some overviews in chapter 5 which can lead us to fulfill our thesis goal.

Our first goal was to find the similarities and differences in between data flows on the basis of usage of JWT. To accomplish this goal we come up with table no 5.1 where we point out different uses of JWT in different data flows and then explained in section 5.1. Now it was not enough to just point out the usage of JWT in particular purposes in different flows because each usage of JWT is related with fulfilling some security objectives and some attacks can also be possible during usage of JWT on those flows to disrupt those objectives which needs to be explained as well including some countermeasures and recommendation for best practices to accomplish the second, third and the fourth goals from section 1.4. It has been done by discussing table no 5.1, 5.5, 5.3 and 5.4 in details. In addition, since using TLS is mandatory in outh 2.0 specification and the uses of TLS can be a very impor-

tant point to prevent some attacks, it is worth it to have an analysis regarding how many attacks can be solved or prevented by just using TLS discussed in section 5.5

## 6.1 Future work

In this section we will discuss what else we could do if we get more time for more expansion of this thesis work and also how our thesis work can be a good source of information for someone who wants to work on something related.

If we look at our overall thesis work and the goals that we achieved then we can see, our work was only limited on knowing the basic structure of JWT and how it works in general at first then we jumped into the uses of jWT in different data flows including the security objectives fulfilled by JWT on those flows and some important attacks on those flows as well with mentioning some countermeasures and recommendations for best practices.

First of all, most of our work throughout this project is based on theoretical discussions. But in future there is some scopes to add some practical demonstration. Although we can see some of the practical demonstrations of attacks on JWT PHP library in appendix A.1, but not all of the attacks have been showcased there. In future we can try to add some more attacks on JWT PHP library and also try to find attacks on other JWT libraries available as well and co-relate them with our main project work. Some important attachment can be to develop a small web app and deploy oauth with it and try to demonstrate attacks on oauth and prevent some of them by adding DPoP-JWTs and mTLS with it. In addition, it can be a good informative source for others as well if they want to do some related work in future.

# Chapter 7

# Conclusion

At the end we can say JWT is an interesting approach for both authentication, authorization and information exchange. JWT can be very useful if it is implemented properly. Although they look a lot simpler, a simple mistake can create a lots of issues such as account hijacking, data loss, and much more.

From our whole thesis work we can see JWT has its many uses nowadays in different data flows. As in digital realm nothing is hundred percent secure, JWT has its own flaws as well but most of them happens due to improper implementation and validation. Some can be solved by using oauth security extensions such as DPoP-JWTs, mTLS or using OIDC authentication protocol. Some can also be prevented by taking some extra precaution during client registration.

We can say the goals we set at the beginning to complete this thesis work has been accomplished. Although we had scope to elaborate some of them more, in this case we can point out some goals as partially accomplished and some of them are completely accomplished. We explained some different data flows where the JWT has its direct involvement for ensuring secure authentication and authorization. We successfully pointed out the particular uses of JWT for each of those data flows which helps us to find some similarities and differences amongst the data flows in regards using JWT for different purposes. Here we could make some more contributions regarding finding differences and similarities in between data flows on the basis of some other aspects. So we can say in our case the first goal has been accomplished partially. In terms of the second goal which was finding the common types of use of JWTs and which security objectives need to be fulfilled is accomplished completely. As in each data flow the JWT has its own role which works to fulfill some security objectives, it was important to mention exactly how JWT does that and how some attacks on those data flows can disrupt the security objectives provided by the JWT which has been explained in details including how they can be prevented or minimized that leads us to accomplish the third goal successfully. The fourth goal which was about the recommendations for best practices

which is partially accomplished as it could have some more recommendations to be added. At the end we made an overall summary and overview to explain all our findings more decorated way with tables which leads us to explain our whole work and contribution step by step and giving it a final touch. At the end, we can say that JWTs can be very-useful if implemented properly and if some precautions can be taken by particular oauth security extentions when there is no requirement for session management.

# Bibliography


[1] *About the OWASP Foundation*. https://owasp.org/about/. Accessed: November 12, 2023.

[2] Dr.Shanti Bhushan Mishra;Dr.Shashi Alok. *HANDBOOK OF RESEARCH METHODOLOGY*. Available at https://www.researchgate.net/publication/319207471_HANDBOOK_OF_RESEARCH_METHODOLOGY (2023/11/06).

[3] Dan Arias. *Adding Salt to Hashing: A Better Way to Store Passwords*. Available at https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/ (2023/10/18).

[4] Yjvesa Balaj. "Token-Based vs Session-Based Authentication:A survey". In: (September 2017). URL: https://www.researchgate.net/publication/320068250_Token-Based_vs_Session-Based_Authentication_A_survey.

[5] B.Campbell;C.Mortimore;M.Jones. *JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants*. Available at https://datatracker.ietf.org/doc/html/rfc7523 (2023/11/19).

[6] B.Campbell;C.Mortimore;M.Jones. *Security Assertion Markup Language (SAML) 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants*. Available at https://datatracker.ietf.org/doc/html/rfc7522 (2023/11/19).

[7] B.Campbell;J.Bradley;N.Sakimura;T.Lodderstedt. *RFC 8705 OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens*. Available at https://www.rfc-editor.org/rfc/rfc8705#name-jwt-certificate-thumbprint- (2023/10/08).

[8] Marianne Swanson;Joan Hash;Pauline Bowen. *Guide for Developing Security Plans for Federal Information Systems*. Available at https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-18r1.pdf (2023/11/26).

[9] D.Fett;B.Campbell;J. Bradley;T.Lodderstedt;M.Jones;D.Waite. "OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer (DPoP)". In: (29 December 2022). URL: https://www.ietf.org/archive/id/draft-ietf-oauth-dpop-12.html#RFC7519.

[10] Akanksha; Akshay Chaturvedi. "Comparison of Different Authentication Techniques and Steps to Implement Robust JWT Authentication". In: (June 2022). URL: https://ieeexplore-ieee-org.zorac.aub.aau.dk/document/9835796.

[11] *Configure OAuth 2.0 Demonstrating Proof-of-Possession*. https://developer.okta.com/docs/guides/dpop/main/. Accessed: October 01, 2023.

[12] Scott Rose;Oliver;Borchert;Stu Mitchell;Sean Connelly. *Zero Trust Architecture*. Available at https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-207.pdf (2023/10/07).

[13] Anthony Critelli. *Base64 encoding: What sysadmins need to know*. Available at https://www.redhat.com/sysadmin/base64-encoding (2023/11/27).

[14] *Cross-site request forgery (CSRF)*. https://portswigger.net/web-security/csrf. Accessed: November 11, 2023.

[15] *Cross-Site Request Forgery Prevention Cheat Sheet*. https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html. Accessed: November 11, 2023.

