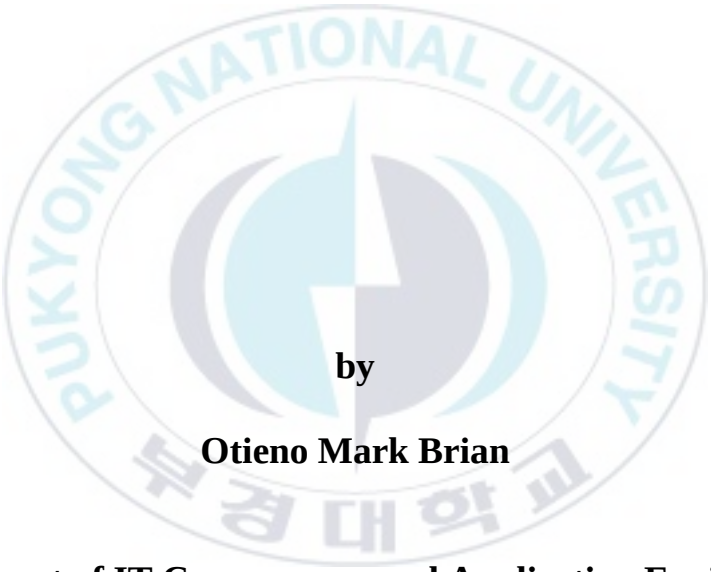
Thesis for the Degree of Master of Engineering

An OAuth Implementation on the OwnCloud for Secure Access by Social Networking Sites



by

Otieno Mark Brian

Department of IT Convergence and Application Engineering

The Graduate School

Pukyong National University

February 2016

An OAuth Implementation on the OwnCloud for Secure Access by Social Networking Sites

(소셜 네트워킹 사이트의 안전한 접근을 위한 OwnCloud 상의 OAuth 구현)

Advisor: Prof. Kyung-Hyune Rhee

by
Otieno Mark Brian

A thesis submitted in partial fulfillment of the requirements
for the degree of

Master of Engineering

Department of IT Convergence and Application Engineering,
The Graduate School,
Pukyong National University

February 2016

An OAuth Implementation on the OwnCloud for Secure Access by Social Networking Sites

A thesis

by

Otieno Mark Brian

Approved by:

(Chairman) Prof. Kim-Chang Soo

(Member) Prof. Man-Gon Park

(Member) Prof. Kyung-Hyune Rhee

February 26, 2016

Table of Contents

Table of Figures

List of Tables

소셜 네트워킹 사이트의 안전한 접근을 위한 OwnCloud 상의 OAuth 구현

오티에노 마크 브라이언

부경대학교 대학원 IT 융합응용공학과

요 약

클라우드 스토리지 서비스뿐만 아니라 소셜 미디어 사이트의 급속한 발전으로 인해, 기존의 웹 기반 서비스를 포함하여 블로그나 사진공유 형태의 소셜 네트워크 서비스가 점점 더 인기를 끌고 널리 사용되고 있는 추세이다. 그러나 이때 여러 사이트 간의 미디어 공유 서비스에 대한 보안도 중요하게 고려할 필요가 있다. 이에, 본 논문은 클라우드 기반의 안전한 데이터 공유시스템을 구현하기 위해 OwnCloud 와 OAuth 를 이용한 접근제어 프로토콜을 설계하였다. OwnCloud 는 사용자 스토리지 서버 시스템으로서, 파일 동기화 및 공유 서비스에 대한 유연성을 제공한다. OAuth 는 다양한 서비스들 간의 식별정보 (identity) 관리를 위한 표준 프로토콜이다. 따라서 OwnCloud 와 OAuth 를 연동하여 페이스북이나 트위터 같은 소셜 네트워킹 서비스를 통해 사용자들 간에 안전하게 파일을 공유할 수 있는 시스템을 제공할 수 있다.

An OAuth Implementation on the OwnCloud for Secure Access by Social Networking Sites

Otieno Mark Brian

IT Convergence and Application Engineering, Graduate School
Pukyong National University

Abstract

There has been advancement in the area of cloud storage services as well as a tremendous growth in the embrace of social media sites. Allowing one Web service to act on our behalf of another has become increasingly important as social Internet services such as blogs, photo sharing, and social networks have become widely popular. With this increased cross-site media sharing arises numerous security implications and thus the need to come up with security protocols and considerations. OwnCloud is an enterprise file sync and share that is hosted in user's data center, on user's servers, using user's storage. OwnCloud's server is flexible to enable Information Technology experts to protect and manage files within the OwnCloud environment; from file storage to user provisioning and data processing. OAuth, a new protocol for establishing identity management standards across services, provides an alternative to sharing our user-names and passwords, and exposing ourselves to attacks on our on-line data and identities. We are therefore proposing an OAuth implementation to the OwnCloud environment for Secure Access by Social Networking Sites like Facebook and Twitter.

1. INTRODUCTION

1.1. Background

Internet-based social networking sites have created a revolution in social connectivity. Social networking sites are Internet-based services that allow people to communicate and share information with a group. Facebook, Twitter, Google+, LinkedIn and other social networks have become an integral part of online lives. Social networks are a great way to stay connected with others through the sharing of information such as photos, videos, and personal messages.

The use physical storage devices is quite limiting in terms of storage scalability. They therefore need a scalable storage and are thus seeking cloud platforms like OwnCloud and Dropbox to store their media.

Dropbox is ideal but it has several limitations in-terms of control and storage. Much of the control lies in the hands of the host client and additional storage is charged. OwnCloud is therefore preferred because the owner has full control of his data, because it is deployed within his own server/ domain. And further more is quite scalable because the storage capability depends on the server space the owner owns.

In the traditional client-server authentication model according to D. Hardt [1], the client requests an access-restricted resource (protected resource)

on the server by authenticating with the server using the resource owner's credentials. In order to provide third-party applications access to restricted resources, the resource owner shares its credentials with the third party. This creates several problems and limitations:

- Third-party applications are required to store the resource owner's credentials for future use, typically a password in clear-text.
- Servers are required to support password authentication, despite the security weaknesses inherent in passwords.
- Third-party applications gain overly broad access to the resource owner's protected resources, leaving resource owners without any ability to restrict duration or access to a limited subset of resources.
- Resource owners cannot revoke access to an individual third party without revoking access to all third parties, and must do so by changing the third party's password.
- Compromise of any third-party application results in compromise of the end-user's password and all of the data protected by that password.

OAuth addresses these issues by introducing an authorization layer and separating the role of the client from that of the resource owner. In OAuth, the client requests access to resources controlled by the resource owner and hosted by the resource server, and is issued a different set of credentials than those of

the resource owner.

Instead of using the resource owner's credentials to access protected resources, the client obtains an access token which is a string denoting a specific scope, lifetime, and other access attributes. Access tokens are issued to third-party clients by an authorization server with the approval of the resource owner. The client uses the access token to access the protected resources hosted by the resource server [24].

1.2. Overview and Contribution

OAuth 2.0 support is a key requirement which will make OwnCloud attractive as a platform for third-party developers who need to integrate OwnCloud into their applications. OAuth is a security protocol that allows third-party applications to request access to protected information without providing the username and the password to other party. Currently, third-party applications have to use Basic Authentication which involves sending the username and password to access the OwnCloud instance which has the following disadvantages:

- Users have to provide their credentials to third-party applications. If one of the third-party providers has been compromised then OwnCloud login will be lost.

- An authentication can only be revoked by changing the user password which is suboptimal.
- Third-party developers do currently have access to the whole OwnCloud instance; they even could change your password.

Instead of using passwords for authorization, OAuth is using unique tokens for every client. In OAuth, the client requests access to the needed resources (scopes) by redirecting the user to a website where he has to approve these permissions.

In this thesis, we propose a secure sharing platform between OwnCloud cloud storage and internet based social networking sites. This resolves the limitation of the current OwnCloud infrastructure and third-party applications by providing a security protocol that allows third-party applications to request access to the protected information without providing the username and the password to other party. The proposed protocol entails integrating third-party login into the OwnCloud framework, thereby allowing third party internet social networking sites to have access to all the media files stored within the cloud storage. In order to achieve these goals, we propose the integration of OAuth 2.0 security protocol into the OwnCloud platform. OAuth 2.0 will ensure security by ensuring the proper authorization flow is followed during communication between OwnCloud and the social

networking sites. In addition, it performs authentication of the resource owner/clients through the Authorization Server. Moreover, it ensures a token is issued that not only authorizes but also dictates the scope and lifetime of a given action.

