Analysis of privacy vulnerabilities in single sign-on mechanisms for multimedia websites

Manuel Urueña · Alfonso Muñoz · David Larrabeiti

Abstract This paper studies the privacy risks for the users of two popular single sign-on platforms for web-based content access: OpenID and Facebook Connect. In particular we describe in detail a privacy vulnerability of the OpenID Authentication Protocol that leads to the exposure of the OpenID user identifier to third parties. We illustrate how OpenID agents leak the (potentially unique) OpenID identifiers of their users to third parties, like advertisement and traffic analysis corporations. This vulnerability is a real and widespread privacy risk for OpenID users. This paper also analyzes the privacy of Facebook Connect –the proprietary single sign-on platform that is gaining a lot of popularity recently– and, we conclude that it is not affected by the same vulnerability but other important privacy issues remain. Finally, this paper studies the solution space of these problems and defines a number of possible countermeasures. In the case of the OpenID vulnerability, we propose three solutions to this problem: one for the long term to avoid the root cause of the vulnerability, and another two short-term mitigations.

Keywords OpenID · Facebook connect · Google connect · Single sign-on (SSO) · Privacy · Security

1 Introduction

The main feature of a single sign-on (SSO) mechanism is to provide a single user identifier to log in all the websites that support it. Furthermore, the user authentication process is centralized in the identity provider, thus usually it is only necessary to authenticate once per

M. Urueña (✉) · A. Muñoz · D. Larrabeiti
Universidad Carlos III de Madrid, Avda. Universidad 30, 28911 Leganés (Madrid), Spain
e-mail: muruenya@it.uc3m.es

A. Muñoz
e-mail: ammunoz@it.uc3m.es

D. Larrabeiti
e-mail: dlarra@it.uc3m.es

web browsing session, hence the *single* sign-on name. This is especially important for websites serving multimedia content, since users are expecting a hassle-free (i.e. no registration or login) content consumption experience, irrespective of the place where the multimedia object is hosted, while site owners have to know their clients in order to provide them with value-added services and increase customer loyalty.

Besides its convenience, SSO mechanisms also provide many benefits from a security point of view, because with SSO the user credentials are only stored in a single, trusted place. This clearly contrasts with the current situation, where each user has multiple usernames and passwords in order to log into many different websites, with the obvious trouble for users. Even worse, because of those inconveniences, most users employ the same login and password for all the websites they visit, and never change it. This means that, if any of the websites where the user was registered is plainly malicious or its security is breached, the attacker could employ the obtained user credentials to gain access to other similar websites. Therefore, with shared usernames and passwords, the security of the whole user's website chain is the one of the weakest link, the website with the lowest security.

On the other hand, if the user credentials and personal information are stored just in one identity provider, which is fully trusted by the user, like in most SSO architectures, it is a lot easier to protect them. Furthermore, and also because of centralization, it would be possible for the SSO identity provider to implement more advanced authentication mechanisms such as one-time password (OTP) tokens or smart cards.

Two of the most popular web SSO mechanisms nowadays are OpenID [7] and Facebook Connect [2] which, although they basically provide the same SSO features, have radically different design perspectives. Facebook Connect leverages the huge popularity of Facebook social site, allowing Facebook users to log into an increasingly number of web sites employing their Facebook accounts. Therefore, Facebook acts as the centralized SSO identity provider of all web sites supporting Facebook Connect. On the contrary, OpenID is not tied to a specific identity provider but anyone can setup an interoperable OpenID provider, even the end user herself. Therefore OpenID users are able to choose which identity provider they trust.

Even if the identity providers were fully trusted by the users, if SSO identity mechanisms become commonplace and each user has a single and unique identifier for all her public information published on the web, like posts, photos, videos, etc. the associated privacy risks are evident. This is the centre of a very interesting debate in the SSO and the social networks communities, and many alternatives, such as temporal, anonymous IDs, or using multiple identities per user, are being proposed as potential solutions to this problem.