[16] *CVE-2018-1000531 Detail*. https://nvd.nist.gov/vuln/detail/CVE-2018-1000531. Accessed: October 22, 2023.

[17] Roman Danyliw. *Javascript Object Signing and Encryption*. Available at https://datatracker.ietf.org/doc/charter-ietf-jose/ (2023/11/11).

[18] D.Cooper;S.Santesson;S.Farrell;S.Boeyen;R.Housley;W.Polk. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. Available at https://datatracker.ietf.org/doc/html/rfc5280 (2023/10/07).

[19] Brian Demers. *A Beginner's Guide to JWTs*. Available at https://developer.okta.com/blog/2020/12/21/beginners-guide-to-jwt (2023/11/14).

[20] D.Fett;D.Tonge. "FAPI 2.0 Security Profile — draft". In: (13 September 2023). URL: https://openid.bitbucket.io/fapi/fapi-2_0-security-profile.html#section-5.3.1.1.

[21] D.Fett;K.Yasuda. *Selective Disclosure JWT (SD-JWT)*. Available at https://datatracker.ietf.org/doc/html/draft-fett-oauth-selective-disclosure-jwt (2023/10/17).

[22] D.Hardt. *The OAuth 2.0 Authorization Framework*. Available at https://datatracker.ietf.org/doc/html/rfc6749 (2023/10/08).

[23] D.Hardt. "The OAuth 2.0 Authorization Framework". In: (October 2012). URL: https://datatracker.ietf.org/doc/html/rfc6749#section-2.1.

[24] *Digest/Hash function*. https://www.ibm.com/docs/en/app-connect-pro/7.5.3?topic=reference-digesthash-function. Accessed: October 18, 2023.

[25] *DIGITAL SIGNATURE STANDARD (DSS)*. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5.pdf. Accessed: November 14, 2023.

[26] *ECMA-404*. https://ecma-international.org/publications-and-standards/standards/ecma-404/. Accessed: November 08, 2023.

[27] Varsharani Hawannaa;V.Y.Kulkarnia;R.A.Ranea;P.Mestrib;S.Panchal. *Risk Rating System of X.509 Certificates*. Available at https://www.researchgate.net/publication/306362676_Risk_Rating_System_of_X509_Certificates (2023/10/07).

[28] Sherry Hsu. "Session vs Token Based Authentication". In: (2018). URL: https://sherryhsu.medium.com/session-vs-token-based-authentication-11a6c5ac45e4.

[29] *HttpOnly*. https://owasp.org/www-community/HttpOnly. Accessed: November 15, 2023.

[30] *Insecure Hash*. https://docs.guardrails.io/docs/vulnerability-classes/insecure-use-of-crypto/insecure-hash. Accessed: October 19, 2023.

[31] *Introducing JSON*. https://www.json.org/json-en.html. Accessed: November 08, 2023.

[32] *Introduction to JSON Web Tokens*. https://jwt.io/introduction. Accessed: November 08, 2023.

[33] *Introduction to JSON Web Tokens*. https://jwt.io/introduction. Accessed: October 22, 2023.

[34] Will Johnson. *RS256 vs HS256: What's The Difference?* Available at https://auth0.com/blog/rs256-vs-hs256-whats-the-difference/ (2023/11/16).

[35] Michael B. Jones;Pieter;Kasselman. "OAuth DPoP (Demonstration of Proof of Possession) is what we're doing about it". In: (). URL: https://self-issued.info/presentations/Identiverse_2022_DPoP.pdf.

[36] S. Josefsson. *The Base16, Base32, and Base64 Data Encodings*. Available at https://datatracker.ietf.org/doc/html/rfc4648 (2023/11/11).

[37] *JSON Defined*. https://www.oracle.com/database/what-is-json/. Accessed: November 08, 2023.

[38] *JSON Web Token Claims*. https://auth0.com/docs/secure/tokens/json-web-tokens/json-web-token-claims. Accessed: November 12, 2023.

[39] *JSON Web Token (JWT)*. https://www.iana.org/assignments/jwt/jwt.xhtml. Accessed: November 12, 2023.

[40] *JWT*. https://jwt.io/. Accessed: October 21, 2023.

[41] *JWT attacks*. https://portswigger.net/web-security/jwt. Accessed: October 01, 2023.

[42] KirstenS. *Cross Site Request Forgery (CSRF)*. Available at https://owasp.org/www-community/attacks/csrf (2023/11/11).

[43] Bruno Krebs. *The OpenID Connect Handbook*. Available at https://auth0.com/resources/ebooks/the-openid-connect-handbook/thankyou (2023/10/11).

[44] Ado Kukic. *The Definitive Guide to Single Sign-On*. Available at https://auth0.com/resources/whitepapers/definitive-guide-to-single-sign-on/thankyou (2023/10/15).

[45] *Man-in-the-Middle (MITM) Attack*. https://snyk.io/learn/man-in-the-middle-attack/. Accessed: November 11, 2023.

[46] *Manipulator-in-the-middle attack*. https://owasp.org/www-community/attacks/Manipulator-in-the-middle_attack. Accessed: November 11, 2023.

[47] Charles P.Pfleeger;Shari Lawrence Pfleeger;Jonathan Margulies. *Security in Computing*. 5. ed. Pearson Education,Inc., 2015.

[48] Emily McKeown. *Single Sign-on vs. Federated Identity Management: The Complete Guide*. Available at https://www.pingidentity.com/en/resources/blog/post/sso-vs-federated-identity-management.html (2023/10/15).

[49] Tim McLean. "Critical vulnerabilities in JSON Web Token libraries". In: (August 21, 2020). URL: https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/.

[50] N.Sakimura;J.Bradley; M.Jones;B.de Medeiros;C.Mortimore. *OpenID Connect Core 1.0 incorporating errata set 1*. Available at https://openid.net/specs/openid-connect-core-1_0.html (2023/10/10).

[51] *Mitigating Cross-site Scripting With HTTP-only Cookies*. https://learn.microsoft.com/en-us/previous-versions//ms533046(v=vs.85)?redirectedfrom=MSDN. Accessed: November 15, 2023.

[52] M.Jones. *JSON Web Algorithms (JWA)*. Available at https://datatracker.ietf.org/doc/html/rfc7518 (2023/11/14).

[53] M.Jones. *JSON Web Key (JWK)*. Available at https://datatracker.ietf.org/doc/html/rfc7517 (2023/11/14).

[54] M.Jones. *JSON Web Key (JWK) draft-jones-json-web-key-03*. Available at https://openid.net/specs/draft-jones-json-web-key-03.html (2023/11/14).

[55] M.Jones;D.Hardt. *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. Available at https://www.rfc-editor.org/rfc/pdfrfc/rfc6750.txt.pdf (2023/10/24).

