**JWT RESEARCHES**

What is JWT

JSON WEB TOKEN Sisteme bir kullanıcı giriş yaptıktan sonra yaptığı tüm isteklerde authorization işlemini gerçekleştirebilmek için gereken tokendir.

3 bölümden oluşur

Header: Token tipi ve imzalamak için gerekli algoritma (JWT ve RSA | ECDSA (for Asymettric encryption) gibi)

Payload: İstemin bulunduğu bölümdür. İstek için gerekli parametreleri içerir. Örnek bir payload olarak;

{

"sub": "1234567890",

"name": "Faruk K",

"admin": true

}

Signature: Bu kısmın oluşturulabilmesi için encoded header(Base64Url), encoded payload(Base64Url) ve gizli anahtar(secret) gereklidir.

Örnek bir Signature bölümü;

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

Bu signature mesajın integrity'sini sağlar aynı zamanda token eğer ki private key ile de imzalanmış ise bu sayede de Authenticity sağlanmış olur JWT sayesinde kullanıcılar sadece yetkilerinin olduğu requestleri yapabilirler ve bunun kontrolü sağlanır.

JWT Avantajları

1- Stateless(Karşı taraf ile iletişim kurulurken bir tcp connectionu kurulur. Bu connection eğer işlem bittikten sonra da açık bırakılıyor ve connection aç-kapa işleminin yükünün altına girilmiyorsa Bu statefuldur. Ama eğer query atıldıktan veya işlem yapıldıktan sonra bağlantı direk kapanıyor ise yani load balancer'da sürekli bir idle bölge bırakılıyor ise bu stateless'tir. İki durumun da avantajlı veya dezavantajlı olduğu durumlar vardır) çalışır. Yani kontrol edecek bir Session bulunmamaktadır. Bilgiler ve son geçerlilik tarihi ne sunucuda ne client tarafında tutulur. Token içerisinde gerekli bilgiler tutulur.

2- Portable çalışır. Birden çok backend ile çalışabilir, yalnızca 2 taraf arasında kullanılmak zorunda değildir. Bu durum hem web uygulamanız hem de mobil uygulamanız aynı web servisi kullandığında oldukça önemlidir.

3- JSON formatını kullanır.

4- Doğrulama işlemi diğer Authorization metodlarına göre daha hızlıdır. Doğrulama işlemi için veritabanı ile bağlantı kurmaya gerek kalmaz

JWT’de Encryption Decryption Nasıl Gerçekleşir

\*Şifreleme işleminde eğer ki bir asymmetric signing algoritma kullanıldıysa sadece authorization server tokenleri imzalayabilir.

Örnek olarak RSA Digital signature algoritması olarak kullanılsın ve SHA256 da asymmetric key exchange algoritması olsun. Bu durumda auth server private keyini kullanarak tokeni oluşturur ve imzalar ardından tokeni alan application serverin public keyi ile tokeni decrypt eder. Genel şema şöyle işler;

1-) Header kısmı base64urlencode ile şifrelenir (Varsayalım şifreleme metodu olarak RSA belirtilsin).

2-) Payload kısmı base64urlencode ile şifrelenir. Bu iki veri art arda eklenir ve buna Signing Input denir. (Birbirlerinden bir nokta ile ayrılırlar sdkjfsjdf.WawEFG gibi)

3-) Daha sonra Signing Input SHA-256 ile hashlenir.

4-) Hashed Signing Input encrypt edilir. Bu aşamada token issuer(Auth server) kendi private keyini kullanarak ecryption işlemini gerçekleştirir(RSA-ECDSA gibi).

5-) Final output JWT'nin signaturesidir. Bu kısma Crypto Segment denir.

6-) JWT'yi alan application JWT içeriklerine ve auth server'ın public keyine sahiptir. Burada şöyle bir durum oluyor, zaten header ve payload base64url ile şifrelendiği için direk decode edilebilir ve application|client bunu decode edebilir. Ama bunların integrity'sini bu verileri tekrar sha256 ile şifreleyerek signature içerisinde bulunan halleri ile karşılaştırdığında sağlayabiliyor. Signature içerisindeki veriye erişebilmek için de Token issuer'ın public keyi ile signature bölmesini açıyor. Bu sayede mesajın yol boyunca değişmediğini kontrol etmiş oluyor. Token issuer'ın signaturesini valide ettikten sonra payload segementin içerisinde bulunan claimlerinde valide edilmesi gerekiyor. O bölmede aynı zamanda token issuer, token expiration, tokeni alan application'un client id'si, tokeni auth requeste bind eden bilgiler. Örneğin:

{

"iss": "https://{you}.authz-server.com",

"aud": "RxHBtq2HL6biPljKRLNByqehlKhN1nCx",

"exp": 1570019636365,

"iat": 1570016110289,

"nonce": "3yAjXLPq8EPP0S",

...

}

- iss (issuer): JWT issuer'ı. Yani auth Serevr

- aud (audience): Tokeni recieve eden applicationun client ID'si

- exp (expiration time): Tokenin bitiş süresi

- iat (issued at time): Tokenin oluşturulduğu zaman

- nonce : Client'ın auth requestini aldığı tokene bind eder. Client bu rastgele değeri oluşturur ve auth request ile yollar. Auth server bu değeri tokeni application'a geri göndermeden önce tokene gömer. Application serverdan dönen tokendeki nonce değeri ile kendi oluşturduğu ve yolladığı nonce değeri aynı mı diye kontrol eder. Böylece application gelen tokenin istemi yaptığı doğru auth serverdan geldiğini kendi kendine kontrol edebilmiş olur.

Bu veriler, reciever app'e, Signature validationunun tek başına sağlayamayacağı önemli ayrıntılar verir. Örneğin, claimlerin incelenmesi, teknik olarak geçerli bir tokenin aslında farklı bir uygulama veya kullanıcı için tasarlandığını, süresinin dolduğunu, uygulamanın hiçbir bağlantısı olmayan bir issuer'dan geldiğini vb. ortaya çıkarabilir.

JWT Potential Vulnerabilities

In Jwt attacks generally a hacker sends modified JWTs to the server to achieve a malicious goal. This goal can be bypass authentication and Access controls by impersonating another authenticated user.

Impact of JWT attacks are most usually severe. Attacker can escalate his priveleges or impersonate other users to take control of their accounts. A flaw in JWT generally means that signature of JWT is not verified properly. This situation let attacker tamper the value passed to the application via JWT’s payload. Even if the signature is verified correctly, it’s reliability lies on the server secret key. If the key is leaked in some way or can be guessed or brute-forced, attacker can generate a valid signature for any token that he want to create. That will be a big problem.

**Vulnerability 1 : Exploiting Flawed JWT Signature Verification**

This attack methods relies on lackness of JWT information storing on servers which they issued. Instead of it, each token accepted as self-contained entity. Therefore, server doesn’t actually know about the content of the original token or even what the original signature was. So if server doesn’t verify signatures properly an attacker can make arbitrary changes on JWT tokens. For example, consider a JWT containing the following claims:

{

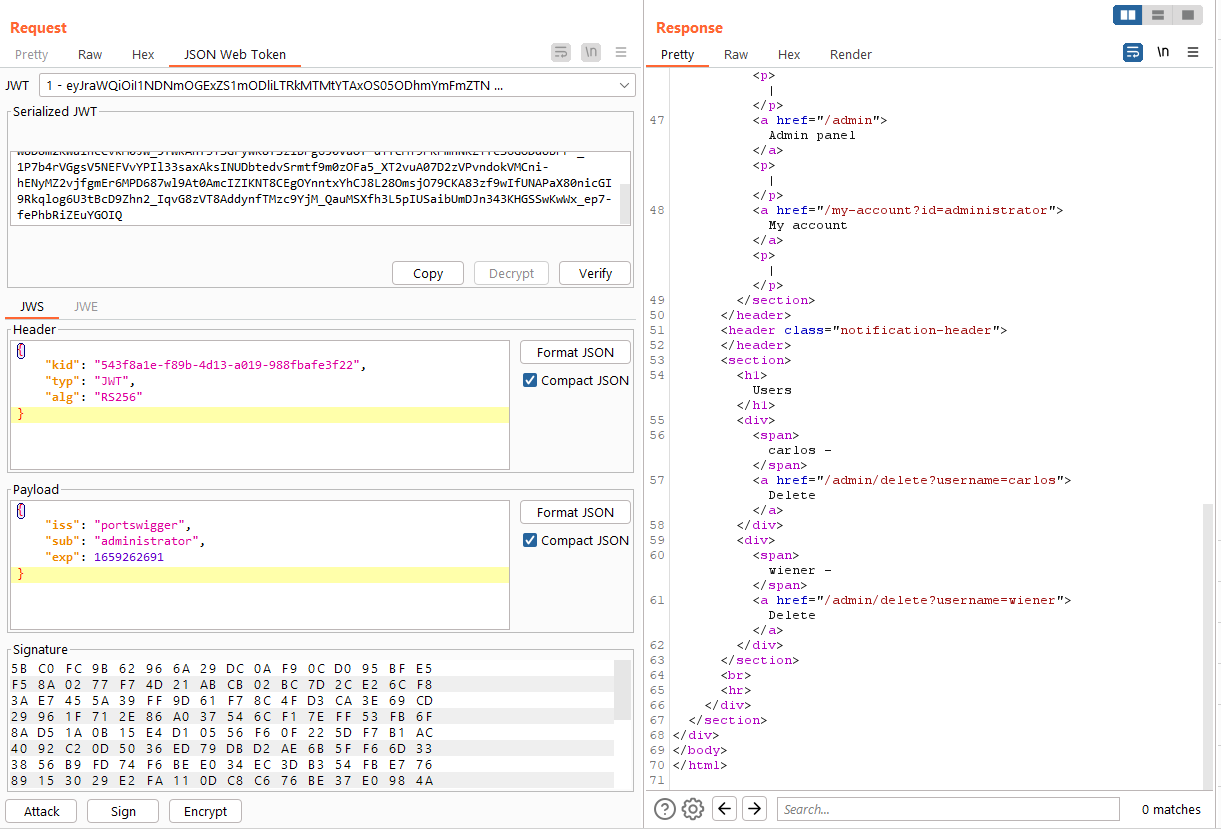
"username": "faruk",

"isAdmin": false

}

If the server identifies the session based on this username, modifying the username field might let an attacker to impersonate other logged-in users. Similarly, if the isAdmin value is used for access control, this could provide a simple vector for privilege escalation.