This implementation will be of benefit to different users, developers and administrators on different levels. Users will be able to grant third-party applications access to their data without providing their passwords or granting access to the whole instance. Users and administrators will be able to see which applications have access to which data and manage them. And lastly, developers will be able to use standard libraries to integrate with OwnCloud.

The rest of this thesis is organized as follows. The next chapter briefly introduces OwnCloud, Secure Socket Layer (SSL) and OAuth 2.0 and their system architectures. In Chapter 3, we present the proposed protocol for secure sharing between OwnCloud and internet based social networking sites through integration of the OAuth 2.0 security protocol. We give the security and performance evaluations of the proposed protocols in chapter 4. Finally, we conclude the thesis in chapter 5.

2. PRELIMINARIES

2.1. OwnCloud

OwnCloud is enterprise file sync and share that is hosted in your data center, on your servers, using your storage. OwnCloud provides Universal File Access through a single front-end to all of your disparate systems. Users can access company files on any device, anytime, from anywhere while IT can manage, control and audit file sharing activity to ensure security and compliance measures are met. [2]

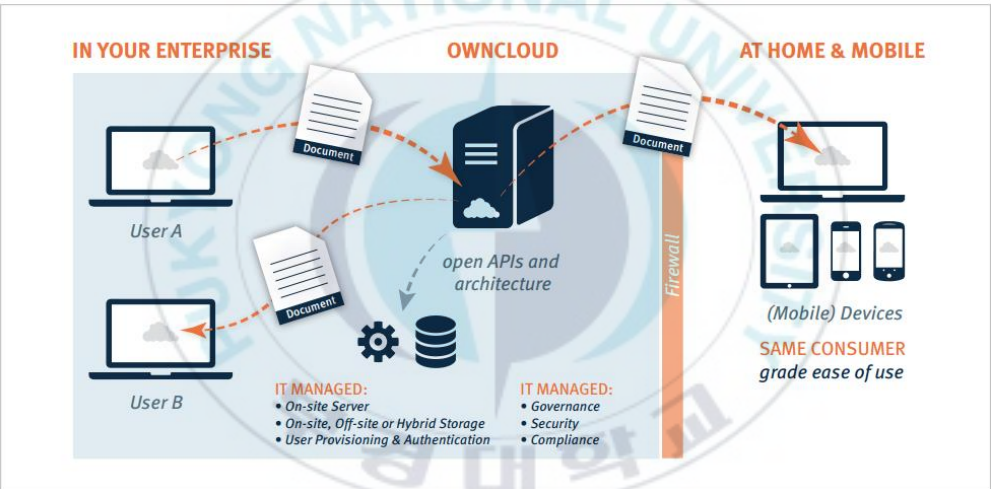


Figure 1: OwnCloud's Architecture

Unlike consumer-grade file sharing services such as Dropbox, OwnCloud's server enables IT to protect and manage files within the OwnCloud environment; from file storage to user provisioning and data processing. OwnCloud can be extended to do far more than basic file sync and

share through the use of mobile libraries, open Application Program Interfaces (APIs) and plug-in applications [19].

2.2. OAuth 2.0

The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf.

The OAuth 2.0 authorization protocol standardizes delegated authorization on the Web. Popular social networks such as Facebook, Google and Twitter implement their APIs based on the OAuth protocol to enhance user experience of social sign-on and social sharing.

In [5], F. Yang *et al.* describes OAuth (open standard for authorization) as a protocol that provides a generic framework to let a resource owner authorize third-party to access the owner's resource held at a server without revealing to the third-party the owner's credentials (such as user-name and password) [1], [6].

OpenID Connect 1.0 is a simple identity layer on top of the OAuth 2.0 protocol. It allows Clients to verify the identity of the End-User based on the

authentication performed by an Authorization Server, as well as to obtain basic profile information about the End-User in an inter-operable and Representational State Transfer (REST)-like manner.

The OAuth 2.0 security protocol defines four roles which helps it accomplish the purpose it is tasked with reliably. [7]

- **Resource owner -** This refers to an entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user.
- **Resource server** - The server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens.
- **Client** - An application making protected resource requests on behalf of the resource owner and with its authorization.
- **Authorization server** - The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

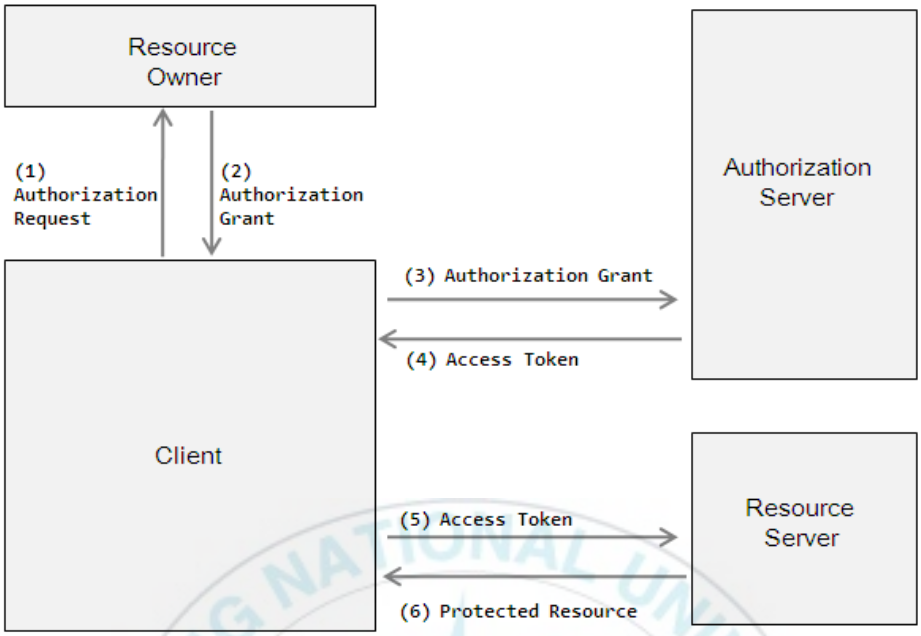


Figure 2: OAuth Protocol Flow

OAuth 2.0 flow illustrated in Figure 3 above describes the interaction between the four roles and includes the following steps:

1. The client requests authorization from the resource owner. The authorization request can be made directly to the resource owner, or preferably indirectly via the authorization server as an intermediary.
2. The client receives an authorization grant, which is a credential representing the resource owner's authorization, expressed using one of four grant types defined in this specification or using an extension grant type. The authorization grant type depends on the method used

by the client to request authorization and the types supported by the authorization server.

3. The client requests an access token by authenticating with the authorization server and presenting the authorization grant.
4. The authorization server authenticates the client and validates the authorization grant, and if valid, issues an access token.
5. The client requests the protected resource from the resource server and authenticates by presenting the access token.
6. The resource server validates the access token, and if valid, serves the request.

2.3. How OAuth Works

In [8], OAuth-based SSO systems are based on browser redirection in which a relying party (RP) redirects the user's browser to an IdP that interacts with the user before redirecting the user back to the RP website. The identity provider (IdP) authenticates the user, identifies the RP to the user, and asks for permission to grant the RP access to resources and services on behalf of the user.

Table 1: Notations and descriptions

Notations	Descriptions
RP	Relying party
IdP	Identity provider
U	User
B	Browser

Once the requested permissions are granted, the user is redirected back to the RP with an access token that represents the granted permissions and the duration of the authorization.

Using the authorized access token, the RP then calls web APIs published by the IdP to access the user's profile attributes.

The OAuth 2.0 specification defines two flows for RPs to obtain access tokens: server-flow (known as the "Authorization Code Grant" in the specification), intended for web applications that receive access tokens from their server side program logic; and client-flow (known as the "Implicit Grant") for JavaScript application running in a web browser.

