Implementation of HKID/HKU OAuth 2.0 Server for Real Identity Verification

Name: Chiew Lik Seng (3035767487)

Supervisor: Dr. Chan, Hubert T. H.

1 Introduction

In this era of globalisation and technology advances, Internet is almost accessible to anyone and anywhere where civilizations exist. The widespread proliferation of internet access throughout many societies worldwide has facilitated not only socially positive practices, but also detrimental ones. There are countless users on the web that remain their anonymity and participate in online discussions without fear of reprisal in the physical world. It is just the nature of Internet. But in recent years, the norm has gradually been broken. We can see more and more applications or services require real name verification before we can access the service for instance the registration process of student Octopus card, booking process of community sports centre, etc. Besides that, countries like China even imposed the real-name registration rules to stabilize public order [1]. The Republic of Korea, the nation with the highest rate of high-speed internet access in the world, created its own “Real Name Verification System” as well by forcing Korean citizens to verify their identities on a public institution website [2].

In Hong Kong, users are generally required to scan their HKID (Hong Kong Identity Card) by their phone camera in these real identity verification procedures. It is annoying that the process is not essentially accurate in times since factors like lighting, shadow and camera resolution could affect the outcome of the verification process. Other than that, the same process is repeated whenever real name verification is required on any other web services. Thus, some users just quit attempting to use certain services.

To better facilitate the real identity verification process, OAuth framework has come across my mind as an auth account basically represents a user’s identity in the scope of client application and HKID is no different from an auth account in real life. Each citizen would have his/her own user account on a resource server regulated by a trusted authority which would be the government and all the personal information regarding the account owner is contained within the resource server. Whenever users log into any third part web applications via OAuth 2.0 protocol, by providing the correct login credentials to the government server, an acknowledgement would be returned by the server and it could serve the same purpose as the HKID and extend to even more possibilities. In this way, identity theft on the internet will be eliminated even if your HKID is stolen by someone else as OAuth 2.0 protocol is extremely secure with correct implementation unless users themselves leak their passwords.

2 Aims and Objectives

2.1 Aims

This project aims to develop an OAuth 2.0 server for real identity verification and investigate the possible extension and contribute to the advancement of knowledge in the field of authentication, cybersecurity, and delegated access control.

2.2 Objectives

The objectives of this project are:

1. To understand the OAuth 2.0 protocol and its underlying principles.
2. To study different OAuth 2.0 grant type (e.g. Authorization Code Flow, Implicit Flow, Resource Owner Password Credentials Flow, and Client Credentials Flow) and pick the most suitable one for the project.
3. To explore the integration of OAuth 2.0 with existing web and mobile applications by incorporating SSO capabilities and delegated access.
4. To inspect the security considerations and best practices in OAuth 2.0, for instance secure token handling, client authentication and protection against common attacks.

3 Literature Review

3.1 OAuth 2.0 Framework

Open Authorization (OAuth) is a widely deployed authorization framework that allows users (known as "resource owner") to grant limited access to their resources on one website (known as the "resource server") to another website or application (known as the "client") without sharing their credentials [3]. In this way, users do not need to create an auth account for every single new client application. The risk of leaking users' login credentials is also minimized as OAuth relies on short-lived access tokens and third-party client application will only be communicating with the resource server via an access token. In general, there will be not less than 2 endpoints in the authorization server. The two mandatory endpoints are authorization endpoint and token endpoint. The authorization endpoint communicates with the resource owner and returns an authorization grant/code whereas the token endpoint interacts with the client application where client presents the authorization grant to exchange for the access token.

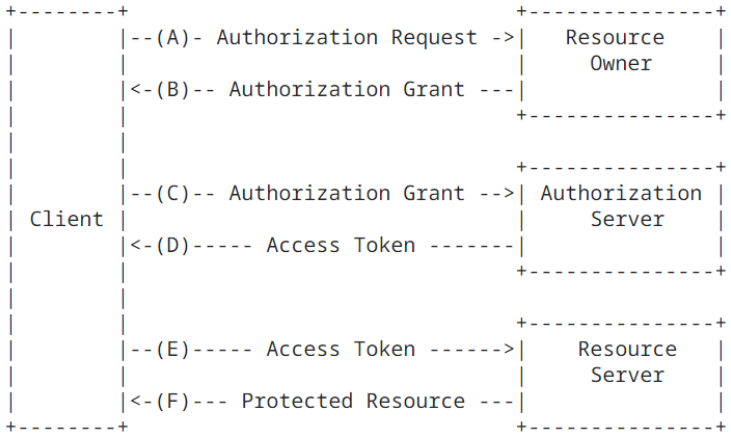

Figure 1: Abstract Protocol Flow [3]