**LAB 1 :** The aim of this lab is to gain /admin panel and delete the user Carlos.

1. Logging into the site with credentials wiener:peter
2. When we logged in website, I tried to go to the /admin panel website didnt allow me because I am not an admin user. So I changed payload of JWT to pretend like an admin user because there was no signature verification on JWTs
3. I Access the admin panel by acting as admin user. And clicked delete user Carlos button. I had to modify that requests payload to change sub as “admin” this time. And lab is solved



**NOTE: Try to remember Copying the Repeater request and paste it to the Intercepted Proxy Request. Change the Intercepted request with the modified Repeater request is the thing we need to do to be able to use modified JWTs**

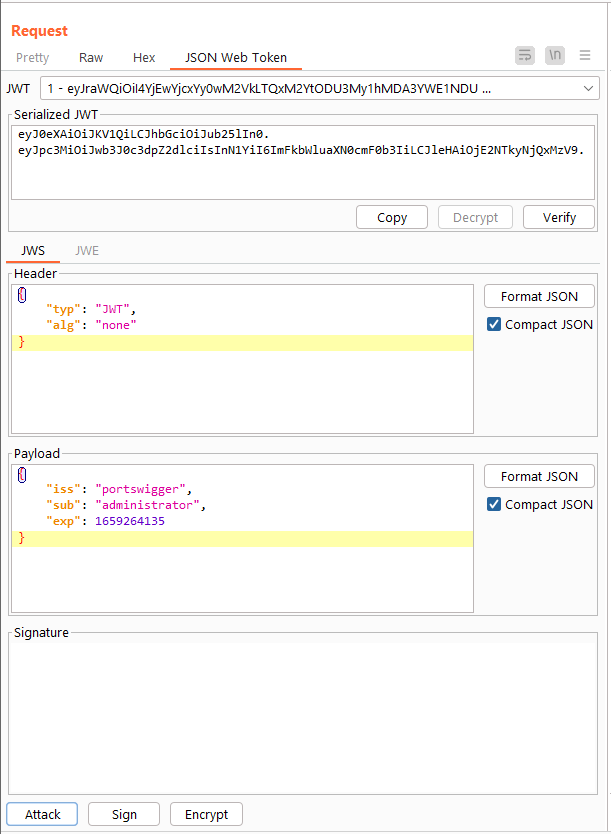
**Vulnerability 2 : Accepting tokens with no Signature**

In tokens header there is an alg parameter which declares algorithm that it need to use when signing the token and for verifying signature. This is flawed because if server doesn’t verify the alg parameter, an attacker can change how the server checks the token trustworthiness(by changing token’s alg parameter). JWT token can be left unsigned. So in this case **alg parameter** set to **none,** which tell to server not to check signature actually so we can left the signature blind in that case. Server generally reject tokens with no signature. However, this kind of filtering relies on string parsing, sometimes we can bypass these filters using classic obfuscation techniques, such as mixed capitalization and unexpected encodings.

Important note : **Even if the token is unsigned, the payload part must still be terminated with a trailing dot.**

**LAB 2 :** The aim of this lab is to gain /admin panel and delete the user Carlos.

1. I try to Access to the admin panel by pretending as admin user like in the first lab, but in this lab server checks JWTs signatures. But this time server does not check alg parameter which can set to be **none** to prevent server for checking signatures.
2. When I inspecting the accessing /admin panel request, In JWT Request editor tab on the Repeater, There was a attack button below the payload and we can select an option that say **Attack with “none” signing algorithm.** I clicked it and the request was modified like this;



1. And the request was accepted and I was able to delete user Carlos(change deleting requests payload accordingly(change user to administrator and redirect the page on repeater)).



**Vulnerability 3 : Brute-forcing secret keys**

Some signing algorithms, such as HS256 (HMAC + SHA-256), use an arbitrary, standalone string as the secret key. Just like a password, it's important that this secret can't be easily guessed or brute-forced by an attacker. Otherwise, they may be able to create JWTs with any header and payload values they like, then use the key to re-sign the token with a valid signature. During implementing JWT apps, developers can make mistakes like forgetting to change default or placeholder secrets. They may copy and paste code parts they find online so the even forget the change the secret value of that example which is hardcoded. So an attacker can [use well-known secrets’ wordlist](https://github.com/wallarm/jwt-secrets/blob/master/jwt.secrets.list) (Burpsuite recommends **HASHCAT** to brute-force secret keys. It is pre-installed on Kali Linux) to be able to make a brute-force attacks. You just need a valid, signed JWT from the target server and a [wordlist of well-known secrets](https://github.com/wallarm/jwt-secrets/blob/master/jwt.secrets.list). You can then run the following command, passing in the JWT and wordlist as arguments:

hashcat -a 0 -m 16500 <jwt> <wordlist>

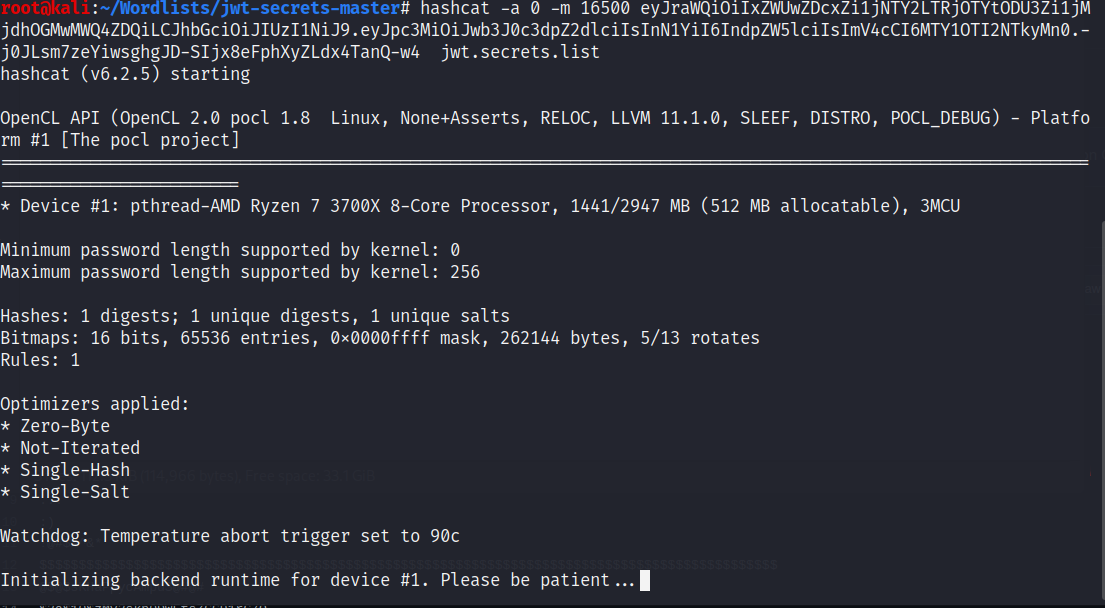
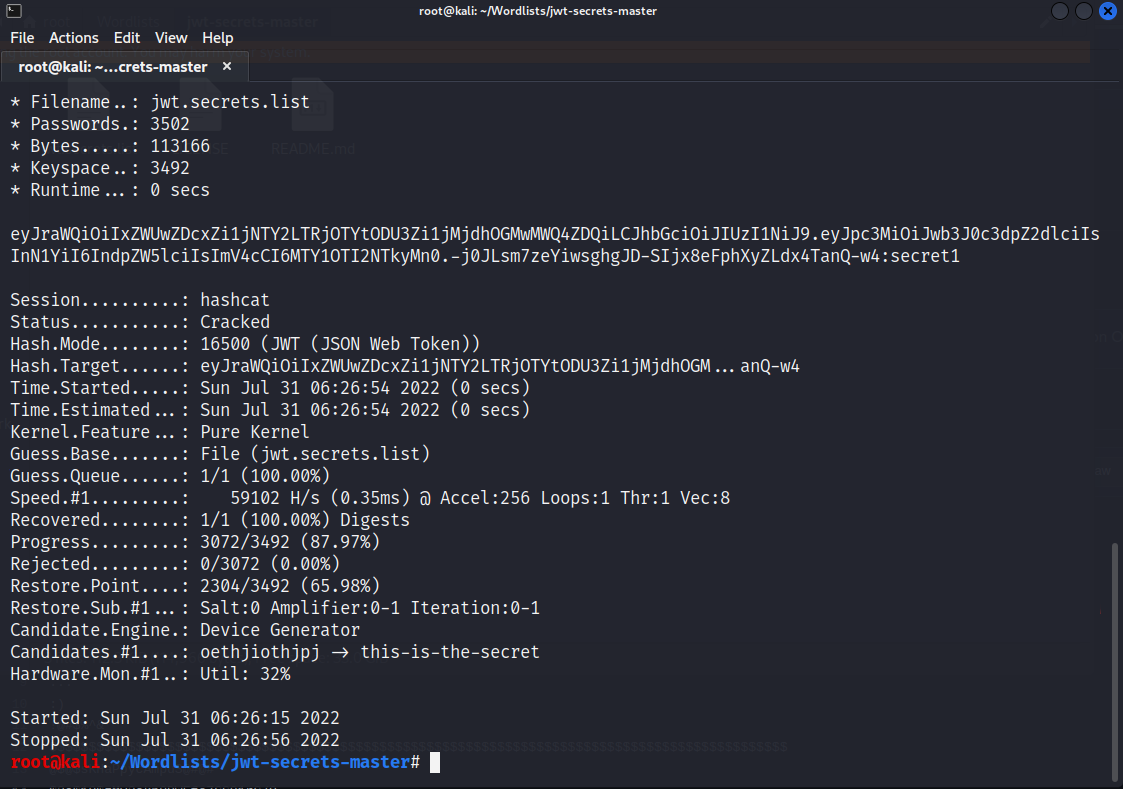
Hashcat signs the header and payload from the JWT using each secret in the wordlist, then compares the resulting signature with the original one from the server. If any of the signatures match, hashcat outputs the identified secret in the following format, along with various other details:

<jwt>:<identified-secret>

**NOTE : If you run the command more than once, you need to include the --show flag to output the results.**

Once an attacker identified the secret key, it can be used to generate any valid signature for any header and payloads. Also if the server uses a very weak secret key, it can even be brute-forced character-by-character instead of using wordlists.

**LAB 3 :** The aim of this lab is to gain /admin panel and delete the user Carlos in a page that using HS256 alg in JWTs. So we need to get the secret value to create arbitrary tokens.

1. I captured the /admin panel request and request’s alg parameter var HS256.
2. I tried to get the secret value with hashcat in my Kali Linux 
3. Hashcat found the secret key inside the wordlist (**”secret1”** is the secret key value); 
4. Then we need to create a new Symmetric Key from the JWT Key editor tabs. When generating key size doesnt matter because extension will update it whenever necessary in the requests. Also, change the **k** value in the generating phase with the **base-64 encoded secret value that we found on the hashcat.** After generating the key modify the request’s sub part in the payload as “administrator” and sign the request with the key that we created. Later parts will be the same for deleting the Carlos(Click follow the redirection when 302 status code in response in repeater). 

**Vulnerability 4 : JWT header parameter injections**

According to the JWS (Json web signature) specification only the **alg** parameter is mandatory on the header. There are some other parameters that we can include in the header part of JWT. Example :

* jwk (JSON Web Key) - Provides an embedded JSON object representing the key.
* jku (JSON Web Key Set URL) - Provides a URL from which servers can fetch a set of keys containing the correct key.
* kid (Key ID) - Provides an ID that servers can use to identify the correct key in cases where there are multiple keys to choose from. Depending on the format of the key, this may have a matching kid parameter.

All of these parameters tell the server which key to use when verfying the signature. So an attacker can inject modified,signed JWT’s with our own arbitrary key rather than the server’s secret key.

**Injecting self-signed JWTs via the jwk parameter:**

With this **jwk** header parameter, servers can embed their public key directly within the token itself in JWK format.

