**OAuth 2.0 authentication vulnerability**

**Explanation, how OAuth Works, Basics:**

While browsing the web, you've almost certainly come across sites that let you log in using your social media account. The chances are that this feature is built using the popular OAuth 2.0 framework. OAuth 2.0 is highly interesting for attackers because it is both extremely common and inherently prone to implementation mistakes. This can result in a number of vulnerabilities, allowing attackers to obtain sensitive user data and potentially bypass authentication completely.

In this section, we'll teach you how to identify and exploit some of the key [vulnerabilities found in OAuth 2.0 authentication](https://portswigger.net/web-security/oauth#exploiting-oauth-authentication-vulnerabilities) mechanisms. Don't worry if you're not too familiar with OAuth authentication - we've provided plenty of background information to help you understand the key concepts you'll need. We'll also explore some [vulnerabilities in OAuth's OpenID Connect extension](https://portswigger.net/web-security/oauth/openid#openid-connect-vulnerabilities). Finally, we've included some guidance on how to [protect your own applications](https://portswigger.net/web-security/oauth/preventing) against these kinds of attacks.

**What is OAuth?**

OAuth is a commonly used authorization framework that enables websites and web applications to request limited access to a user's account on another application. Crucially, OAuth allows the user to grant this access without exposing their login credentials to the requesting application. This means users can fine-tune which data they want to share rather than having to hand over full control of their account to a third party.

The basic OAuth process is widely used to integrate third-party functionality that requires access to certain data from a user's account. For example, an application might use OAuth to request access to your email contacts list so that it can suggest people to connect with. However, the same mechanism is also used to provide third-party authentication services, allowing users to log in with an account that they have with a different website.

\*\*Although OAuth 2.0 is the current standard, some websites still use the legacy version 1a. OAuth 2.0 was written from scratch rather than being developed directly from OAuth 1.0. As a result, the two are very different. Please be aware that the term "OAuth" refers exclusively to OAuth 2.0 throughout these materials.

**How does OAuth 2.0 work?**

OAuth 2.0 was originally developed as a way of sharing access to specific data between applications. It works by defining a series of interactions between three distinct parties, namely a client application, a resource owner, and the OAuth service provider.

* **Client application** - The website or web application that wants to access the user's data.
* **Resource owner** - The user whose data the client application wants to access.
* **OAuth service provider** - The website or application that controls the user's data and access to it. They support OAuth by providing an API for interacting with both an authorization server and a resource server.

There are numerous different ways that the actual OAuth process can be implemented. These are known as OAuth "flows" or "grant types". In this topic, we'll focus on the "authorization code" and "implicit" grant types as these are by far the most common. Broadly speaking, both of these grant types involve the following stages:

1. The client application requests access to a subset of the user's data, specifying which grant type they want to use and what kind of access they want.
2. The user is prompted to log in to the OAuth service and explicitly give their consent for the requested access.
3. The client application receives a unique access token that proves they have permission from the user to access the requested data. Exactly how this happens varies significantly depending on the grant type.
4. The client application uses this access token to make API calls fetching the relevant data from the resource server.

**OAuth Grants/Flows** – vital piece of OAuth which is necessary in learning how to exploit it.

What is OAuth Grant type?

The OAuth grant type determines the exact sequence of steps that are involved in the OAuth process. The grant type also affects how the client application communicates with the OAuth service at each stage, including how the access token itself is sent. For this reason, grant types are often referred to as "OAuth flows".

An OAuth service must be configured to support a particular grant type before a client application can initiate the corresponding flow. The client application specifies which grant type it wants to use in the initial authorization request it sends to the OAuth service.

There are several different grant types, each with varying levels of complexity and security considerations. We'll focus on the "**authorization code**" and "**implicit**" grant types as these are by far the most common.

**OAuth Scopes**

For any OAuth grant type, the client application has to specify which data it wants to access and what kind of operations it wants to perform. It does this using the scope parameter of the authorization request it sends to the OAuth service.