OAuth 2.0 defines four main authorization grant types: authorization code, implicit, resource owner password credentials, and client credentials. As for now, authorization code flow would be our choice of authorization grant as it provides an additional layer of security compared to other grant types. It involves an extra step of exchanging authorization code between the client and the authorization server which better enhance the security of the process [3]. The code is short-lived as well as the token and it needs to be served together with appropriate client credentials like client secret for successful exchange of the token. Besides that, the user is redirected to the authorization server's login page as they are providing their login credentials. This makes sure that users' login credentials are never exposed to the client application and in turn reassure users and earn their trusts to select the OAuth login method.

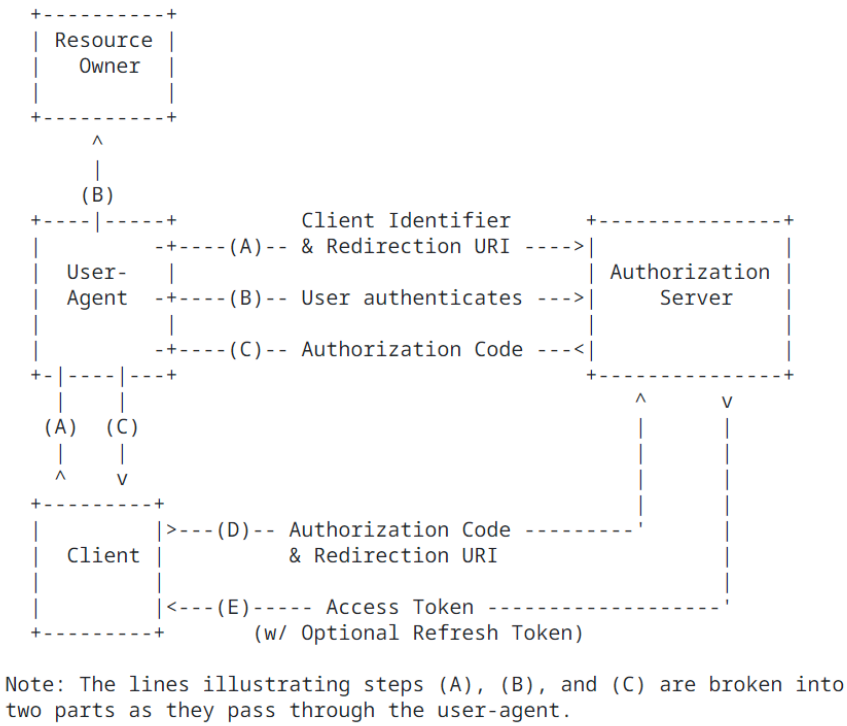

Figure 2: Authorization Code Flow [3]

The flow illustrated in Figure 2 includes the following steps:

(A) The flow starts with the client directing the resource owner' user-agent (in this case the browser) to the authorization endpoint in authorization server. The client identifier, requested scope, local state and a redirection URI is sent along.

(B) The resource owner is authenticated by the authorization server via the browser and resource owner is asked for its permission to grant the client's access request.

(C) Once access is granted by the owner, the user-agent is redirected back to client following the redirection URI mentioned earlier and the authorization code is included in the URI.

(D) The client then makes a request for the access token by presenting the appropriate authorization code and the redirection URI to the authorization server.

(E) The authorization server authenticates the client by validating the authorization code and verifying that the redirection URI is the one used to exchange for authorization code in the previous steps. The authorization server returns an access token if the authentication is valid.

The communication route to the authorization server is built upon Transport Layer Security (TLS) hence it greatly enhances the security of the framework.

3.2 Proof Key for Code Exchange

As illustrated in the previous section, we can see that the correct implementation of Authorization Code Flow is sufficiently secure, but there is still a common security weakness for OAuth 2.0 public clients signing in with Authorization Code Grant such that it is prone to authorization code interception attack [4].