**Not : A JWK (JSON Web Key) is a standardized format for representing keys as a JSON object.**

Example of this in JWT header :

{

"kid": "ed2Nf8sb-sD6ng0-scs5390g-fFD8sfxG",

"typ": "JWT",

"alg": "RS256",

"jwk": {

"kty": "RSA",

"e": "AQAB",

"kid": "ed2Nf8sb-sD6ng0-scs5390g-fFD8sfxG",

"n":"yy1wpYmffgXBxhAUJzHHocCuJolwDqql75ZWuCQ\_cb33K2vh9m"

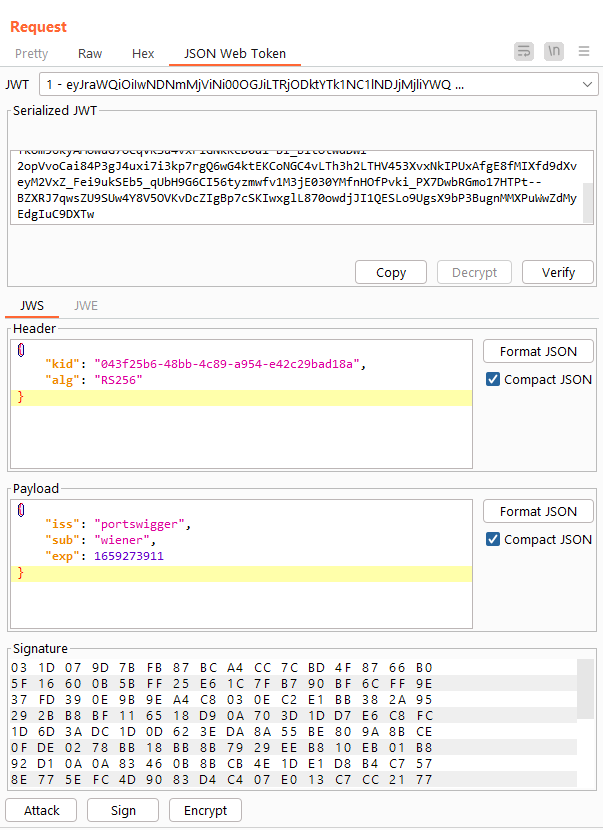
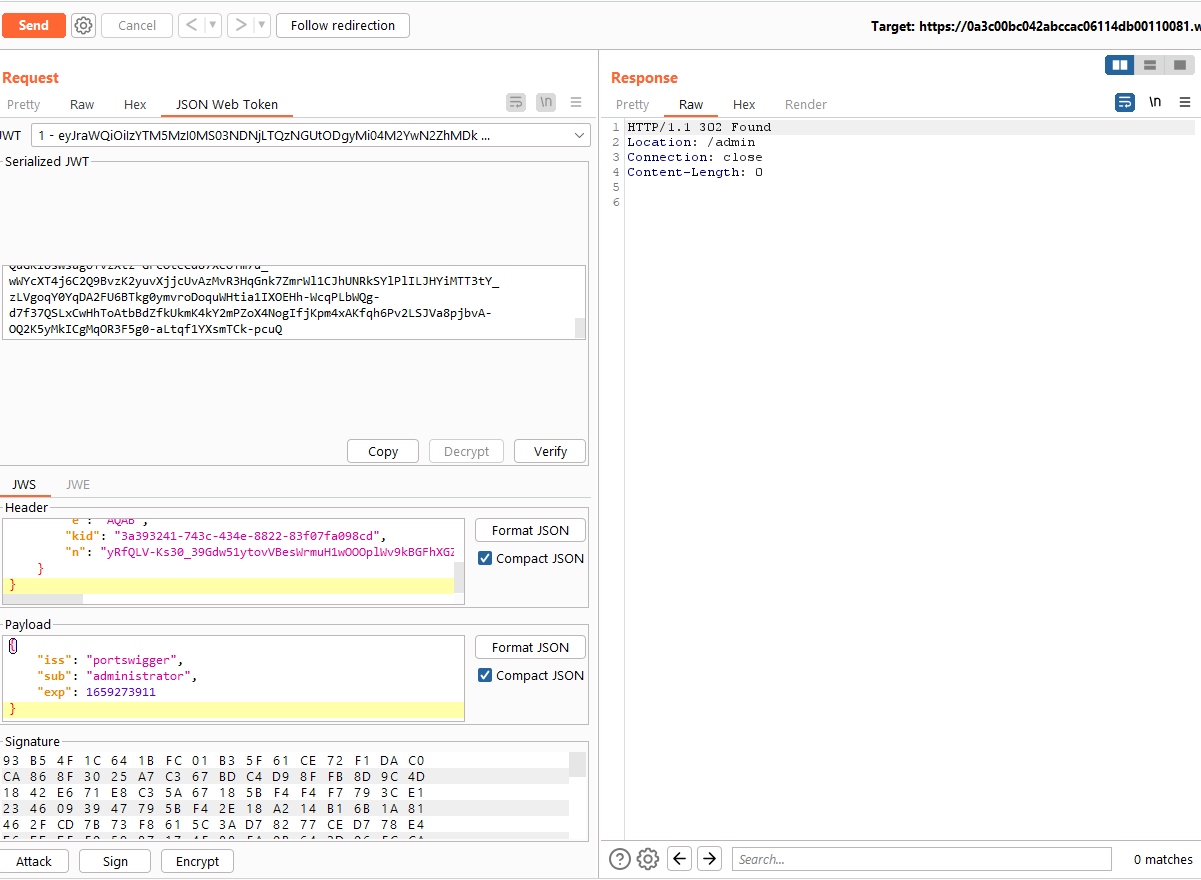
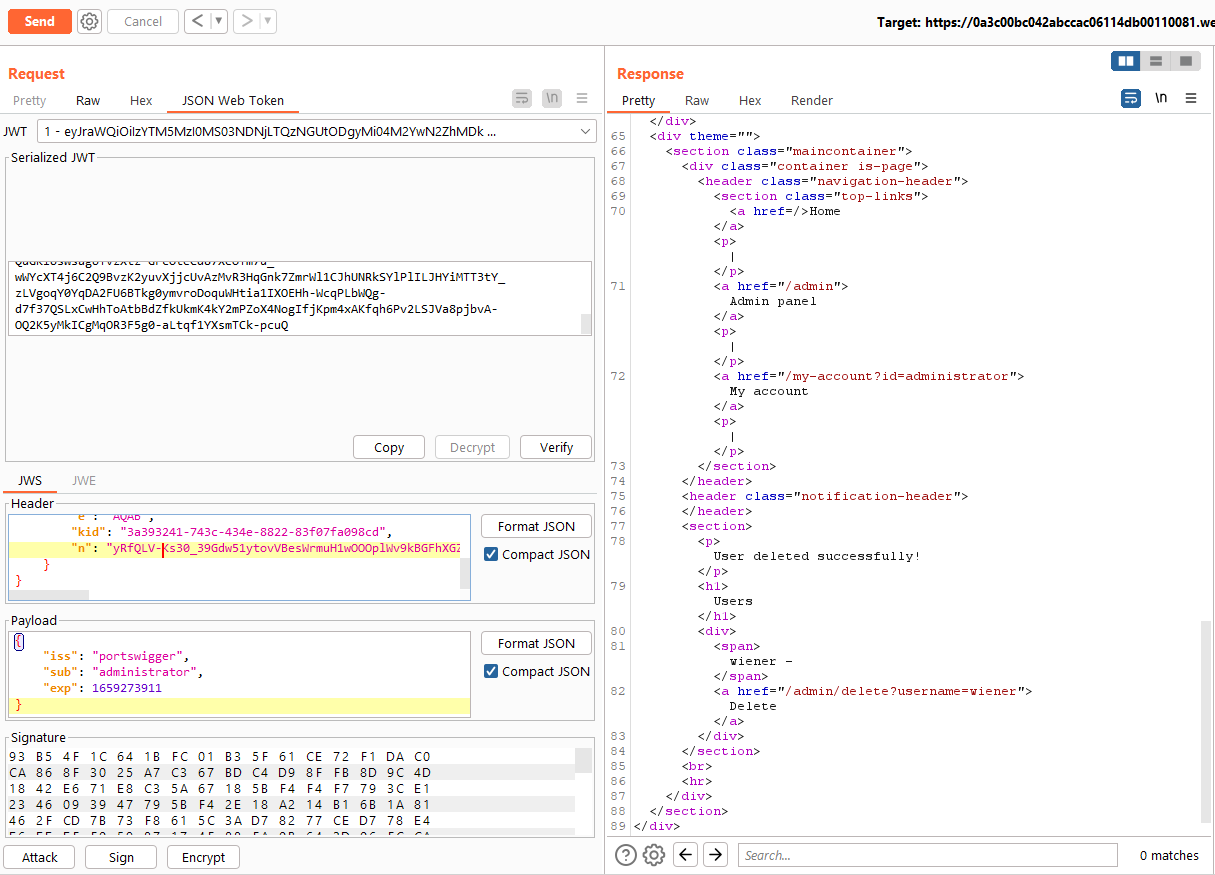
}

}

Normally, servers should only use a whitelisted public keys to verify the JWT signatures. However if the server misconfigured, server can use any key that’s embedded in the jwk parameter. We can exploit this vulnerability by signing a modified JWT using our own RSA private key, then embedding the matching public key in the jwk header. There is steps on Portswigger to exploit these vulnerability by using JWT Editor Extension. Here are the steps:

1. With the extension loaded, in Burp's main tab bar, go to the **JWT Editor Keys** tab.
2. Generate a new RSA key.
3. Send a request containing a JWT to Burp Repeater.
4. In the message editor, switch to the extension-generated **JSON Web Token** tab and modify the token's payload however you like.
5. Click **Attack**, then select **Embedded JWK**. When prompted, select your newly generated RSA key.
6. Send the request to test how the server responds.

**LAB 4 :** The aim of this lab is to gain /admin panel and delete the user Carlos in a page that uses jwk header in the JWT header.

1. Captured **get /admin** request and the request was like this; 
2. Then I clicked the **Attack->Embedded JWK** and selected one of the RSA keys that I have created before(changed sub parameter to administrator)(**CLICK FOLLOW REDIRECTION**). 
3. The lab is done by making the same steps on deleting Carlos request. 



Injecting self-signed JWTs via the jku parameter

Instead of embedding public keys directly using the jwk header parameter, some servers let you use the jku (JWK Set URL) header parameter to reference a JWK Set containing the key. When verifying the signature, the server fetches the relevant key from this URL.

**Note : A JWK Set is a JSON object containing an array of JWKs representing different keys. Example :**

{

"keys": [

{

"kty": "RSA",

"e": "AQAB",

"kid": "75d0ef47-af89-47a9-9061-7c02a610d5ab",

"n": "o-yy1wpYmffgXBxhAUJzHHocCuJolwDqql75ZWuCQ\_cb33K2vh9mk6GPM9gNN4Y\_qTVX67WhsN3JvaFYw-fhvsWQ"

},

{

"kty": "RSA",

"e": "AQAB",

"kid": "d8fDFo-fS9-faS14a9-ASf99sa-7c1Ad5abA",

"n": "fc3f-yy1wpYmffgXBxhAUJzHql79gNNQ\_cb33HocCuJolwDqmk6GPM4Y\_qTVX67WhsN3JvaFYw-dfg6DH-asAScw"