[56] M.Jones;E.Rescorla;J.Hildebrand. *JSON Web Encryption (JWE) draft-jones-json-web-encryption-02*. Available at https://openid.net/specs/draft-jones-json-web-encryption-02.html (2023/11/13).

[57] M.Jones;J.Bradley;H.Tschofenig. *Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)*. Available at https://www.rfc-editor.org/rfc/pdfrfc/rfc7800.txt.pdf (2023/10/08).

[58] M.Jones;J.Bradley;N.Sakimura. *JSON Web Signature (JWS)*. Available at https://datatracker.ietf.org/doc/html/rfc7515 (2023/11/13).

[59] M.Jones;J.Bradley;N.Sakimura. *JSON Web Token (JWT)*. Available at https://datatracker.ietf.org/doc/html/rfc7519 (2023/10/22).

[60] M.Jones;J.Bradley;N.Sakimura;D.Balfanz;Y.Goland;J.Panzer;P.Tarjan. *JSON Web Signature (JWS) draft-jones-json-web-signature-04*. Available at https://openid.net/specs/draft-jones-json-web-signature-04.html (2023/11/13).

[61] M.Jones;J.Hildebrand. *JSON Web Encryption (JWE)*. Available at https://datatracker.ietf.org/doc/html/rfc7516 (2023/11/13).

[62] M.Jones;N.Sakimura. "JSON Web Key (JWK) Thumbprint". In: (September 2015). URL: https://www.rfc-editor.org/rfc/pdfrfc/rfc7638.txt.pdf.

[63] Ch.Jhansi Rani;SK.Shammi Munnisa. "A Survey on Web Authentication Methods for Web Applications". In: (2016). URL: https://ijcsit.com/docs/Volume%207/vol7issue4/ijcsit2016070406.pdf.

[64] *non-repudiation*. https://csrc.nist.gov/glossary/term/non_repudiation. Accessed: November 26, 2023.

[65] *nowakowskir/php-jwt*. https://github.com/nowakowskir/php-jwt. Accessed: November 01, 2023.

[66] N.Sakimura;J.Bradley;E.Jay. "Financial-grade API Security Profile 1.0 - Part 2: Advanced". In: (March 12, 2021). URL: https://openid.net/specs/openid-financial-api-part-2-1_0.html.

[67] N.Sakimura;J.Bradley;M.Jones. *OpenID Connect Dynamic Client Registration 1.0 incorporating errata set 1*. Available at https://openid.net/specs/openid-connect-registration-1_0.html (2023/10/14).

[68] N.Sakimura;J.Bradley;N.Agarwal. "Proof Key for Code Exchange by OAuth Public Clients". In: (September 2015). URL: https://datatracker.ietf.org/doc/html/rfc7636.

[69] *OpenID Connect Explained*. https://connect2id.com/assets/oidc-explained.pdfl. Accessed: October 11, 2023.

[70] O.Terbu;D.Fett. *SD-JWT-based Verifiable Credentials (SD-JWT VC)*. Available at https://datatracker.ietf.org/doc/html/draft-ietf-oauth-sd-jwt-vc (2023/10/17).

[71] *OWASP Top Ten*. https://owasp.org/www-project-top-ten/. Accessed: November 11, 2023.

[72] Priya Pedamkar. *Types of Research Methodology*. Available at https://www.educba.com/types-of-research-methodology/ (2023/11/06).

[73] Sebastian Peyrott. *JWT Handbook*. Available at https://auth0.com/resources/ebooks/jwt-handbook/thankyou (2023/10/22).

[74] *PKI Fundamentals*. https://pki.treas.gov/pki_funds3.htm. Accessed: October 07, 2023.

[75] Irfan Darmawan; Aditya Pratama Abdul Karim; Alam Rahmatulloh; Rohmat Gunawan; Dita Pramesti. "JSON Web Token Penetration Testing on Cookie Storage with CSRF Techniques". In: (October 2021). URL: https://ieeexplore-ieee-org.zorac.aub.aau.dk/document/9701965.

[76] *Public client and confidential client applications*. https://learn.microsoft.com/en-us/azure/active-directory/develop/msal-client-applications. Accessed: October 03, 2023.

[77] Raja Rao. "JSON Web Tokens (JWT) are Dangerous for User Sessions—Here's a Solution". https://redis.com/blog/json-web-tokens-jwt-are-dangerous-for-user-sessions/ (accessed September 30, 2023).

[78] *RFCs*. https://www.ietf.org/standards/rfcs/. Accessed: November 06, 2023.

[79] Luca Compagna;Hugo Jonker; Johannes Krochewski;Benjamin Krumnow; Merve Sahin. *A preliminary study on the adoption and effectiveness of SameSite cookies as a CSRF defence*. Available at https://ieeexplore.ieee.org/document/9583694 (2023/10/07).

[80] *Security policy and objectives*. https://www.ibm.com/docs/en/i/7.1?topic=security-policy-objectives. Accessed: November 09, 2023.

[81] *Server-Side Request Forgery (SSRF)*. https://owasp.org/www-community/attacks/Server_Side_Request_Forgery. Accessed: December 21, 2023.

[82] *Server-Side Request Forgery (SSRF)*. https://portswigger.net/web-security/ssrf. Accessed: December 21, 2023.

[83] *Session hijacking attack*. https://owasp.org/www-community/attacks/Session_hijacking_attack. Accessed: November 11, 2023.

[84] *Session Hijacking Attack: Definition,Damage,Defense*. https://www.okta.com/identity-101/session-hijacking/. Accessed: November 11, 2023.

[85] *Session vs Token Authentication*. https://www.authgear.com/post/session-vs-token-authentication. Accessed: September 30, 2023.

[86] *Single Sign-On (SSO)*. https://www.pingidentity.com/en/resources/identity-fundamentals/authentication/single-sign-on.html. Accessed: October 15, 2023.

[87] *Single Sign-On (SSO)*. https://www.ibm.com/topics/single-sign-on. Accessed: October 15, 2023.

[88] S.Kelly;S.Frankel. *Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec*. Available at https://datatracker.ietf.org/doc/html/rfc4868 (2023/11/11).

[89] S.Legg. *Lightweight Directory Access Protocol (LDAP):Syntaxes and Matching Rules*. Available at https://datatracker.ietf.org/doc/html/rfc4517 (2023/10/09).

[90] Michael Stepankin. *Hidden OAuth attack vectorse*. Available at https://portswigger.net/research/hidden-oauth-attack-vectors (2023/10/16).

[91] T.Bray;ED.Textuality. *The JavaScript Object Notation (JSON) Data Interchange Format*. Available at https://datatracker.ietf.org/doc/html/rfc8259 (2023/11/08).

[92] T.Dierks;E.Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. Available at https://datatracker.ietf.org/doc/html/rfc5246 (2023/10/07).