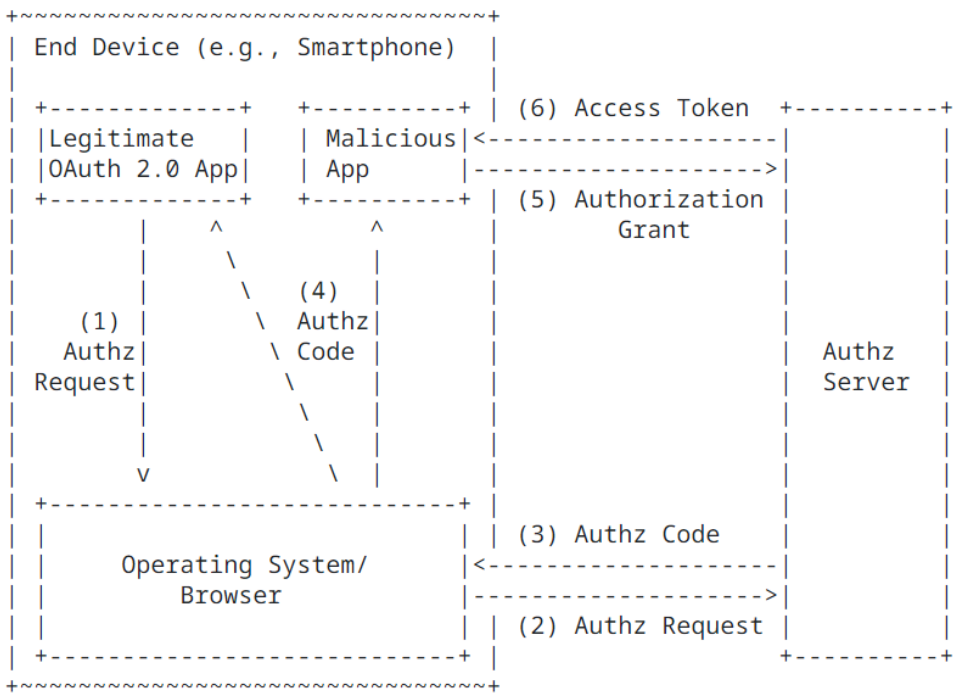

Figure 3: Authorization Code Interception Attack [4]

As shown in Figure 3 above, if a malicious application is planted on the client device, when the authorization code is sent back to the client application via the browser in step (4), the malicious application could intercept the authorization code and in turn request and obtain an access token by going through steps (5) and (6) respectively. As the communication path in steps (1) and (2) is directed to the authorization server, if OAuth Protocol is implemented correctly with TLS version, the transport channel is secure from any attempts to intercept [5].

To counter against this kind of attack, we have decided to utilize an extension to the Authorization Code flow called Proof Key for Code Exchange (PKCE), pronounced as ‘pixey’. PKCE basically creates a randomly-generated cryptographical key called “code verifier” for every authorization request and then the code verifier is transformed into a value called “code challenge” [4]. The “code challenge” derived from the “code verifier” is then sent to the authorization endpoint of the authorization server along with the request for the authorization code. The authorization code is then sent to the token endpoint along with the “code verifier”. Afterwards, the authorization server compares the “code verifier” and “code challenge” to verify that the client requesting for access token is the same as the one asking for the authorization code. This could effectively mitigate the interception attack as the attacker would not know this one-time key as it is sent over TLS. The abstract flow utilizing PKCE can be visualized in Figure 4.

