Lecture Notes for Advanced Web Security 2017

Part 5 — Web Based Single Sign-On and Access Control

Martin Hell

1 Introduction

Letting users use information from one website on another website can in many cases make life more simple for the users. It also allows cooperation between different services on the Internet. Since users have accounts on several different websites, and stores different information with each website, some way of controlling the information flow between the different websites is sometimes needed. These notes will discuss two aspects of this, namely how one username and password can be used throughout several websites without revealing the password to all websites, and how one website can be allowed to access user data on another website. The solutions are more general and involved parties can sometimes be implemented as applications on a device.

2 Web Based Single Sign-On

The web offers a large number of services. Many services require, and offer, the possibility to have personalized content which can be changed at any time from any computer. In these situations, maintaining a state using cookies is not enough. Instead, users are returning to a previous state by authenticating to the service and from that point (session or persistent) cookies are used to maintain the state. At logout time, the cookie is destroyed and the only way to return to the last state is to authenticate again. This will allow users to authenticate on different computers every time, and also allow several users to authenticate to the same service from one computer (but typically not at the same time, although the user-agent potentially can handle such situations).

The exact type of authentication is not important in this context, but the fact that passwords are most often used is one important motivation for developing web based single sign-on solutions. Having to use different usernames and passwords for every single web service has several disadvantages. First, if all passwords are different, users tend to either write them down or to choose bad passwords. Choosing bad passwords can have serious consequences if the database with hashed passwords is stolen since they would most likely be revealed through dictionary or brute force attacks. Writing passwords down on a piece of paper is in many cases a better solution, but exposes the passwords to people that can get physical access to the piece of paper. Another disadvantage is that users must trust every single web service to handle the credentials in a proper way. Numerous examples show that even large websites treat passwords in a very uneducated way, neglecting the importance of a salt value, opening up for TMTO attacks on stolen hashes. Sometimes, the passwords are even stored in clear on the server side.

A web based single sign-on (SSO) solution solves these problems. The user only has to have one password, which can be used to log into all services. There are several proposed ways of actually doing this in practice. Apart from some university developed solutions, the first proposed method was Microsoft's Passport, later changed to Windows Live ID and in 2012 it changed name to Windows Account. This is a solution specific for Microsoft. This section will discuss two more general solutions for web based SSO, namely the Security Assertion Markup Language (SAML) and OpenID. In web based SSO, typically the user is redirected to a sign-on service and is authenticated there before being redirected back to the requested web service. However, the SSO concept also includes the situation where a user first signs on to one ID provider and this authentication will automatically be valid for all other participating web services that are subsequently accessed.

While OpenID is a solution dedicated to web based SSO, with a very defined and simple way of accomplishing it, SAML is a much larger and more abstract framework in which SSO is just one profile.

The different parties involved in an SSO (using the terminology in SAML) are

- **The Identity Provider (IdP)** This is the party that authenticates a subject and makes assertions, also called the *asserting party*. The service provider is relying on the IdP to correctly authenticate the subject. In OpenID, this is called the OpenID provider (OP).
- **Subject** The subject is typically an end user that wishes to sign in to some service provider.
- **Service Provider (SP)** This is the web service that the subject wishes to log into and that relies on the IdP to authenticate the subject. It is thus also called the *relying party* as it relies on the assertions made by the asserting party. In OpenID, this is called the relying party (RP).

2.1 SAML and the Web Browser SSO Profile

The current version of SAML is 2.0, which was finalized in 2005. It is a framework for describing and exchanging security information, developed and maintained by the Organization for the Advancement of Structured Information Standards (OASIS). SAML is based on four main notions.

- **Assertions** An assertion is a statement about a subject. There are three kinds of assertion statements possible. An *authentication* statement says that a subject was authenticated at some time by some means. An *attribute* statement says that a subject is associated with some specific attribute, and an *authorization decision* statement grants or denies access for some subject to some resource.

- **Protocols** A protocol describes how assertions should be exchanged in a certain situation. The authentication request is an example of a protocol.
- **Bindings** The bindings describe how the assertions should be transported to another party, e.g., using the HTTP or SOAP protocols.
- **Profiles** A profile defines how the assertions, protocols and bindings are to be used in a specific usage scenario. One profile is the SAML Web Browser SSO Profile, which is the one used for web based SSO.

