Security Analysis of Double Redirection Protocols

Francisco Corella and Karen P. Lewison

Pomcor

February 7, 2011

Update (November 23, 2011)

There is an error in Section 3.1 SAML Web Browser SSO Profile. The *recipient* attribute in the *subject confirmation data* of a bearer assertion is not optional. It is required, and it must be verified by the service provider. We are grateful to Brian Campbell for pointing out the error.

Abstract

In a double-redirection protocol, such as OpenID or OAuth, an application redirects the user's browser to a third-party that interacts with the user before redirecting the user back to the application; the third-party identifies the application to the user, authenticates the user, and asks for permission to identify the user to the application and grant the application access to resources and services on behalf of the user. In this paper we analyze the security of the double-redirection mechanism, as well as the security of several specific double-redirection protocols. We also propose solutions to several outstanding problems concerning the design of secure double-redirection protocols. We conclude by arguing the need for a double-redirection protocol that provides strong security for unregistered applications.

Acknowledgements

This work was supported in part by NSF grant 1013594.

The ideas presented here originated from a proposal for the social login protocol PKAuth [1] and have been shaped by several discussions on the OAuth and OpenID mailing lists. We are very grateful to all those who participated in the discussions, and to Marius Scurtescu for a detailed reading of an early PKAuth paper and very useful comments on the paper.

Table of Contents

1. Introduction

A double-redirection protocol is a security protocol where an application redirects the user's browser to a third-party that interacts with the user before redirecting the user back to the application. The third-party identifies the application to the user, authenticates the user, and asks for permission to identify the user to the application and grant the application access to resources and services on behalf of the user.

The application may be a traditional Web application running on a Web server; a rich application running within the browser and implemented, e.g., in JavaScript, Flex or Silverlight; or a native application running on a desktop or mobile platform. The third party may be, for example, a social site such as Facebook, a microblogging site such as Twitter, a photo-sharing site such as Flickr, a multipurpose Web site such as Yahoo, Google or Windows Live, a dedicated Web identity provider, a cloud services identity provider, or an enterprise authorization server. The third party may grant the application access to resources and services that it controls, or to resources and services controlled by fourth parties.

The prototypical double-redirection protocols are OpenID [2] and OAuth [3], but there are others, including combinations of OpenID and OAuth [4,5,6] and the Web Browser Single-Sign-On (SSO) Profile of SAML [7]. OpenID and OAuth are described respectively as an authentication protocol and an authorization protocol, but the distinction between authentication and authorization is not sharp, and the two protocols have overlapping functionalities and use cases. Double-redirection protocols are becoming key elements of the Web ecosystem, as they are used for social login [8]. Social login adds to the advantages of Web SSO the compelling advantage for the application of gaining access to the user's social context.

Prior work on the security of double-redirection protocols includes security-consideration sections of protocol specifications, criticism of OpenID [9], criticism of OAuth 2.0 by the very editor of the specification [10], and discussions of the OAuth 2.0 specification, which is still under development, on the mailing list [11] of the IETF OAuth working group [12]. We will refer to this prior work throughout the paper.

In this paper we add the following contributions to the prior work:

1. We describe a recently-discovered attack against double-redirection protocols to which both OpenID and OAuth are vulnerable;
2. We specify conditions that we believe to be sufficient for achieving security with a double-

redirection protocol;

3. We propose new building blocks for designing secure double-redirection protocols, including:
   (a) Methods of identifying the application to the user by means of the application's ordinary TLS certificate and/or a holder-of-key assertion;
   (b) A new method for authenticating the application without ad-hoc signatures;
   (c) Methods for authenticating rich applications and native applications without exposing application secrets to the user; and
   (d) New methods for passing access tokens to rich applications and native applications.

The paper is organized as follows. In section 2 we discuss the security of the double-redirection mechanism itself, independently of any particular protocol. In section 3 we discuss the following protocols: SAML Web Browser SSO Profile, OpenID 2.0, OAuth 1.0, WRAP, and OAuth 2.0. In section 4 we propose solutions to outstanding security problems and building blocks for designing secure double-redirection protocols. In section 5 we conclude by arguing the need for a double-redirection protocol that provides strong security for unregistered applications.

2. Security of Double Redirection

A double-redirection protocol consists of:

1. The following core steps, in sequence:
   (a) (First redirection.) The application redirects the browser to a third party user-interaction endpoint, requesting user data which may include identity data, and/or permission to access resources or services on behalf of the user.
   (b) The third party identifies the application to the user, describes to the user the request made by the application, asks for permission to fully or partially satisfy the request, and authenticates the user.
   (c) (Second redirection.) If permission was granted in step 1(b), the third party redirects the browser to a callback endpoint of the application, sending a user identifier and/or user data and/or a token; the token may be an access token that provides access to requested resources or services, or a provisional token that the application exchanges for an access token.
   (d) The application verifies and/or obtains data. If the response conveyed by the second redirection in step 1(c) included a user identifier and/or user data, the application may verify the origin of the data by verifying a signature included in the response. If it included an access token (or a provisional token that was exchanged for an access token) and the protocol is used for social login, the application uses the access token to obtain a user identifier and/or user data from the third party.
   (e) If the response conveyed by the second redirection in step 1(c) included an access token (or a provisional token that was exchanged for an access token), the application uses the access token to request access to resources and services on behalf of the user at access endpoints controlled by the third party or by fourth parties.
2. An application authentication step, which is placed at different positions in the above sequence by different protocols, and is omitted by some protocols.

In OAuth 1.0, WRAP and OAuth 2.0, the third party sends a provisional token rather than an access token in step 1(c), presumably to compensate for the fact that a non-secure channel is used to send the token. (The provisional token is the request token of OAuth 1.0, the verification code of WRAP, and the authorization code of OAuth 2.0.) We shall see that the use of a provisional token does not achieve security if the channel is not secure, and we believe that it does not improve security if the protocol is otherwise designed correctly.