This paper does not study the problem of unique identifiers for public multimedia contents, but the potential ability of third parties (e.g. advertisement and audience metering agencies) to track all the web visited by a given SSO user, because of the way these SSO technologies work and exchange information on top of the HTTP protocol.

The structure of this paper is as follows: Section 2 provides a summary of the OpenID Authentication Protocol, in order to fully understand the discovered privacy vulnerability, which is described in section 2.1. Then, different countermeasures to this problem are presented in section 2.2. Section 3 explains the Facebook Connect authentication mechanism and its privacy is evaluated in section 3.1, including whether it also suffers from the reported OpenID vulnerability. It also proposes different countermeasures to the privacy issues of Facebook Connect. Finally the conclusions of this work are presented in section 4.

2 OpenID

OpenID [7] was one of the first single sign-on mechanisms for the Web that achieved a considerable popularity. It is backed by big Internet players like Google, Yahoo, Flicker, WordPress or AOL. For instance the Google Connect/Login with Google service is also based in OpenID. There are hundreds of different sites that provide OpenID identifiers, and tens of thousands of web sites support OpenID. The OpenID Authentication Protocol has been developed by the OpenID Foundation, which is being sponsored by big corporations such as Microsoft, Google, IBM, PayPal, VeriSign or Yahoo.

First of all, it is necessary to define the terminology employed by OpenID. The OpenID architecture is composed by three types of agents:

- *User Agent*: The browser of the User that wants to access a website (Relying Party) with an OpenID Identifier, obtained from the OpenID Provider of her choice.
- *OpenID Provider (OP)*: An authentication server that provides one or more unique identifiers for the User, and validates the User's credentials on behalf of the Relying Parties.
- *Relying Party (RP)*: The website where the User tries to log in employing her OpenID Identifier. This identifier will be validated by the OpenID Provider of the User.

In OpenID, Users are uniquely identified by means of a particular `http:` URL, which is called *OpenID Identifier*[1] (e.g. `http://alice.myid.org`). Initially, this is the only information provided by the User to the Relying Party when trying to log in. There are two benefits of using an URL as an identifier: First of all, URLs are hierarchical in nature by means of its domain part, thus each OpenID Provider is able to define its own set of globally unique identifiers without colliding with other OpenID Providers. Also, by providing a URL from the OpenID Provider DNS domain, Relying Parties could easily found the IP address of the OpenID authentication server and begin with the discovery and authentication phases of the OpenID Authentication Protocol.

The OpenID Authentication Protocol is the cornerstone of the OpenID single sign-on (SSO) mechanism. Nowadays there are two versions of this protocol in use, version 1.1 [9] and version 2.0 [8], which is the latest one. The OpenID Authentication Protocol allows Relying Parties and OpenID Providers to exchange information on top of the Hypertext Transfer Protocol (HTTP)[4] both, directly or transparently through the User Agent. A very important feature of the OpenID Authentication Protocol is that it does not require any kind of special support from User Agents. Any HTTP 1.1 standard browser can be employed as an OpenID User Agent, even if it does not support JavaScript or Cookies have been disabled by the user.

Following, and with the aid of Fig. 1, we will explain in detail the different phases of the OpenID Authentication Protocol and the format of its messages, as well as its interaction with the underlying HTTP protocol:

1) The OpenID authentication process starts when a User, employing a User Agent, visits a website (Relying Party) and tries to log in. To that end, she provides her OpenID URL identifier (e.g. `http://alice.myid.org`) to the Relying Party, usually in a HTML form with a HTTP POST operation (the recommended name for the HTML form field is "`openid_identifier`").

[1] Instead of `http:` URLs, OpenID users can employ a XRI URL pointing to an XML document as an alternative to identify themselves. For simplicity, in this paper we will only consider plain OpenID `http:` URL identifiers. Nevertheless all the results of this work are still valid, irrespectively of which kind of OpenID identifier is employed.