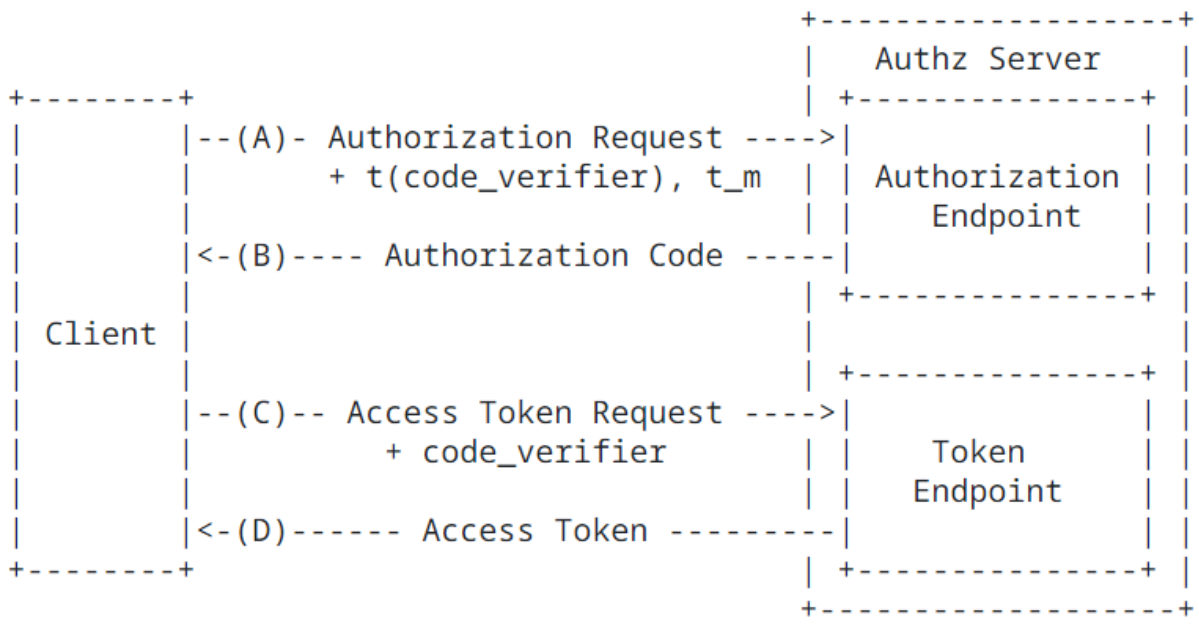

Figure 4: Abstract Authorization Code Flow with PKCE [4]

4 Products & Deliverables

4.1 Authorization Server

The Authorization Server is the main component of this project as it contains the two authorization endpoints, authorization endpoint and token endpoint, to handle all the authentication processes. It will be a backend server that can handle RESTful APIs in the form of HTTP request and return appropriate HTTP responses.

4.2 Authorization Server Login Page

The Login Page acts as a middleman for resource owners to interact with the authorization server where users input their login credentials. The login page then fires authentication requests to the authorization server.

4.3 Resource Server

Resource server and authorization server sometimes can be the same server but, in our case, we aim to handle resources from multiple different resource holder in the future so we decided to separate them into different entity. This original goal of this project is to have the resource server containing personal information of all Hong Kong citizens but it is impossible to obtain access these data from government at the moment hence an alternative “HKU students” resource server is created to replace the “HKID” resource server as they are interchangeable components. Successful communication between the “HKU students” and our authorization server proves the correct implementation of the authorization server working with “HKID” resource server. The resource server allows retrieval of data upon receiving a valid access token according to the scope and permissions associated with the access token by setting up routes to handle HTTP requests.

5 Technology Framework

5.1 Backend Server

The two servers will be implemented in Node.js/Express.js Framework as Node.js is asynchronous and non-blocking in nature. This allows the server to handle a large number of concurrent connections and requests efficiently. In OAuth, it is common that multiple clients are making simultaneous authorization requests and Node.js performs well in these situations, making our OAuth project scalable and responsive. Besides that, Node.js has a rich ecosystem of open-source packages available through npm (Node Package Manager), thus greatly reducing development time and effort.

5.2 Frontend Login Pages

As for the authorization server login page, React.js will be used to create the webpages because it is also a JavaScript framework and thus it works well with the backend servers.

5.3 Database

Our choice of database would be MongoDB because it is designed to scale horizontally, allowing developers to distribute the OAuth server across multiple machines to handle high traffic loads.

5.4 Cloud Platform / Deployment

AWS (Amazon Web Services) is best known for its scalability and elasticity so it very suitable for us to easily scale our OAuth server based on demand. It is also easy to deploy Node.js servers on the AWS EC2 instances.

6 Methodology

6.1 Requirement Analysis

Firstly, a comprehensive analysis of the requirements for the OAuth server must be conducted. We must first identify the core functionalities, security measures, and the OAuth flow.

6.2 System Design

Based on the analysis, we first conduct the system design to define the architecture, components, and interactions of the OAuth server. It includes deciding the tech stack, selecting appropriate frameworks or libraries

6.3 Database Design

Database and tables are to be designed and tailored for the OAuth system to efficiently store access tokens, authorization grant, and users' information. Good database design ensures the scalability of the application when the userbase grows to certain extent.

6.4 Implementations

This phase involved coding out the OAuth server using Node.js, importing the chosen frameworks and libraries. The OAuth protocols are implemented by accurate following the documentation, ensuring compliance with OAuth 2.0 and related standards such as PKCE and TSL.

6.5 Testing

Unit testing is conducted simultaneously during development process to ensure the individual component work as expected in the early stages. Integration testing is carried out at the final stage of development to ensure each functional component interact correctly with each other.

6.6 Production Deployment

This phase involved deploying the OAuth server on the Amazon Web Services (AWS) cloud platform, deploying a production MongoDB database, and utilizing suitable DevOps tools to manage the project. Configuring settings, such as environment variables and security parameters, are properly set up to ensure the secure and efficient operation of the OAuth server.

7 Scope, Project Plan & Timeline

7.1 Project Scope

The scope of this project includes:

- Development of an OAuth 2.0 with PKCE server
- Development of a resource server that stores the personal information of the resource owner

The scope of this project does not include:

- Client registration process with the authorization server

7.2 Project Plan & Timeline

A Gannt Chart is produced to ensure that the project progresses smoothly and can be monitored easily. The tasks allocated as well as the timelines are defined in the Gantt Chart show in Figure 5.