Several methods are available for implementing each of the two redirections:

- The source of the redirection (the application in the first redirection, the third party in the second redirection) may initiate the redirection by sending an HTTP redirect response.
- The source of the redirection may initiate the redirection by sending an HTTP response that contains a form and a script that submits the form automatically.
- When the application is a rich application, it may implement the first redirection by sending an HTTP request from a pop-up window or frame.
- When the application is a native application, it may implement the first redirection by launching a browser [13, section 2.3] pointed to the target of the redirection. (Section 2.3 or [13] suggests using an external browser or an embedded browser. An embedded browser, however, would defeat one of the purposes of OAuth, since it would allow the application to observe the user's credentials for the third party.)
- Many different methods have been proposed for implementing the second redirection when the application is a rich application or a native application [13,14,15].

Each redirection has two steps: a first step that transfers data from the source of the redirection to the browser, and a second step that transfers data from the browser to the target of the redirection. When the second step is an HTTP GET request we refer to the redirection as a GET redirection. When the second step is an HTTP POST request we refer to the redirection as POST redirection. A GET redirection has the drawback that data sent with the redirection is included in the redirection URI, which may cause it to leak via referrer headers, browser history, or server logs.

If the user is not logged in, the first redirection will typically take the user to an HTML page containing a login form. A basic defense against phishing attacks is for the user not to submit credentials on a page whose URI does not use the https scheme. Therefore the third party's user-interaction endpoint must be protected by TLS in order to be acceptable to knowledgeable users.

It is a widespread misconception that TLS is not needed to protect the callback endpoint. In fact, it is essential. Without it, double redirection cannot provide security. Existing double-redirection protocols, including OpenID and OAuth, fail to require TLS and are vulnerable to both man-in-the-middle and passive attacks.

To mount a man-in-the-middle attack, the attacker can, for example, use an ordinary laptop to implement a rogue WiFi access point, using off-the-shelf hacker tools [16]. The attacker can then route all traffic from the laptop of a legitimate user (the victim) to the Internet through the attacker's machine. When the attacker observes the unencrypted HTTP request from the victim's machine to the callback endpoint, the attacker can read any user data contained in the request. The attacker forwards the request to the callback endpoint, but then the attacker disconnects the victim's machine and takes over the interactions with the application. If the protocol is used for social login, the application will log in the user and return an authentication cookie with the HTTP response, which the attacker will install in its own browser, thus impersonating the victim. If a token was sent to the callback endpoint, whether provisional or not, the attacker will be able to use the application to access resources and services using the token, on behalf of the victim but for the benefit of the attacker, after the application has exchanged the token if provisional.

To carry out a passive attack the attacker can simply observe the traffic between the victim's machine and the callback endpoint, using for example the technique used by the Firesheep add-on to Firefox, which has reportedly [17] shamed Facebook into using TLS. The attacker will obtain any user data sent with the HTTP request to the callback endpoint. By replaying the HTTP request, the attacker is likely to be logged in by the application and, as we shall see in section 3.5.3, the victim may be logged out.

On the other hand, if TLS is used at the callback endpoint, we believe that the double redirection mechanism achieves security, without using a provisional token, provided that:

1. The user is securely authenticated in step 1(b).
2. The third party knows the callback endpoint of the application, or the protocol includes an application authentication step (step 2) that takes place before the second redirection and the third party learns the callback endpoint securely during the authentication step.
3. The second redirection is implemented by an HTTP response protected by TLS followed by an HTTP request to the callback endpoint.
4. The callback endpoint satisfies one of the following conditions:
   (a) It uses the scheme https, i.e. it is protected by TLS; or
   (b) It uses the scheme http and its host component is localhost; or
   (c) It uses a custom scheme.
5. Local applications can be trusted, and custom schemes are handled by local applications.
6. If the callback endpoint is protected by TLS, the certificate it uses binds the key pair used in the TLS handshake to the domain name in the host portion of the callback endpoint, the user's browser verifies the certificate and its certificate chain using a root certificate store that only contains trustworthy CA certificates, the user's browser issues an invalid certificate warning to the user if verification of the certificate or certificate chain fails, and the user heeds such a warning.
7. When the application uses the access token, it sends it to a remote party (the third party or a fourth party) over a TLS connection with authentication of the destination, and
   (a) Either the access token is intended for a specific party and the remote party verifies that it is the intended party, or
   (b) The application authenticates itself to the remote party, and the access token allows the remote party to verify that the application is the one that access token was intended for.

Stating precisely what it means to achieve security, let along proving this conjecture, is difficult, if at all possible. We leave it for future work.

3. Double-Redirection Protocols

3.1 SAML Web Browser SSO Profile

In this protocol [7, section 4.1], the application is the *service provider* and the third party is the *identity provider*. In [7, figure 1] the first redirection is step 3 and the second redirection is step 5. The service provider delegates user authentication to the identity provider, then returns a resource to the browser if authentication is successful. Notice that the resource resides in the service provider, which is the application, not the third party.

The second redirection carries to the service provider a SAML assertion about the identity of the user signed by the identity provider. The assertion is a bearer assertion [7, section 3.3], i.e. a statement about the identity of whatever party is capable of producing it.

A bearer assertion can be used to authenticate an entity as the subject of the assertion to the party that relies on the assertion only if the following conditions are satisfied:

1. The assertion is sent by the asserting party to the subject over a channel that provides confidentiality and destination authentication.
2. The assertion is sent by the subject to the relying party over a channel that provides confidentiality and destination authentication.
3. The asserting party includes the identity of the intended relying party in the signed statement, and the actual relying party verifies that it is the intended relying party.