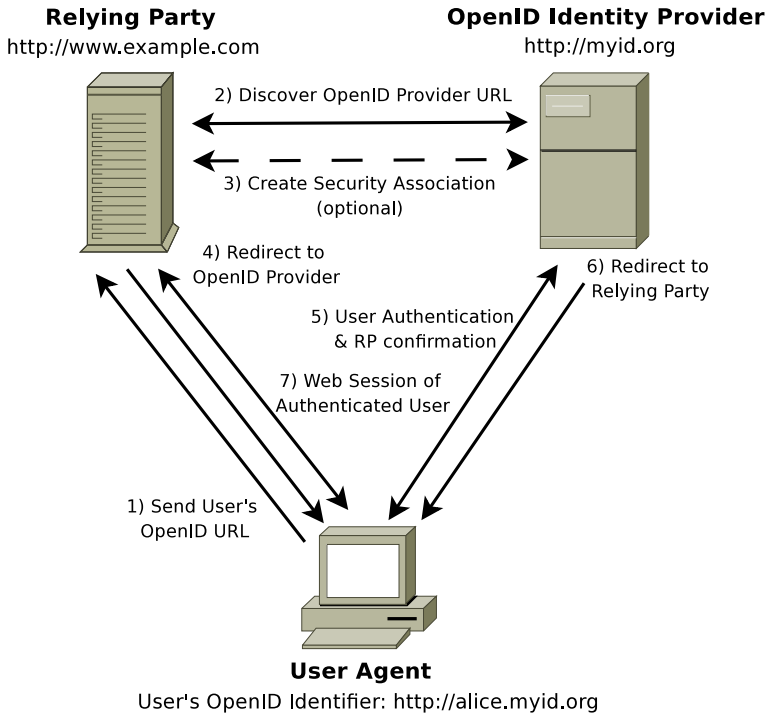

Fig. 1 The different phases of the OpenID Authentication Protocol [8]

2) The Relying Party employs the URL supplied by the User in order to discover the endpoint of the OpenID Provider to be employed for the authentication of the User. Although there are different OpenID discovery mechanisms, this is usually performed just by requesting the web page pointed by the User's OpenID Identifier URL and looking for a particular `<link>` tag within the `<head>` part of this HTML page. This `<link>` tag must include the `rel="openid2.provider"` or `rel="openid.server"` attribute values (depending on the OpenID protocol version), plus a href attribute that defines the OpenID Provider endpoint URL where the user authentication takes place (e.g. `href="http://myid.org/openid-auth/"`).
3) After discovering the OpenID Provider endpoint, the Relying Party may (optionally) create a security association with the OpenID Provider, in order to negotiate a shared secret using a Diffie-Hellman [1] secure key exchange. This key is later employed to sign and verify the messages exchanged between them. This security association is recommended because otherwise it is necessary to address direct requests between them in order to verify each authentication request/response message.
4) Then, the Relying Party redirects the User Agent to the discovered OpenID Provider endpoint by means of a 302 HTTP Temporary Redirect message, where the HTTP Location header specifies the target OpenID Provider endpoint. This URL must also contain all the parameters of the OpenID Authentication Request (`openid.mode=checkid_setup`). The most important ones are the User's OpenID Identifier (`openid.identity`), the identifier of the optional security association established between the RP and the OP (`openid.assoc_handle`), and the URL of the Relying Party where the result of the Authentication operation

should be sent back (`openid.return_to`). Additionally, the Relying Party can request, by means of protocol extensions, some additional information about the User such as her full name, gender, or e-mail. Figure 2 shows an (quite simplified[2]) example of an OpenID Authentication Request message, sent by the User Agent to the OP because of the RP redirect reply.

