Introduction to OAuth

Problems with existing authentication mechanisms

We have already looked at HTTP authentication as one way to allow the user of a client (such as the fan site) to authenticate with a web service (such as HitTastic!) However there are problems with this approach:

- Username and password might be sent across in plain text if HTTPS is not used;
- We are trusting the client (e.g. the fan site) to look after our username and password. What if their security is not as good as that of the web service? This is quite a likely scenario as clients are likely to be developed by smaller organisations (with less resources to focus on security) than web services.

What we need, ideally, is some way in which **the user never has to supply their login credentials to the client site at all**. Enter ***OAuth***, a standard protocol for precisely this scenario. OAuth allows client sites to authenticate with services via ***tokens***. With OAuth, we have a ***provider*** which provides an OAuth service (this would typically be the web service) and a ***consumer*** which uses the service (typically the web service client). The user logs in on the ***provider*** site and then receives a ***token*** to access any pages which need authentication.

OAuth needs to be implemented as follows.

Client needs to register with the server

Firstly the client site (consumer) needs to identify itself to the service (provider). This is done as follows:

- Each consumer (client, e.g. the fan site) ***registers*** with the provider (web service). The web service gives the consumer a unique key and secret, both unique codes which can be used to identify the consumer to the provider.
- The consumer includes the key and secret in their code (the secret needs to be hidden, e.g. by putting it in an include file not accessible to the web)
- When registering, the consumer typically provides their name and email (so that the provider knows who they are) and a ***callback*** - a page on

the consumer site which the user will be redirected to once they have logged in on the provider site.

User authentication with OAuth

Once a consumer site has registered itself with the provider, an OAuth client can be written on the consumer. This works as follows:

1. User clicks an "authenticate" link;
2. Request containing the client key and secret sent to the OAuth provider.
3. OAuth provider checks key and secret, and if valid, sends back an ***request token***. This is a temporary code used during the authentication process only. Initially, this is ***unvalidated*** (see below). The request token is paired with a secret, for enhanced security: this is also passed to the OAuth server.
4. If an request token is received, the client then redirects the user to the provider's website, which prompts them for their login details. The login form contains a hidden field with the request token. If the user logs in correctly, and the request token is correct, the provider ***validates the request token*** (typically by updating a field in the database), and ***informs the user that the consumer wishes to perform certain sensitive operations on the provider on the user's behalf*** (e.g. buy products, make bookings, etc).
5. The user then has to agree to the consumer performing such operations on their behalf, typically by clicking a link on a special page on the provider's site (I will demo this in class)
6. The user is then redirected back to the consumer with the validated request token plus a verification code.
7. The consumer then sends the validated request token and secret, and verification code (sent back in the last step) and the provider sends back an ***access token***. This is used from now on for all sensitive operations: each time the client wishes to ask the provider to perform a sensitive operation, the access token will be sent. Since the only way that an access token can be obtained is through the user both logging into the provider site AND agreeing to let the consumer carry out sensitive operations on the user's behalf, this will be secure.
8. Access tokens remain active until the user ***revokes*** them. Typically, a user can log onto the provider's site and view all their active tokens. Each token can then be revoked by the user. Until a token is revoked, that token will remain active: an approach taken by some developers (e.g OSM How Did You Contribute?), is to store the user's access token as a cookie so that they don't get a new token each time they use a consumer; instead they are automatically authenticated using the access token stored in the cookie. An OAuth provider might also wish to provide an API call to revoke a token.

Implementation details

PHP has an OAuth extension which makes it relatively easy to develop OAuth clients and servers. Building an OAuth client is not too difficult.

Writing an OAuth client (consumer)

We use the PHP OAuth extension, which is documented here. A summary of how to develop a client is presented below. You can see this example working on the Neptune server.