In the present case, the asserting party is the identity provider (the third party in our terminology), the subject is the user equipped with his or her browser, and the relying party is the service provider (the application in our terminology).

None of the three conditions are satisfied by the protocol. Conditions 1 and 2 would be satisfied if the protocol required the use of TLS for the two HTTP exchanges involved in the second redirection, but TLS is only recommended, not required. The *subject confirmation data* of the assertion includes a *recipient* attribute that can be used to specify the relying party, but it is only optional, and the processing rules do not require the service provider to verify the value of the attribute. Since the conditions are not satisfied, a malicious service provider can obtain an assertion and use it to impersonate the user and obtain a resource belonging to the user from another service provider.

TLS should also be required for the second request of the first redirection as a defense against phishing attacks, but TLS is only recommended, and phishing attacks are not discussed.

3.2 OpenID 2.0

In this protocol [18] the third party is the *OpenID provider* (*OP*) and the application is the *relying party* (*RP*). The main purpose of the protocol is to allow the RP to delegate user authentication to an OP, without prior knowledge of the OP by the RP. The user specifies the OP (by typing in an OP identifier or a user identifier that includes an OP identifier) and the RP discovers the user-interaction endpoint of the OP using one of three discovery protocols.

OpenID has two variants. In the first variant, the RP and the OP use Diffie-Hellman to establish a shared secret before the first redirection. After authenticating the user, the OP uses the shared secret as a symmetric key to sign an assertion about the user's identity. Then, after the second redirection, the RP authenticates the user by verifying the signature. In the second variant, the OP signs the assertion using a private symmetric key that it does not share with the RP. When the RP receives the assertion, it sends a message to the OP asking it to verify the signature. The second variant is clearly not stronger than the first and we will not discuss it further.

OpenID has numerous security flaws. Some of them are acknowledged in the security considerations section of the specification [18, section 15], some are not.

The discovery protocols are not secure, the OP's user authentication endpoint is not protected by TLS, and Diffie-Hellman is used without authentication. This makes it possible for an attacker to impersonate the OP vis-a-vis the RP, sign an assertion specifying the identity of a legitimate user, and present the assertion to the RP in the second redirection to impersonate the legitimate user. This is acknowledged in the security considerations section.

An attacker can also take advantage of the lack of TLS protection for the OP's user-authentication endpoint to impersonate the OP vis-a-vis the user. In particular, the attacker can mount a phishing attack and capture the user's credentials for the OP; phishing attacks are discussed in the security considerations section [18, section 15], and OpenID has been criticized for allowing them [9]. The attacker can also mount a man-in-the-middle attack using a rogue RP that proxies the connection to the OP through itself, and again capture the user's credentials; this attack is acknowledged in the security considerations section.

Like the SAML assertion used in the SAML Web SSO Profile, the assertion signed by the OP is a bearer assertion that authenticates any party that can produce it. Therefore the same three conditions enumerated in section 3.1 must be satisfied to guard against impersonation.

The third condition is satisfied, because the RP verifies the return_to URL (the URL of the callback endpoint); OpenID is better than the SAML Profile in this respect. The confidentiality conditions, on the other hand are not satisfied, since use of TLS is not required. The specification attempts to compensate for this by using a nonce. The rationale for the nonce is given in the specification [18, section 15.1.1]: an attacker that observes the assertion cannot reuse it because the nonce protects against replay attacks. However this fails in two different ways.

1. Use of the assertion by the attacker is a replay attack only if the legitimate user presents the assertion first to the RP. There is a race between the legitimate user's browser and the attacker, and the attacker may win the race if the attacker has a shorter Internet route to the RP.
2. The specification does not even recommend the use of TLS for the return_to URL (the callback endpoint). Therefore the attacker can mount a man-in-the-middle attack between the user's browser and the RP, intercept the assertion, and present it to the RP while preventing the legitimate user from doing the same. This is the man-in-the-middle attack described in section 2, profiled for OpenID.

There is also a security flaw in the way in which the OP identifies the RP to the user, which an attacker can exploit to mount an RP misidentification attack, tricking a legitimate user into giving the OP permission to provide the user's identity data to the wrong RP. This may be only a minor concern if the identity data is a single identifier. But OpenID is typically used in conjunction with the SREG extension [19], and then the identity data may include the user full name, birthdate, email address, etc.

OpenID identifies the RP to the user by showing a *realm*, which is essentially a domain name with a wild card. For example, if the host component of the return_to URL is pomcor.example.com, the realm may be http://*.example.com. Using this realm would help the user realize that the RP may have nothing to do with Pomcor. But the realm is specified by the RP, which, in a misidentification attack, is the attacker. If the attacker, who owns example.com, is impersonating Pomcor and wants to trick the user into allowing the OP to send identity data to pomcor.example.com, the attacker will simple use http://pomcor.example.com as the realm. A realm can have a wild card, but does not have to.

3.3 OAuth 1.0

This protocol has three specifications:

1. The specification of the original version of the protocol, OAuth 1.0 (the *original community specification*) [20];
2. A specification of a revised version of the protocol, OAuth 1.0a, that corrected a security flaw (the *revised community specification*) [21]; and
3. A rewrite and further revisions by the editor of the specification, published as an informational

RFC (the *informational specification*) [22].

Unfortunately the informational specification uses different terminology than the community specifications. Each terminology is better suited than the other for some particular purpose, so we will use both in this section. In the community specifications the application is called the *consumer* and the third party is called the *service provider*. (Notice the difference with OpenID, where the service provider is the application.) In the informational specification the application is called the *client* and the third party is called the *server*.