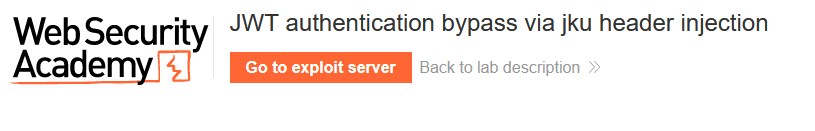
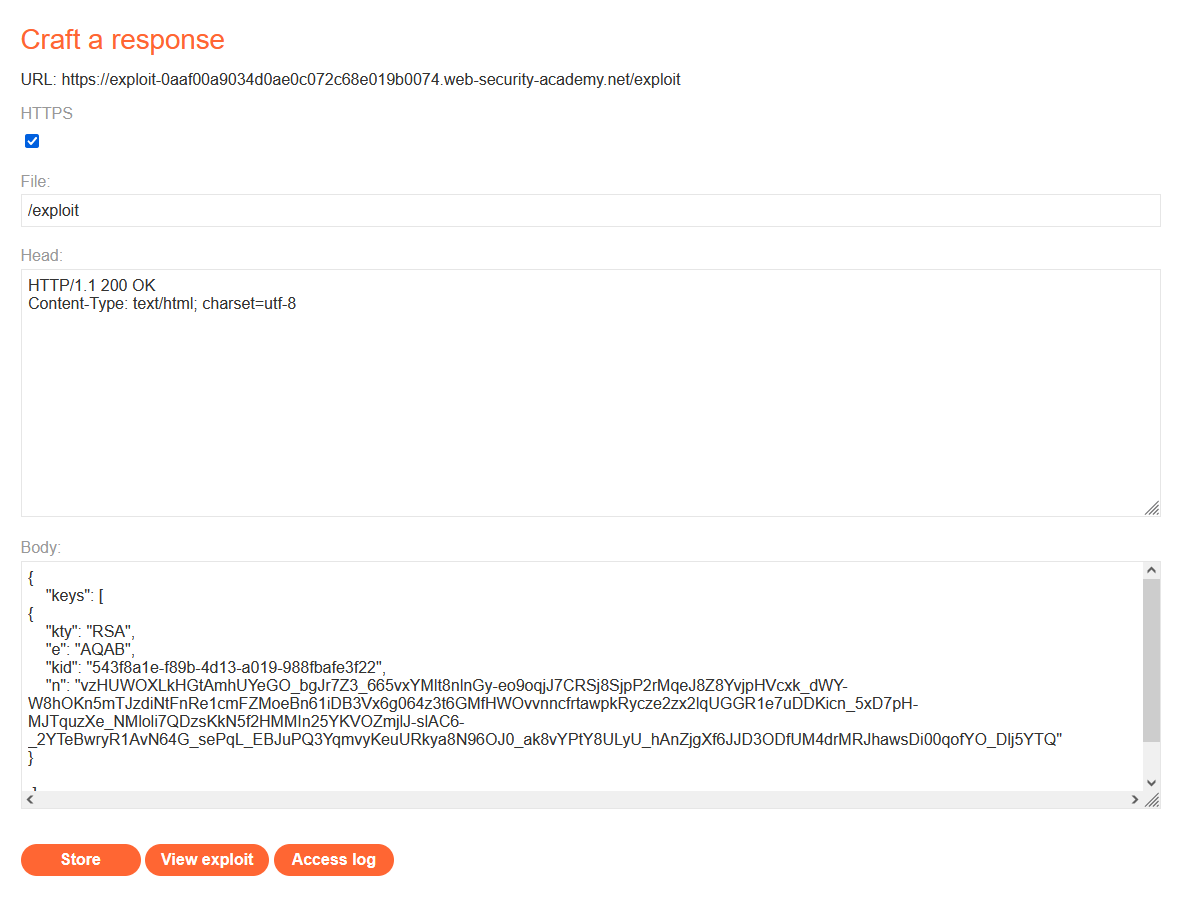
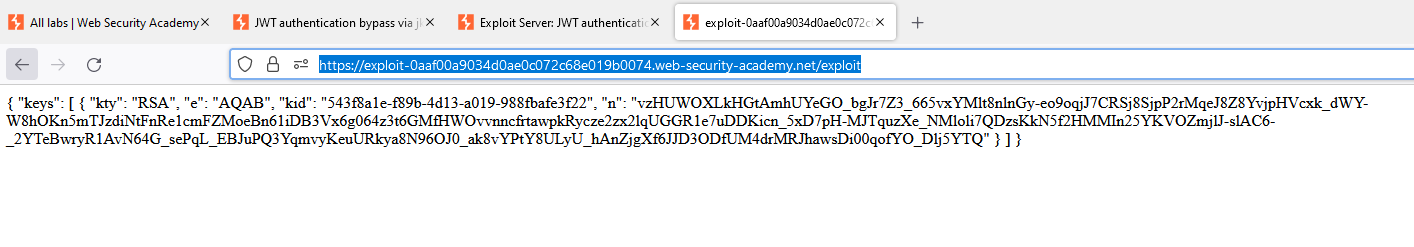
}

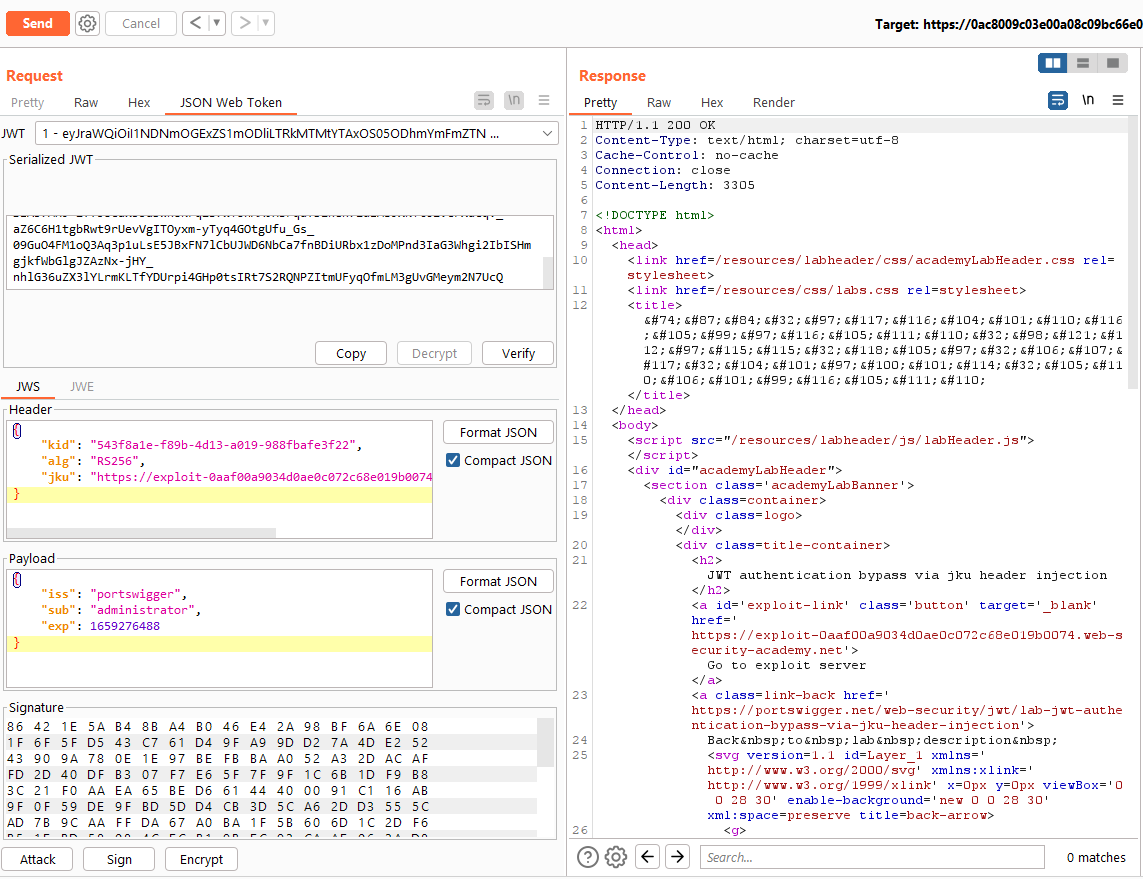
]

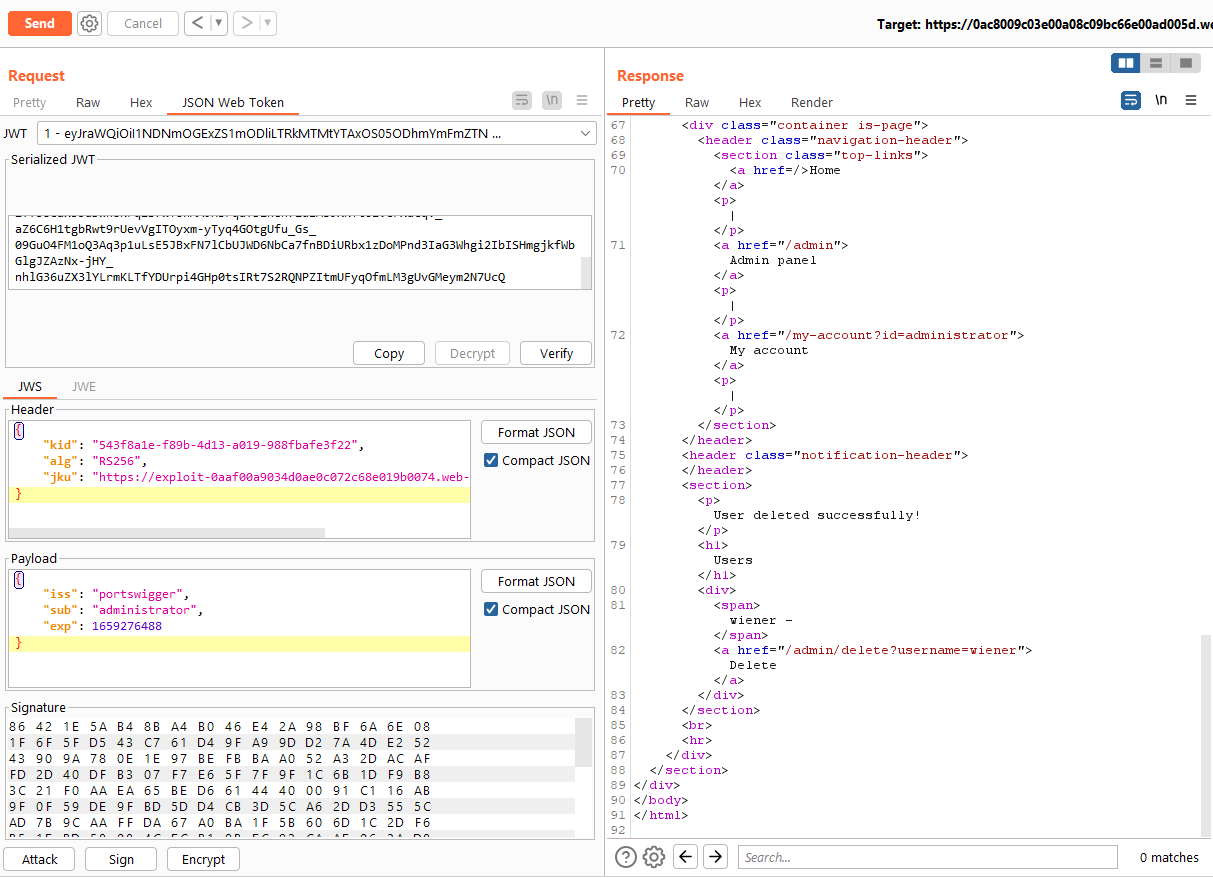
}

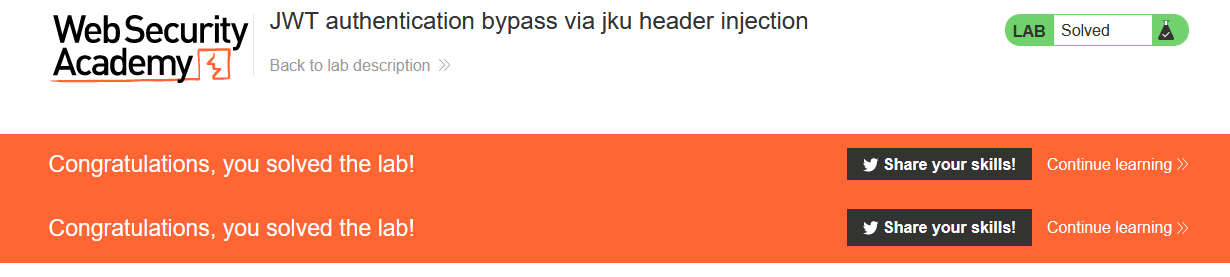
Secure websites only fetch keys from trusted domains but attacker can use URL parsing discrepancies to bypass filtering.

**LAB 5 :** The aim of this lab is to gain /admin panel and delete the user Carlos in a page that uses jku header parameter in the JWT header.