Figure 3 below illustrates the following steps, which demonstrate how the server-flow works:

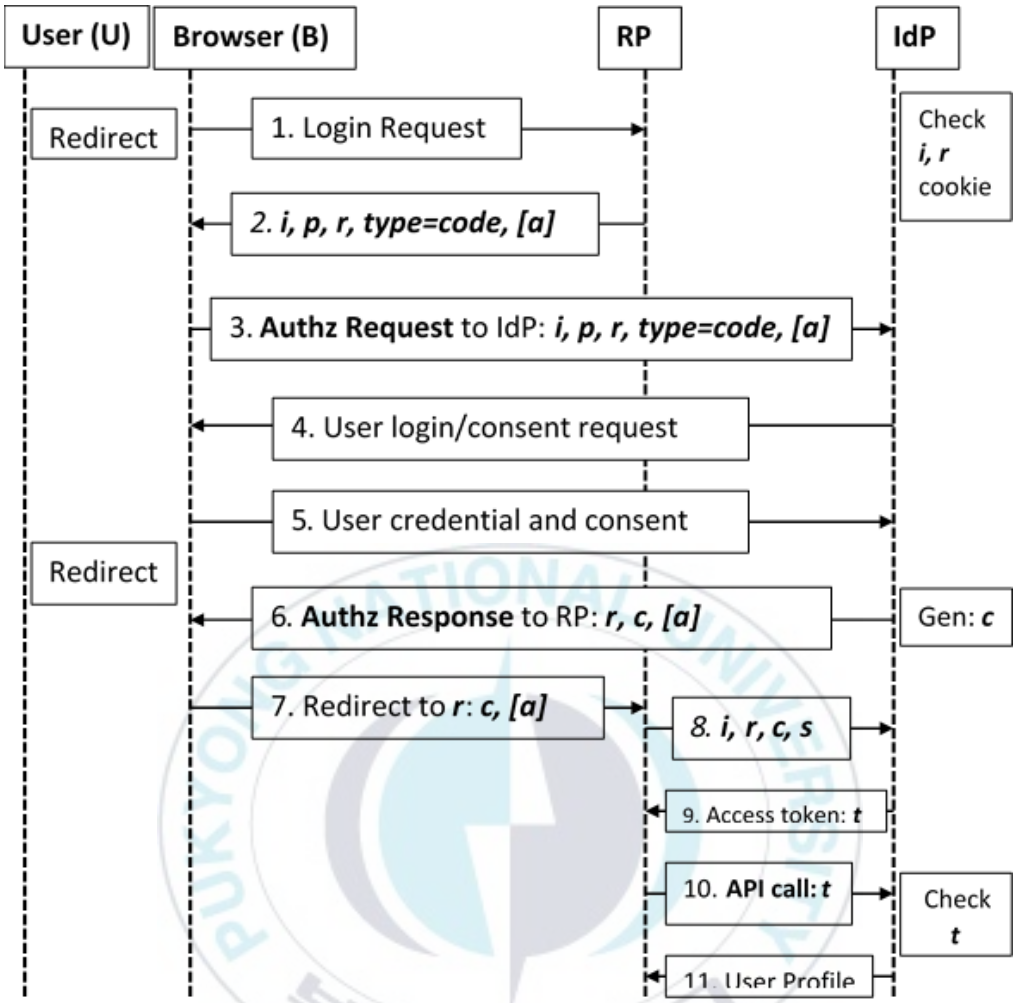


Figure 3: The OAuth server-flow protocol sequence

1. User **U** clicks on the social login button rendered by the **RP** to initiate an SSO process. The browser **B** then sends this login HTTP request to **RP**.
2. RP sends response_type=code, client ID *i* (assigned during registration with the **IdP**), requested permission scope *p*, and *a* redirect Uniform

Resource Locator (URL) r to **IdP** via **B** to obtain an authorization response. The redirect URL *r* is where **IdP** should return the response back to **RP** (via **B**). **RP** could also include an optional state parameter *a*, which will be appended to *r* by **IdP** when redirecting **U** back to **RP**, to maintain state between the request and response.

3. **B** sends response_type=code, *i*, *p*, *r* and optional *a* to **IdP**. **IdP** checks *i* and *r* against its own local storage. If a cookie that was previously set after a successful authentication with **U** is presented in the request, and the requested permissions *p* has been granted by **U** before, **IdP** could omit the next two steps (4 and 5).
4. **IdP** presents a login form to authenticate the user.
5. **U** provides her credentials to authenticate with **IdP**, and then consents to the release of her profile information.
6. **IdP** generates an authorization code *c*, and then redirects **B** to *r* with *c* and *a* (if presented) appended as parameters.
7. **B** sends *c* and *a* to *r* on **RP**.
8. **RP** sends *i*, *r*, *c* and *a* client secret *s* (established during registration with the **IdP**) to **IdP's** token exchange endpoint through a direct communication (i.e., not via **B**).
9. **IdP** checks *i*, *r*, *c* and *s*, and returns an access token *t* to **RP**.

10. **RP** makes a web API call to **IdP** with *t*.
11. **IdP** validates *t* and returns **U's** profile attributes for **RP** to create an authenticated session.
12. The client-flow is designed for applications that cannot embed a secret key, such as JavaScript clients running in browsers. The access token is returned directly in the redirect response, and its security is handled in two ways:
    - The IdP validates the redirect URI matches a pre-registered URL to ensure the access token is not sent to unauthorized **RPs**;
    - The token itself is appended as an URI fragment (#) of the redirect URI so that the browser will never send it to the server, and hence prevents the token from exposing in the network.

    Figure 4 below illustrates how the client-flow works:

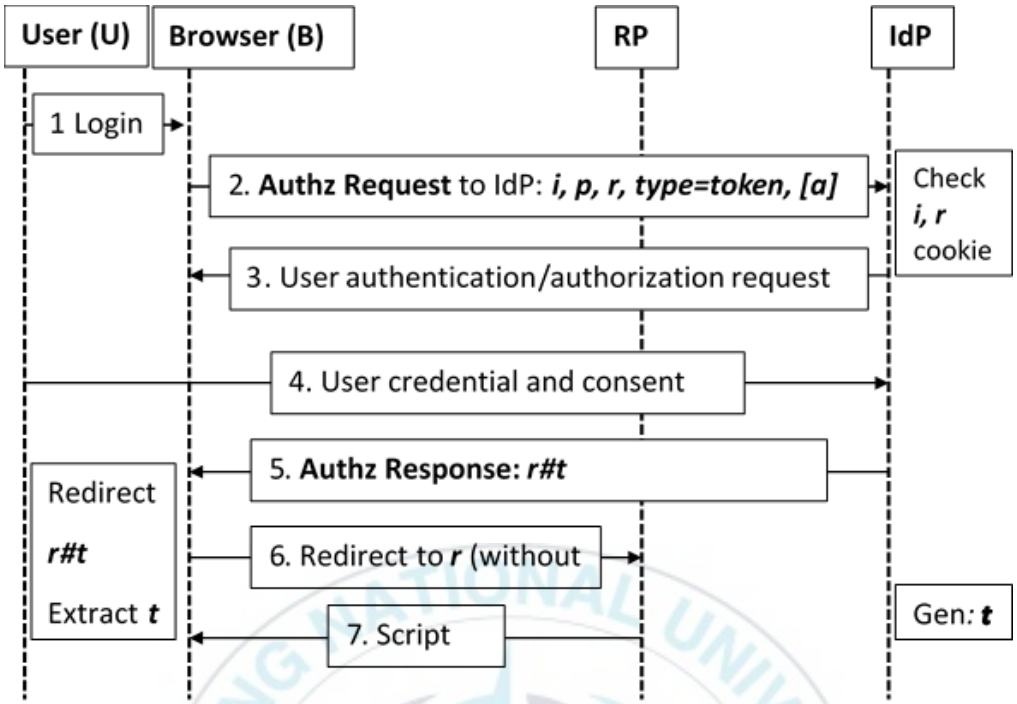


Figure 4: The OAuth client-flow protocol sequence

1. User **U** initiates an SSO process by clicking on the social login button rendered by **RP**.
2. B sends response_type=token, client ID *i*, permission scope *p*, redirect URL *r* and an optional state parameter a to **IdP**.
3. **IdP** presents a login form to authenticate the user, followed by an authorization consent form. The authentication step could be omitted if the user has logged to **IdP** in the same browser session; and the consent step could be skipped if the requested permissions have already been granted before.

4. **U** signs into **IdP**, and grants the requested permissions.
5. **IdP** returns an access token *t* appended as an URI fragment of *r* to **RP** via **B**. State parameter a is appended as a query parameter if presented.
6. **B** sends *a* to *r* on **RP**. Note that **B** retains the URI fragment locally, and does not include *t* in the request to **RP**.
7. **RP** returns a web page containing a script to **B**. The script extracts t contained in the fragment using JavaScript command such as document.location.hash. With *t*, the script could call **IdP's** web API to retrieve **U's** profile that is bounded to *t*.

To ensure protocol security, several approaches based on formal methods [11, 12, 13] were used to analyze the OAuth protocol. The results of that analysis suggest that the protocol is secure, provided that the comprehensive security guidelines from the OAuth working group included in "OAuth threat model" [14] are followed by the IdP and RP.

However, given that the formal proofs are executed on abstract models, some important implementation details could be inadvertently left out. Furthermore, it is unclear whether real implementations actually do follow the above guidelines. Thus, the research question regarding the security of OAuth implementations remains open [18].