When writing an OAuth client we need to use a try/catch to handle any exceptions thrown by OAuth:

```
try
{
    ...
```

Within this try block, the first thing we do is to create a new OAuth object using our client key and secret, e.g:

```
$key='a171b4510fb7996cba0a5ec4a9ff11688913e4af';
$secret='2fb0487e5ad3e56e2f1b818fb6185bb823d20f78';
$oauth = new OAuth ($key, $secret);
```

The best way of writing the OAuth client is to have a single index.php script which responds to actions passed in via a query string, e.g set ***$action*** to either the query string "action" or a blank string if not supplied:

```
$action = isset($_GET["action"]) ? $_GET["action"] : ""
```

The first action we will look at is "login", which will occur if the user adds ***action=login*** as a query string. The code for handling a login is as follows, notice how we first obtain a request token and then, if we have, redirect the user to the provider's login page:

```
if($action=="login")
{
    // We try to get a request token from the provider
    if($requestToken = $oauth->getRequestToken("http://
    {
        // This will return a request token and secret,
        // returned store it in a session for later use
        $_SESSION['request_secret'] = $requestToken['oa

        // Redirect to login page on the oauth server,
        // request token
        header("Location: http://ephp.solent.ac.uk/oaut
```

```
    }
    else
    {
        die("Could not obtain a request token : ".  $oa
    }
}
```

The request token will be embedded in the provider's login form, and checked along with the username and password on the provider. If all correct, the provider will redirect to the consumer (client)'s callback, which is often the index page, passing the request token and a verification code (for extra security) back as query strings. So if these query strings are present,we request an access token via the URL of the provider's access token API:

```
// If the oauth_token and oauth_verifier query string p
// the user has logged in to the provider successfully,
// request an access token
elseif(isset($_GET["oauth_token"]) && isset($_GET['oaut
{
    // function to show page's head section
    showHead();

    // Get the access token by sending the request toke
    // and verifier to the provider
    $oauth->setToken($_GET['oauth_token'], $_SESSION['r
    $at = $oauth->getAccessToken("http://ephp.solent.ac

    // If we get one..
    if($at && !empty($at))
    {
        // We no longer need the request secret - remov
        unset($_SESSION["request_secret"]);

        // Save the access token and corresponding acce
        // sessions so we can reload the page and they
        $_SESSION["access_token"] = $at["oauth_token"];
        $_SESSION["access_secret"] = $at["oauth_token_s

        // Save them in the oauth object as well
        $oauth->setToken($_SESSION["access_token"], $_S

        // Make a web service call which requires an ac
        $user=getUser($oauth);
    }
```

```
    else
    {
        $error="error with access token: ".  $oauth->ge
      }
}
```

Note how we call a web service which requires an access token, via the ***getUser()*** function (our own custom function). The workings of this function will be described below.

Note how we store the access token and secret in session variables, so that if we reload the page, they are still available to us:

```
// If the access token and secret have been saved in se
// to get the user details from the provider's API
elseif(isset($_SESSION["access_token"])&& isset($_SESSI
{
    showHead();
    $oauth->setToken($_SESSION["access_token"], $_SESSI
    $user=getUser($oauth);
}
```

If the user logs out, we destroy the session, but note an alternative approach would be to store the access token in a cookie so that the user could use the site next time without authenticating again.

```
if($action=="logout")
{
    session_destroy();
    header("Location: index.php");
}
```

Finally, if no special query strings were supported we just show the head section as normal:

```
// when we first load the page (no query strings suppli
else
{
    showHead();
}
```

Finally we must close the ***try*** and catch the OAuthExceptions that might be thrown as part of the process.

```
catch(OAuthException $e)
{
    $error = "error:" .$e->getMessage();
}
```

and finish up with some general page logic:

```
// If we have a logged in user, show it.
// Otherwise provide a login link.
if($user!==false)
{
    showUser($user);
}
else
{
    echo "<a href='?action=login'>Login (provided by Bi
}

// Print errors if there were any
if($error!=false)
{
    showError($error);
}

// If it's not a login action, show the page content ar
if($action!="login")
{
    showContent();
    closePage();
}
```

