**Attacking Access control & Security Misconfigurations**

* **Security Misconfigurations**
* **Attacking Access control**
  + **Horizontal Access control** 
    - **IDOR**
  + **Vertical Access Control**
  + **Context Based Access Control**
* **Attacking Access control**
  + Access control vulnerability are very simple the web application let you do something you shouldn’t do there are 3 types of access controls
  + **Vertical access controls**
    - if a user can gain access to functionality that they are not permitted to access then this is vertical privilege escalation. For example, if a non-administrative user can in fact gain access to an admin page where they can delete user accounts, then this is vertical privilege escalation.
    - **Unprotected functionality**
      * At its most basic, vertical privilege escalation arises where an application does not enforce any protection over sensitive functionality. For example, administrative functions might be linked from an administrator's welcome page but not from a user's welcome page. However, a user might simply be able to access the administrative functions by browsing directly to the relevant admin URL.
      * For example, a website might host sensitive functionality at the following URL:
        + https://insecure-website.com/admin
      * This might in fact be accessible by any user, not only administrative users who have a link to the functionality in their user interface. In some cases, the administrative URL might be disclosed in other locations, such as the robots.txt file:
        + <https://insecure-website.com/robots.txt>
      * Admin panel of some CMS
        + /administrator on Joomla!,
        + /wp-admin/ on WordPress,
        + /index.php/admin on Magento,
        + /user/login on Drupal.
    - **Parameter-based access control methods**
      * Some applications determine the user's access rights or role at login, and then store this information in a user-controllable location, such as a hidden field, cookie, or preset query string parameter. The application makes subsequent access control decisions based on the submitted value. For example:
        + https://insecure-website.com/login/home.jsp?admin=true
        + https://insecure-website.com/login/home.jsp?role=1
      * This approach is fundamentally insecure because a user can simply modify the value and gain access to functionality to which they are not authorized, such as administrative functions.
    - **HTTP header based access control methods**
      * **Referrer based access control**
        + Some websites base access controls on the Referer header submitted in the HTTP request. The Referer header is generally added to requests by browsers to indicate the page from which a request was initiated.
        + For example, suppose an application robustly enforces access control over the main administrative page at /admin, but for sub-pages such as /admin/deleteUser only inspects the Referer header. If the Referer header contains the main /admin URL, then the request is allowed.
        + In this situation, since the Referer header can be fully controlled by an attacker, they can forge direct requests to sensitive sub-pages, supplying the required Referer header, and so gain unauthorized access.
      * **Location based access control**
        + Some web sites enforce access controls over resources based on the user's geographical location. This can apply, for example, to banking applications or media services where state legislation or business restrictions apply. These access controls can often be circumvented by the use of web proxies, VPNs, or manipulation of client-side geolocation mechanisms.
  + **Horizontal access controls**
    - Horizontal privilege escalation arises when a user is able to gain access to resources belonging to another user, instead of their own resources of that type. For example, if an employee should only be able to access their own employment and payroll records, but can in fact also access the records of other employees, then this is horizontal privilege escalation
    - Often, a horizontal privilege escalation attack can be turned into a vertical privilege escalation, by compromising a more privileged user
    - **Insecure direct object references (IDOR)**
      * Insecure direct object references (IDOR) are a type of access control vulnerability that arises when web developer uses an identifier for direct access to an internal implementation object but provides no additional access control and/or authorization checks
      * represents a vulnerable Direct Object Reference. It involves replacing the entity name such as “id”, “uid”, “pid”  in the url or cookie or any other place with a different value without the user’s authorization. As a result, users will be directed to links, pages, or sites other than the ones they intended to visit, without having the slightest clue about it.
      * **Examples** 
        + The value of a parameter is used directly to retrieve a database record.

http://foo.bar/somepage?invoice=12345

* + - * + The value of a parameter is used directly to perform an operation in the system

http://foo.bar/changepassword?userID=someuser

* + - * + The value of a parameter is used directly to retrieve a file system resource

http://foo.bar/showImage?img=img00011

* + - * + The value of a parameter is used directly to access application functionality