Whereas in OpenID 2.0 the RP discovers the OP, in OAuth 1.0 the client must have registered with the server before initiating the protocol. The protocol uses three different pairs of credentials, each consisting of a non-secret component and a secret component: *client credentials* obtained during registration, consisting of a *consumer key* and a *consumer key secret*; *temporary credentials* consisting of a *request token* and a *request token secret*, and *token credentials* consisting of an *access token* and an *access token secret*.

Before the first redirection, the client uses its client credentials to obtain temporary credentials, and both client and server create sessions. (Sessions are not explicitly mentioned in the specification, but they are implicit; the flaw addressed by OAuth 1.0a was a session fixation attack.) After the first redirection the server asks the user for permission to grant the client limited access to the user's account with the server, and records the grant in the session. After the second redirection, the client exchanges the temporary credentials for token credentials, then uses the token credentials to access the user's account to the extent that was authorized by the user.

The client signs requests to the server using symmetric keys constructed from the secret components of credentials. The consumer key secret is used to sign the request for the temporary credentials, a combination of the consumer key secret and the request token secret is used to sign the request for the token credentials, and a combination of the consumer key secret and the access token secret is used to access the user's account. However the protocol allows two alternatives to such signatures. Instead of sending a signature, the client can send the symmetric key that would have been used for signature, over a TLS connection. And instead of using a symmetric key, the client can sign with a private RSA key, in which case the shared secrets do not seem to be used in the protocol.

In the first redirection, the client sends the request token to the server and the server uses it to associate the client's request with the server's session. The original version of the protocol was vulnerable to a session fixation attack, where the attacker impersonated a legitimate user (the victim) of a legitimate client and a legitimate server [23]. The attacker initiated the protocol, causing the server to create a session, and obtaining the request token. The attacker used a rogue client to redirect the victim to the server, sending along the attacker's own request token. The server authenticated the victim and recorded the victim's grant of access in the attacker's session. This vulnerability was addressed in OAuth 1.0a by adding a *verifier* parameter to the protocol, sent by the server to the client with the second redirection, and needed to obtain the token credentials. In the attack scenario, the verifier is sent via the browser of the legitimate user rather than via the browser of the attacker. The verifier, however, is sent in the clear, because the callback endpoint is not protected by TLS (the specification does not require TLS, and the scheme of the callback URI is http in the examples), so the attacker can observe it. Therefore, although the verifier parameter complicates the attack, it does not prevent it.

Because the callback endpoint is not protected by TLS, all protocol variants are vulnerable to a man-in-the-middle attack where the attacker intercepts the redirected request to the client's callback endpoint, and then impersonates the legitimate user by presenting the request token and the verifier to the client. This is the man-in-the-middle attack of section 2, profiled for this protocol. The use of temporary credentials does not prevent this attack, and we believe that it does not improve security.

The security considerations section has a subsection [22, section 4.7] about phishing attacks, which says that users should be careful to verify the authenticity of the server. However users will not be able to follow that advice unless the user-authentication endpoint of the server is protected by TLS; and this is not required or recommended by the specification, which considers the issue to be out of scope [22, section 2.2].

Both client and server allocate storage, for the sessions they create, before the user has authenticated, and must keep the storage allocated while waiting for user input. This makes them vulnerable to a denial of service attack by exhaustion of available storage. The security considerations section has a subsection on denial of service by resource exhaustion [22, section 4.10] but does not point this out.

3.4 Web Resource Authorization Profiles (WRAP)

The WRAP specification [24] was merged into OAuth 2.0 and is now superceded by OAuth 2.0, but OAuth 2.0 is still a work in progress, and implementations of WRAP are still being used.

WRAP includes five different variants, called *profiles*. The *client account and password profile*, the *assertion profile*, and the *username and password profile* are not double-redirection profiles and we will not discuss them further. The *rich app profile* refers to applications running in the browser (which we are calling *rich applications* in this paper) and native applications; we touch on that profile in section 3.5.4 below. In this section we focus on the *Web app profile*.

The WRAP specification refers to the application as the *client*, to the third party as the *authorization server*, to the access endpoints as *protected resources*, and to the user-interaction endpoint of the third party as the *user authorization URL*. Notice that the term protected resources in this protocol refers to endpoints, not to data accessible through endpoints.

There is no preliminary step before the first redirection. The client initiates the protocol by redirecting the browser to the user authorization URL, which asks the user for permission to grant the client access to the protected resources, and authenticates the user. Client identification is not an issue because, as in OAuth 1.0, the client must have registered with the authorization server. The authorization server sends the client a verification code with the second redirection. The client exchanges the verification code for a short-lived access token and a long-lived refresh token that can be used to obtain a new access token when an old one expires. The access token provides access to the protected resources.

The protocol requires TLS protection only for HTTP exchanges that transmit the refresh token. TLS is required for the *access token URL* where the client exchanges the verification code for the access token and the refresh token. It is not required nor even recommended for the user authorization URL and the callback endpoint of the application. This means that:

- The protocol is vulnerable to phishing attacks by impersonation of the authorization server.
- The protocol is vulnerable to the passive and active attacks of section 2 above. An attacker can obtain the verification code by observing or intercepting the connection from the browser to the callback endpoint of the application in the second redirection. The attacker can then use the verification code to make the client access protected resources for the benefit of the attacker.

Both redirections are GET redirections. Since the second redirection is a GET redirection the verification code is included in the callback URI and may be leaked via referrer headers, browser history, and server logs.

After obtaining an access token the client sends it to a protected source without TLS protection and no authentication of the client or the protected resource. Nothing prevents an attacker from obtaining the access token and gaining unauthorized access.

The protocol allows the client to include a client state parameter in the first redirection, which the authorization server returns in the second redirection. The intended use of this parameter is as a reference to a session that the client creates before the first redirection. However, creating such a session requires allocating storage before user authentication, and keeping the storage allocated while waiting for the user to authenticate. This makes the client vulnerable to a denial of service attack by storage exhaustion.