SAML is a modular and abstract framework. The core functionality is given in [5] and some specific profiles are described in [7]. A technical overview of SAML can be found in [8]. The description of SAML given here is focused on the Web Browser SSO Profile. Within this profile, there are different options for realizing the SSO. One option that affects the communication is if the SSO is initiated by the IdP or the SP. The most common is that the communication is initiated by the SP. Another option is how the messages are transferred, i.e., the bindings, between the IdP and the SP.

The Web Browser SSO Profile uses the SAML Authentication Request protocol together with the HTTP Redirect, HTTP POST and HTTP Artifact bindings. The user is assumed to have a standard web browser without any special extra capabilities.

The profile defines the following steps for achieving SSO.

1. The subject tries to access a resource at the SP.
2. The SP determines the correct IdP to use for authentication.
3. The SP sends an authentication request <AuthnRequest> to the IdP via the subject. This message may be digitally signed.
4. The IdP authenticates the subject in some way. It could be the case that the subject already has an authenticated session with the IdP. Then, no new authentication is needed.
5. The IdP sends a <Response> message back to the SP with some proof that authentication was successful.
6. The SP verifies that the IdP authenticated the subject and the subject is allowed access to the requested resource.

The exact way in which the subject is authenticated in step 4 above is not specified. It could be a password, but it could also be some other type of authentication mechanism. SAML also does not explicitly specify how the SP determines which IdP to contact in step 2. SAML specified another profile, called the *SAML identity provider discovery profile* that can be used for this, but it can also be achieved in other ways. The SP could use a separate service to identify the IdP and then the <AuthnRequest> could be relayed through this service. This is an example of the modularity provided by SAML.

If the IdP initiates the communication, the communication starts at step 5 above.

In order for the SP to verify that the IdP has authenticated the subject, the SP and the IdP must have some prior agreement that defines how this is done.

2.1.1 Authentication Request Protocol

There are a large number of protocols in SAML. In the Web Browser SSO profile, the Authentication Request Protocol is the protocol that is used when the SP wants an IdP to authenticate a subject. It is used in step 3 above. The protocol specifies how the authentication request data is described when it is sent from the SP to the IdP. The main element is the <AuthnRequest> and it can contain information about the subject that an assertion is requested for. It can also contain other elements that describe which IdP, or a list of IdPs, that are trusted by the requester. Details about the protocol can be found in [5].

The <AuthnRequest> message sent from the SP to the IdP results in a <response> messages, containing one or more assertions about the subject.

2.1.2 Bindings

A binding is a way to transmit the messages between the IdP and the SP. The specifications for the different protocol bindings for SAML can be found in [6]. In Web Browser SSO, there are two types of messages sent. The SP sends an *authentication request* to the IdP and the IdP sends back a response with an assertion to the SP. The authentication request can either be sent as an HTTP redirect, as an HTTP POST or as an artifact binding. The response message can be an HTTP POST or an artifact binding. The sender is free to choose the binding it finds most appropriate and the choice depends on, e.g., the message size.

An HTTP redirect binding is a 302 or 303 HTTP redirect response which simply redirects the browser to another server using the *location* header. In an authentication request, the request is given in a URL query parameter called SAMLRequest. In order to remember the current state the SP also submits a query parameter named RelayState, which is to be returned by the IdP.

```
http://IdP-example.com/redirect?SAMLRequest=req&RelayState=token
```

The user-agent will now send an HTTP GET request with the authentication request data to the server at IdP-example.com. The response is not allowed to use the HTTP redirect binding since a response (assertion) typically is too large to fit within the maximum URL length supported by user-agents.

In an HTTP POST binding, the message is sent through the user-agent in a POST using an HTML form. When it is an authentication request, the SP sends an HTML page to the subject which contains an HTML form with hidden fields. Then JavaScript is used to automatically submit this form to the IdP.

```
<form method="post" action="https://IdP-example.com/redirect>
<input type="hidden" name="SAMLRequest" value="req" />
<input type="hidden" name="RelayState" value="token" />
<input type="submit" value="Submit" />
</form>
```

The situation is similar for a response message. The response is named SAMLResponse and the RelayState is also sent back together with the response. When HTTP POST binding is used for the response, the assertion is sent in an HTML form to the subject, which in turn automatically submits the form to the SP. The assertion is digitally signed by the IdP. If the SP accepts the assertion, the value of the RelayState token is used to reconstruct the state and an HTTP redirect is sent by the SP to the subject in order to redirect the subject to the SP.