[93] *Testing for Weak Encryption*. https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/09-Testing_for_Weak_Cryptography/04-Testing_for_Weak_Encryption.html. Accessed: October 19, 2023.

[94] T.Lodderstedt;J.Bradley;A.Labunets;D.Fett. *OAuth 2.0 Security Best Current Practice*. Available at https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics-18 (2023/11/18).

[95] T.Lodderstedt;J.Bradley;A.Labunets;D.Fett. "OAuth 2.0 Security Best Current Practice". In: (13 March 2023). URL: https://datatracker.ietf.org/doc/html/draft-ietf-oauth-security-topics-22.

[96] T.Lodderstedt;M.McGloin;P.Hunt. *OAuth 2.0 Threat Model and Security Considerations*. Available at https://datatracker.ietf.org/doc/html/rfc6819 (2023/11/18).

[97] Jennifer Cawthra;Michael Ekstrom;Lauren Lusty;Julian Sexton;John Sweetnam;Anne Townsend. *Data Integrity: Detecting and Responding to Ransomware and Other Destructive Events*. Available at https://www.nccoe.nist.gov/publication/1800-26/VolA/index.html (2023/11/09).

[98] T.Pornin. *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*. Available at https://datatracker.ietf.org/doc/html/rfc6979 (2023/11/16).

[99] *User impersonation for Connect apps*. https://developer.atlassian.com/cloud/jira/software/user-impersonation-for-connect-apps/. Accessed: December 20, 2023.

[100] V.Bertocci. *JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens*. Available at https://datatracker.ietf.org/doc/html/rfc9068 (2023/11/16).

[101] *Verifiable Credentials Data Model v1.1*. https://www.w3.org/TR/vc-data-model/. Accessed: October 18, 2023.

[102] *What is mutual TLS (mTLS)?* https://www.cloudflare.com/learning/access-management/what-is-mutual-tls/. Accessed: October 08, 2023.

[103] *Which OAuth 2.0 grant should I implement?* https://oauth2.thephpleague.com/authorization-server/which-grant/. Accessed: December 23, 2023.

[104] *White Paper OpenID Connect 101*. https://www.pingidentity.com/en/resources/content-library/white-papers/3127-openid-connect-101.html. Accessed: October 10, 2023.

[105] *X.509 CERTIFICATES*. https://www.brainkart.com/article/X-509-Certificates_8470/. Accessed: October 08, 2023.

[106] M.Jones;D.Balfanz;J.Bradley; Y.Goland;J.Panzer;N.Sakimura;P.Tarjan. *SON Web Token (JWT) draft-jones-json-web-token-07*. Available at https://openid.net/specs/draft-jones-json-web-token-07.html (2023/11/08).

[107] Y.Sheffer;D.Hardt;M.Jones. *JSON Web Token Best Current Practices*. Available at https://datatracker.ietf.org/doc/html/rfc8725.html (2023/11/03).

# Appendix A

# Appendixes

## A.1 Some practical demonstration on JWT vulnerability

Table A.1 shows the tools, libraries, and services along with the two columns **Description** and **Purpose** that provide a brief explanation of the tools as well as the goal of using them.

### A.1.1 Brute forcing on weak secrete keys

If a weak secrete key has been used for signing JWT, it can be revealed by using hashcat tool. If we use HS256 (HMAC + SHA-256) as a signing algorithm then it is possible to use an arbitrary string as the secret key. If it is the case then the attacker can easily guess or brute-force the key and generate JWTs with any header and payload. [41] [107] Below is the step by step procedure to reveal weak keys from attackers perspective–

Step 1: Sending request with postman to the login endpoint. figure A.1

Step 2: : Intercepting response with burp suit. Point to be noted, the interceptor needs to be turned on before submitting login file. figure A.2

Step 3: : Copying jwt from the burp suite and use hashcat command with it to reveal weak secrete key. The hashcat command was hashcat -a 0 -m 16500 <jwt>

| Name | Description | Purpose |
|---|---|---|
| PHP-JWT library | Encode and decode JSON Web Tokens (JWT) in PHP | To find the vulnerability. |
| MySQL | open-source relational database management system | Storing user login information. |
| Composer | dependency manager for managing PHP libraries | Download PHP-JWT library. |
| Postman | API platform for building and using APIs | Interecting with API endpoints. |
| Burp Suite | A software used for penetration testing | Intercepting traffic. |
| Hashcat | Password recovery tool | Brute force attack. |