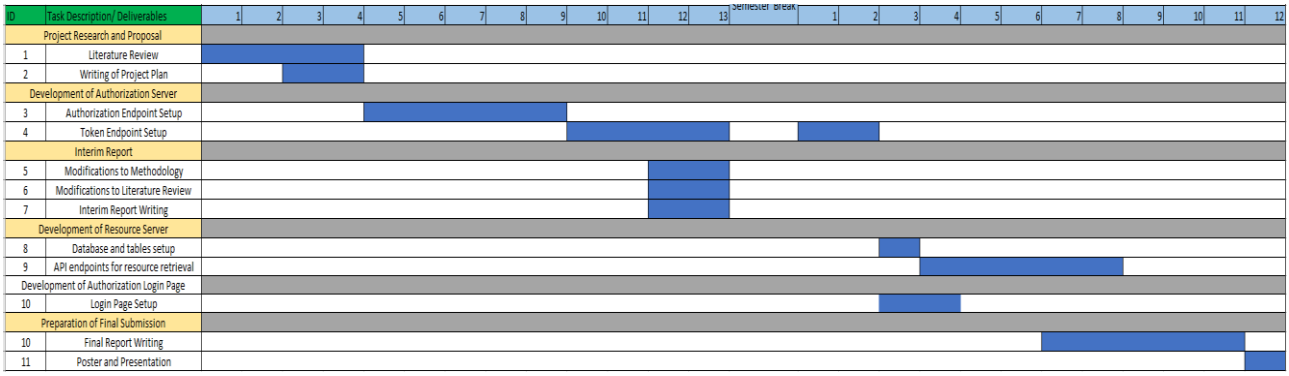

Figure 5: Project Gantt Chart

8 Risk Management Plan

8.1 Risk Assessment Matrix

Consequence and Likelihood	Insignificant	Minor	Serious	Disastrous	Catastrophic
Rare	L	L	L	L	M
Unlikely	L	L	M	M	S
Possible	L	M	M	S	H
Likely	L	M	S	H	E
Almost Certain	M	S	H	E	E

L = Low Risk, M = Medium Rish, S = Substantial Risk, H = High Risk, E = Extreme Risk

8.2 Non-OHS Risk Assessment

Project Risk	Risk	Likelihood	Consequence	Risk Level	Mitigation	Residual Risk
Resource over-utilization in AWS EC2 instance	Users fail to be authenticated	Possible	Serious	M	Utilize Elastic Load Balancing (ELB) Service to allow flexible adaptions to intensive CPU usage	Lost users' trust in our OAuth application
Incorrect implementation of OAuth 2.0 protocol	Weak security, susceptible to malicious attack	Possible	Disastrous	S	Careful implementation following research documentation and thorough integration testing	Users data leaked and reputation harm
Inappropriate database design	Non-scalable when userbase grows, server operations become irresponsive	Possible	Disastrous	S	Consulting MongoDB experts to better model the tables	All existing database structures are required to be erased

9 References

[1] J. Lee, C. Liu, “Real-Name Registration Rules and the Fading Digital Anonymity in China,” Washington International Law Journal, Vol 25, No. 1, 2016.
https://digitalcommons.law.uw.edu/cgi/viewcontent.cgi?article=1717&context=wilj
[2] L. John, “Identifying the Problem: Korea’s Initial Experience with Mandatory Real Name Verification on Internet Portals,” Journal of Korean Law, Vol. 9, 83-108, December 2009. https://s-space.snu.ac.kr/bitstream/10371/85159/1/4.%20Identifying%20the%20Problem%20Korea%E2%80%99s%20Initial%20Experience%20with%20Mandatory%20Real%20Name%20Verification%20on%20Internet%20Portals.pdf

[3] D. Hardt, “RFC 6749 The OAuth 2.0 Authorization Framework,” Internet Engineering Task Force (IETF), October. https://datatracker.ietf.org/doc/html/rfc6749

[4] N. Sakimura, J.Bradley, and N. Agarwal, “RFC 6749 The OAuth 2.0 Authorization Framework,” Internet Engineering Task Force (IETF), October. https://datatracker.ietf.org/doc/html/rfc7636

[5] T. Lodderstedt, M. McGloin, P. Hunt, “OAuth 2.0 Threat Model and Security Considerations”, RFC 6819, DOI 10.17487/RFC6819, January 2013, http://www.rfc-editor.org/info/rfc6819.