OAuth-based SSO systems are built upon the existing web infrastructure, but web application vulnerabilities (e.g., insufficient transport layer protection, cross-site scripting (XSS), cross-site request forgery (CSRF)) are prevalent [15] and constantly being exploited [16, 17]. Moreover, as the protocol messages are passed between the RP and IdP via the browser, a vulnerability found in the browser could also lead to significant security breaches.

To enhance the security of OAuth SSO systems, firstly, further understanding of how those well-known web vulnerabilities could be leveraged to compromise OAuth SSO systems. Next, is the fundamental enabling causes and its consequences? Thirdly, is how prevalent they are, and lastly how to prevent them in a practical way. These particular issues are still poorly understood by researchers and practitioners. [8, 20]

2.4. Related Work

There has been several implementations and integrations of secure an extensive sharing to internet social networking sites. These implementations normally involve the integration of the capability to login into host sites via a popular social networking site account. A good example is the Picasa Web Albums Data API allows for websites and programs to integrate with Picasa Web Albums, enabling users to create albums, upload and retrieve photos, comment on photos.

Dropbox and OwnCloud are the two most adopted cloud storage platforms. OwnCloud does not however support the ability to directly share the media stored in it to the social networking sites. The integration involves allowing the capability to login into OwnCloud via social networking sites such as Facebook. OwnCloud have not embraced this feature because of the implications that arise from such integration. From a security perspective, as the popularity of these social sites grows, so do the risks of using them. Hackers, spammers, virus writers, identity thieves, and other criminals follow the traffic [20]. Over sharing has also been cited as a major concern for the reluctance in the integration of this model. This is because once information is posted to a social networking site, it is no longer private. According to [9], the Federal Bureau of Investigation (FBI) warns that the more information you

post, the more vulnerable you may become. Even when using high security settings, friends or websites may inadvertently leak your information. From a business perspective, this move tends markets the social internet networking site instead of promoting the hosts application. And with the embrace of a single sign-on (SSO) feature, most users will prefer to use an existing account as compared to creating a new account which requires management as well. For users and customers, the ability to log into a site using Facebook Connect means they have one less password to remember and therefore a much faster path to signing up for and using your site in the first place.

OwnCloud currently uses the Lightweight Directory Access Protocol (LDAP) and the Web Distributed Authoring and Versioning (WebDAV). LDAP is a directory service protocol that runs on a layer above the TCP/IP stack. It provides a mechanism used to connect to, search, and modify Internet directories. The LDAP directory service is based on a client-server model.

Web Distributed Authoring and Versioning (WebDAV) is an extension of the Hypertext Transfer Protocol (HTTP) that allows clients to perform remote Web content authoring operations. The WebDAV protocol provides a framework for users to create, change and move documents on a server, typically a web server or web share. LDAP on one hand is used for user authentication while WebDAV on the other hand is used for file access. There

is also a "*user_webdavauth*" app, but this is to allow authenticating to OwnCloud via any http service which, upon retrieval of login data, returns 100 if login should be successful or any other value if not.

The external API inside OwnCloud allows third party developers to access data provided by OwnCloud apps. Methods are registered inside the *appinfo/routes.php* using OCP\API. Once the API backend has matched your URL, your callable function as defined in *$action* will be executed. This method is passed as array of parameters that you defined in *$url*. To return data back to the client, you should return an instance of OC_OCS_Result. The API backend will then use this to construct the XML or JSON response. Because Representational State Transfer (REST) is stateless you have to send user and password each time you access the API. Therefore running OwnCloud with SSL is highly recommended otherwise everyone in your network can log your credentials.

3. PROPOSED SCHEME

3.1. System Model

The proposed framework involves implementing the OAuth 2.0 security protocol into the OwnCloud platform. The OAuth 2.0 server authentication flow is used whenever an OwnCloud account uses the integration for the first time. The OwnCloud user must login to their account and give permission to the third-party social networking sites application to access their OwnCloud account. The server authentication flow consists of two main transactions:

- The third-party application makes an authorization request
- The authorization server responds with an authorization code
- The third-party application then makes an access token request using the authorization code
- The authorization server responds with an access token.

The interactions between the user, the third-party application, the OwnCloud, and the OAuth 2.0 Authorization Server are illustrated in the figure 6 below.

According to the proposed system model, these are the four main roles;

1. The Resource Server is represented by OwnCloud
2. The User represents the Owner/User

3. The Authorization Server is represented by the OAuth 2.0 protocol
4. The Third-Party Application represents the Social Networking Sites such as Facebook and Twitter

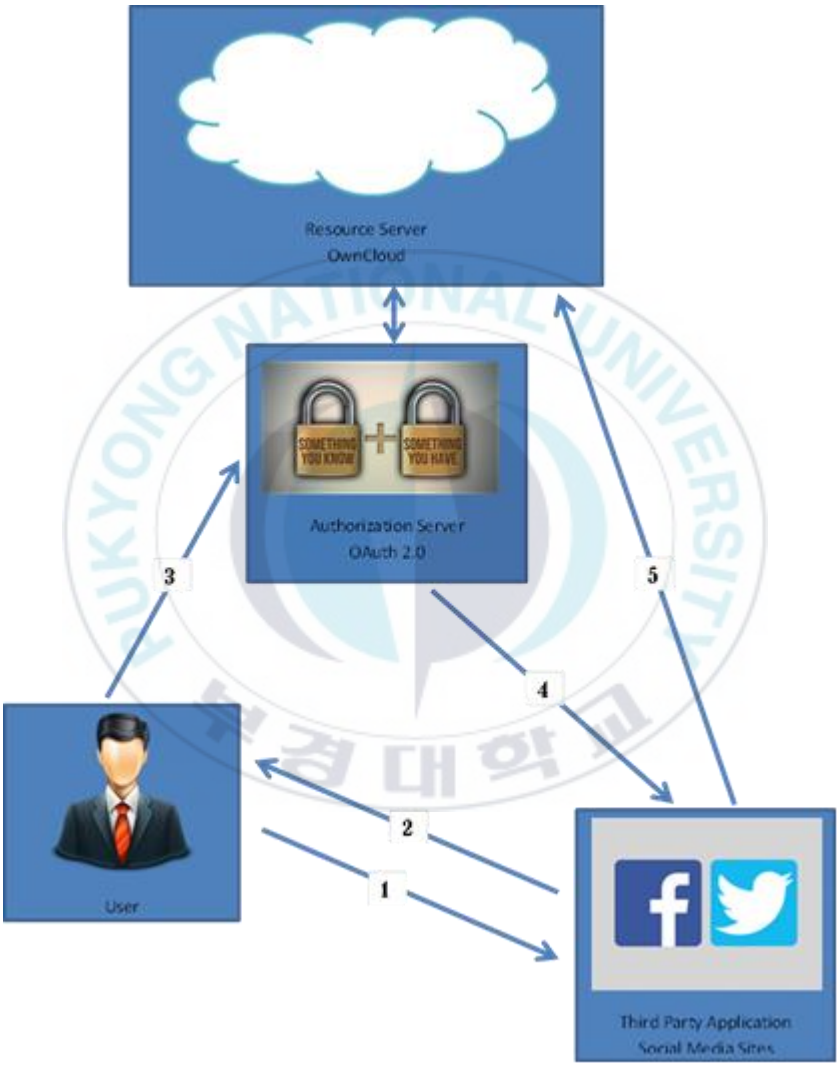


Figure 5: proposed System Model

The proposed process flows as follows:

1. The User visits the Third-Party Application (Social Media Sites) webpage
2. The application directs the user to the OAuth 2.0 authorization server security protocol (Authorization Server)
3. The User authenticates with OAuth 2.0 and grants the Third-party Application access to their account.
4. The Authorization Server redirects the User to the application using the redirect URI, and provides an authorization code if the user granted access to the application. The User is redirected to the application's redirection endpoint, the `redirect_url`, with an authorization code in the `code` URL parameter,

Table 2: Application redirection endpoint

```
HTTP/1.1 302 Found
Location:
https://client.example.com/cb?code=SplxlOBeZQQYbYS6WxSbIA&
state=xyz
```

5. The application then exchanges the authorization code for an access token for use in all API calls for that account.

   The application uses the Authentication Code to obtain Access

and Refresh Tokens using a **POST** request to the `login.owncloud.com/auth/oauth2/token` endpoint. The **POST** request should include a **JSON body** with the following parameters;

Table 3: JSON response parameters and values

Parameter	Value	Required?
`grant_type`	Must be `authorization_code`	Yes
`code`	The authorization code	Yes
`redirect_uri`	Application's registered redirection endpoint	Yes

The authorization server validates the authorization code and, if valid, responds with a JSON body containing the access token, refresh token, access token expiration time, and token type, as indicated in the table below:

Table 4: Successful access token JSON response

```
HTTP/1.1 200 OK
Content-Type: application/json
{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
   "token_type":"bearer",
   "expires_in":3600,
  "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA"
}
```

In this proposed model, the resource servers assert the token issued by the authorization server. According to the OAuth 2.0 specification, RFC6749 [1], it very specifically punts on this issue in section 7: “The methods used by the resource server to validate the access token (as well as any error responses) are beyond the scope of this specification but generally involve an interaction or coordination between the resource server and the authorization server.”