3.5 OAuth 2.0

In this protocol [25] the third party is called the *authorization server*, the application is called the *client*, the user is called the *resource owner*, the third party's user-interaction endpoint is called the authorization endpoint, and the access endpoints are called *resource servers*. The resource servers provide access to protected resources. Notice the difference with WRAP: the protected resources are not endpoints.

The OAuth specification is still a work in progress, but implementations have already been developed and deployed by Facebook, Twitter, LinkedIn, Google, Yahoo, and others. Although OAuth is viewed as an authorization protocol, implementations of OAuth 2.0 are being used for social login, and hence for authentication.

3.5.1 Flows

As of version 12 [25] the protocol has four variants, called *flows*: the *authorization code flow* [25, figure 3], the *implicit grant flow* [25, figure 4], the *resource owner password credentials flow* [25, figure 5] and the *client credentials flow* [25, figure 6]. The latter two flows are not double-redirection protocols and we will not discuss them further.

The authorization code flow is essentially identical to the Web app profile of WRAP. The implicit grant flow is intended for rich applications running in the browser, with support from an ancillary Web server. Version 12 [25] says that the flow is also applicable to native applications, but it is not clear how; version 11 [13] suggested a different handling of native applications.

3.5.2 Authorization Endpoint

OAuth 2.0 improves on WRAP by recommending the use of TLS for the authorization endpoint. It does not require it, however, so it should still be considered vulnerable to phishing attacks.

(Note: the protocol requires TLS for the authorization endpoint “if the response returns an access token”. This seems to be an inaccuracy in the specification. The response should be a page that asks the user for permission to act on the client's request, and may ask the user to enter credentials. It is only after the user has granted permission that an access token may be returned. In any case, an access token is only returned by the second redirection in the implicit grant flow, discussed below. Therefore TLS protection for the authorization endpoint seems to never be required for the authorization code flow.)

3.5.3 Callback Endpoint

Like WRAP, OAuth 2.0 does not require or recommend TLS protection for the application's callback endpoint. Therefore OAuth 2.0 is vulnerable to the passive and active attacks described above in section 2. In the authorization code flow, an attacker can obtain the authorization code by observing or intercepting the HTTP request sent from the browser to the callback endpoint by the second redirection.

The attacker can then use the authorization code to impersonate the user in a social login and make the client access protected resources for the attacker's benefit.

OAuth 2.0 makes an implementation suggestion that makes the passive attack of section 2 more likely to succeed. It says that the authorization server may revoke an access token issued upon presentation of an authorization code when the authorization code is presented again. This suggests that a second presentation of the authorization code should be honored and should take precedence over an earlier presentation. Suppose the client uses the protocol to implement social login, and an attacker observes the authorization code in an unencrypted HTTP request sent from the legitimate user's browser to the callback endpoint. As the attacker tries to log in to the client by presenting the authorization code, he or she is in a race with the legitimate user's browser that the browser is more likely to win. But being second is to the attacker's advantage if client and server act as suggested: the second login will succeed and the legitimate user will be logged out.

3.5.4 Implicit Grant Flow

The client has a shared secret with the authorization server, established during registration. In the authorization code flow the client uses this secret to authenticate itself when it exchanges the authorization code for an access code and an optional refresh token. In the implicit grant flow, on the other hand, the client does not use the shared secret and does not authenticate itself. The authorization server sends the access token rather than the authorization code to the callback endpoint of the application. There is no exchange of the authorization code for an access token, and therefore no client authentication.

In WRAP there is a similar difference between the Web app profile and the rich app profile. In the rich app profile the client does not use its client secret. The verification code is used and exchanged for an access code and refresh token, but the exchange is not authenticated.

The OAuth 2.0 specification improves on the WRAP specification by providing an explanation for the difference between the treatment of rich applications and the treatment of traditional Web applications. It points out that a client application running on the user's machine cannot use its client secret because that would require storing the secret in the resource owner's machine, where it would be exposed to the resource owner.

To justify the seemingly less secure treatment of rich applications, the OAuth 2.0 specification says that client authentication is based on the browser's same-origin policy. But the same-origin policy does not authenticate the client. In the second redirection of the implicit grant flow, the access token is sent in the fragment portion of the callback URI. JavaScript code in the HTTP response from the callback URI reads the access token from the fragment and delivers it to the client application running in the browser. The same-origin policy ensures that only JavaScript code originating from the ancillary server will be able to read the token, but it does nothing to authenticate the ancillary server itself.

The specification points out that the fragment portion is not forwarded by the browser to the ancillary server. This is presumably the reason for including the access token in the fragment portion rather than the query portion of the callback URI: if it were included in the query portion it would be forwarded in the clear, given that the protocol does not recommend TLS protection for the request to the callback endpoint.

The observation that a rich application cannot store an application secret in the client is true, but that doesn't mean that it cannot authenticate. We propose an authentication method in section 4.3 below. The same can be said of native applications. A native application cannot store an application secret in the user's machine, but it can authenticate nevertheless. We show how that can be done in section 4.2

below.

Another security problem with the implicit grant flow is the fact that the access token (and the refresh token if used in addition to the access token) may leak through referrer headers, browser history and server logs. We propose a method that does not have this problem in section 4.5.

The passive attack of section 2 does not work against the implicit grant flow, since the access token stays in the browser. And the man-in-the-middle attack does not work if the ancillary server is not involved in setting an authentication cookie. However a more elaborate man-in-the-middle attack works: the attacker modifies the JavaScript code in the response from the callback endpoint so that, after reading the access token from the fragment, it uploads it to the ancillary server; then the attacker intercepts the upload and obtains the access token.

3.5.5 Access Tokens

The specification does not explain how access tokens are used, but it refers, as examples, to two drafts describing two kinds of access tokens: bearer tokens [26] and MAC tokens [27]. We discuss the latter first.