<http://foo.bar/accessPage?menuitem=12>

* + - * + Static file

<https://insecure-website.com/static/12144.txt>

* + - * **Tools**
        + Burp plugins

Burp Suite plugin Authz

Burp Suite plugin AuthMatrix

Burp Suite plugin Authorize

* + - * **IDOR can leads to :** 
        + **Information Disclosure:** When the attacker will have control over your account via this vulnerability, it is obvious that an attacker will be able to come across your personal information.
        + **Authentication Bypass:** As the attacker can have access to millions of account with this vulnerability, it will be a type of Authentication bypass mechanism.
        + **Alteration of Data:** An attacker may have privileges to access your data and alter it. By this, an attacker may have permission to make changes to your data, which may lead to manipulation of records.
        + **Account Takeover:** While an attacker may have multiple access to user accounts just by changing the “UID” values, this will lead to account takeover vulnerability. When one vulnerability leads to another vulnerability(like in this case), It is known as Chaining of BUGS
  + **Context-dependent access controls**
    - Many web sites implement important functions over a series of steps. This is often done when a variety of inputs or options need to be captured, or when the user needs to review and confirm details before the action is performed. For example, administrative function to update user details might involve the following steps:
      * Load form containing details for a specific user.
      * Submit changes.
      * Review the changes and confirm.
    - Sometimes, a web site will implement rigorous access controls over some of these steps, but ignore others. For example, suppose access controls are correctly applied to the first and second steps, but not to the third step. Effectively, the web site assumes that a user will only reach step 3 if they have already completed the first steps, which are properly controlled. Here, an attacker can gain unauthorized access to the function by skipping the first two steps and directly submitting the request for the third step with the required parameters
* **Security Misconfigurations**
  + **Example of some security misconfigurations**
    - Unpatched flaws
    - Default configurations
    - Unused pages
    - Unprotected files and directories
    - Unnecessary services
  + **Oauth2** 
    - **Definition**
      * OAuth stands for Open Authorization Framework and is an authorization framework that enables applications to obtain limited access to user accounts on an HTTP service, such as Facebook, GitHub, and DigitalOcean. It works by delegating user authentication to the service that hosts the user account, and authorizing third-party applications to access the user account. OAuth 2 provides authorization flows for web and desktop applications, and mobile devices.If you were ever asked by web or mobile application to give permissions to access your personal data, you have probably used OAuth 2.0.  Vulnerability in OAuth flow leads to takeover of victim account .
      * Allows a server to authenticate a user without any password. The app uses a service provider to authenticate users.  You can use OAuth 2.0 to read data of a user from another application
      * EX
        + User clicks login with Facebook.
        + User gets redirected to Facebook [facebook.com/oauth?redirect\_uri=target.com%2fcallback&state=xyz](http://facebook.com/oauth?redirect_uri=target.com/callback&state=xyz)
        + If the login is successful, he will be redirected to [target.com](http://target.com/)
    - **Oauth2 in Details**
      * **OAuth Roles : OAuth defines four roles:**
        + **Resource Owner (user)** : the *user* who authorizes an *application* to access their account.
        + **Client (The Third-Party Application )**:  The client is the application that is attempting to get access to the user's account. It needs to get permission from the user before it can do so.
        + **Resource Server** :  The resource server is the API server used to access the user's information
        + **Authorization Server** :  This is the server that presents the interface where the user approves or denies the request. In smaller implementations, this may be the same server as the API server, but larger scale deployments will often build this as a separate component
      * **Authorization**

****

* + - * + **step** **1 :** of OAuth 2 is to get authorization from the user. For browser-based or mobile apps, this is usually accomplished by displaying an interface provided by the service to the user.
        + OAuth 2 provides several "grant types" for different use cases. The grant types defined are:

**Authorization Code** :  used with server-side Applications  for apps running on a web server, browser-based and mobile apps

**Password** :  used with trusted Applications, such as those owned by the service itself  for logging in with a username and password (only for first-party apps)