* 1. In this type of vulnerability, we need to create-find a page that contains JWK SET to instruct the server take the public key value from the JKU. Either we can do this with SSRF from our create page or we can do that with a page that is inside of the webapp that we try to attack.
  2. In this website there was a button that say go to exploit page; 
  3. When we click that value server redirects us to the page that I mentioned in the step 1; 
  4. I have edited Body part of the page with corresponding to the JWK Set. In the **keys** array **I have copied one of my RSA key’s public key**, and pasted it inside of the “keys” array. When I stored that value and clicked View exploit button, I saw a page that contain JWK Set with the public key value that derived from my RSA key. URL of that page is the **JKU** value that I need to forge the JWT
  5. I have modified the JWT in the repeater JWT extension. I have added

**jku:** [**https://exploit-0aaf00a9034d0ae0c072c68e019b0074.web-security-academy.net/exploit**](https://exploit-0aaf00a9034d0ae0c072c68e019b0074.web-security-academy.net/exploit) parameter on the header and I have modified **sub** parameter in the payload to the administrator. After that I have to change **the kid parameter with the kid parameter on my RSA key.** Then I have signed the token with the same key that I used before. After all of these steps my request is accepted. 

* 1. When I entered the /admin panel did the same thing with earlier to make the request valid. 



Injecting self-signed JWTs via the kid parameter

Header of a JWT can contain **kid**(Key ID) parameter which helps server to identify which key to use when verifying a signature. Verification signature are often stored as JWK Set. So the server simply look for the JWK with the same kid as the token. Most importantly JWS specification doesn’t define a proper structure for kid, **it is just an arbitrary string of the developer’s choosing.** This means that an attacker can use the kid parameter to point a particular entry in database or a name of the file. Also if this parameter is also vulnerable to **directory traversal,** attacker can force server to use a arbitrary file from server’s system as the verification key. For example :

{

"kid": "../../path/to/file",

"typ": "JWT",

"alg": "HS256",

"k": "asGsADas3421-dfh9DGN-AFDFDbasfd8-anfjkvc"

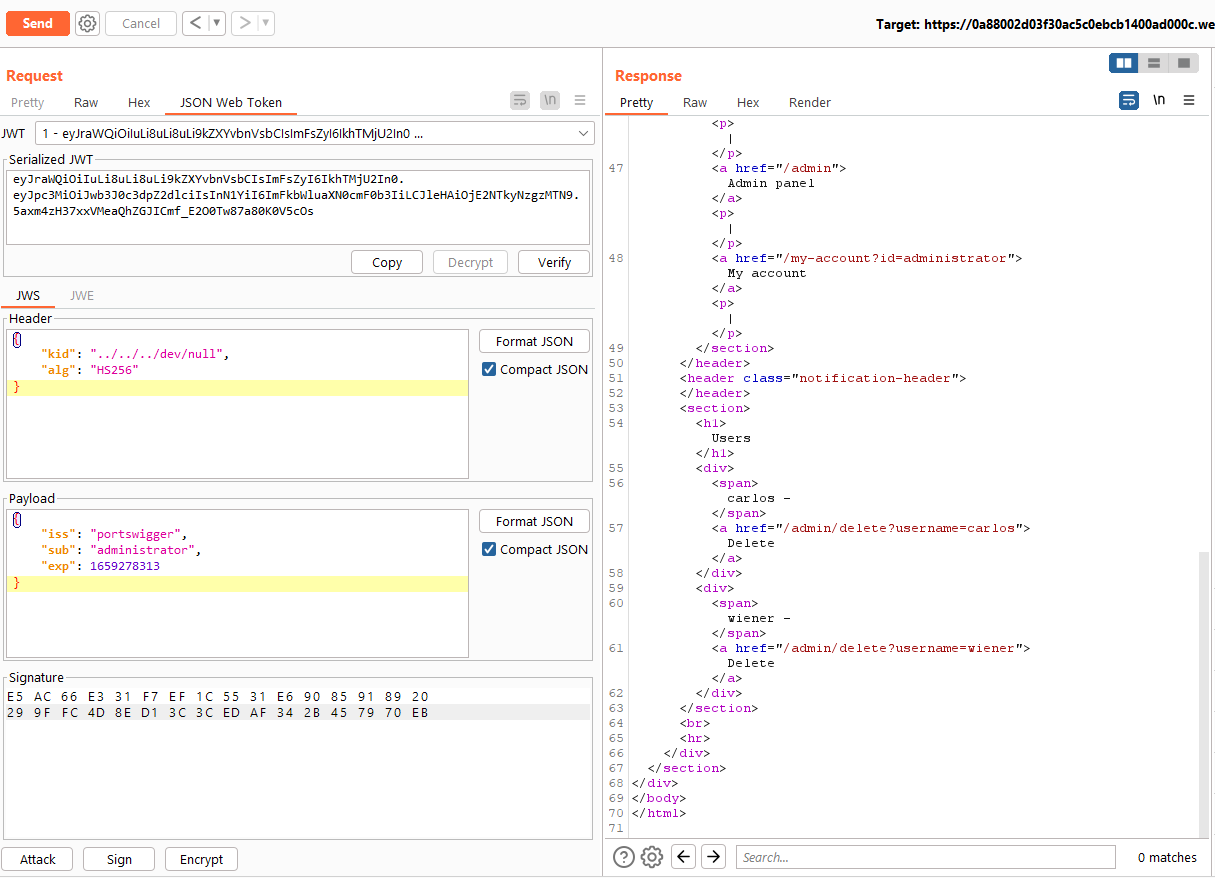
}

This is especially dangerous because if server support signed JWTs with symmetric algorithm, in this case an attacker could potentially point the kid parameter to a predictable, static file (a file with null value maybe), then sign JWT using a secret that matches the contents of that file. Theoretically attacker can do this with any file, but to give an example, /dev/null file is mostly exist on Linux systems. As this is an empty file, fetching it will return null. Therefore, signing token with a Base64-encoded null byte will result in a valid signature.

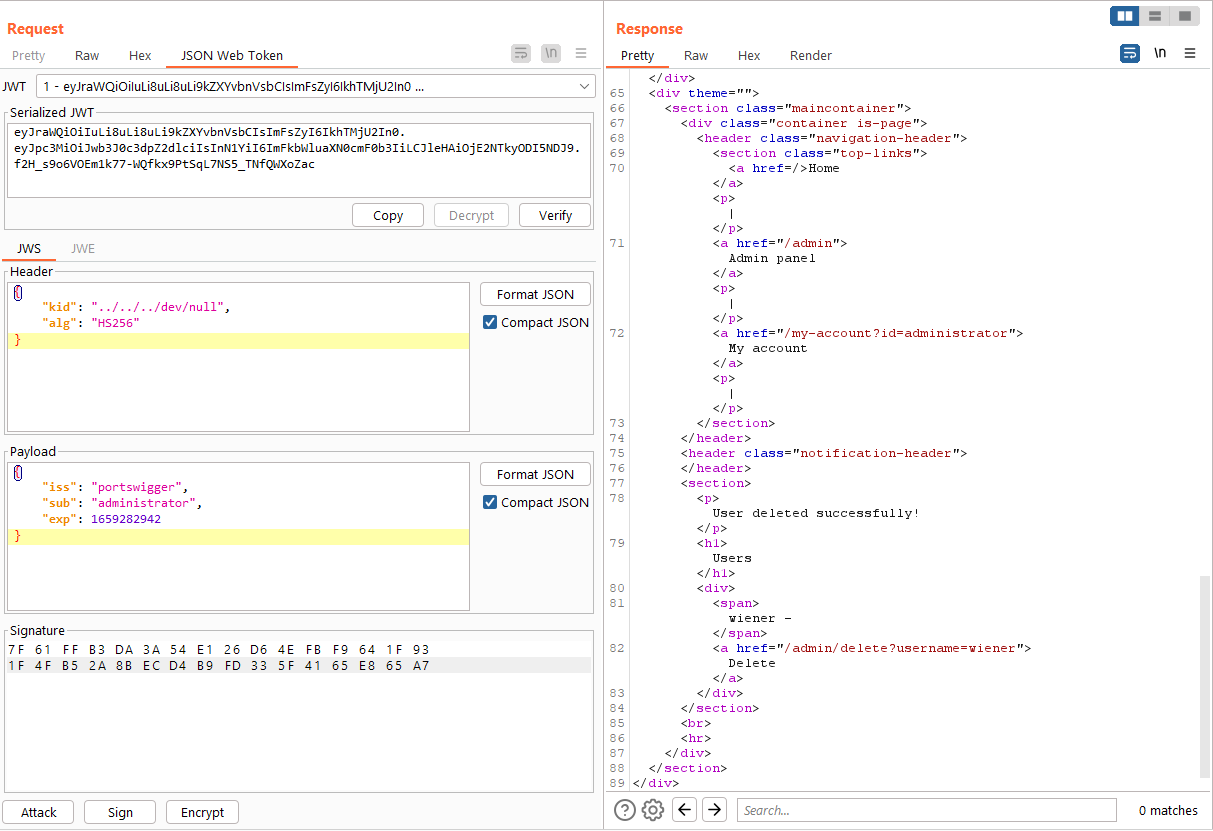
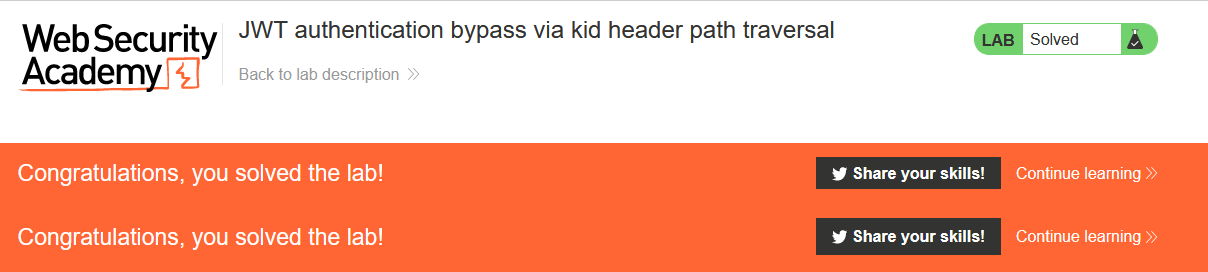
Important note : If server stores its verification keys in a database, the **kid** header can also be used for SQL injections to get the key values.

**LAB 6 :** The aim of this lab is to gain /admin panel and delete the user Carlos in a page that uses kid parameter in the JWT header to fetch the relevant key from the system. Be careful, server is using Symmetric key enc. on JWT :) (so we can make directory traversal attacks an deven SQLi maybe to get the kid value because that string(kid parameter value) is not correctly checked mostly)

1. Send the JWT request to the repeater and we will see a kid parameter on the header and also we gonna see the alg is set to **alg:HS256.**
2. As I mentioned before when we see HS256 Symmetric key encrypted JWTs we will think about directory traversal because kid header value is a string that is not checked properly. **So we can specify a location on the server which is null f.e and create a symmetric encryption key with the k value is set to base-64 encoded null byte which is (AA==**). Then we modify the JWTs kid header with directory traversal attack to be able to use a **/dev/null** file which is **null.** When we sign the JWT using a secret that matches the contents of the file JWT will become valid.
3. So I modified the key value with ../../ until I find the exact number of ../ . Dont forget to sign the JWT everytime to test if ../ number is enough or not to reach the file. And also change sub parameter on the payload to the “administrator”;



1. Do the same steps for deleting Carlos after reaching the /admin panel;

Other Interesting JWT Header parameters

**cty (Content Type)** : Declares media type for the content in the JWT payload. This is usually removed from the header, but underlying parsing library may support it. So if we found any way to bypass signature verification, we can try injecting **cty** header to change the content type to text/xml or application/x-java-serialized-object which can potentially enable new vectors for **XXE** and **deserialization** attacks

**Vulnerability 5 : JWT Algorithm Confusion**

Even if the server uses a robust secrets that unable to brute-force or guess, we may still create valid JWTs by signing the token with algorithm that developers haven’t been expecting. Example, we force the server to verify the signature of JWT using a different algorithm than website is intended to use.