3.5.5.1 MAC Tokens

A MAC token is the same as an access token in OAuth 1.0. It has an associated secret, shared with the authorization server, which is used to sign an access request to a resource server. The only difference with OAuth 1.0 is that the signing key in OAuth 2.0 consists only of the token secret, whereas in OAuth 1.0 it is a combination of the token secret and a client secret.

The MAC token specification [27] recommends the user of TLS when using a token to access a resource server, but does not require it. Failure to use TLS allows an easy man-in-the-middle attack that gives the attacker access to protected resources.

The signature includes a nonce for protection against replay attacks. This makes a passive attack difficult, but not impossible. If an attacker observes a token sent over a connection not protected by TLS and tries to use it, the attacker is in a race with the client to present the access token to the resource server. The attacker may win the race if he or she has a shorter Internet route to the resource server.

3.5.5.2 Bearer Tokens

A bearer token is used as a bearer assertion. It is presented by the client to the server in an unsigned request. To achieve security, the three conditions on bearer assertions stated above in section 3.1 must be satisfied.

The first condition is satisfied, at least in the authorization code flow, because the client uses TLS and authenticates itself when exchanging the authorization code for the access token and optional refresh token.

The second condition is satisfied because the client presents the access token to a resource server over a TLS connection with authentication of the resource server.

However the third condition is not satisfied. The specification [26, section 3.2] says that "...it is important for the authorization server to include the identity of the intended recipients, namely a single resource server (or a list of resource servers)". However the identity of the intended recipient is verified only by the client as it presents the access token to a resource server. It is not verified by the resource server. Therefore a malicious resource server can use an access token to obtain resources from a different resource server.

Bearer tokens work if the resource servers trust each other. So it works in the most common use case where the authorization server and the resource server all belong to one site; i.e., in our terminology, if there are no fourth parties. If there are fourth parties, then bearer tokens work if each resource server verifies that the access token it receives is intended for the party to which the resource server belongs; this, however, this goes counter to the spirit of the OAuth 2.0 specification, which does not contemplate the possibility of a user having to obtain different access tokens for different resource servers, and having to interact repeatedly with the authorization server to obtain those different access tokens.

3.5.5.3 The Signature Issue

An important difference between OAuth 1.0 and WRAP was that WRAP did not require cryptographic signatures. When WRAP was merged into OAuth 2.0, signatures were removed from the OAuth specification [25, change log for version 6]. There was however a disagreement about this [10], with some people thinking signatures are too difficult to implement, but others thinking they are needed for stronger security. The competing types of tokens seem to reflect this disagreement.

In section 4.1 we show how it is possible to take advantage of the security provided by signatures without requiring developers to implement them.

3.5.6 Denial of Service by Storage Exhaustion

Like WRAP (see section 3.4), OAuth 2.0 has an optional client state parameter that the client sends to the authorization server with the first redirection, and the authorization server returns with the second redirection. As we saw for WRAP, the intended use of this parameter as a reference to a client session makes the client vulnerable to a denial of service attack by storage exhaustion.

3.5.7 Denial of Service Attacks on the Callback Endpoint

If the client refrains from using the client state parameter to avoid the denial of service attack by storage exhaustion of section 3.5.6, then, in the authorization code flow, the client must unconditionally forward the authorization code received at the callback endpoint to the authorization server. An attacker can take advantage of this to launch a different denial-of-service attack, by sending requests with bogus authorization codes to the callback endpoint.

It has been pointed out [28] that, if the callback endpoint is not protected by TLS, then the attacker triggers an https connection from the client to the authorization server (which requires a computationally expensive RSA decryption by the server) with a simple http connection from the attacker to the callback endpoint of the client, achieving cost magnification. However, we have seen in section 3.5.3 above that lack of TLS protection at the callback endpoint allows an attacker to impersonate a legitimate user, which is a graver threat than denial of service. Also, similar cost magnification can be achieved by sending ordinary TLS requests directly to the authorization server. (A TLS client that does not use a client certificate does only an RSA encryption, which is orders of magnitude less computationally expensive than an RSA decryption; and the client can even skip the encryption and send a bogus ciphertext to the server.)

Whether or not the callback endpoint is protected by TLS, however, the denial-of-service attack on the callback endpoint can be very effective against the client. As the client unconditionally forwards bogus authorization codes to the server, the server may lock out the client to protect itself, and may even revoke the client's registration. So the client should validate the authorization code as suggested in

[28], for its own sake. Unfortunately that requires signatures such as those that where removed from the core specification in version 6.

Signatures are not a problem, however, if the same party signs and verifies. In a double-redirection protocol such as our own PKAuth [1, section 9.5], the application (the client in OAuth terminology) can protect itself against attacks on the callback endpoint by verifying a signature that it has produced itself.

4. Solutions to Outstanding Security Issues

The more than 5000 messages in the OAuth 2.0 mailing list archives show that there are many unresolved security problems related to the design of double redirection protocols. In this section we provide solutions to some of those problems. Our PKAuth protocol for social login [1] is an example of how many of these solutions can fit together into one protocol.

4.1 How to Authenticate Applications without Ad-Hoc Signatures

As the application sends an access token to an access endpoint, security must be provided to prevent a range of attacks that could give an attacker unauthorized access to the resources or services made available by the token.

Two approaches to providing security have been proposed:

1. The application may authenticate itself by signing the request. In OAuth 1.0 [22], it signs the request using a symmetric key obtained by combining a secret associated with the token and a secret associated with the application, or a private key associated with the application. In the MAC token draft [27] it signs the request using a secret associated with the token; this is an indirect method of application authentication.
2. The token may be used as bearer assertion, which requires sending it through a confidential channel to an authenticated access endpoint which verifies that it is the intended relying party. The bearer token draft [26] uses this approach, as discussed above in section 3.5.5.2.