Here is an example of a call to a protected web service on the server. Note that we do not use cURL; we use the ***fetch()*** method of our OAuth object. We check that there were no errors with the response: here, an error will be returned if the JSON contains a field called "error".

```
function getUser($oauth)
{
    $oauth->fetch ("http://ephp.solent.ac.uk/oauth_serv
    $resp = json_decode($oauth->getLastResponse(), true
    if(isset($resp["error"]))
    {
        $e = new OAuthException();
        $e->lastResponse = $resp["error"];
```

```
            throw $e;
        }
        else
        {
            return $resp;
        }
}
```

Writing an OAuth server

Writing an OAuth server is tricky, even with the PHP OAuth extension to help us. There is a lot of work involved in implementing a secure OAuth provider in compliance with the specification; luckily, as web developers you are going to find yourselves consuming OAuth services much more frequently than providing them. For this reason, and due to time constraints, we are not going to develop an OAuth server from scratch, or examine how it is done. That said, it is something you can do if you wish as additional research to implement in the assignment.

On a basic, insecure level you can simply create and check tokens using the OAuthProvider generateToken() method (see the OAuth docs) and store them in the database. However there are additional checks that a secure OAuth server must implement, for instance the ***timestamp and nonce handler***. This is a function which checks that tokens are not older than a certain time (to decrease the chance of tokens being stolen) and also checks the ***nonce***, which is a random string generated on each request (which minimises the chance of 'replay attacks' in which a third-party can intercept the request and repeat it to authenticate as the legitimate user).

To mitigate the complexity of OAuth server development, a simple library (for demo purposes only, not guaranteed to be fully secure) was developed by Freek Lijten and available on GitHub here. A modified version of this, suitable for our use, can be downloaded here. This modified version has been installed on the Neptune server with the following URLs:

- **http://ephp.solent.ac.uk/oauth_server/register.html** - register a new OAuth consumer
- **http://ephp.solent.ac.uk/oauth_server/request_token.php** - get a request token
- **http://ephp.solent.ac.uk/oauth_server/login.php** - login to the provider
- **http://ephp.solent.ac.uk/oauth_server/hittasticlogin.php** - login to the provider (HitTastic version)
- **http://ephp.solent.ac.uk/oauth_server/access_token.php** - get an access token
- **http://ephp.solent.ac.uk/oauth_server/api_user.php** - a sample API call which gets the details of the current user as JSON.

Advanced Exercise

Like promises this is one of the unit's most difficult topics, and is not a requirement for a high grade in the assignment (it is one of many ways in which you can achieve a high grade) and therefore this exercise is ***only recommended for strong web developers confident with the use of query strings to control how a script operates, and happy with object-oriented syntax***. You will likely find it difficult otherwise. An alternative is to finish off the Canvas and further event handling exercises from last week, which you may have missed due to the assignment hand in.

Work on Neptune (ephp.solent.ac.uk); you have a temporary account on Neptune with your Edward2 login details which will be active for the duration of WAD (only) and will be deleted when DFTI begins next term as EPHP will be in use again then. **Note that Neptune deliberately has minimal security (as it's the EPHP server); please do NOT upload anything even slightly confidential to it, such as PHP scripts containing your Edward2 password.**

Create a new copy of your fan site and upload to Neptune. Modify the clientdownload.php page to include OAuth authentication, as in the example above. When a user visits clientdownload.php, the full OAuth process must take place, including a redirection to the provider. Here, HitTastic! is the provider. Use the URLs above to authenticate (use the HitTastic! version of the login script).

You will also need to register your application, see the register url above. This will give you a key and secret.

Test that it works by checking that ***api_user.php*** gives you back valid data, i.e. the details of the current user.

Then, write your own download API, to do ths look at api_user.php in the example above and do something similar. It should take in a song id and quantity as POST data and reduce the balance, use SQL queries similar to the ones in api_user.php. ***To send POST data*** to your web service over OAuth you need to supply it as the second parameter to fetch(), and specify a method of OAUTH_HTTP_METHOD_POST, i.e:

```
$oauth->fetch("url", ["field1"=>"value1"], OAUTH_HTTP_M
```