For basic OAuth, the scopes for which a client application can request access are unique to each OAuth service. As the name of the scope is just an arbitrary text string, the format can vary dramatically between providers. Some even use a full URI as the scope name, similar to a REST API endpoint. For example, when requesting read access to a user's contact list, the scope name might take any of the following forms depending on the OAuth service being used:

scope=contacts

scope=contacts.read

scope=contact-list-r

scope=https://oauth-authorization-server.com/auth/scopes/user/contacts.readonly

When OAuth is used for authentication, however, the standardized OpenID Connect scopes are often used instead. For example, the scope openid profile will grant the client application read access to a predefined set of basic information about the user, such as their email address, username, and so on. We'll talk more about [OpenID Connect](https://portswigger.net/web-security/oauth/openid) later.

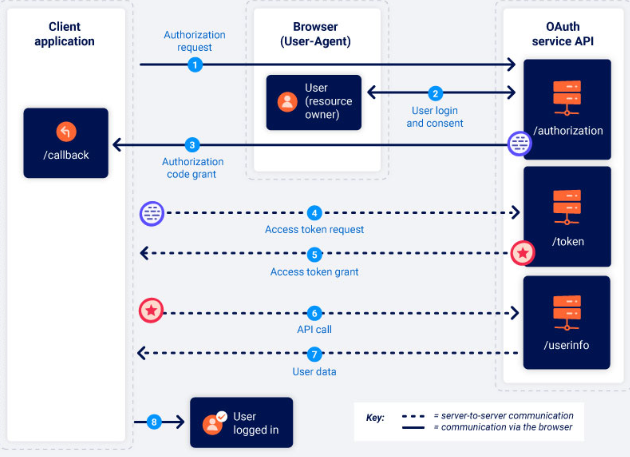
**Authorization Code Grant Type:**

The authorization code grant type initially looks quite complicated, but it's actually simpler than you think once you're familiar with a few basics.

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

All communication that takes place from the code/token exchange onward is sent server-to-server over a secure, preconfigured back-channel and is, therefore, invisible to the end user. This secure channel is established when the client application first registers with the OAuth service. At this time, a client\_secret is also generated, which the client application must use to authenticate itself when sending these server-to-server requests.

As the most sensitive data (the access token and user data) is not sent via the browser, this grant type is **arguably the most secure**. Server-side applications should ideally always use this grant type if possible.



**Authorization Code Steps Explained (steps from the above pic explained in detail):**

#### 1. Authorization request

The client application sends a request to the OAuth service's /authorization endpoint asking for permission to access specific user data. Note that the endpoint mapping may vary between providers - our labs use the endpoint /auth for this purpose. However, you should always be able to identify the endpoint based on the parameters used in the request.

GET /authorization?client\_id=12345&redirect\_uri=https://client-app.com/callback&response\_type=code&scope=openid%20profile&state=ae13d489bd00e3c24 HTTP/1.1

Host: oauth-authorization-server.com

This request contains the following noteworthy parameters, usually provided in the query string:

* client\_id

Mandatory parameter containing the unique identifier of the client application. This value is generated when the client application registers with the OAuth service.

* redirect\_uri

The URI to which the user's browser should be redirected when sending the authorization code to the client application. This is also known as the "callback URI" or "callback endpoint". Many OAuth attacks are based on exploiting flaws in the validation of this parameter.

* response\_type

Determines which kind of response the client application is expecting and, therefore, which flow it wants to initiate. For the authorization code grant type, the value should be code.

* scope

Used to specify which subset of the user's data the client application wants to access. Note that these may be custom scopes set by the OAuth provider or standardized scopes defined by the OpenID Connect specification. We'll cover [OpenID Connect](https://portswigger.net/web-security/oauth/openid) in more detail later.

* state

Stores a unique, unguessable value that is tied to the current session on the client application. The OAuth service should return this exact value in the response, along with the authorization code. This parameter serves as a form of [CSRF](https://portswigger.net/web-security/csrf) token for the client application by making sure that the request to its /callback endpoint is from the same person who initiated the OAuth flow.

#### 2. User login and consent

When the authorization server receives the initial request, it will redirect the user to a login page, where they will be prompted to log in to their account with the OAuth provider. For example, this is often their social media account.

They will then be presented with a list of data that the client application wants to access. This is based on the scopes defined in the authorization request. The user can choose whether or not to consent to this access.

It is important to note that once the user has approved a given scope for a client application, this step will be completed automatically as long as the user still has a valid session with the OAuth service. In other words, the first time the user selects "Log in with social media", they will need to manually log in and give their consent, but if they revisit the client application later, they will often be able to log back in with a single click.