HTTP redirect and POST bindings deliver the message to the recipient immediately. For an artifact binding, the Artifact Resolution profile is used. An artifact is a reference to a request or a response and it is used in a situation when the user-agent will not agree to relay the complete message in the URL or in an HTML form. When seeing the artifact, the receiver makes a callback directly to the sender and the message is exchanged directly between the two parties. This communication in turn can use a different binding, such as a SOAP binding.

2.1.3 Messages

SAML assertions and the protocol data sent are encoded in XML. The XML elements that are used to construct the assertion are given in the SAML core document [5]. Elements can, e.g., be <subject> which describes the subject for which the assertion is made. This element has also a number of subelements. For the actual statements there are the elements <AuthnStatement>, <AttributeStatement> and <AuthzDecisionStatement> corresponding to the three possible assertion statements possible.

2.1.4 Other Profiles

There are several other SSO profiles that take advantage of the SAML framework. One is the Enhanced Client and Proxy (ECP) Profile which supports SSO with clients that have more capabilities than plain web browsers. The Single Logout Profile is used to simultaneously logout from all services that a user has been logged into using SSO. There are also other types of profiles that are not directly related to SSO.

2.2 OpenID

OpenID has many similarities with the Web Browser SSO profile in SAML, but the protocol is more lightweight and specific in the description. The latest specification of OpenID 2.0 was finalized in 2007 and is given in [9]. This section will focus on that version.

OpenID is an open source project with free open source plugins available for, e.g., WordPress, and there are also modules available for Apache.

The term OpenID Provider (OP) is used to denote the party corresponding to the IdP in SAML and Relying Party (RP) corresponds to the SP in SAML. The subject is called an end user.

2.2.1 The OpenID Protocol

The main purpose of the protocol is for an end user to prove that it is in control of an identifier, e.g., a URL. This is actually just what happens when logging in to a service, namely that the end user proves, using a password, that it is in control of a username. Here the username is replaced by a URL, which is globally unique, and the user proves it is in control of this.

Following [9], the authentication protocol can be summarized as follows.

1. The user tries to access the RP by presenting a *user-supplied identifier* to the RP.
2. The RP uses the identifier to determine which OP to use for authentication. This is called *discovery*.
3. Diffie-Hellman is used to establish an association containing a shared secret between the RP and the OP. The OP signs messages and the RP verifies the messages using this shared secret. This step is optional and variants exist, see Section 2.2.4.
4. The RP redirects the user-agent to the OP with an *authentication request*.
5. The OP authenticates the user.
6. The user-agent is redirected back to the RP with a message claiming that the user is controlling the identifier (or that authentication failed). This message is authenticated using the shared key between the OP and the RP.
7. The RP verifies the message and allows the user to access the resource. The verification can be done either by the RP sending a direct request to the OP or by the OP sending a signed response to the RP via the user-agent.

This procedure is very similar to the one used in SAML. A notable difference is step 3 in OpenID where the RP and OP can establish an association even though they have no prior agreement or knowledge of each other. This step is optional since they might have an association since before. In that case, a new one is not needed.

The specification differs between *direct communication* and *indirect communication*. Direct communication can only be initiated by the RP. The RP sends direct requests to the OP. This occurs when an association is being established (step 3) and when the authentication is verified in step 7. Indirect communication can be initiated by either the RP or by the OP. They are indirect in the sense that the messages are passed through the user-agent. Indirect communication is used in authentication requests (step 4) and authentication responses (step 6). The indirect communication are HTTP requests and the information can be sent using either HTTP redirects or HTML form submission using the HTTP POST method. This is a subset of the alternatives in the SAML Web Browser SSO profile.

For the exact parameters that are sent in an authentication request, refer to the specification [9].

The message authentication algorithms specified for use in OpenID 2.0 are HMAC-SHA1 and HMAC-SHA256, with the latter being the recommended algorithm.