**Table A.1:** Tools, libraries, and services used for fulfilling the attack demonstration.

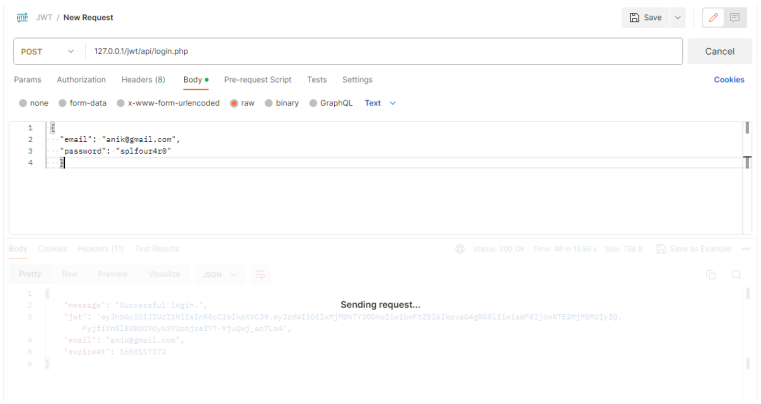


**Figure A.1:** Sending request with postman to the login endpoint.

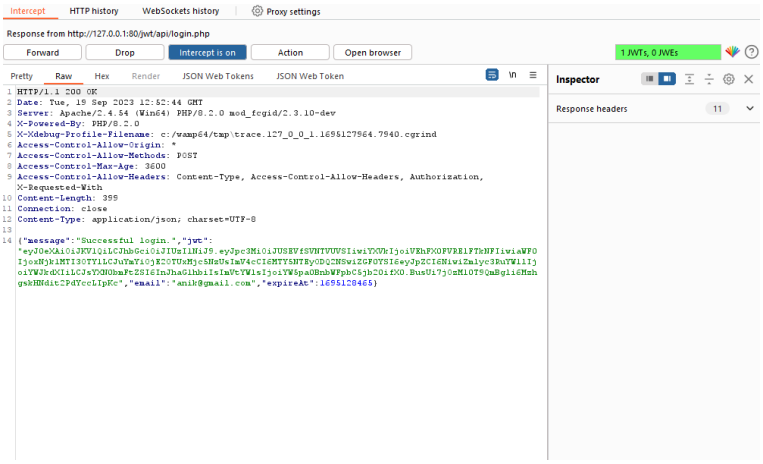


**Figure A.2:** Intercepting response with burp suit.

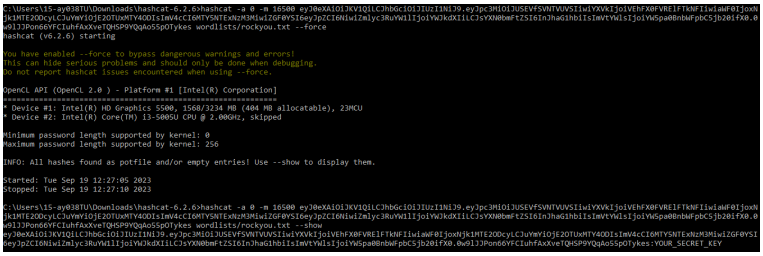


**Figure A.3:** Using hashcat command.

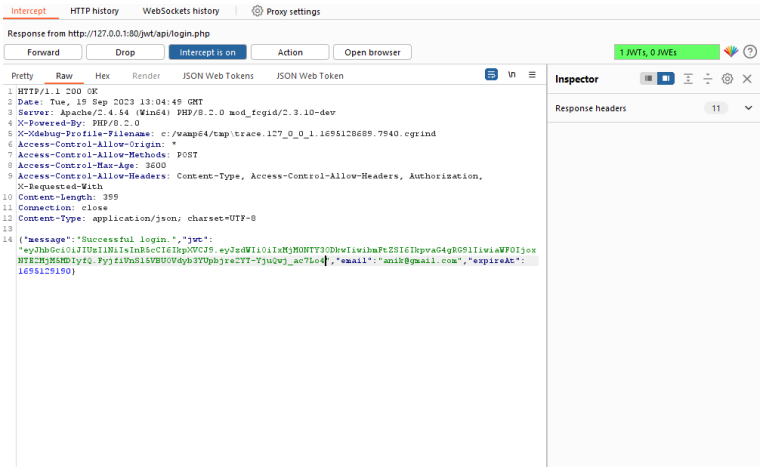


**Figure A.4:** Sending request again replacing new jwt with the old one.

<wordlists/rockyou.txt> –force. 16500 used here for JWT. -a for attack mode and -m for hash type, 0 for MD5 hash. figure A.3

Step 4: create new jwt using revealed secrete key via jwt.io website. [40] figure 1.1

Step 5: Copy paste new jwt replacing with the captured one in burp suit and send the request again. figure A.4

Step 6: Observing the login success message from postman. figure A.5

### A.1.2 "None" Algorithm vulnerability

Most of the JWT libraries should support one special algorithm known as "none" algorithm. Normally it is being used where the integrity of the token has already been verified. But some implementations may accept our JWT as correctly signed, If we use none algorithm in the header and leave the signature [49] [16]. Below we will see the practical example with explanation–

Step 1: Turn on the burp interceptor and sends the request with postman to the API endpoint for user login. figure A.6