#### 3. Authorization code grant

If the user consents to the requested access, their browser will be redirected to the /callback endpoint that was specified in the redirect\_uri parameter of the authorization request. The resulting GET request will contain the authorization code as a query parameter. Depending on the configuration, it may also send the state parameter with the same value as in the authorization request.

GET /callback?code=a1b2c3d4e5f6g7h8&state=ae13d489bd00e3c24 HTTP/1.1

Host: client-app.com

**4. Access token request**

Once the client application receives the authorization code, it needs to exchange it for an access token. To do this, it sends a server-to-server POST request to the OAuth service's /token endpoint. All communication from this point on takes place in a secure back-channel and, therefore, cannot usually be observed or controlled by an attacker.

POST /token HTTP/1.1

Host: oauth-authorization-server.com

…

client\_id=12345&client\_secret=SECRET&redirect\_uri=https://client-app.com/callback&grant\_type=authorization\_code&code=a1b2c3d4e5f6g7h8

In addition to the client\_id and authorization code, you will notice the following new parameters:

* client\_secret

The client application must authenticate itself by including the secret key that it was assigned when registering with the OAuth service.

* grant\_type

Used to make sure the new endpoint knows which grant type the client application wants to use. In this case, this should be set to authorization\_code.

**5. Access token grant**

The OAuth service will validate the access token request. If everything is as expected, the server responds by granting the client application an access token with the requested scope.

{

"access\_token": "z0y9x8w7v6u5",

"token\_type": "Bearer",

"expires\_in": 3600,

"scope": "openid profile",

…

}

#### 6. API call

Now the client application has the access code, it can finally fetch the user's data from the resource server. To do this, it makes an API call to the OAuth service's /userinfo endpoint. The access token is submitted in the Authorization: Bearer header to prove that the client application has permission to access this data.

GET /userinfo HTTP/1.1

Host: oauth-resource-server.com

Authorization: Bearer z0y9x8w7v6u5

#### 7. Resource grant

The resource server should verify that the token is valid and that it belongs to the current client application. If so, it will respond by sending the requested resource i.e. the user's data based on the scope of the access token.

{

"username":"carlos",

"email":"carlos@carlos-montoya.net",

…

}

The client application can finally use this data for its intended purpose. In the case of OAuth authentication, it will typically be used as an ID to grant the user an authenticated session, effectively logging them in.

**Implicit Grant Type:**

The implicit grant type is much simpler. Rather than first obtaining an authorization code and then exchanging it for an access token, the client application receives the access token immediately after the user gives their consent.

You may be wondering why client applications don't always use the implicit grant type. The answer is relatively simple - it is far less secure. When using the implicit grant type, all communication happens via browser redirects - there is no secure back-channel like in the authorization code flow. This means that the sensitive access token and the user's data are more exposed to potential attacks.

The implicit grant type is more suited to single-page applications and native desktop applications, which cannot easily store the client\_secret on the back-end, and therefore, don't benefit as much from using the authorization code grant type.

A screenshot of a computer

Description automatically generated

**Implicit Grant Steps:**

#### 1. Authorization request

The implicit flow starts in much the same way as the authorization code flow. The only major difference is that the response\_type parameter must be set to token.

GET /authorization?client\_id=12345&redirect\_uri=https://client-app.com/callback&response\_type=token&scope=openid%20profile&state=ae13d489bd00e3c24 HTTP/1.1

Host: oauth-authorization-server.com

#### 2. User login and consent

The user logs in and decides whether to consent to the requested permissions or not. This process is exactly the same as for the authorization code flow.

#### 3. Access token grant

If the user gives their consent to the requested access, this is where things start to differ. The OAuth service will redirect the user's browser to the redirect\_uri specified in the authorization request. However, instead of sending a query parameter containing an authorization code, it will send the access token and other token-specific data as a URL fragment.

GET /callback#access\_token=z0y9x8w7v6u5&token\_type=Bearer&expires\_in=5000&scope=openid%20profile&state=ae13d489bd00e3c24 HTTP/1.1

Host: client-app.com

As the access token is sent in a URL fragment, it is never sent directly to the client application. Instead, the client application must use a suitable script to extract the fragment and store it.