2.2.2 Identifiers, XRI and URL

OpenID distinguishes between four different types of identifiers

- OP Identifier. This is an identifier for an OpenID provider. It defines which OP the end user is using, e.g., http://www.myopenid.com/.
- Claimed Identifier. This is the identifier that the user claims to own and the identifier that the RP should use when saving data about a user, i.e., the account should correspond to this identifier.
- User-Supplied Identifier. This is the identifier that the end user is presenting to the RP and that the RP will use for discovery. After discovery, the RP will get either an OP Identifier or a Claimed Identifier.
- OP-Local Identifier. This is an identifier that the user has locally with the OP.

The User-Supplied Identifier provided by the end user to the RP is first normalized. There are two variants of the identity that is supplied by the user, namely an Extensible Resource Identifier (XRI) or a URL.

An XRI is an identifier that can be used on the Internet. The idea is that one XRI can resolve to a user's email address, skype name, personal webpage and many other things. By just typing the XRI in the "To:" field in an email client, the correct email address is resolved and the email is delivered to that address. The XRI resolves to an XRDS document (similar to a URL being resolved to resource record in DNS), which is an XML document with information related to the identifier, such as the preferred email address. If a user wants to change email address or any other information, he or she just changes the information in the XML document, while leaving the XRI unchanged. For OpenID, a user can change between OPs while keeping the same XRI. XRIs starting with "=" typically refers to a person while XRIs starting with "@" refers to companies. These symbols are called *global context symbols*. The actual names are called *I-names*. Browsers and operating systems can not currently resolve XRIs themselves, but it is possible to use a proxy URL, such as http://xri.net/. To go to the website of a company with I-name @company, one can type http://xri.net/@company in the web browser address bar. Exactly which type of information from the XRDS document is used is situation specific. An email client would use email information, while a browser would use web page information. I-names can be registered for free at, e.g., http://freexri.com but there are also commercial alternatives providing more features.

It is also possible to use a URL as an identifier. The URL location can contain either an HTML page or an XRDS document as will be shown in Section 2.2.3.

2.2.3 Discovery

The first action taken by the RP is to perform discovery, i.e., to get information about how to proceed with the authentication request. The RP must, e.g., know which OP is handling the supplied identifier. The discovery is either based on finding an XRDS document with the necessary information, or by finding an HTML page which holds the information.

XRDS-based Discovery uses information given in an XRDS document and can be performed both if the User-Supplied Identifier is an XRI or a URL. The User-Supplied Identifier is after normalization either an OP Identifier or a Claimed Identifier. If it is an OP Identifier, the XRDS document will provide an Endpoint URL that the RP will use for authentication. The Claimed Identifier will then be chosen by the user when authenticating to the OP. As an example, if the user chooses http://www.exampleOP.com/id/ as (normalized) identifier, by e.g., clicking a link or a button, this URL can host a XRDS document with the following information.

```
<Service>
  <Type>http://specs.openid.net/auth/2.0/server</Type>
  <URI>https://exampleOP.com/auth</URI>
</Service>
```

The type element is a static string indicating that this is information for an OP Identifier. The URI element gives the Endpoint URL that the RP should use for retrieving authentication information. Google provides a comprehensive overview of how this is done when Google is used as an OP [1].

If the User-Supplied Identifier is a Claimed Identifier, the XRDS document will also contain an Endpoint URL, but it can optionally be accompanied by an OP-Local Identifier. As an example, a user wants to identify himself using the User-Supplied Identifier "=john.doe" and submits that XRI to the RP as xri://=john.doe. The RP normalizes this to =john.doe" and then either uses a proxy URL to get the corresponding XRDS document, or it makes its own XRI resolution to retrieve the document. The document may contain the following information.

```
<Service>
  <Type>http://specs.openid.net/auth/2.0/signon</Type>
  <URI>https://exampleOP.com/auth</URI>
  <LocalID>https://john-doe.exampleOP.com/</LocalID>
</Service>
```

The type element specifies the protocol version, the URI element is the Endpoint URL which tells the RP whom to send the authentication request to, and the LocalID is the OP-Local Identifier. In this way, the user can switch to another OpenID provider, while keeping the same Claimed Identifier.

If a URL, instead of an XRI, is used it is normalized by, e.g., adding the "http://" protocol identifier if it is not already there and stripping off the fragment identifier # and anything that follows it. For URLs, the Yadis protocol is used to retrieve an XRDS document. Yadis is a protocol for retrieving an XRDS document from a URL. The RP takes the URL, called Yadis URL, and makes an HTTP request to that URL. The server then responds either with the Yadis (XRDS) document, or a link to such a document.