For small deployments, it can look it up in a database. In many instances, the RS and the AS are usually co-located and very tightly bound so they have access to the same data store. When the AS part of the server mints a token, it drops the token or a hash of it into a database along with all of the information about the token that will be needed to make an authorization decision. When that token comes back in later, the RS part of the server just looks up the token value or its hash and plucks any other bits of data that it needs from that record in order to authorize or deny the request being made.

Alternatively, in an instance where there are multiple RS's and a single AS, then there is need for a means to communicate all that meta-information surrounding the token including what scopes it has, who authorized it, what client it was authorized for, when it expires from the AS to the RS.

The solution is to use a structured token value like JSON Web Token (JWT). JWT is a compact, URL-safe means of representing claims to be

transferred between two parties. The claims in a JWT are encoded as a JavaScript Object Notation (JSON) object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed or MACed and or encrypted. (https://tools.ietf.org/html/draft-ietf-oauth-json-web-token-32)

JWTs are good constructs being that it's a blob of JSON that can be signed and encrypted in a way that won't get mucked up in transit. JWTs define a set of common claims, such as issuer, audience, subject, and other bits needed for a security object like this. The RS gets handed a JWT, it parses the JWT, checks the signature or decrypts it, reads the claims, sees who the token is for and what it is for and if it’s expiry. The RS could get everything it needs from that.

JWTs define a set of common claims, such as issuer, audience, subject etc. but packing all this information into the token could raise some issues:

- The token could become rather large and unwieldy if there is a lot to say about the authorization context.
- There is the risk of the client being able to read what is in the token, which might leak sensitive information.

OAuth tokens are opaque to clients, which mean they do not have to read the token to use it, but that does not mean that a client cannot try to read the token and get something useful out of it. This can be combatted by encrypting the token, but even the JWT specification says that the best way to avoid privacy leakage issues is to just not put sensitive information inside the token itself. And it also assumes that the owner is okay with tokens being good until they expire, because if the RS is parsing the token on its own, there is no good way to revoke a token once it is in flight. However, this can be combatted by having short enough timeouts on the tokens.

Alternatively, the RS can have a service it calls at runtime to get information about the token in the context of its authorization decision, then find out in real time if the token has been revoked or not. And if it is making that call, it could also just as easily find out all of the important meta-information about that token. Token Introspection defines a very simple HTTP service that lets an RS send the token over in a POST and get back a JSON document that says what the token is good for. Introspection re-uses the claims defined in JWT and adds a few of its own. The RS authenticates to the AS during this call so that not just anyone can go search for token information.

3.2. System Setup

The proposed framework involves implementing the OAuth 2.0 security protocol into the OwnCloud platform. The OAuth 2.0 is written in both PHP and JavaScript programming language. The first step is the installation of OwnCloud into the Apache web server. Upon successful setup installation, OwnCloud should be accessible via the browser either through localhost or domain used.

Figure 6: OwnCloud login screen

The OAuth 2.0 source code is then integrated into the Apps folder of OwnCloud. As shown in figure 8 below, successful implementation of

Facebook connect will be integrated as a third party application under the Apps menu in the OwnCloud backend. Its functionality can be enable or disabled by the administrator as preferred.

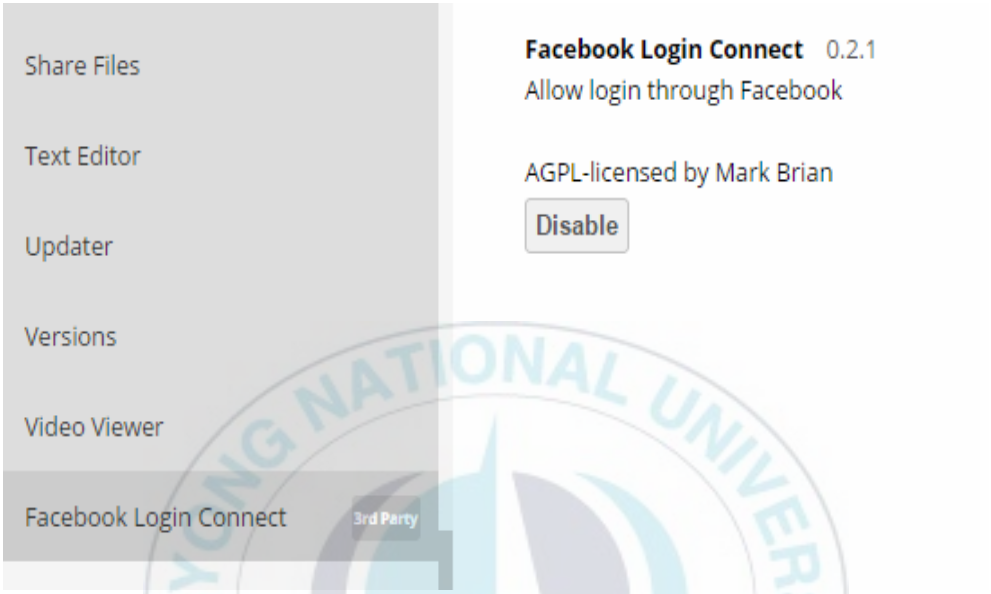


Figure 7: Integration of Facebook Login Backend

When the login connect is enabled, a configuration screen will be formed at the Admin menu section of OwnCloud. At the Admin section, an App ID and App Secret will be displayed in the Facebook login settings as shown in figure 9 below. The App credentials are retrieved from the chosen internet social networking site of choice. In the case of Facebook, one needs to register the OwnCloud application at the developer section of Facebook under the link (http://developers.facebook.com/apps). The registration requires an

App Name and declaring our app as a Website and therefore giving a Website URL. The App ID and App Secret will be automatically generated uniquely for each application created. And the credentials are added to the code end of the Facebook login connect so that it is able to pull the data of the account to which the credentials belong to.

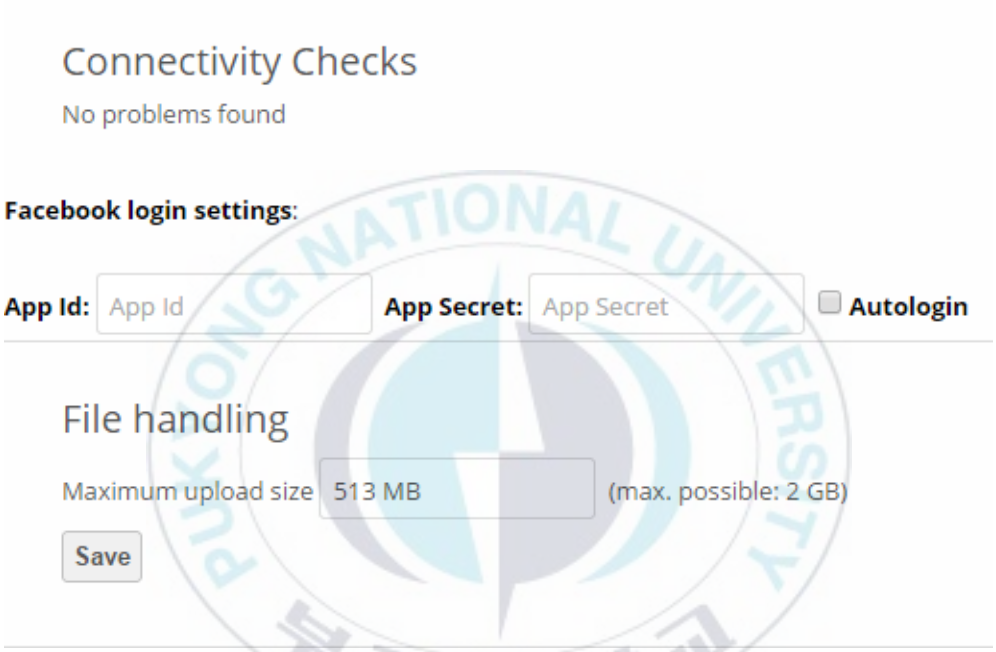


Figure 8: Facebook login backend

OAuth2.0 source code is downloaded and edited appropriately so it can integrate well within the OwnCloud infrastructure. The edited source code is then added to the Apps folder within the OwnCloud architecture which is hosted within the Apache environment. OwnCloud picks OAuth2.0 as an internal app and thus is accessible at the OwnCloud backend under the Apps

menu as shown in the figure 10 below. Enabling the OAuth will establish a security protocol which will run mostly as a background process.