**4. API call**

Once the client application has successfully extracted the access token from the URL fragment, it can use it to make API calls to the OAuth service's /userinfo endpoint. Unlike in the authorization code flow, this also happens via the browser.

GET /userinfo HTTP/1.1

Host: oauth-resource-server.com

Authorization: Bearer z0y9x8w7v6u5

**5. Resource grant**

The resource server should verify that the token is valid and that it belongs to the current client application. If so, it will respond by sending the requested resource i.e. the user's data based on the scope associated with the access token.

{

"username":"carlos",

"email":"carlos@carlos-montoya.net"

}

The client application can finally use this data for its intended purpose. In the case of OAuth authentication, it will typically be used as an ID to grant the user an authenticated session, effectively logging them in.

**OAuth Authentication:**

Although not originally intended for this purpose, OAuth has evolved into a means of authenticating users as well. For example, you're probably familiar with the option many websites provide to log in using your existing social media account rather than having to register with the website in question. Whenever you see this option, there's a good chance it is built on OAuth 2.0.

For OAuth authentication mechanisms, the basic OAuth flows remain largely the same; the main difference is how the client application uses the data that it receives. From an end-user perspective, the result of OAuth authentication is something that broadly resembles SAML-based single sign-on (SSO). In these materials, we'll focus exclusively on vulnerabilities in this SSO-like use case.

OAuth authentication is generally implemented as follows:

1. The user chooses the option to log in with their social media account. The client application then uses the social media site's OAuth service to request access to some data that it can use to identify the user. This could be the email address that is registered with their account, for example.
2. After receiving an access token, the client application requests this data from the resource server, typically from a dedicated /userinfo endpoint.
3. Once it has received the data, the client application uses it in place of a username to log the user in. The access token that it received from the authorization server is often used instead of a traditional password.

**How do OAuth Vulnerabilities arise?**

OAuth [authentication vulnerabilities](https://portswigger.net/web-security/authentication) arise partly because the OAuth specification is relatively vague and flexible by design. Although there are a handful of mandatory components required for the basic functionality of each grant type, the vast majority of the implementation is completely optional. This includes many configuration settings that are necessary for keeping users' data secure. In short, there's plenty of opportunity for bad practice to creep in.

One of the other key issues with OAuth is the general lack of built-in security features. The security relies almost entirely on developers using the right combination of configuration options and implementing their own additional security measures on top, such as robust input validation. As you've probably gathered, there's a lot to take in and this is quite easy to get wrong if you're inexperienced with OAuth.

Depending on the grant type, highly sensitive data is also sent via the browser, which presents various opportunities for an attacker to intercept it.

**Identifying OAuth Authentication:**

Recognizing when an application is using OAuth authentication is relatively straightforward. If you see an option to log in using your account from a different website, this is a strong indication that OAuth is being used.

The most reliable way to identify OAuth authentication is to proxy your traffic through Burp and check the corresponding HTTP messages when you use this login option. Regardless of which OAuth grant type is being used, the first request of the flow will always be a request to the /authorization endpoint containing a number of query parameters that are used specifically for OAuth. In particular, keep an eye out for the client\_id, redirect\_uri, and response\_type parameters. For example, an authorization request will usually look something like this:

GET /authorization?client\_id=12345&redirect\_uri=https://client-app.com/callback&response\_type=token&scope=openid%20profile&state=ae13d489bd00e3c24 HTTP/1.1

Host: oauth-authorization-server.com

**Recon:**

Doing some basic recon of the OAuth service being used can point you in the right direction when it comes to identifying vulnerabilities.

It goes without saying that you should study the various HTTP interactions that make up the OAuth flow - we'll go over some specific things to look out for later. If an external OAuth service is used, you should be able to identify the specific provider from the hostname to which the authorization request is sent. As these services provide a public API, there is often detailed documentation available that should tell you all kinds of useful information, such as the exact names of the endpoints and which configuration options are being used.

Once you know the hostname of the authorization server, you should always try sending a GET request to the following standard endpoints:

* /.well-known/oauth-authorization-server
* /.well-known/openid-configuration

These will often return a JSON configuration file containing key information, such as details of additional features that may be supported. This will sometimes tip you off about a wider attack surface and supported features that may not be mentioned in the documentation.