This kind of vulnerabilities arise due to flawed implementation of **JWT libraries**. Verification process differs depending to the algorithm, many library provide algorithm-agnostic(algoritmadan bağımsız) method for verfying signature. And this methods only look for the algs parameter in the tokens header to verify the verfying encryption method. To give an example one of the signature verify method on JWT library;

function verify(token, secretOrPublicKey){

**algorithm = token.getAlgHeader();**

**if(algorithm == "RS256"){**

**// Use the provided key as an RSA public key**

**} else if (algorithm == "HS256"){**

**// Use the provided key as an HMAC secret key**

}

}

Problem arise when developers assume that the method will handle JWTs signed with asymmetric algorithm such as RS256. Due to this flawed assumption, they could pass a fixed public key to the method. For example :

publicKey = <public-key-of-server>;

token = request.getCookie("**session**");

verify(token, publicKey);

(Important Note : JWT tokens are sended server in the request via session cookie parameter)

In this case, anyone who have server’s public key (but it must be the correct public key) can create arbitrary JWT that signed with servers public key as a HMAC secret. This means that attacker could sign the token using HS256 and the public key, and server will use the same public key to verify the signature.

**Performing an algorithm confusion attack**

1. Obtain the server's public key
2. Convert the public key to a suitable format
3. Create a malicious JWT with a modified payload and the alg header set to HS256.
4. Sign the token with HS256, using the public key as the secret.

**Step 1 : Obtain the server's public key**

Servers sometimes expose their public keys as JWK objects via standard endpoint mapped to **/jwks.json or /.well-known/jwks.json** for example. This endpoint may store arrays with JWKs keys. Example ;

{

"keys": [

{

"kty": "RSA",

"e": "AQAB",

"kid": "75d0ef47-af89-47a9-9061-7c02a610d5ab",

"n": "o-yy1wpYmffgXBxhAUJzHHocCuJolwDqql75ZWuCQ\_cb33K2vh9mk6GPM9gNN4Y\_qTVX67WhsN3JvaFYw-fhvsWQ"

},

{

"kty": "RSA",

"e": "AQAB",

"kid": "d8fDFo-fS9-faS14a9-ASf99sa-7c1Ad5abA",

"n": "fc3f-yy1wpYmffgXBxhAUJzHql79gNNQ\_cb33HocCuJolwDqmk6GPM4Y\_qTVX67WhsN3JvaFYw-dfg6DH-asAScw"

}

]

}

**Step 2 : Convert the public key to a suitable format**

Even if we find the servers true public key in JWK format, when verifying a signature of the token, server will use its own copy of the key from its local file system or database. So it can be stored in different format.

To make this attack work, version of our key must be identical with the server’s local copy. Addition to being a same format, **every byte must match**. Even the non-printing characters. Assume that we need a key in X.509 PEM format. We can convert JWK to a **PEM** using **JWT Editor.** Here are the steps of doing this;

1. With the extension loaded, in Burp's main tab bar, go to the **JWT Editor Keys** tab.
2. Click **New RSA** Key. In the dialog, paste the JWK that you obtained earlier.
3. Select the **PEM** radio button and copy the resulting PEM key.
4. Go to the **Decoder** tab and Base64-encode the PEM.
5. Go back to the **JWT Editor Keys** tab and click **New Symmetric Key**.
6. In the dialog, click **Generate** to generate a new key in JWK format.
7. Replace the generated value for the k parameter with a Base64-encoded PEM key that you just copied.
8. Save the key.

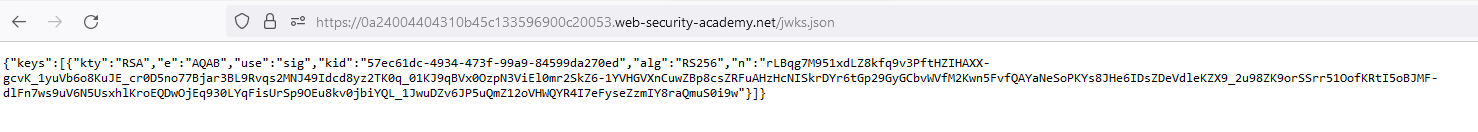
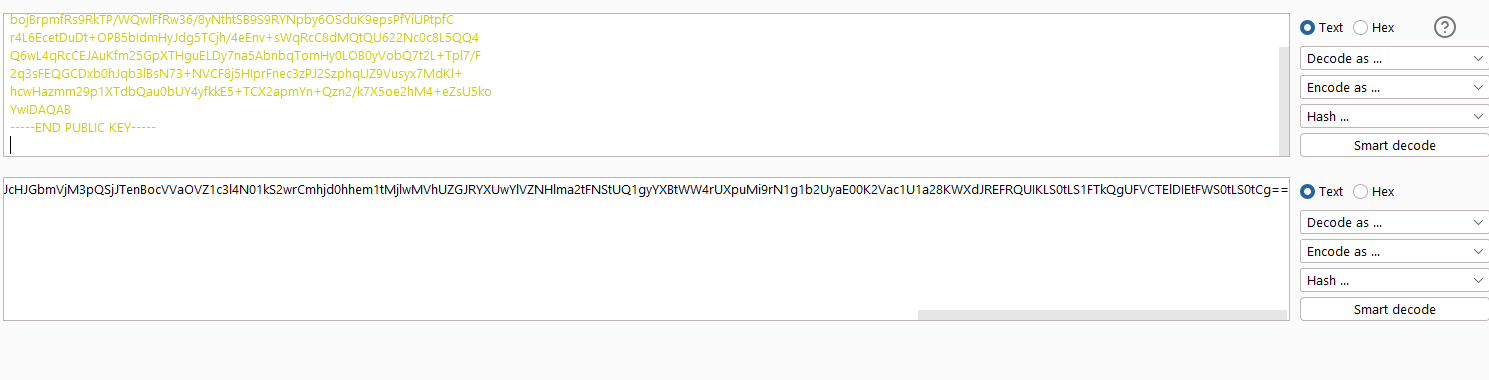
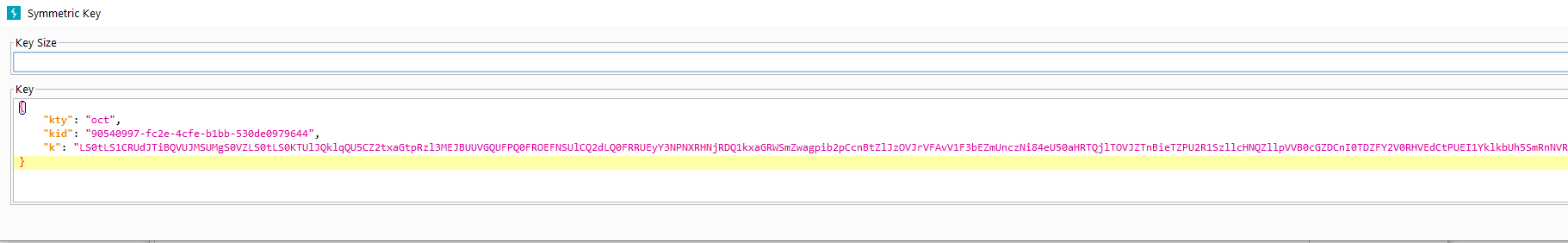
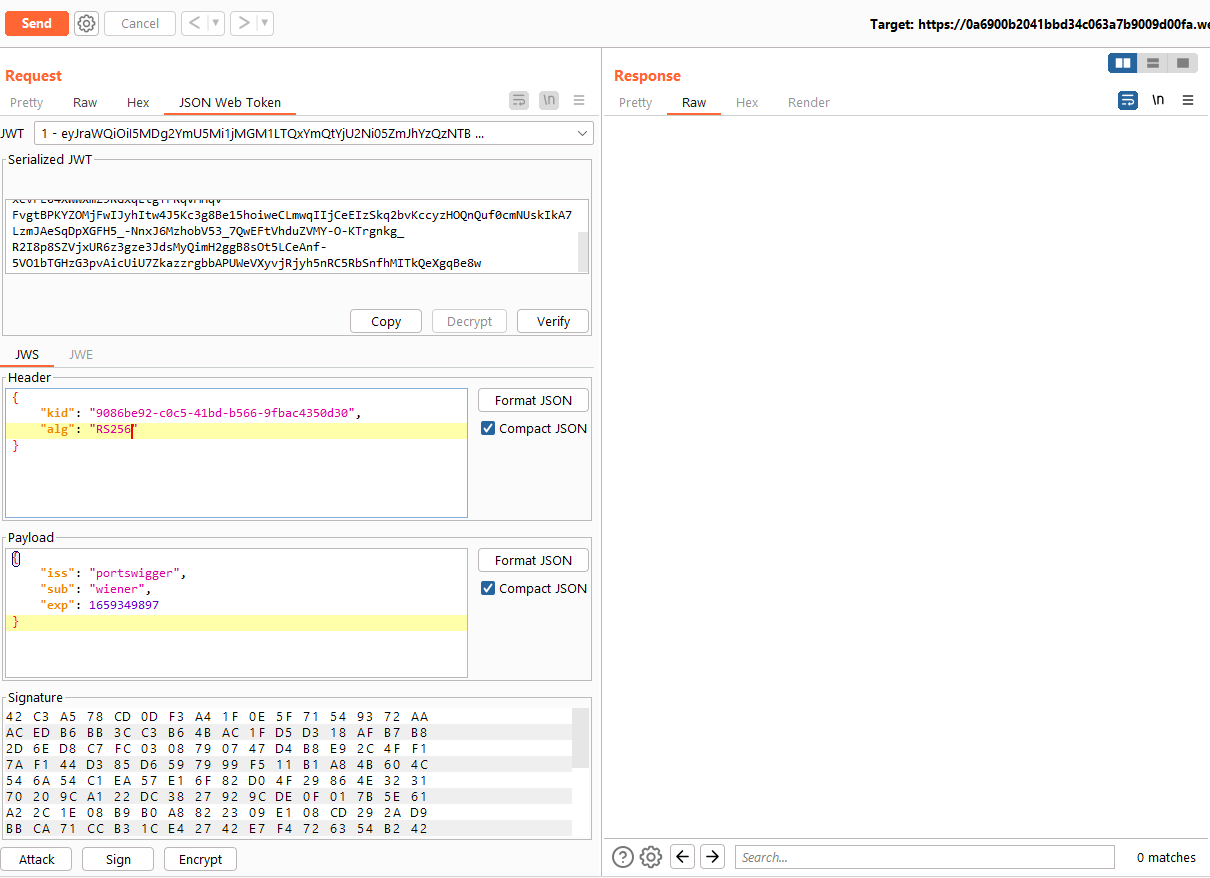
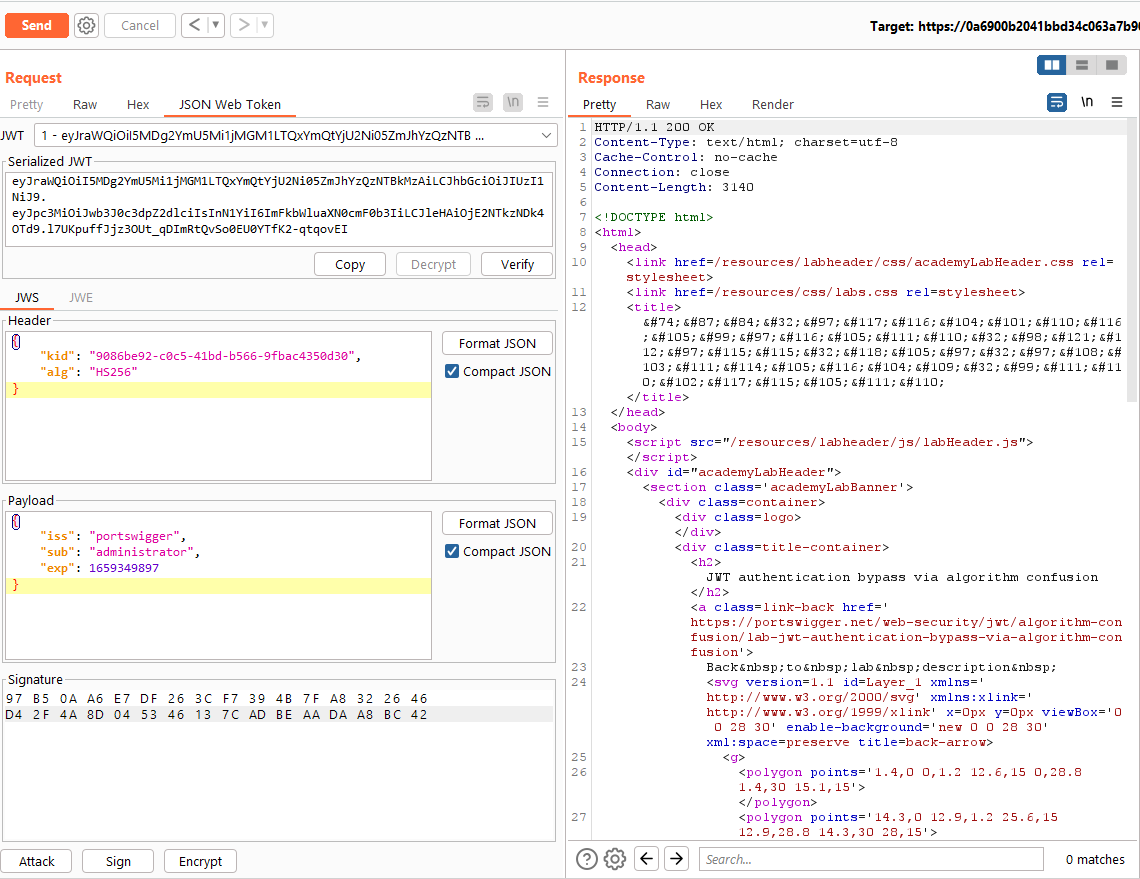
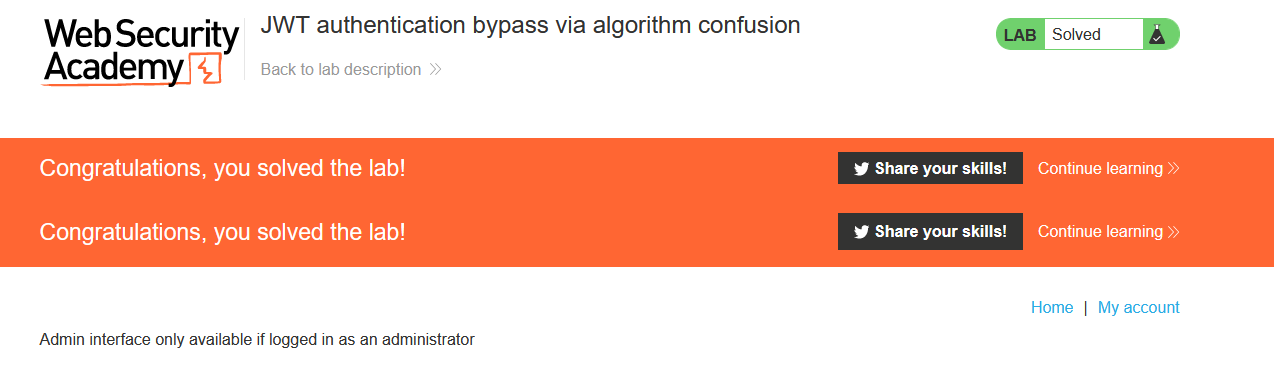
**Step 3 : Modify JWT**