Step 2: Intercepting request with burp suite. figure A.7

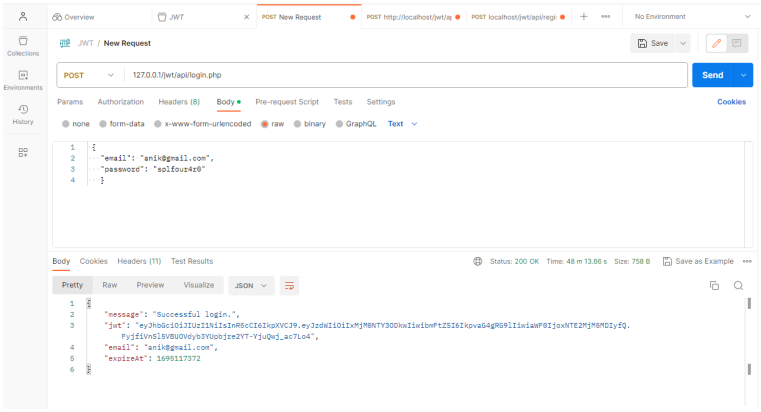


**Figure A.5:** Login successful.

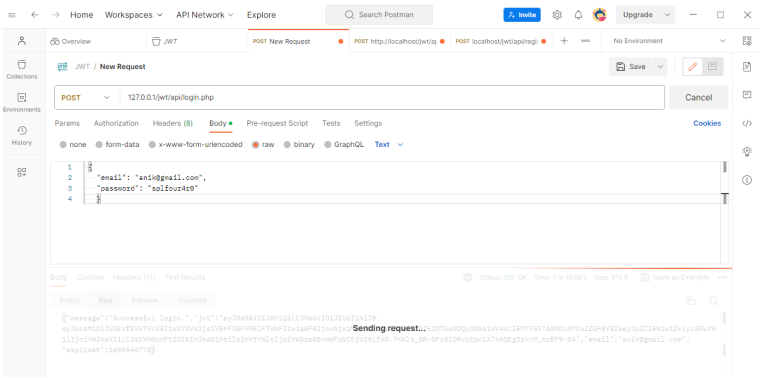


**Figure A.6:** Sending login request using postman.

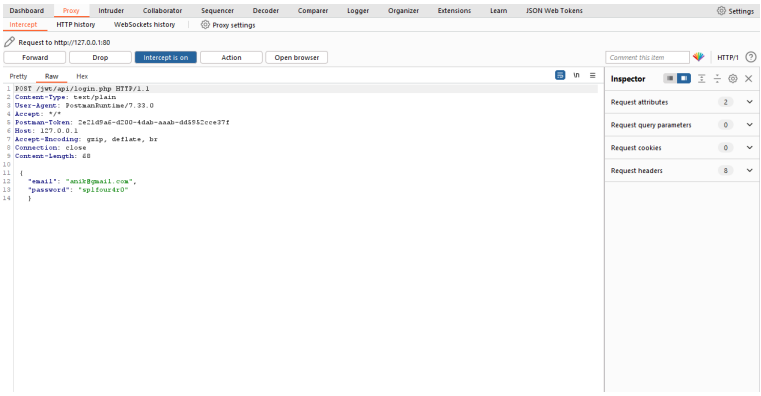


**Figure A.7:** Intercepting request with burp suite.

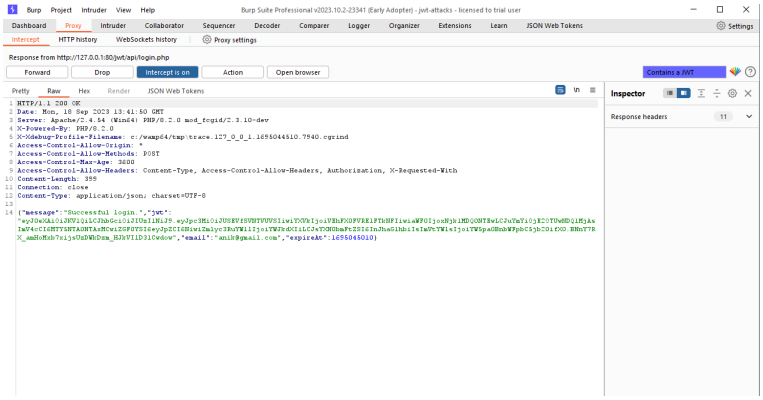

**Figure A.8:** Capturing request.

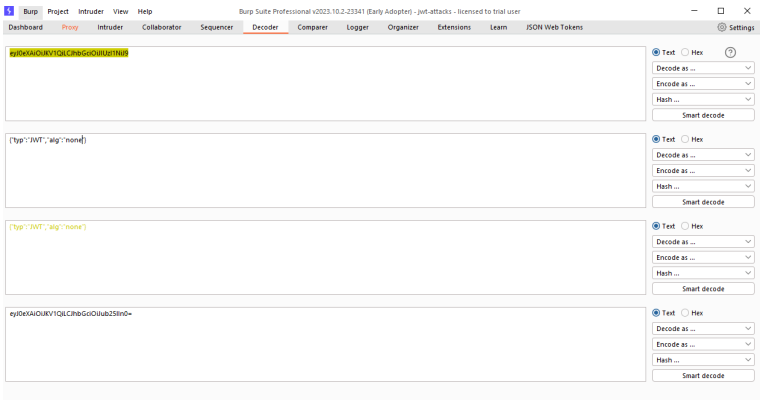

**Figure A.9:** Editing token header.

Step 3: Go to the action button and choose “do intercept”-> then choose response to the request and it will capture the request. figure A.8

Step 4: Send the first part of json web token that means the header to the encoder section and decode it using base64 option and change the HS256 algorithm into none algorithm and encode it again using base64 option and copy and paste the result inside the captured token. figure A.9

Step 5: similar way copy the payload part and decode it. Edit the decoded payload where there is an extra third bracket at the end. Also delete the signature then encode it and replace it with captured jwt payload and send it again using burp suit. figure A.10

Step 6: we can see that we can successfully login with our changed jwt in the login endpoint through postman. figure A.11

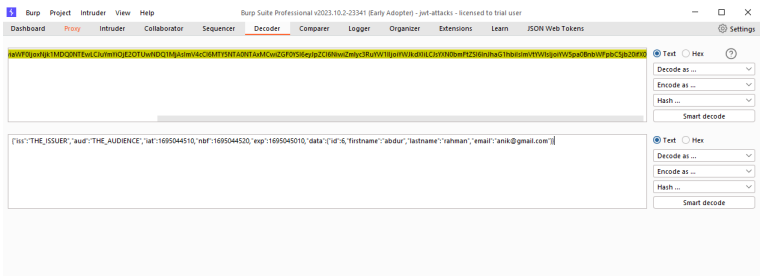


**Figure A.10:** Editing token payload.

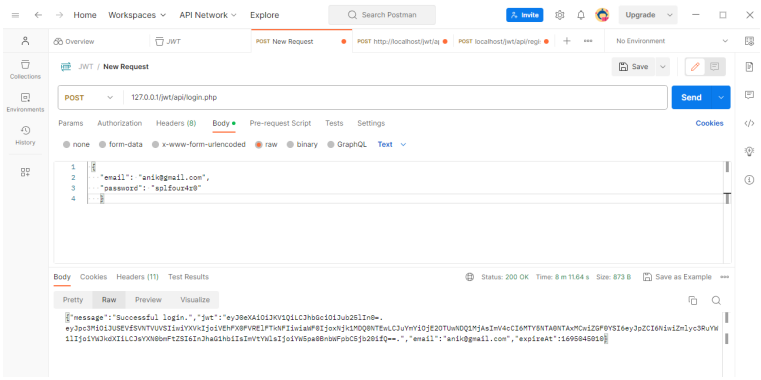


**Figure A.11:** Login Successful.

# Appendix B

# Some Important technical terms

In this chapter we will discuss about some important technical terms which will help us for a better understanding of different client-server data flows explained in chapter 4

## B.1 Confidential and Public Application

According to the OAuth 2.0 specification, applications can be two types which are either confidential or public. [23]

**Confidential applications** can hold clients credentials (such as a client ID and secret) securely without exposing them to unauthorized parties during authentication procedure with the authorization server. example of confidential applications are web apps, web API apps, or service/daemon apps.

On the other hand, **Public applications** cannot hold credentials securely. example of this kinds of apps are that runs on devices, desktop computers or in a web browser. [76]

## B.2 TLS

Transport layer security protocol or TLS protocol provide a safe channel between two communicating parties over internet which ensures privacy and data integrity among those parties.

First of all, TLS Handshake Protocol is a tls sub-protocol works on top of the TLS record layer.

Basic TLS handshake procedures are as below:

1. During first time communication to generate shared secrete both client and server needs to agreed upon protocol version, cryptographic algorithms, public-key encryption techniques. In the first step, client sends client hello includ-

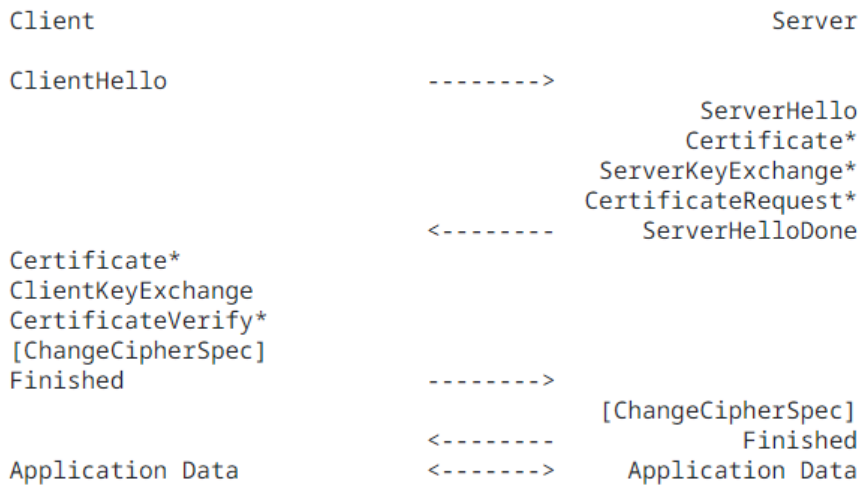


**Figure B.1:** TLS full handshake procedure. [92]

ing above mentioned parameters. (such as- protocol version,session id,cipher suit).

2. In return server responds with server hello including server certificate, server key exchange message which includes key exchange procedure, certificate request message from client and finishes with server hello done.