HTML-based Discovery is used if the User-Supplied Identifier is a URL and the Yadis protocol is not successful. The URL provided by the user to the RP is then the Claimed Identity which points to an HTML document with the OP Endpoint URL and the OP-Local Identifier in the HEAD section of the document. If John Doe owns the domain johndoe.com, and uses exampleOP as an OpenID provider, having https://john-doe.exampleOP.com/ as the OP-Local Identifier, he could host an HTML document at http://myid.johndoe.com with the following in the `<head>` section of that document.

```
<link rel="openid2.provider openid.server"
      href="https://exampleOP.com/auth"/>
<link rel="openid2.local_id openid.delegate"
      href="https://john-doe.exampleOP.com/"/>
```

By using myid.johndoe.com as the User-Supplied Identifier, the RP will first normalize this to http://myid.johndoe.com/ which will be the Claimed Identifier. When performing discovery, the RP will find that the user is using https://john-doe.exampleOP.com/ as an OP-Local Identifier with the OP that can be reached at https://exampleOP.com/auth.

2.2.4 Associations and Verification of Assertions

The secret key shared between the RP and the OP is negotiated using Diffie-Hellman key exchange. It is also possible for the OP to send the secret key in clear text to the RP, but this must never be used without some other protection of the messages, e.g., SSL/TLS.

A handle to the association is included in the authentication request message sent by the RP to the OP. In the response, the OP computes a MAC on the assertion using the key and the algorithm defined in the association. The RP verifies the MAC to make sure that the message originated from the OP that it has an association with. Included in the response is also a nonce to protect against replay attacks. It may be the case that there is no association between the RP and the OP, or it could be the case that the OP thinks that the secret key is invalid or compromised. Then, the OP creates a private association and computes a MAC with this. Upon receiving the response, the RP initiates direct communication with the OP in order to verify the MAC.

2.3 Security

It is important to understand the security provided by the SSO solutions. When logging in to a website, the user and the website itself often have different interests and the exact interest can depend on the application. A user's main interest is that no one else will be able to use his or her account. The website, on the other hand, might have an interest in knowing the true identity of the user logging in. This is the case when accountability is required. Operating systems in multi-user environments is a typical example of this. Users cannot be held accountable for actions if they are not properly authenticated since authentication is the basis for correct event logging. Sensitive user information that is accessible online also requires the web service to know the true identity of the user. This can only be provided by the web based SSO technologies if the SP/RP can trust the IdP/OP to collect, maintain and provide such information. However, many online services do not require the SP/RP to know the true identity of the users, e.g., blogs, social networks, email providers or any other service that just provide personalized websites. OpenID is particularly appropriate in these applications.

SAML requires a trust relationship between the SP and the IdP. This is not the case in OpenID, where a user can (potentially) use any OP with any RP. This opens the possibility for malicious RPs in OpenID to attempt phishing attacks against users. The RP can write its own fake IdP webpage identical to a legitimate IdP login page and redirect users to that page. Once the user enters the credentials, the fake login page can redirect the user back to the malicious RP. Thus, the RP can steal the OpenID login credentials if the user is not careful.

OpenID is also vulnerable to man-in-the-middle attacks if SSL/TLS is not used between the OP and RP.

Several large websites, or companies such as Microsoft, have their own variant of SSO which is neither OpenID nor SAML, but use similar ideas. By using plugins from Facebook, website owners can let users log in to their website using their Facebook account. In a similar manner, many "implementations" of OpenID have their own variant of the protocol and they do not necessarily follow the OpenID protocol in all aspects. This sometimes introduces weaknesses that can not be found by analysing the protocol, as the weakness is in the implementation or in false assumptions made by implementers. An analysis of several real SSO implementations can be found in [10].

3 OAuth

An increasing number of web services results in users storing different kinds of information and data with different services. In some situations, the user may want to allow one website, or application, to access data stored with a web service. The OAuth protocol is designed for this situation. It allows one party to access some specified resources on an HTTP based web service without having the user disclose any login credentials. In a more general sense, a user can delegate access to the web service's API to a client. OAuth 1.0 was proposed in 2007, and published as RFC 5849 [2] in 2010 after being moved to IETF for standardization. OAuth v2 is the most recent version [3]. This section will be based on OAuth 2.0. Several examples are also inspired by that document. It should be noted that OAuth 2.0 is not backwards compatible with OAuth 1.0.