Once we have public key in suitable format we can modify JWT whatever we want to do. Remember to edit alg header to **alg : HS256**

**Step 4 : Sign the JWT using the public key**

Sign the token using HS256 algorithm with our created RSA public key as a secret.

**LAB 7 :** The aim of this lab is to gain /admin panel and delete the user Carlos in a page that uses RSA key pair to sign and verify the tokens, However we can do algorithm confusion attack by changing the alg parameter on the JWT header with different algorithm (HSA256 f.e.) After the change we can use the Server’s public key as HS256’s secret by converting it to the suitable format.

1. First, we need to find a public key of the server. We will try to look for a url which contain the Public key. Some server uses standard endpoints to store public keys like **/jwks.json or /.well-known/jwks.json** . I tried /jwks.json and I get the public key value of the server.(Remember that the public key value is starts with **{“kty”:** and ends with a  **}”** just copy that value) 
2. After I get the Public key value, I need to create a new RSA key with correct format that matches with the server. I dont know the exact format that server is using so I will create RSA key with that value and will use both JWK and PEM formatted keys to sign the documents.(example Picture is in PEM format)(Just click the **Copy Public Key As PEM** option when we right clicked the key value on JWT editor extension tool) **(---Begin Public Key---- and end of public key fileds must stay and most importantly even the space character on the bottom must stay for true encoding**)
3. Take the key value from the RSA key and Base64-encode the value. After that create a Symmetric Key and change the **k** value with the value that we **Base64-encoded.** (value after the encoding) (Created Symmetric key in the picture) 
4. When I try to Access the admin page the JWT contents was like this ; 
5. I change the JWT according to my purposes and signed the JWT with Symmetric key that I have created recently. 
6. After reaching the admin panel, the rest of the steps was the same as before. 

**Deriving public keys from existing tokens**

In case where public key is not stored any open endpoint, we can still try to derive the public key from a pair of existing JWTs. This process can be made with a tool like **jwt\_forgery.py.** This script can be found in [rsa\_sign2n GitHub repository](https://github.com/silentsignal/rsa_sign2n) with other several useful scripts. Also portswigger created a simplified version of that tool can be used like this ;