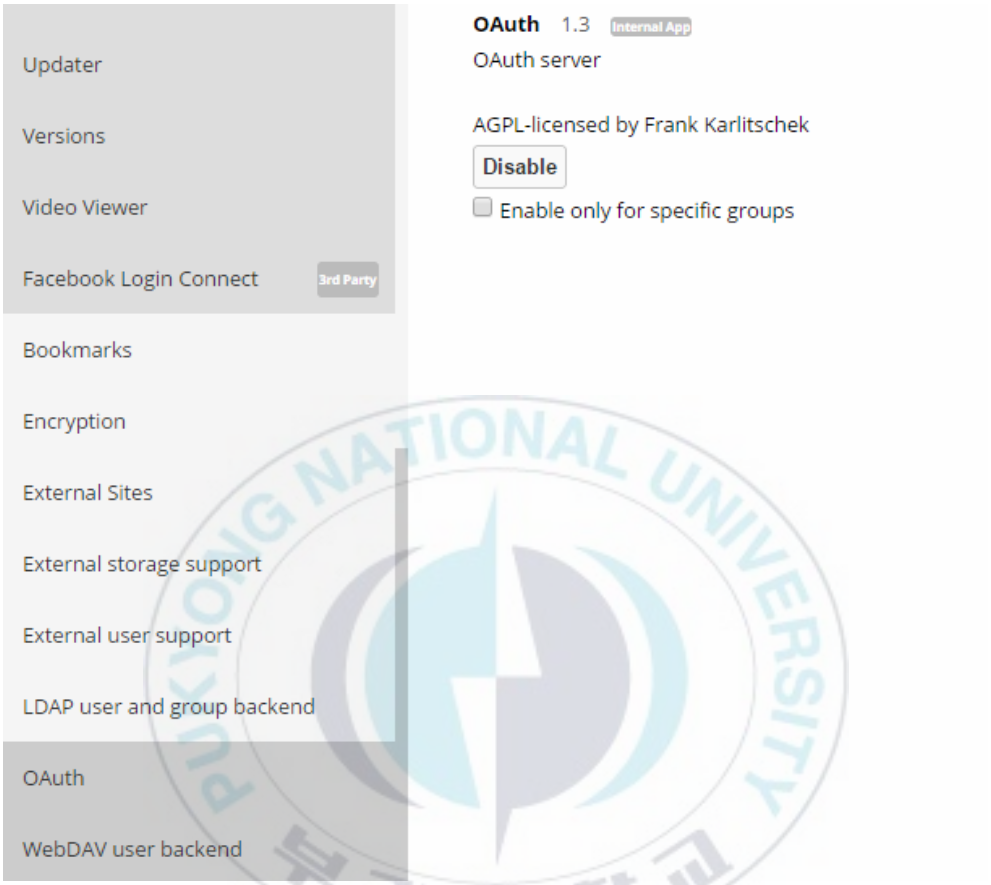


Figure 9: OAuth2.0 integration into OwnCloud infrastructure

The Introspection Endpoint is an OAuth 2.0 Endpoint that responds to HTTP POST requests and optionally HTTP GET requests from token holders, particularly including Resource Servers and Clients. The endpoint takes a single parameter representing the token and optionally further authentication

and returns a JavaScript Object Notation (JSON) document representing the Meta information surrounding the token. The endpoint MUST be protected by TLS or equivalent. The endpoint MAY allow other parameters to provide context to the query. The retrieved metadata can be used in order to determine the appropriateness of the token being presented or the approved scopes and the context in which a token was issued.

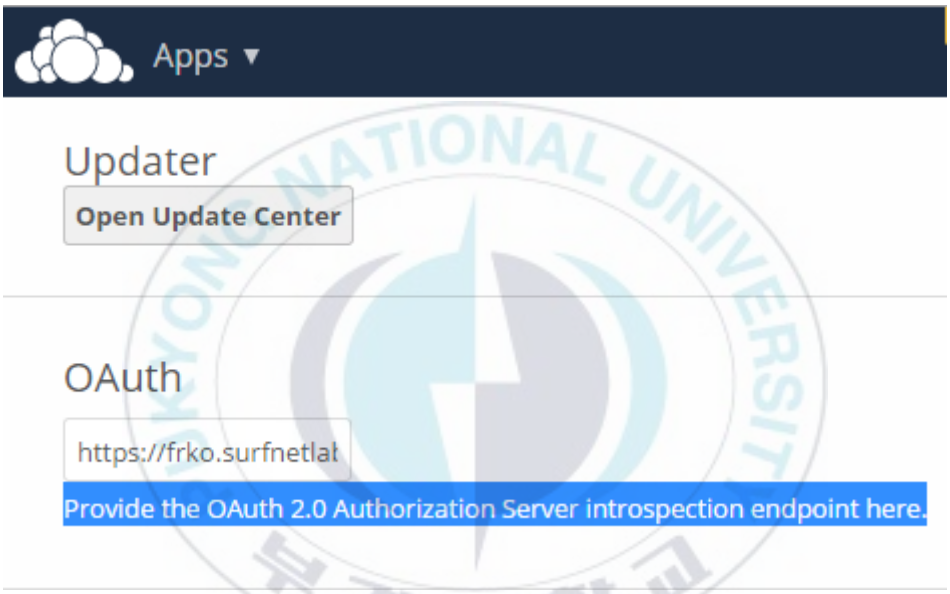


Figure 10: OAuth2.0 Authorization Server introspection endpoint

Since the introspection endpoint takes in OAuth 2.0 tokens as parameters, it MUST be protected by TLS or equivalent. A server MAY require an HTTP POST method only to the endpoint.

4. IMPLEMENTATION AND SECURITY ANALYSIS

4.1. Third Party Integration

When OwnCloud connects to an internet social networking site, then it can always access the site's public profile. OwnCloud may optionally ask for other pieces of information as well. This can include the list of friends using the app, their email, the events that they are attending, their hometown or the things they have liked. All of these are available behind optional permissions, which are asked for during the login process. You can ask for additional permissions later, after a person has logged in. A user may optionally choose to decline sharing data that the OwnCloud application asks for.

Figure 11: Alternative logins via social networking sites

The login dialog creates a trusted link between the user, OwnCloud and their information. It displays consistent messaging across all devices, enabling OwnCloud to request permissions anywhere. The permission requested is clearly explained in the dialog. The dialog also lets a user to decline to share data that OwnCloud has requested on a piece by piece basis, except for the

public profile which is always shared. Some permission are extra-sensitive, like publishing or access to a user's pages, and a user will have to confirm in an extra screen that they allow OwnCloud to have access to those capabilities.