**Client credentials**:  used with Applications API access  for application access without a user present

**Implicit** : used with Mobile Apps or Web Applications (applications that run on the user’s device) was previously recommended for clients without a secret, but has been superseded by using the Authorization Code grant with PKCE

Ex of  authorization code link

<https://authorization-server.com/auth?response_type=code&>client\_id=CLIENT\_ID&redirect\_uri=REDIRECT\_URI&scope=photos&state=1234zyx

**response\_type=code** - Indicates that your server expects to receive an authorization code

**client\_id** - The client ID you received when you first created the application

**redirect\_uri** - Indicates the URI to return the user to after authorization is complete

**scope** - One or more scope values indicating which parts of the user's account you wish to access

**state** - A random string generated by your application, which you'll verify later to maintain state between the request and callback. Used to prevent csrf attacks

* + - * + **Step 2:** User Authorizes Application

When the user clicks the link, they must first log in to the service, to authenticate their identity (unless they are already logged in). Then they will be prompted by the service to authorize or deny the application access to their account.

* + - * + **Step 3:** Application Receives Authorization Code

If the user clicks “Authorize Application”, the service redirects the user-agent to the application redirect URI, which was specified during the client registration, along with an authorization code. The redirect would look something like this

<https://example-app.com/cb?code=AUTH_CODE_HERE&state=1234zyx>

**code** - The server returns the authorization code in the query string

**state** - The server returns the same state value that you passed

* + - * + **Step 4**: Application Requests Access Token

The application requests an access token from the API, by passing the authorization code along with authentication details, including the client secret, to the API token endpoint

POST <https://api.authorization-server.com/token> grant\_type=authorization\_code&code=AUTH\_CODE\_HERE&redirect\_uri=REDIRECT\_URI&client\_id=CLIENT\_ID&client\_secret=CLIENT\_SECRET

grant\_type= authorization\_code - The grant type for this flow is authorization\_code

code=AUTH\_CODE\_HERE - This is the code you received in the query string

redirect\_uri=REDIRECT\_URI - Must be identical to the redirect URI provided in the original link

client\_id=CLIENT\_ID - The client ID you received when you first created the application

client\_secret=CLIENT\_SECRET - Since this request is made from server-side code, the secret is included

* + - * + **Step 5**: Application Receives Access Token

If the authorization is valid, the API will send a response containing the access token (and optionally, a refresh token) to the application. The entire response will look something like this:

{"access\_token":"ACCESS\_TOKEN","token\_type":"bearer","expires\_in":2592000,"refresh\_token":"REFRESH\_TOKEN","scope":"read","uid":100101,"info":{"name":"Mark E. Mark","email":"mark@thefunkybunch.com"}}

Now the application is authorized! It may use the token to access the user’s account via the service API, limited to the scope of access, until the token expires or is revoked. If a refresh token was issued, it may be used to request new access tokens if the original token has expired.

* + - **Testing for Oauth misconfiguration**
      * For example <https://www.readcted.com/> is the application.
      * Register to the application using email account and complete all the registration process
      * Observe if the application supports OAuth functionality service providers like Facebook and Google which you can link your social accounts to the application <https://www.redacted.com/>
      * Intercept the application using Burpsuite and Now click on Facebook icon for linking of social account to the account in <https://www.redacted.com/>
      * Observer the request and lookout for whether state parameter is implemented or not
      * If state parameter is not there which means it is vulnerable to CSRF attack
      * Once you successfully authenticated then intercept the callback request from Facebook looks like below
        + GET /auth/facebook/callback?code=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX HTTP/1.1
        + Host: [redacted.com](http://redacted.com)
      * Generate a CSRF poc on this page and save it as poc.html
      * Now create another account. which is victim account on <https://www.redacted.com/> go to the setting page where you can link the social account.
      * Now open poc.html page in the browser and click on submit button
      * Facebook account is successfully linked with victim account on <https://www.redacted.com>
      * Logout from the application and try to login from your social account
      * Successfully logged into the victim account of <https://www.redacted.com/>
    - **Attacks**
      * Attacking the ‘Connect’ request
        + This attack exploits the first , the request generated when a user clicks ‘Connect’ or ‘Sign in with’ button. Many websites allow users to connect additional accounts like Google, Facebook, Twitter, etc. 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).
        + Steps:

The attacker creates a dummy account with some Provider.