docker run --rm -it portswigger/sig2n <token1> <token2>

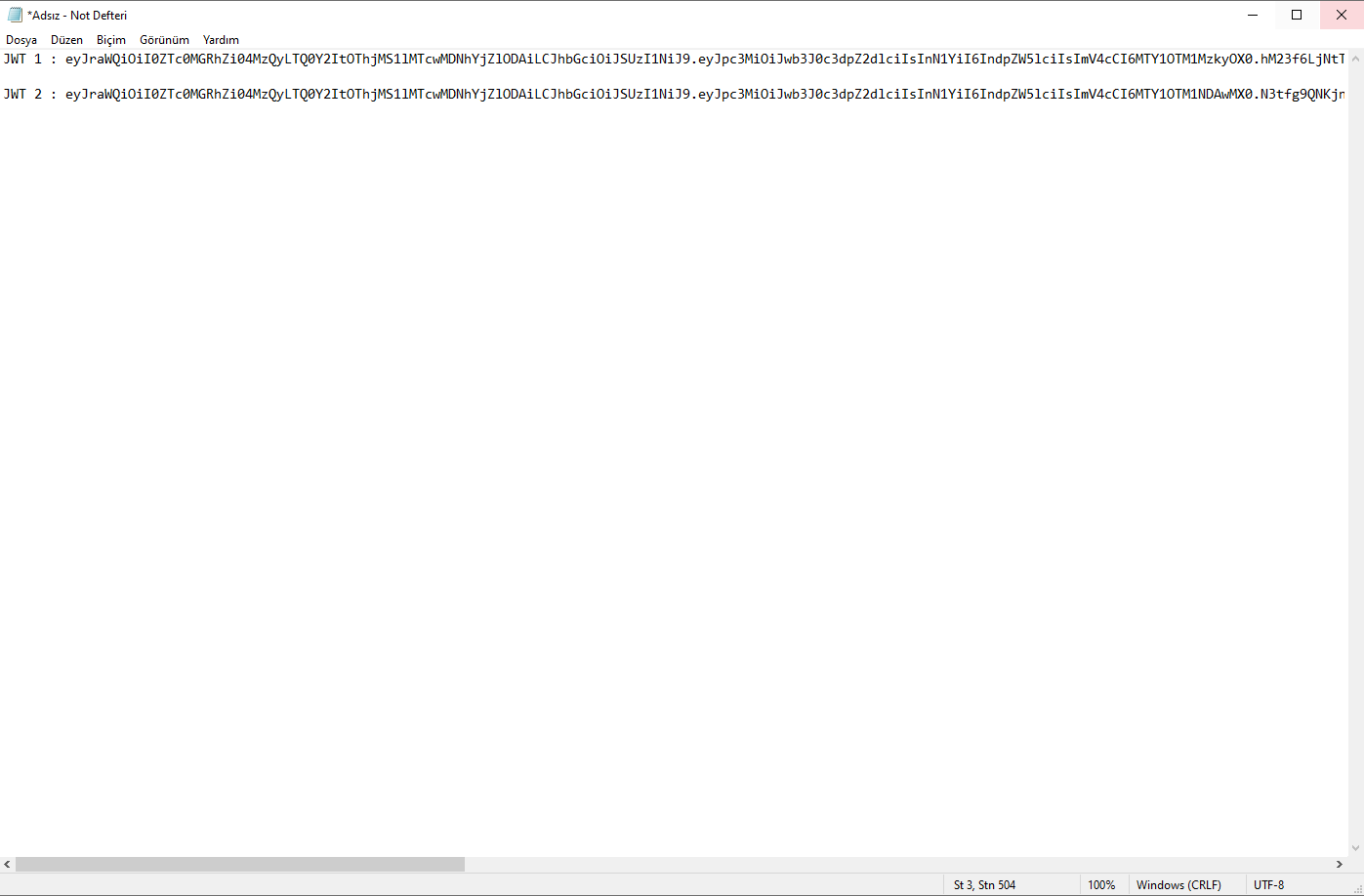
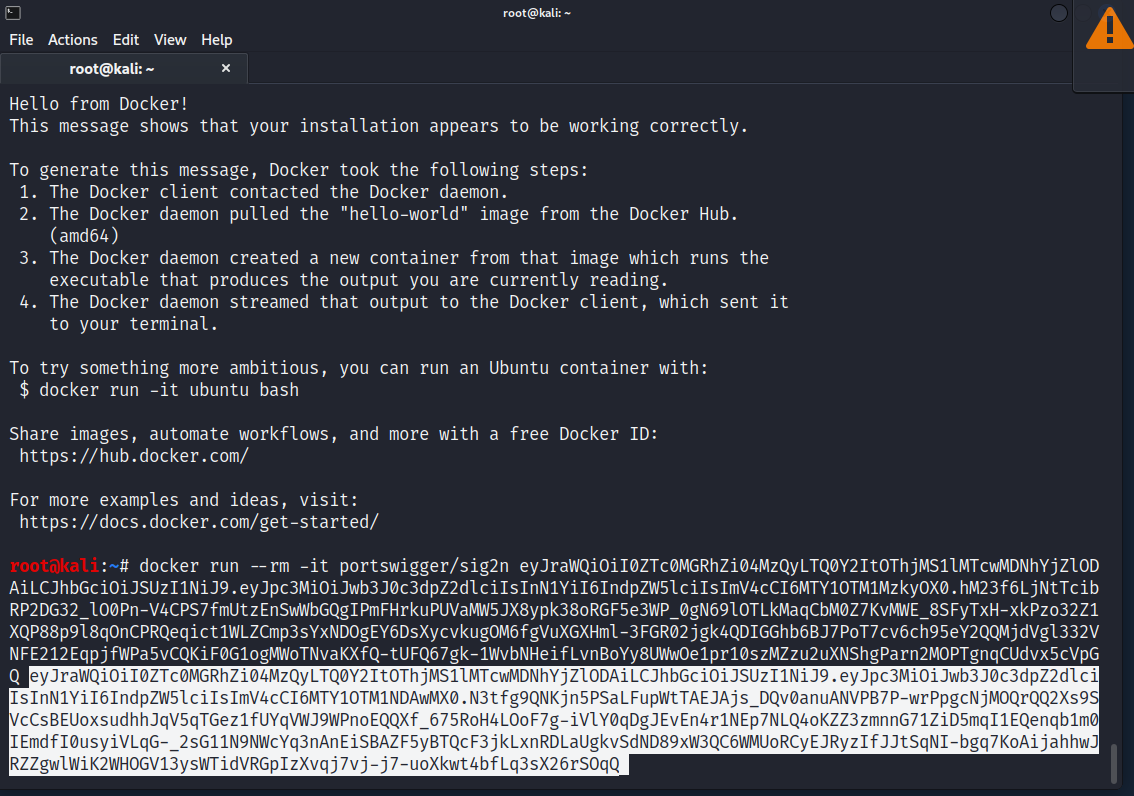
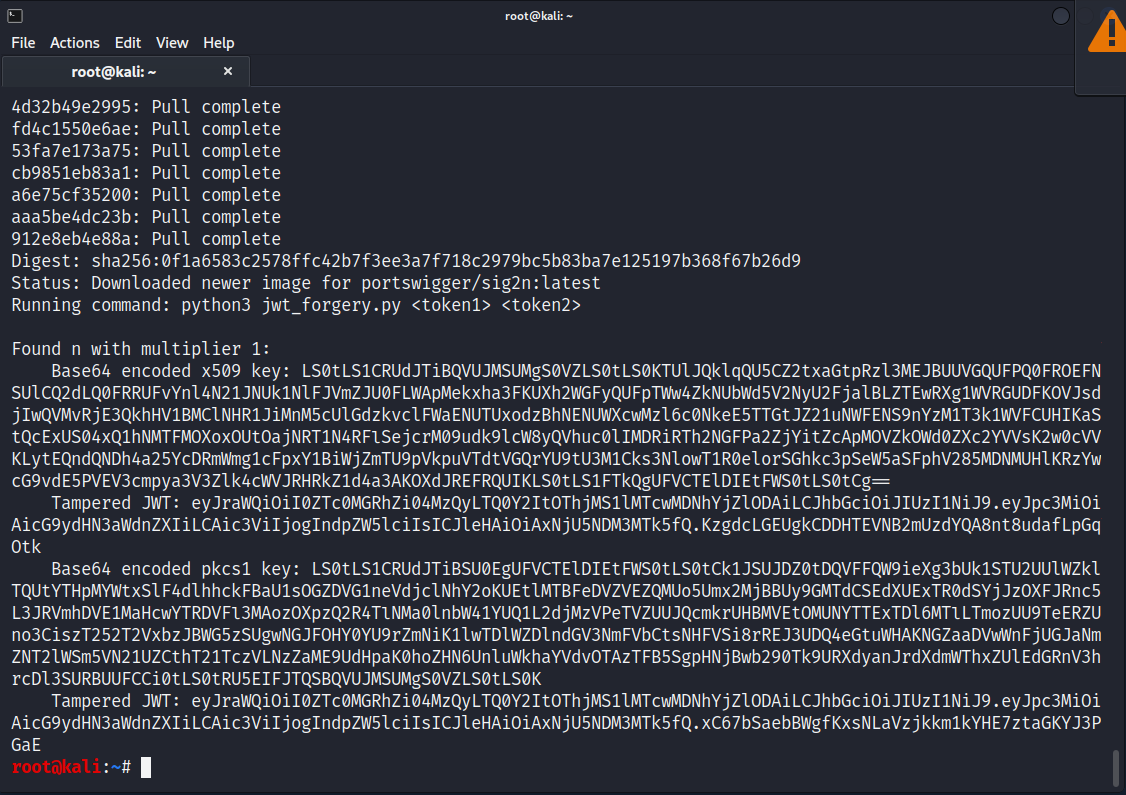
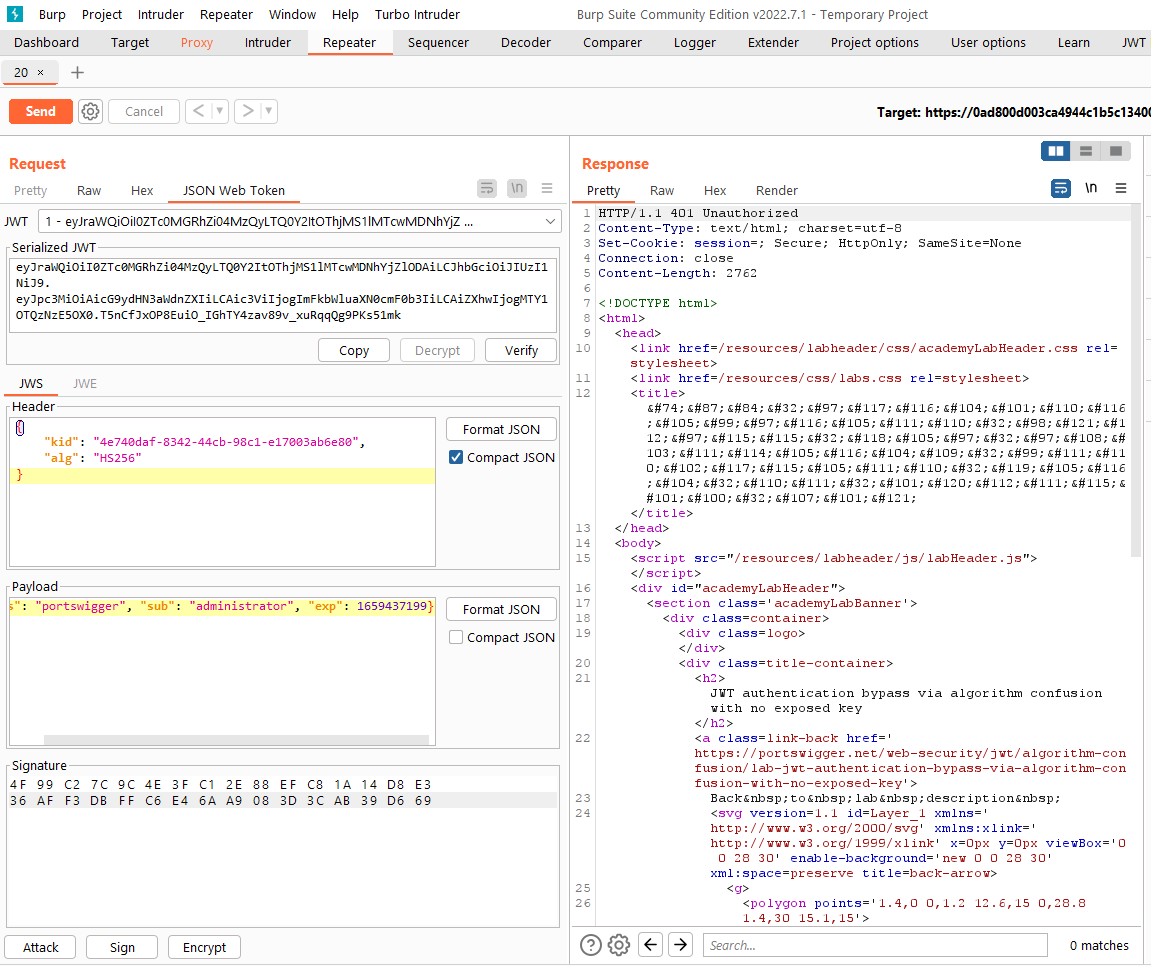
**Important Note : We need Docker CLI to run either version of the tool. When running command for the first time, it can take a few minutes**

This script uses the JWTs that we provided to calculate potential **n** values. Only one of the potential values matches with **n** used by the server’s key. For each potential value script outputs;

* A Base64-encoded PEM key in both X.509 and PKCS1 format.
* A forged JWT signed using each of these keys.

To find the true key, we need to send a request containing each of forged JWTs via Burp Repeater. Only one of the JWTs will be accepted by a server so when one of the forged JWT accepted, then we would find the correct public key value. Then we can use that key to construct an algorithm confusion attacks (go to algorithm confusion attack part and start from step 1).

**LAB 8 :** The aim of this lab is to gain /admin panel and delete the user Carlos in a page that uses RSA key pair to sign and verify the tokens, However we can do algorithm confusion attack by changing the alg parameter on the JWT header with different algorithm (HSA256 f.e.) After the change we can use the Server’s public key as HS256’s secret by converting it to the suitable format. But in this version we dont have any exposed Public Key. So we try to derive them form pair of existing JWTs via docker.

1. I copied 2 JWT sample from the 2 different login request to the page. 
2. After I get 2 samples of JWTs, I try to use the code that can derive public key from existing token (code provided via PortSwigger); 
3. This uses the JWTs that we provide to calculate one or more potential values of n. Only one of these matches the value of n used by the server's key. To identify the correct key sende requests via Repeater by using one of these JWTs. Only one of these will be accepted by the server. Then we can use the matching key of that JWT to construct algorithm confusion attacks; (The result is below page) 
4. The tampered JWTs that we tried can give 2 **HTTP Status Code as a Response**. Either HTTP 302 Redirection or HTTP 200 OK. **The Tampered JWT that gives 302 as response is the false one.** **200 HTTP OK response shows us the true Tampered JWT and also true Public key to use.** In the picture above the 1st one gives us HTTP 200 code so we will use it accordingly.(**Sometimes both can give 302, so try both of the key as Symmetric key’s k (secret) value to see which one is the true one**)(Keys provided in this script is already Base64-Encoded so we just copy and paste the value to the Symmetric key’s secret field)
5. Also when we Intercept the request(/admin) and send it to the repeater, We should have change the JWT value on the request with the Tampered JWT that we want to use it’s public key in a algorithm confusion attack. When I do this with the 2nd result 401 unauthorized response came from the server; 
6. Now I tried it with the first Tampered JWT and Public key but server return 401 not authorized error. I looked at solutions from the PortSwigger. The keys that script is returning was working in videos but for me they are not working. There could be an issue about script so I must leave this lab, because I dont have any true derived public keys or jwt tokens.

**How to prevent JWT attacks**

* Use an up-to-date libraries for JWT handling and make developers understand JWT mechanism with any security implications.(dont hardcode the JWT public keys via code etc)
* Make sure that you perform robust signature verification on any JWTs that you receive to be able to prevent JWTs signed using unexpected algorithms.
* Enforce a strict whitelist of permitted hosts for the jku header.
* Most importantly, make sure that you're not vulnerable to path traversal or SQL injection via the kid header parameter.

**How to prevent JWT attacks**

Recommended practises about JWTs;

* Always set an expiration date for any tokens that you issue (payload claimlerindeki **exp** parametresi)
* Avoid sending tokens in URL parameters where possible
* Include the aud (audience) to specify the intended recipient of the token. This prevents it from being used on different websites. (aud : client id of the application that recieved the token)
* Enable the issuer server to revoke tokens (f.e on logout).

**SC JWT Tickets**

[https://support.kron.com.tr:8443/browse/SC-5576?jql=summary%20~%20%22JWT\*%22%20OR%20description%20~%20%22JWT\*%22%20ORDER%20BY%20lastViewed%20DESC](https://support.kron.com.tr:8443/browse/SC-5576?jql=summary%20~%20%22JWT*%22%20OR%20description%20~%20%22JWT*%22%20ORDER%20BY%20lastViewed%20DESC)

<https://support.kron.com.tr:8443/browse/SC-5727>

<https://support.kron.com.tr:8443/browse/SC-5663>

<https://support.kron.com.tr:8443/browse/SC-5610>

**REFERENCES**

<https://portswigger.net/web-security/jwt>

<https://portswigger.net/web-security/all-labs#jwt>

<https://jwt.io/introduction>

<https://dev.to/kimmaida/signing-and-validating-json-web-tokens-jwt-for-everyone-25fb>