The first approach complicates the protocol because it requires an ad-hoc cryptographic signature whose details must be carefully specified and implemented because the signer and the verifier are different parties.

The second approach can work well if all the access endpoints belong to the same Web site, which should be true in most cases. But in use cases where service endpoints do not trust each other, the approach does not work well because it requires different access tokens for different endpoints.

We propose a third approach: the application may authenticate itself by using its TLS certificate as client certificate when it establishes a TLS connection to the access endpoint. This provides the benefits of a cryptographic signature, but the signature is constructed and verified within the TLS handshake, without complicating the specification and implementation of the protocol.

4.2 How to Authenticate a Native Application when It Requests Access

In section 4.1 we have observed that an application can authenticate using a TLS client certificate when connecting to an access endpoint. The application demonstrates knowledge of the corresponding private key during the TLS handshake. In the case of a native application running on the user's machine, however, the application cannot use its private key because that would expose it to the user, as observed in the OAuth 2.0 specification [25, section 4.2]

We propose that an instance of a native application running on the user's machine can use a key pair and certificate specific to the instance, the certificate being signed by the application and backed by a certificate chain consisting of an application certificate, followed by a certificate chain that backs the application certificate and ends with a generally trusted root certificate. The native application instance can then use its instance certificate as TLS client certificate, and use its private key in the TLS handshake without exposing any application secret to the user.

4.3 How to Authenticate a Rich Application when It Requests Access

A rich application running in the user's browser is another case where the application's TLS private key cannot be used because that would expose it to the user.

We propose in this case that the application instance running on the user's machine can proxy its request to an access endpoint through a Web server belonging to the application, something which is usually required anyway by the same-origin policy of the browser. As the application server proxies the request, it can use its TLS certificate for authentication.

4.4 How to Securely Identify an Application Without Registration

In section 3.2 we saw that OpenID uses a realm to identify the application, but it is vulnerable to an application misidentification attack where the realm is specified by the attacker.

OAuth does not have this problem because the application can be identified based on information established during the registration process. But compulsory registration is undesirable, for reasons discussed below in section 5.

In this section we propose three methods of identifying an application without registration, which are more secure than the method used by OpenID.

4.4.1 How to Securely Identify an Application using a Domain Name

Instead of showing the user a realm specified by the application, and thus possibly by an attacker, the third party (the OP in the case of OpenID) can identify the application to the user by displaying the portion of the host component of the callback endpoint that is a registered domain. For example, the third party would identify the application as example.com if the host component of the callback endpoint is pomcor.example.com, and as example.co.uk if the host component is pomcor.example.co.uk. Browsers and browser add-ons use a similar method to help users know what site they are visiting.

The third party should also warn the user about suspicious domain names, such as domain names where characters belong to a mix of Unicode scripts.

4.4.2 How to Identify an Application using a TLS certificate

The Web has a public key infrastructure, consisting of the TLS certificates of Web sites and applications, and the CA certificates installed in the root certificate stores of browsers. We have seen that the application must use TLS to protect its callback endpoint; it must therefore have a TLS certificate. The third party can use the information contained in the certificate to identify the application to the user. Whereas a browser lacks the real-estate to display that information, the third party has an entire page to display it as it asks the user for permission to respond to the application's request.

CAs only verify some of the information in a certificate. The third party should display all relevant information in the certificate, pointing out which certificate fields have been verified and which have not. The third party should also display the certificate chain, and refuse to satisfy the application's request if the certificate chain is not rooted on a CA certificate that the third party trusts.

The information in the certificate should be displayed in addition to the registered domain portion of the host portion of the callback URI as explained above.

The third party can obtain the application's TLS certificate in two different ways:

1. The application can establish a TLS connection to the third party before the first redirection, as in OAuth 1.0. The application can then use its TLS certificate as client certificate for the connection, and the third party can obtain the application's certificate and certificate chain during the TLS handshake.
2. The third party can establish a TLS connection to an application authentication endpoint after the first redirection and before asking the user for permission to respond to the application's request. In this case the application can use its TLS certificate as a server certificate, and the server again can obtain the certificate and certificate chain during the TLS handshake.

4.4.3 How to Identify an Application using a Holder-of-Key Assertion

A holder-of-key assertion, such as a SAML holder-of-key assertion [7, section 3.1], can bind the application's TLS certificate to additional information that may be useful in identifying the application to the user. The application can transmit the assertion to the third party in the HTTP exchange where the third party obtains the certificate as explained above in section 4.4.2. If the application connects to the third party before the first redirection and uses its TLS certificate as client certificate, the application can include the assertion in the HTTP request. If the third party connects to the application after the first redirection and the application uses its TLS certificate as server certificate, the assertion can be included in the HTTP response.

4.5 A Better Way of Passing a Token to a JavaScript Application

In the implicit grant flow of Oauth 2.0 [25, section 4.2], the authorization server redirects the client to an ancillary server that supports a JavaScript application, passing the access token in the fragment portion of the redirect URL. This has two drawbacks:

1. The access token can be leaked through referrer headers, browser history or server logs.
2. The authorization server must behave differently for traditional Web applications and for JavaScript applications.

These drawbacks can be avoided by using a POST redirection, sending the access token in the body of the HTTP POST request, and having the ancillary server set a cookie whose value is the access token, as it sends the HTTP response. The cookie can be set by a general-purpose middleware module, such as an Apache module, that sets cookies as specified by a collection of name-value pairs in the body of a POST request. The ancillary server can include a script in the response that notifies the main application script running in the browser of the fact that the cookie has been set.