The attacker initiates the ‘Connect’ process with the Client using the dummy account on the Provider, but, stops the redirect mentioned in request 3(in the Authorization code grant flow). i.e. The attacker has granted Client access to his/her resources on the Provider but the Client has not yet been notified.

The attacker creates a malicious webpage simulating the following steps:

Logging out the user on Provider(using CSRF).

Logging in the user on Provider with the credentials of his/her dummy account(using CSRF).

Spoofing the 1st request to connect the Provider account with Client. This can be easily done, as it is just another GET request. It is preferred to do this within an iframe so that the victim is unaware of this.

When the victim visits the attacker’s page, he/she is logged out of Provider and then gets signed in as the dummy account. The ‘Connect’ request is then issued which results in the attacker’s dummy account to be connected with the victim’s account on Client. Note that the victim will not be asked for granting access to the client as the attacker has already approved it in Step 2.

Now, the attacker can log in to the victim’s account on Client by signing in with the dummy account on Provider.

* + - * + Mitigation

Although the vulnerability exists on the Provider itself(allowing CSRF log in and log out), it is even better to protect the ‘Connect’ page from allowing requests that do not originate from the user. This can be ensured by using a csrf\_token within the client to protect the 1st request. The OAuth 2.0 standard should specify this

* + - * CSRF on Authorization response
        + By performing a Cross Site Request Forgery attack, an attacker can link a dummy account on Provider with victim’s account on Client(as mentioned in the first attack). This attack uses the 3rd request of the Authorization code grant.
        + Steps:

The attacker creates a dummy account on Provider.

The attacker initiates the ‘Connect’ process with the Client using the dummy account on the Provider, but, stops the redirect mentioned in request 3(in the Authorization code grant flow). i.e. The attacker has granted Client access to his/her resources on the Provider but the Client has not yet been notified. The attacker saves the authorization\_code.

The attacker forces the victim to make a request to: https://client.com/<provider>/login?code=AUTH\_CODE. This can be easily done by making the victim opening a malicious webpage with any img or script tag with the above URL as src.

If the victim is logged in Client, the attacker’s dummy account is now connected to his/her account.

Now, the attacker can log in to the victim’s account on Client by signing in with the dummy account on Provider.

* + - * + Mitigation

OAuth 2.0 provides security against such attacks through the state parameter passed in the 2nd and 3rd request. It acts like a CSRF token. The attacker cannot forge a malicious URL without knowing the state which is user session specific. However, in the current implementation of OAuth, this parameter is NOT required and is optional. Developers not well versed with security are susceptible to ignore this.

OAuth 2.0 should force clients to send a state parameter and handle requests that are missing this parameter as ‘error requests’. Proper guidelines should also be given for generating and handling csrf tokens.

Note: Using the state parameter does not prevent the first attack mentioned above(Attacking the ‘Connect’ request

* + - * Reusing an access token - One access\_token to rule them all
        + OAuth 2.0 considers access\_token to be independent of any client. All it ensures is that an access\_token stored on the authorization server is mapped to appropriate scopes and expiration time. An access token generated for client1 can be used for client2 as well. This poses a danger to clients using the Implicit grant.
        + Steps:

The attacker creates an authentic client application client1 and registers it with a Provider.

The attacker somehow manages to get the victim use client1. Thereby, he/she has access to the access token of the victim on client1.

Assume that the victim uses client2 which further uses the Implicit grant. In Implicit grant, the authorization server redirects the user-agent to a URL such as: https://client2.com/callback#access\_token=ACCESS\_TOKEN. The attacker visits this URL with the access\_token of the client.

client2 authenticates the attacker as the victim. Hence, a single access token can be used on many different clients that use Implicit grant.

* + - * + Mitigation

Clients must ensure that the access token being used was indeed issued by them. Some OAuth server like Facebook, provide endpoints to get the \_\_ a particular access\_token was issued to: https://graph.facebook.com/app?fields=id&access\_token=ACCESS\_TOKEN.