The different parties involved in OAuth are the following:

- **Resource Server** This is the server that hosts the resources that a client will get access to.
- **Resource Owner** This is typically an end user that uses the resource server to store some resources, e.g., files or information.
- **Client** This is the party that gets limited access to resources on the resource server on behalf of the resource owner. It can be another web service on a web server, an application running in the browser, or a native application running on the computer or on another device.
- **Authorization Server** This is the server that issues access tokens for the resource server to the client. It can be the same server as the resource server but it can also be another server that the resource server trusts.

Whenever sensitive credentials are sent across the network in cleartext, TLS must be used. This is assumed throughout the description.

3.1 Protocol Overview

The protocol steps in OAuth can be summarized as follows.

1. The client sends an authorization request to the resource owner. This is typically done through the authorization server using redirects. The client first redirects the user-agent to the authorization server.
2. At the authorization server, the resource owner provides its credentials, e.g., username and password and acknowledges that the client is allowed access to a particular resource.

3. The resource owner issues an authorization grant to the client. Continuing the previous example, this can be done by the authorization server redirecting the user-agent back to the client with the grant.

4. The client sends the authorization grant to the authorization server and asks for an access token to the resources. The client authenticates with the authorization server.

5. The authorization server issues an access token based on the information given in the authorization grant.

6. The client uses the access token to query resources from the resource server.

The client must register with the authorization server before initiating the protocol in order to provide, e.g., its redirection URI. The client also receives a unique client identifier that can be used to identify the client when it is communicating with the authorization server, e.g., through redirection of the user-agent.

An authorization code is the most commonly used authorization grant, but there are other alternatives available that are suitable in some special situations.

The client can be running completely separated from the resource owner, e.g., on a web server which is accessible by the resource owner. These clients are called *confidential* as their credentials are not available to the resource owner. If the client is run in the web browser or as a native application it is assumed that the resource owner has full access to the client credentials. These clients are called *public*. Confidential clients using an authorization code grant will be the focus in the sequel.

3.2 Obtaining a Grant

In the first step, the client sends an authorization request, typically by redirecting the resource owner's user-agent to the authorization server. An example of a URL encoded GET HTTP request is given below.

```
GET /oauthreq?response_type=code&client_id=nSv89drLh
    &state=jSasdhfia4y&scope=read,make
    &redirect_uri=https%3A%2F%2Fclient%2Ecom%2Foauth HTTP/1.1
Host: server.com
```

The *response_type* specifies which type of grant the client requests, in this case an authorization code, and the *client_id* specifies the identifier for the client. These two parameters are required. The *state* parameter can be used by the client to maintain a state while waiting for a response. The authorization server will return the same state value. The *redirect_uri* is the URI used by the authorization server when responding and the *scope* can be used by the client to specify which type of access it requests. It is a comma separated list of strings and the options available are defined by the authorization server.

Upon receiving the authentication request, the authorization server authenticates the resource owner and checks so that the owner agrees to allow the client access to the resources. Exactly how this is done is not included in the specification, but is instead up to the authorization server. The server redirects the user back to the client.

```
GET /oauth?code=hdjE75hjGDbsju35h9&state=jSasdhfia4y HTTP/1.1
Host: client.com
```

The *code* is the authorization code to be used when the client requests an access token and the *state* is the same string as initially supplied by the client in the authorization request.

3.3 Obtaining an Access Token

Using the code, the client can request an access token from the authentication server. This request is sent directly to the authentication server and is not redirected through the user-agent. It is sent in a POST message in the same way as when HTML forms are submitted. When the client authenticates to the authorization server it can, e.g., do so using Basic Access Authentication through the authorization HTTP request header.

```
POST /tokenreq HTTP/1.1
Host: server.com
Authorization: Basic c2VydmVyLmNvbTpwYXNzd29yZA==
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&code=hdjE75hjGDbsju35h9
&redirect_uri=https%3A%2F%2Fclient%2Ecom%2Foauth
```

The *grant_type* specifies which grant is used, an authorization code in this case. The authorization code previously received is sent in the *code* parameter. The *redirect_uri* parameter, if present, must be the same as the one that was sent in the authorization request. If the client is not authenticating with the authorization server, the *client_id* must also be present.