3. In return, client sends his certificate, client key exchange message with type of the key exchange, verify server's certificate and finishes with change cipher spec message which copies the pending CipherSpec into the current CipherSpec that means to notify receiver that records will be preserved or protected under newly negotiated CipherSpec and keys. Point to be noted that it is not mandatory for client to present its certificate in typical TLS procedure. [102].

4. After getting those information from client server responds with the change cipher spec message and finished.

   The client and the server now ready to exchange application-layer data in between them. [92]

Now to understand how mTLS works properly which is explained in section 4.6, it is important to know about the basic flow of TLS first. Because mTLS is just an extension of TLS where it provides an extra security layer over TLS. In TLS only the server needs to presents the certificate but in mTLS the Client also presents a client certificate to establish a mutual trust in between them.

## B.3 X.509 Certificate and Public key infrastructure (PKI)

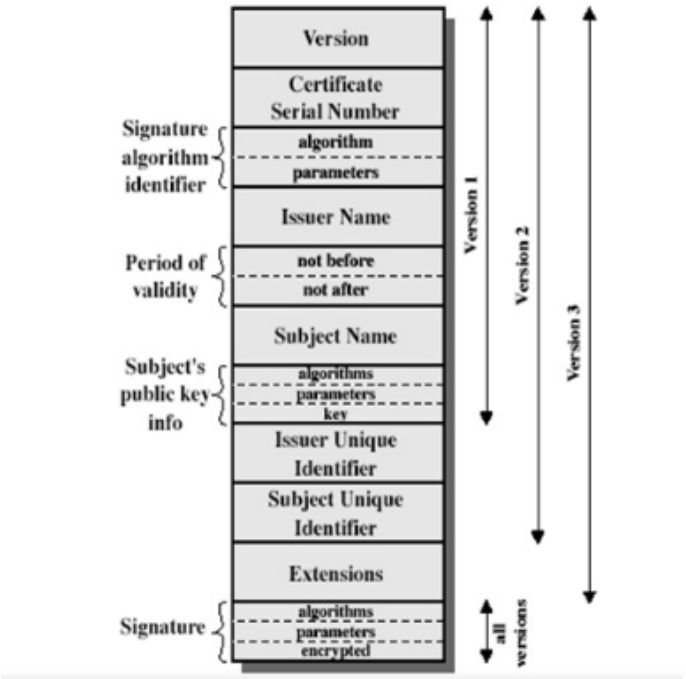


**Figure B.2:** X.509 certificate. [27]

In digital realm, Public key infrastructure (PKI) can be used to protect and authenticate digital communications. PKI is one of the logical components of Zero Trust architecture where trust is never granted implicitly and must be continually evaluated. [12]. A PKI certificate is a trusted digital identity which is comparable with a physical identity card or a passport. Most importantly it is digitally signed and delivered securely by a trusted third party called certificate authority (CA). X.509 Certificate is a digital certificate which defines the format of public key infrastructure (PKI) certificates. In another way, X.509 certificate is the standard for the most commonly used digital certificate formats.

The basic structure of an X.509 v3 digital certificate is shown in figure B.2.

Basic components of Public-Key Infrastructure using X.509 (PKIX) specifications are as below–

1. end entity: Subject of a certificate. It can be user or end user system.
2. CA: certification authority. It signs the digital certificate with their own private key and then publishes the public key that can be accessed upon request. figure B.3

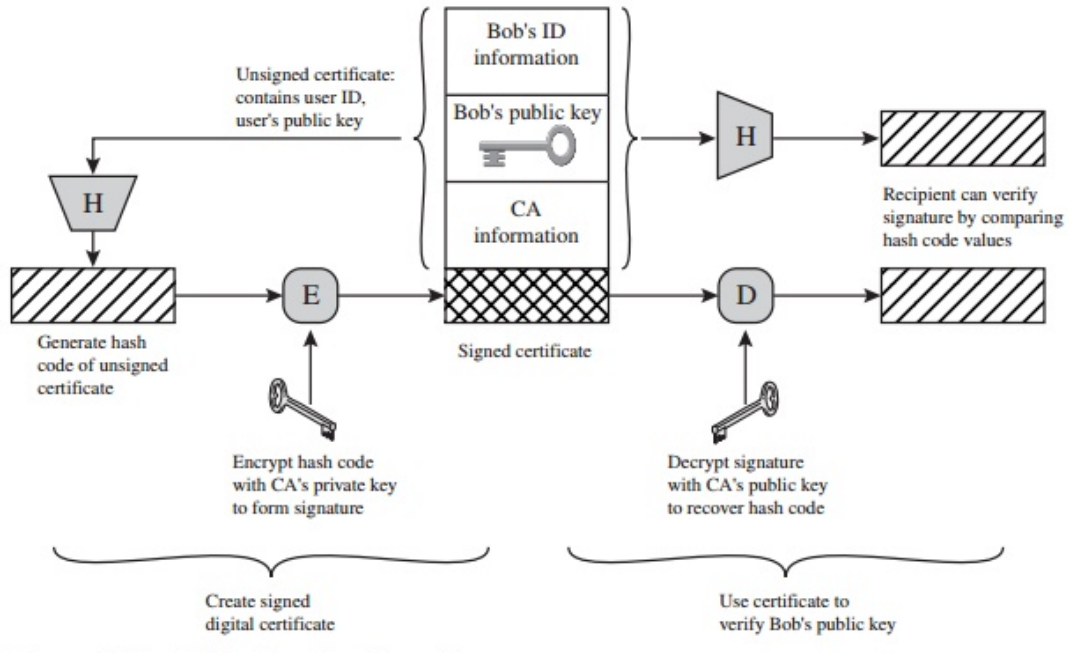


**Figure B.3:** Creation of digital signature. [105]

3. RA: registration authority that verifies user requests for a digital certificate and tells the certificate authority (CA) to issue it. it works as an intermediary in between CA and user which helps CA to receive user or device certificate requests, validate users or devices, authenticate users or devices, revoke credentials if the certificate is no longer valid.

4. CRL issuer: a system that generates and signs a certificate revocation list (CRL).

5. repository: Collection of certificates and CRLs. [18] [74].

The description of how the certificate can be used for mTLS is in section 4.6

## B.4 Proof Key for Code Exchange (PKCE)

The Proof Key for Code Exchange (PKCE) is an extension which is used to improve the security for public clients. It ensures that the same application were involved in the whole authentication flow. The Basic PKCE flow are as below–

(A.) it generates a "code_verifier" which is a high-entropy cryptographic random string and transformed it into "t(code_verifier)" known as "code_challenge" and sends it with the transformation method "t_m" to the authorization endpoint.

(B.) Authorization endpoint records the challenge and the transformation method and sends the authorization code to the client.