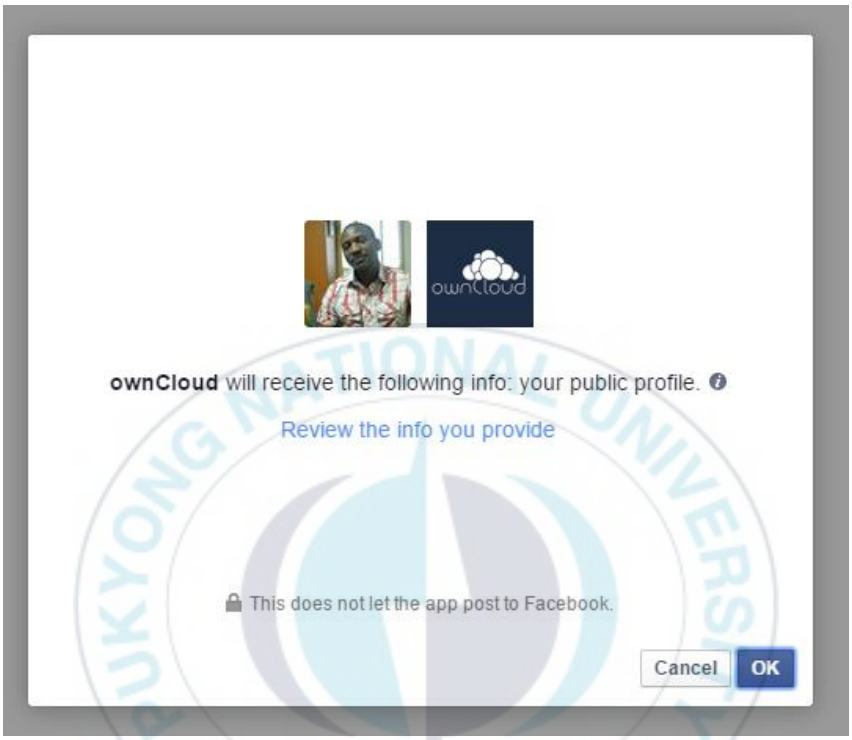


Figure 12: Permission dialog to access Facebook public profile

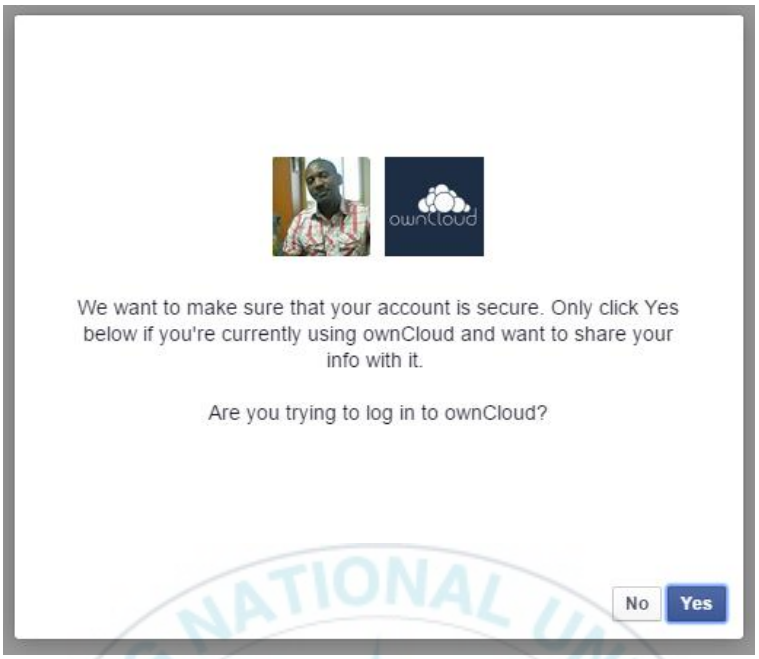


Figure 13: OwnCloud-Facebook Authentication dialog

Figure 14: *Authentication dialog for login via Twitter*

4.2. Security Analysis

Access tokens can be used as proof of authentication. Since an authentication usually occurs ahead of the issuance of an access token, it is possible to consider reception of an access token of any type proof that such an authentication has occurred.

Access of a protected API can also be used as a proof of authentication. Since the access token can be traded for a set of user attributes, it is viable to assume that possession of a valid access token is enough to prove that a user is

authenticated. This is especially true in cases where the token was freshly minted in the context of a user being authenticated at the authorization server.

Authentication of the user who is trying to access OwnCloud via a third party is also established based on the integration of the OAuth 2.0 protocol which acts to establish identity management. A dialogue notification pops up upon any access giving the owner (OwnCloud) the authority to either grant access or deny access and also to determine the scope and sharing limitations of the access. Figure 14 and 15 above illustrate the permission and scope being granted by OwnCloud other third party applications namely Facebook and Twitter respectively.

Token Assertion is done by the by Resource Server. This works on the assumption that the RS will be able to call the AS for each token that it sees, and that there is no problem with the extra network traffic. Thus there is the accuracy/performance tradeoff in most networked systems: you can have live information by calling the authoritative source in real time (using introspection) or you can have self-contained information that you don't have to make a network call for (using JWT). You can, of course, cache the introspection call, and most implementations do this on the client side, at least to an extent.

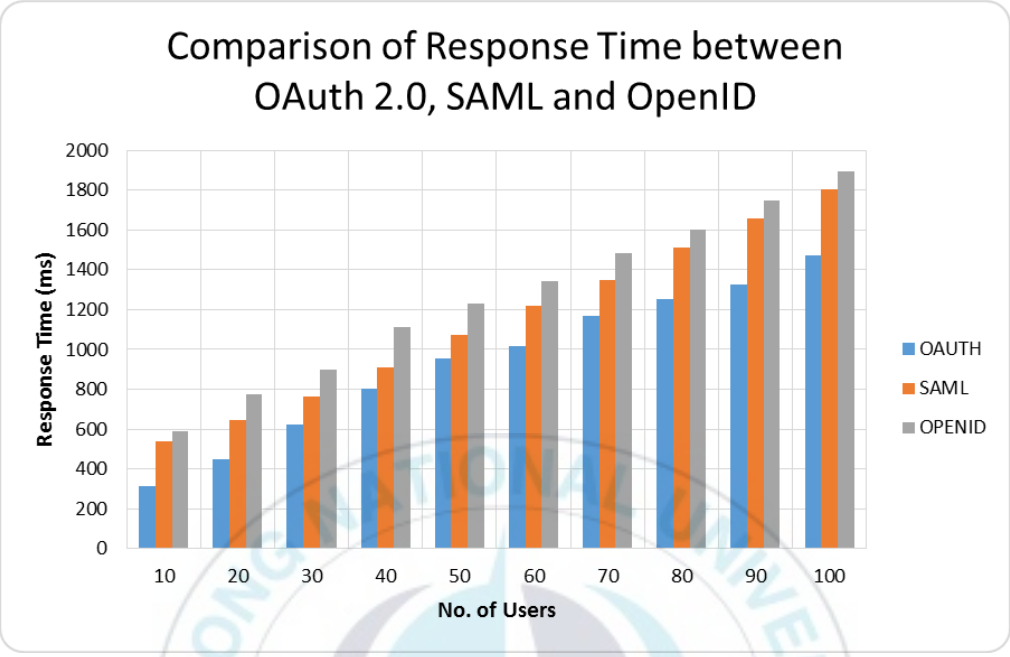
4.3. Performance Evaluation

To evaluate the performance of the server after applying after integration of the OAuth 2.0 security protocol, a load tester JMeter is used. The Apache JMeter is an open source java software application designed to load test functional behavior and measure performance. This works by simulating multiple users to have access to the server concurrently. There are three main parameters that are considered in this simulation.

- Number of threads: This represents the number of users connected to the target website.
- Ramp-Up period: This denotes the time it takes for a user to start a new session.
- Loop count: This denotes the number of request for each user.

The number of users is then varied in sets of tens from 10 all the way to 100 for this particular scenario. The application is set to handle 10 user requests with a Ramp-Up period of 1 second. The Rump-Up period determines how long the next user should take to begin a new session. The table 5 below shows a comparison of the performance results between three different identity management protocols OAuth 2.0, SAML and OpenID.

Table 5: Comparison between performance test results



The result shows that the performance of OwnCloud as the number of users' increases considerably increases the response time.

Using the same values in the required parameters, it was observed that the response time according to the increase of the users was higher in an implementation instance where OpenID protocol was integrated as compared to an instance where either OAuth 2.0 or SAML protocol was used. This means that the integration of OAuth security protocol within the OwnCloud environment resulted in a better performance compared to other identity management protocols.

Source code

Authentication Endpoint

The authentication endpoints used in this implementation include the following ones:

- **/oauth/initiate** - this endpoint is used for retrieving the Request Token.
- **/oauth/authorize** - this endpoint is used for user authorization (Client).
- **/admin/oauth_authorize** - this endpoint is used for user authorization (Admin).
- **/oauth/token** - this endpoint is used for retrieving the Access Token

Table 6: Authentication endpoint code

```
<?php
// INCLUDE LIBS
require('OAuth2/client.php');
require('OAuth2/GrantType/IGrantType.php');
require('OAuth2/GrantType/AuthorizationCode.php');
// SET URL PARAMS
$client_id = 'YOUR CLIENT KEY';
$client_secret = 'YOUR SECRET KEY';
$state = 'test';
$scope = 'brian.profile'; // FETCH BRIAN DATA
$redirect_uri = 'https://localhost';
$authorize_uri = 'https://eu.battle.net/oauth/authorize';
$token_uri = 'https://eu.battle.net/oauth/token';
// CREATE NEW OAUTH2
$client = new OAuth2\Client($client_id, $client_secret);
// IF NO CODE PARAM REQUEST TOKEN
if (!isset($_GET['code'])) {
 $auth_url = $authorize_uri.'?client_id='.$client_id.'&scope='.$scope.
'&state= '.$state.'&redirect_uri='.$redirect_uri.'&response_type=code';
 header('Location: ' . $auth_url);
 die('Redirect');
}
else {
 // ESLE GET TOKEN AND ACCESS DATA
```

```
        $params = array('code' => $_GET['code'], 'redirect_uri' => $redirect_uri);
        $response = $client->getAccessToken($token_uri, 'authorization_code', $params);
        $info = $response['result'];
        $client->setAccessToken($info['access_token']);
        $response = $client->fetch('https://eu.api.battle.net/brian/user/characters');
        var_dump($response);
       }
```

Introspection Endpoint

This specification defines a method for a client or protected resource to query an OAuth authorization server to determine meta-information about an OAuth token. The Introspection Endpoint responds to HTTP POST requests and HTTP GET requests from token holders, particularly including Resource Servers and Clients. The endpoint takes a single parameter representing the token and returns a JSON document representing the Meta information surrounding the token.