4.6 Another Method of Passing a Token to a Native Application

Several methods of passing tokens to a native application running on the user's machine have been proposed [13,14,15] We propose the following simple method as an additional method suitable for desktop applications that can rely on an ancillary Web server. The token is passed in the body of an

HTTP POST request that targets the ancillary server, as part of a POST redirection. A general purpose middleware module in the server, such as an Apache module, copies the body of the HTTP POST request to the body of the HTTP response, and sets the content-type header of the response to a media-type handled by the application.

5. Conclusion

Double redirection is a mechanism used by an application to delegate user authentication and authorization to a third party in protocols such as OpenID and OAuth, which are becoming more and more important now that they are being used for social login. We have analyzed the security of the double-redirection mechanism in general terms, examined the security posture of particular double-redirection protocols, and proposed solutions to outstanding security problems related to double redirection.

Providing strong security is made easier by requiring the application to register with the third party, as OAuth does. Compulsory registration, however, is dangerous. Social login has compelling advantages for users and application providers, and may well become the de facto user authentication method of the Web. When that happens, if social login requires registration, every application will have to register with the dominant social site, currently Facebook, just to be able to authenticate its users. The dominant site will then have the power to disable any application by revoking its registration. This will be bad for the Web and for all parties involved, including the dominant social site which will no doubt face regulation by many different governments.

We believe that it is possible to design authentication, authorization and social login protocols that provide strong security without compulsory registration. To show how this could be done we have sketched out a social login protocol, PKAuth [1], using some of the ideas that we have discussed in this paper.

References

[1] F. Corella and K. P. Lewison. PKAuth: A Social Login Protocol for Unregistered Applications. Revised 2/7/2011. At http://www.pomcor.com/whitepapers/PKAuth.pdf.
[2] OpenID Wikipedia article. At http://en.wikipedia.org/wiki/OpenID.
[3] OAuth Wikipedia article. At http://en.wikipedia.org/wiki/OAuth.
[4] OpenID and OAuth Hybrid Extension.
At http://wiki.openid.net/w/page/12995194/OpenID-and-OAuth-Hybrid-Extension.
[5] OpenID Connect. At http://openidconnect.com/.
[6] OpenID Artifact Binding. At https://bitbucket.org/openid/ab/wiki/Home.
[7] S. Cantor et al. Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0. OASIS Standard, March 2005. At http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf.
[8] We believe that the term social login was coined by Janrain.
See http://www.janrain.com/products/engage/social-login.
[9] Stefan Brands. The Problem(s) with OpenID. August 2007.
At http://www.untrusted.ca/cache/openid.html.
[10] Eran Hammer-Lahav. OAuth 2.0 (without Signatures) Is Bad for the Web. September 2010.
At http://hueniverse.com/2010/09/oauth-2-0-without-signatures-is-bad-for-the-web/.
[11] Mailing list of the OAuth Working Group of the IETF.
At https://www.ietf.org/mailman/listinfo/oauth.
[12] Web page of the OAuth Working Group. At http://datatracker.ietf.org/wg/oauth/charter/.
[13] The OAuth 2.0 Protocol Framework, version 11. December 1, 2010.

At http://tools.ietf.org/html/draft-ietf-oauth-v2-11.
[14] Marius Scurtescu. OAuth 2 for Native Apps. June 2010.
At http://wiki.oauth.net/w/page/27249271/OAuth-2-for-Native-Apps.
[15] Marius Scurtescu. Native Client Extension. December 29, 2010.
At http://www.ietf.org/mail-archive/web/oauth/current/msg04892.html.
[16] P. Moceri and T. Ruths. Cafe Cracks: Attacks on Unsecured Wireless Networks.
At http://www1.cse.wustl.edu/~jain/cse571-07/cafecrack.htm.
[17] PCWorld. Facebook Offers Protection Against Wireless Firesheep Attack. January 26, 2011. At http://www.pcworld.com/businesscenter/article/217851/facebook_offers_protection_against_wireless_firesheep_attack.html.
[18] OpenID Authentication 2.0 – Final. December 5, 2007.
At http://openid.net/specs/openid-authentication-2_0.html.
[19] J. Hoyt et al. OpenID Simple Registration Extension 1.0. June 2006.
At http://openid.net/specs/openid-simple-registration-extension-1_0.html.
[20] OAuth Core 1.0. December 4, 2007. At http://oauth.net/core/1.0/.
[21] OAuth Core 1.0 Revision A. June 24, 2009. At http://oauth.net/core/1.0a/.
[22] E. Hammer-Lahav, Ed. The OAuth 1.0 Protocol. IETF RFC 5849. April 2010.
At http://tools.ietf.org/html/rfc5849.
[23] Eran Hammer-Lahav. Explaining the OAuth Session Fixation Attack. April 23, 2009.
At http://hueniverse.com/2009/04/explaining-the-oauth-session-fixation-attack/.
[24] D. Hardt, Ed. OAuth Web Resource Authorization Profiles. January 15, 2010.
At http://tools.ietf.org/html/draft-hardt-oauth-01.
[25] E. Hammer-Lahav, Ed. The OAuth 2.0 Authorization Protocol, version 12. January 21, 2011.
At http://tools.ietf.org/html/draft-ietf-oauth-v2-12.
[26] M. Jones et al. The OAuth 2.0 Protocol: Bearer Tokens, version 2. January 28, 2011.
At http://tools.ietf.org/html/draft-ietf-oauth-v2-bearer-02.
[27] E. Hammer-Lahav. HTTP Authentication: MAC Authentication, version 2. January 22, 2011.
At http://tools.ietf.org/id/draft-hammer-oauth-v2-mac-token-02.txt.
[28] Peifung Eric Lam. Message to the OAuth mailing list: validate authorization code in draft 12. January 21, 2011. At http://www.ietf.org/mail-archive/web/oauth/current/msg05125.html.