**Exploiting OAuth Vulnerabilities:**

Vulnerabilities can arise in the client application's implementation of OAuth as well as in the configuration of the OAuth service itself. In this section, we'll show you how to exploit some of the most common vulnerabilities in both of these contexts.

* Vulnerabilities in the client application
  + [Improper implementation of the implicit grant type](https://portswigger.net/web-security/oauth#improper-implementation-of-the-implicit-grant-type) LABS
  + [Flawed CSRF protection](https://portswigger.net/web-security/oauth#flawed-csrf-protection) LABS
* Vulnerabilities in the OAuth service
  + [Leaking authorization codes and access tokens](https://portswigger.net/web-security/oauth#leaking-authorization-codes-and-access-tokens) LABS
  + [Flawed scope validation](https://portswigger.net/web-security/oauth#flawed-scope-validation)
  + [Unverified user registration](https://portswigger.net/web-security/oauth#unverified-user-registration)

### Vulnerabilities in the OAuth client application

* Client applications will often use a reputable, battle-hardened OAuth service that is well protected against widely known exploits. However, their own side of the implementation may be less secure.
* As we've already mentioned, the OAuth specification is relatively loosely defined. This is especially true with regard to the implementation by the client application. There are a lot of moving parts in an OAuth flow, with many optional parameters and configuration settings in each grant type, which means there's plenty of scope for misconfigurations.

#### Improper implementation of the implicit grant type

Due to the dangers introduced by sending access tokens via the browser, the [implicit grant type](https://portswigger.net/web-security/oauth/grant-types#implicit-grant-type) is mainly recommended for single-page applications. However, it is also often used in classic client-server web applications because of its relative simplicity.

In this flow, the access token is sent from the OAuth service to the client application via the user's browser as a URL fragment. The client application then accesses the token using JavaScript. The trouble is, if the application wants to maintain the session after the user closes the page, it needs to store the current user data (normally a user ID and the access token) somewhere.

To solve this problem, the client application will often submit this data to the server in a POST request and then assign the user a session cookie, effectively logging them in. This request is roughly equivalent to the form submission request that might be sent as part of a classic, password-based login. However, in this scenario, the server does not have any secrets or passwords to compare with the submitted data, which means that it is implicitly trusted.

In the implicit flow, this POST request is exposed to attackers via their browser. As a result, this behavior can lead to a serious vulnerability if the client application doesn't properly check that the access token matches the other data in the request. In this case, an attacker can simply change the parameters sent to the server to impersonate any user.

**\*\* To test for this we would need to create 2 users and record their auth data, try to auth in with one user and swap details from the other. This should be done whenever we come across an implicit OAuth meaning authentication details are being sent via the browser\*\***

**Flawed CSRF Protection:**

Although many components of the OAuth flows are optional, some of them are strongly recommended unless there's an important reason not to use them. One such example is the state parameter.

The state parameter should ideally contain an unguessable value, such as the hash of something tied to the user's session when it first initiates the OAuth flow. This value is then passed back and forth between the client application and the OAuth service as a form of CSRF token for the client application. Therefore, if you notice that the authorization request does not send a state parameter, this is extremely interesting from an attacker's perspective. It potentially means that they can initiate an OAuth flow themselves before tricking a user's browser into completing it, similar to a traditional [CSRF attack](https://portswigger.net/web-security/csrf). This can have severe consequences depending on how OAuth is being used by the client application.

Consider a website that allows users to log in using either a classic, password-based mechanism or by linking their account to a social media profile using OAuth. In this case, if the application fails to use the state parameter, an attacker could potentially hijack a victim user's account on the client application by binding it to their own social media account.

Note that if the site allows users to log in exclusively via OAuth, the state parameter is arguably less critical. However, not using a state parameter can still allow attackers to construct login CSRF attacks, whereby the user is tricked into logging in to the attacker's account.

**\*\* So if we are proxying OAuth traffic and notice lack of state header we should examine it very carefully and try one of these traditional CSRF attacks. \*\***

**A lot more here that I Didn’t get to see** [**https://portswigger.net/web-security/oauth**](https://portswigger.net/web-security/oauth) **for more.**