If the authorization server accepts the request, i.e., the code is valid, the client authenticates correctly and there are no other errors, it responds with an access token. The data is formatted as a JSON structure. An example is given below.

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"Gfr53SSwfwUnb9kGd4XaeCBV",
  "token_type":"example",
  "expires_in":3600,
  "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
  "scope":"read"
}
```

The cache-control and pragma HTTP headers are used to make sure that the access token is not cached. The access_token, token_type and expires_in define the token, what type it is and for how many seconds it is valid. The type defines how the client should use the token in order to access protected resources. It is also possible for the authorization server to issue a refresh token, which can be used to request a new access token when the current one has expired, without requesting a new grant. If the scope is different from the one sent in the authorization request, the authorization server must specify which scope the token is valid for.

Now, the client can access protected resources on the resource server using the access token. The access is granted as long as the token is still valid.

3.4 Other Authorization Grants

Apart from the authorization code, there are three other variants of grants. In an *implicit* grant, the client immediately receives the access token as a result of an authorization request. It does not go through the intermediate step of receiving a grant, which is used to request an access token. Thus, the grant is implicit in this case. This affects the security in several ways. One is that the resource owner, and potentially other parties, can see the access token.

Another type of grant is the *Resource Owner Password Credentials Grant*. In this case, the resource owner provides the client with its login credentials, e.g., the username and password, for the authorization server. Using the login credentials, the client receives an access token from the authorization server. This is only suitable in situations where the resource owner fully trusts the client.

The last variant is the *Client Credentials Grant*. In this case the client receives an access token directly from the authorization server upon submitting its own credentials. The resource owner is not involved in the communication. This method can be suitable if the client is in control of the resources itself.

3.5 Security in OAuth 2.0

The security of the communication in OAuth heavily relies on the use of SSL/TLS. The access token, which the client uses instead of username and password (or some other credentials), is basically as sensitive as a password, though the scope and lifetime is more limited. Since the access token is submitted in the clear (on application level) between first the authorization server and the client, and later between the client and the resource server, these connections must be protected by SSL/TLS. Some other transport layer security protocol could in principle be used, but the specification explicitly specifies TLS.

The generation of access tokens must also be secure. Similar to cookies, which are also used as replacement for username and password and very similar in concept to the access tokens, it must not be easy to predict the value of the token.

The specification includes a number of security considerations and a more comprehensive analysis of the security can be found in [4] (draft).

References

[1] Google. Federated login for Google account users, 2012. Available at: *https://developers.google.com/accounts/docs/OpenID*.

[2] E. Hammer-Lahav. The OAuth 1.0 Protocol. RFC 5849 (Informational), April 2010. Available at: *http://www.ietf.org/rfc/rfc5849.txt*.

[3] D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749 (Standards Track), October 2012. Available at: *http://www.ietf.org/rfc/rfc6749.txt*.

[4] T. Lodderstedt, M. McGloin, and P. Hunt. OAuth 2.0 Threat Model and Security Considerations draft-ietf-oauth-v2-threatmodel-06, 2012. Available at: *http://tools.ietf.org/html/draft-ietf-oauth-v2-threatmodel-06*.

[5] OASIS. Assertions and protocols for the OASIS security assertion markup language (SAML) v2.0, 2005. Available at: *http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf*.

[6] OASIS. Bindings for the OASIS security assertion markup language (SAML) v2.0, 2005. Available at: *http://docs.oasis-open.org/security/saml/v2.0/saml-bindings-2.0-os.pdf*.

[7] OASIS. Profiles for the OASIS security assertion markup language (SAML) v2.0, 2005. Available at: *http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf*.

[8] OASIS. Security assertion markup language (SAML) v2.0 technical overview, 2008. Available at: *http://www.oasis-open.org/committees/download.php/27819/sstc-saml-tech-overview-2.0-cd-02.pdf*.

[9] OpenID. Openid authentication 2.0, 2007. Available at: *http://openid.net/specs/openid-authentication-2_0.html*.

[10] X. Wang R. Wang, S. Chen. Signing me onto your accounts through Facebook and Google: a traffic-guided security study of commercially deployed single-sign-on web services. In *33th IEEE Symposium on Security and Privacy*. 2012.