* + - * Open Redirect in OAuth 2.0
        + The OAuth 2.0 standard specifies the following guidelines for handling errors in Authorization requests:
        + If the request fails due to a missing, invalid, or mismatching redirection URI, or if the client identifier is missing or invalid, the authorization server SHOULD inform the resource owner of the error and MUST NOT automatically redirect the user-agent to the invalid redirection URI.
        + If the resource owner denies the access request or if the request fails for reasons other than a missing or invalid redirection URI, the authorization server informs the client by adding the following parameters to the query component of the redirection URI using the “application/x-www-form-urlencoded” format, per Appendix B:
        + Some OAuth servers, misinterpret this and interchange the order of the two checks. That is, if the request fails for reasons other than redirection URI, such as invalid scope, the server informs the client by redirecting it to the URL passed by the client without validating it. This makes the OAuth server to serve as an open redirector. A possible URL crafted by the attacker can be https://provider.com/oauth/authorize?response\_type=code&client\_id=CLIENT\_ID&scope=INVALID\_SCOPE&redirect\_uri=http://attacker.com/.
        + This vulnerability was once present in Facebook, Microsoft, and Google.
        + Mitigation

The mitigation is trivial: the authorization server should first validate the redirect\_uri parameter and continue accordingly..

* + - * Attacking ‘redirect\_uri’
        + Presently, to prevent attackers using arbitrary redirect\_uri, many OAuth servers partially match this parameter with a redirect\_uri prespecified during client registration. Generally, during registration, the client specifies the domain and only those redirect\_uri on that particular domain are allowed. This becomes dangerous when an attacker is able to find a page vulnerable, to say XSS, on the client’s domain. The attacker can subsequently steal authorization\_code.
        + Steps:

The attacker is able to leak data(say through XSS) from a page on the client’s domain: https://client.com/vuln.

The attacker injects Javascript code(if XSS) on that page that sends the URL loaded in the browser(with parameters as well as fragments) to the attacker.

The attacker creates a webpage that forces the user to visit a malicious link such as: https://provider.com/oauth/authorize?client\_id=CLIENT\_ID&response\_type=code&redirect\_uri=https%3A%2F%2Fclient.com%2Fvuln

When the victim loads this link, the user-agent is redirected to https://client.com/vuln?code=CODE. This CODE is then sent to the attacker.

The attacker can use this code at his/her end to issue an access token by passing it to the authentic redirect\_uri such as https://client.com/oauth/callback?code=CODE.

This attack is even more dangerous if the authorization server supports the Implicit grant. By passing response\_type=token, the attacker can steal the token directly.

* + - * + Mitigation

To prevent the attack for Authorization code grant, OAuth already specifies the following in the standard for an access token request: The authorization server MUST:

ensure that the “redirect\_uri” parameter is present if the “redirect\_uri” parameter was included in the initial authorization request as described in Section 4.1.1, and if included ensure that their values are identical.

With this, the attacker will be unable to perform Step 5. The client will request for an access token with authentication\_code and authentic redirect\_uri which will not match with https://client.com/vuln. Hence, the authorization server will not grant an access token. However, developers rarely take this into consideration. Individually, this does not represent any real threat, but with other vulnerabilities(as mentioned above), this can lead to leaking of access tokens. Note that, this will not prevent attacking authorization servers using Implicit grant.

Another protective measure, which in my opinion is more secure and handles both the above cases is that the authorization server should whitelist a list of redirect\_uri. Also, while sanitizing this parameter, exact matches should be made instead of partial matches. Usually, clients have predefined redirect\_uri and they rarely need to change them.

* + - **Tools to use:**
      * Generate CSRF PoC in Burp Suite
      * You can use this <https://security.love/CSRF-PoC-Genorator/> to generate CSRF PoC (If you don’t have BurpSuite Pro)
    - **Resources**
      * <https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2>
      * <https://aaronparecki.com/oauth-2-simplified/>
      * <https://dhavalkapil.com/blogs/Attacking-the-OAuth-Protocol/>