(C.) Client sends access token request to the token endpoint including authorization code and the "code_verifier".

(D.) Authorization server transforms "code_verifier" and compares it to "t(code_verifier)" from (B). Access is granted if both of them matched with each other. [68] figure B.4

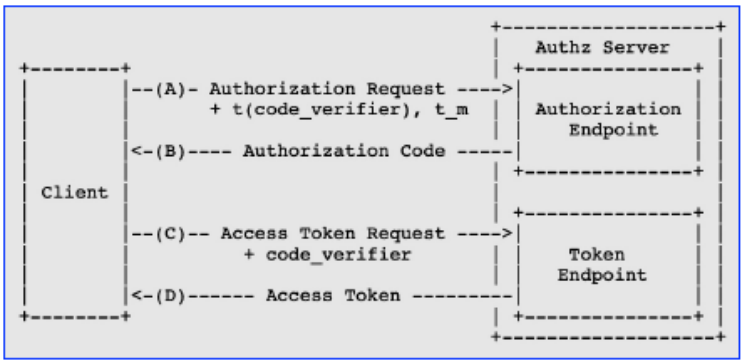


**Figure B.4:** Proof Key for Code Exchange (PKCE) flow. [68]

## B.5 Single-Sign On (SSO)

Single sign-on (SSO) is an authentication procedure which enables users to sign in by using a single set of login credentials, and gain secure access to multiple related applications and services. For example, if an user log into Gmail the user is automatically authenticated to other Google apps such as YouTube, Google Analytics etc. In a similar way, Logging out from Gmail automatically logged out the user from all the apps. figure B.5 [44] [86]

### B.5.1 Federated Identity Management (FIM)

Now the above mentioned scenario is possible due to the presence of Federated identity management or identity federation or federated SSO which creates a trusted relationship between third parties and allows different apps to share identities and authenticate users across domains. Now in traditional non SSO app it does not allow browsers from sharing cookies between domains that means domain2 would have no way of knowing or accessing data from domain1 which push the user to authenticate himself every time in different domains. on the other hand, in a centralized authorization system it allows many apps to talk directly to the authorization server to check if the user is authenticated and grant them access. FIM can

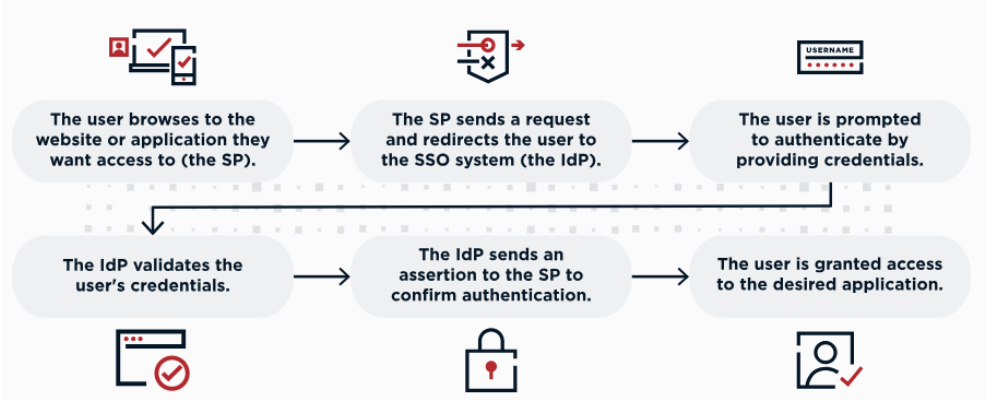


**Figure B.5:** SSO with OIDC. [86]

be achieved via using the standard protocols like SAML, OAuth, OpenID Connect which enables the secure transmission of authentication and access information across domains. Now the main difference in between SSO and FIM is SSO used for enabling access to applications and resources within a single domain whereas FIM enables single-sign on to applications across multiple domains. figure B.6 [44] [86] [48]

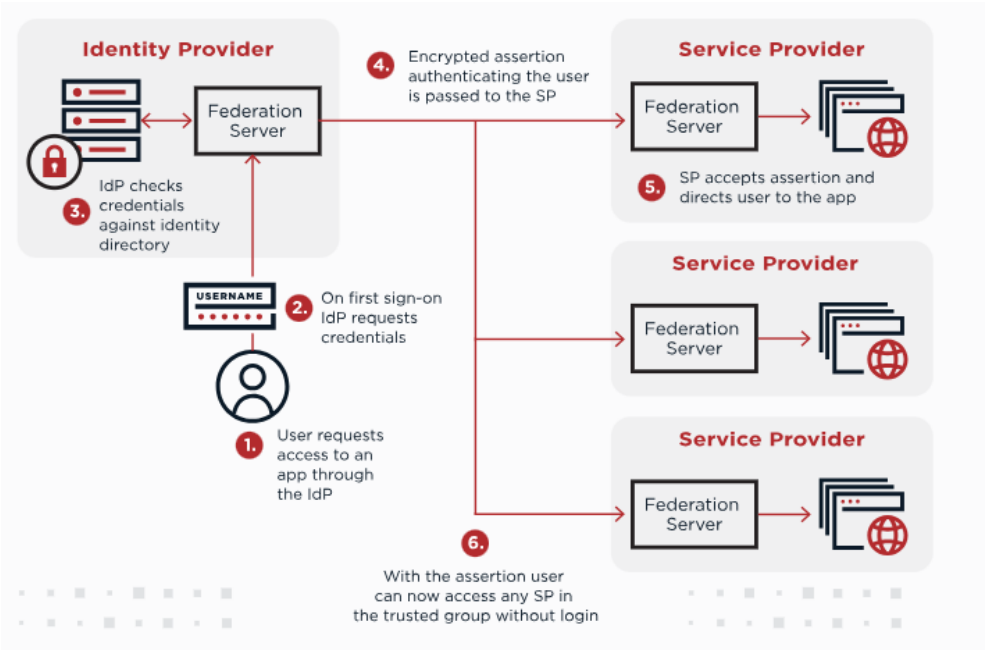


**Figure B.6:** IDP initiated federated SSO use case. [48]

According to IBM's Cost of a Data Breach 2021 report, most frequent initial attack vector for a data breach is based on compromised credentials. Using SSO gives

the attacker lesser chance to attack since it reduces the uses of many password. On the other hand, single password can become a reason of high risk as well if it is compromised. Some precaution can be taken to reduce this risk such as – creating long and complex password, implementing SSO with multi-factor authentication, or MFA which requires one additional authentication factor besides password ex– a code sent to a mobile phone, a fingerprint etc. [87]

Through single sign-on (SSO), OpenID Connect (OIDC) allows users to authenticate themselves using OpenID Providers (OPs). More details about OIDC explained in section 4.7