Table 7: Introspection endpoint code

```
        <?php
        OCP\User::checkAdminUser();
        OCP\Util::addScript( "user_oauth", "admin" );
        $tmpl = new OCP\Template( 'user_oauth', 'settings');
        $tmpl->assign('introspectionEndpoint',
OCP\Config::getSystemValue( "introspectionEndpoint",
'https://frko.surfnetlabs.nl/workshop/php-oauth/introspect.php' ));
return $tmpl->fetchPage();
```

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "access_token": "907c762e069589c2cd2a229cdae7b8778caa9f07",
    "expires_in": 3600,
    "refresh_token": "43018382188f462f6b0e5784dd44c36f476ccce6",
    "scope": null,
    "token_type": "Bearer"
}
```

Figure 15: Successful request JSON document

5. CONCLUSION

This thesis proposes the implementation of OAuth 2.0 security protocol into the OwnCloud platform environment. This implementation is to enable secure access of the OwnCloud stored data during communication between OwnCloud and other third-party applications. The third-party applications considered in this particular case are the online social networking sites. We have used Facebook and Twitter as the demonstration social networks in order to implement this particular protocol. The implementation has thus proved that the integration has provided secure access by allowing OAuth 2.0 security protocol to act as the identity management tool between OwnCloud and the Social Networking Sites by providing user verification and authentication. OwnCloud is therefore tasked with the authority to grant or deny access to any third-parties and to also determine the scope of what can be accessed or shared. It has also further prevented the conventional method of sharing user credentials which include username and password.

It was also concluded that the access tokens could be used as proof of authentication. Since an authentication usually occurs ahead of the issuance of an access token, it was possible to consider the reception of an access token of any type proof that such an authentication has occurred. Secondly, the access of a protected API could also be used as proof of authentication. Since the

access token can be traded for a set of user attributes, it is viable to assume that possession of a valid access token is enough to prove that a user is authenticated. Authentication of the user who is trying to access OwnCloud via a third party is also established based on the integration of the OAuth 2.0 protocol which acts to establish identity management. The perfomance result testing for OwnCloud with OAuth 2.0 security protocol integration was plotted in a comparison graph with the use of the same values in the required parameters. Based on the performance results, it was observe that the response time with an increase in the number of users was higher in an OwnCloud instance where OpenID and SAML protocols were used as opposed to an instance where OAuth protocol was integrated. This means that the integration of OAuth security protocol within the OwnCloud environment resulted in a better performance.

References

1. D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749 (Proposed Standard), October 2012.
2. ownCloud, “ownCloud’s Architecture Overview” Whitepaper at https://owncloud.com/whitepapers , March 2014.
3. J. Wang, Z. A. Kissel, “Introduction to Network Security: Theory and Practice”, John Wiley & Sons, September 2015, pp 183 – 186
4. Sarath Pillai, “Understanding the working of Secure Socket Layer (SSL)”, Slashroot, January 2015.
5. Feng Yang; Manoharan, S., "A security analysis of the OAuth protocol," in *Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on* , vol., no., pp.271-276, 27-29 Aug. 2013
6. E. Hammer-Lahav, "The OAuth 1.0 protocol," The Internet Eng. Task Force RFC 5849, April 2010.
7. Er. Gurleen Kaur, Er. Deepak Aggarwal, “A Survey Paper on Social Sign-On Protocol OAuth 2.0,” Journal of Engineering Computers & Applied Sciences (ISSN: 2319-5606), Volume 2, No 6, June 2013, pp 93-96
8. S. San-Tsai, K. Beznosov, “Simple But Not Secure: An Empirical

Security Analysis of OAuth 2.0-Based Single Sign-On Systems," CCS′12, October 16−18, 2012

9. U.S. Department of Justice, Federal Bureau of Investigation, "Internet Social Networking Risks" https://www.fbi.gov/about-us/investigate/counterintelligence/internet-social-networking-risks-1
10. Covert Redirect and its real impact on OAuth and OpenID Connect, http://www.thread-safe.com/2014/05/covert-redirect-and-its-real-impact-on.html
11. S. Pai, Y. Sharma, S. Kumar, R. M. Pai, and S. Singh. Formal verification of OAuth 2.0 using Alloy framework. In Proceedings of the International Conference on Communication Systems and Network Technologies (CSNT), pages 655–659, 2011.
12. S. Chari, C. Jutla, and A. Roy. Universally composable security analysis of OAuth v2.0. Cryptology ePrint Archive, Report 2011/526, 2011.
13. Q. Slack and R. Frostig. OAuth 2.0 implicit grant flow analysis using Murphi. http://www.stanford.edu/class/cs259/WWW11/ , 2011.

14. T. Lodderstedt, M. McGloin, and P. Hunt. OAuth 2.0 threat model and security considerations. http://tools.ietf.org/html/ draft-ietf-oauth-v2-threatmodel-01, 2011.
15. OWASP. Open web application security project top ten projects. http://www.owasp.org/, 2010.
16. WhiteHat Secuirty. Whitehat website secuirty statistics report. https://www.whitehatsec.com/resource/stats.html , 2011. [Online; accessed 16-May-2012].
17. NIST. National vulnerability database. http://web.nvd.nist.gov/view/vuln/statistics, 2011.
18. ownCloud, “ownCloud’s Data Encryption 2.0 Model” Whitepaper at https://owncloud.com/whitepapers
19. ownCloud, "Optimizing ownCloud Security" Whitepaper at https://owncloud.com/whitepapers
20. Khash Kiani, “Four Attacks on OAuth - How to Secure Your OAuth Implementation”, A technical study of an emerging open-protocol technology, SANS Working Papers in Application Security
21. Suhas Pai, Yash Sharma, Sunil Kumar, Radhika M Pai and Sanjay Singh. “Formal Analysis of OAuth 2.0 using Alloy Framework” 2011 International Conference on Communication Systems and

Network Technologies. 978-0-76954437-3/11, 2011 IEEE DOI 10.1109/CSNT.2011.141

22. Chetan Bansal, Karthikeyan Bhargavan and Sergio Maffeis. “Discovering Concrete Attacks on Website Authorization by Formal Analysis” 2012 IEEE 25th Computer Security Foundations Symposium, 2012 IEEE 10.1109/CSF.2012.27
23. Xingdong Xu, Leyuan Niu and Bo Meng. “Automatic Verification of Security Properties of OAuth 2.0 Protocol with CryproVerif in Computational Model” 2013 Asian Network for Scientific Information. Information Technology Journal 12 ISSN 1812-5638 /DOI:10.3923/itj.2013.2273.2285
24. A. Santana de Oliveira, G. Serme, Y. Lehmann, "Platform-level support for Authorization in Cloud Service with OAuth 2," Intercloud workshop co-located with IEEE International Conference on Cloud Engineering (IC2E), March 2014.

Acknowledgement

First and foremost, I am grateful to the Almighty God for the good health and sound wellbeing that was necessary to successfully accomplish my graduate study at PKNU. I would like to express my sincere gratitude to my advisor Prof. Kyung-Hyune Rhee for the continuous support of my graduate study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my graduate study.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Kim-Chang Soo, and Prof. Man-Gon Park, for their insightful comments and encouragement which incented me to widen my research from various perspectives. My sincere thanks also goes to Dr. Park and Dr. Chul Sur, who equipped me with the necessary research knowledge and advice through seminars. Without their precious support it would not be possible to conduct this research.

I thank my fellow lab mates Lewis, Bayu, MyeongHak and Sam for the insightful discussions, reassurance, and for all the pleasurable moments we shared in the course of my study. Also, I thank my friends for their support and encouragement. In particular, I am grateful to Bright Gameli Mawudor for enlightening me on the availability of this research and for his invaluable input throughout the writing of this research.

Last but not the least, I would like to thank my family: my parents, my brothers and sister for supporting me spiritually throughout the writing of this thesis and for my life in general.