5) When the User Agent receives the HTTP Redirect message from the Relying Party, it starts a new connection with her OpenID Provider in order to be authenticated. How the end user is authenticated by the OpenID provider is out of the scope of the OpenID specification [8], and thus varies among the different OpenID Providers that have been tested. This enables innovation since each OpenID Provider may deploy its own (hopefully secure) authentication mechanisms, like SSL and certificates. In most cases the User needs to log in, employing a username and a password, and then she is requested to confirm the authentication request from the Relying Party, including which of the requested personal information should be returned. For User's convenience, all these choices for each particular Relying Party can be remembered by the OpenID Provider, thus they will not be asked again the next time. In fact, if this is the case and the OpenID Provider also employs a HTTP Cookie in order to authenticate the user subsequently, it is quite possible that the User would not even see the OpenID Provider web page because the whole process occurs automatically and transparently, which of course it is the main selling point of Single Sign-On (SSO).
6) When the OpenID Provider authenticates the User and validates the Authentication Request from the RP, it redirects the User Agent again to the endpoint specified by the Relying Party (i.e. `openid.return_to`). The target URL must contain all the parameters of the Authentication Response (`openid.mode=id_res`), including the OpenID User Identity (`openid.identity`), a copy of the base RP endpoint (`openid.return_to`), a nonce (`openid.response_nonce`), the optional security association handle (`openid.assoc_handle`), and a cryptographic signature (`openid.sig`), along with the list of parameters that have been signed by it (`openid.signed`). Figure 3 shows a (quite simplified) example of an OpenID Authentication Request HTTP message, sent by the User Agent to the RP because of the OP redirect message.
7) Finally, after the reply from User's OpenID Provider is validated by the Relying Party, the User will log in successfully and the web session will continue as usual.

2.1 A privacy vulnerability in OpenID

From the previous explanation it can be seen that the OpenID Authentication Protocol heavily relies on URL parameter encoding in order to exchange information between the Relying Party and the OpenID Provider. The parameters of the Authentication Reply are signed in order to secure the authentication process and avoid tampering. However these parameters are not encrypted, which means that all these parameters, and in particular the (probably unique[3]) User OpenID Identifier (`openid.identity`), can be seen by anyone

[2] OpenID Authentication parameters must be percent-encoded before being added as URL parameters. This encoding has been ignored in all the examples of the paper in order to ease it understanding by the reader.
[3] Google Connect service includes a privacy enhancement that replaces the User's OpenID identifier with a random-looking identifier that is fixed for a given RP but different among RPs. However all the tested Google Connect RPs also request the user's e-mail address and/or full name, which is then included in the return URL. Thus in practical terms user's data also appears in the return URL of the Google Connect service.

```
GET /openid-auth?openid.mode=checkid_setup
  &openid.identity=http://alice.myid.org
  &openid.return_to=http://www.example.org/openid-login
  &openid.assoc_handle=xxxxxxxxxxxxxxxx
  HTTP/1.1
Host: myid.org
```

Fig. 2 An example of an OpenID Authentication Request HTTP message sent by the User Agent to its OpenID Provider (Step 5 of the OpenID Authentication process)

that has access to the full URLs of the Authentication Request or the Authentication Reply messages. Here resides the privacy vulnerability of the OpenID Authentication Protocol that has been found by this work.

The question then is how these URLs, which in principle should be only processed by the Relying Party or the OpenID Provider, can be leaked to a third party. The answer does not lie in OpenID itself but in the underlying Hypertext Transfer Protocol (HTTP) [4] that is employed to access websites, and thus also to transport the URL parameter-encoded OpenID authentication messages. In particular when accessing to a resource (e.g. image, link) the HTTP message header should include a Referer[4] field that specifies the full URL of the web page where this resource was accessed from. The objective of this Referer field is to help webmasters to locate invalid links, and can be very useful for web traffic analysis in order to know how users actually browse the website or what web pages are linking to a particular page or resource.

However this means that the OpenID Authentication URLs, including the User OpenID Identity parameter, can appear inside the HTTP Referer field when accessing any resource linked by the web page that is generated when processing such URLs. If the OpenID Authentication URLs were always processed by a CGI or Servlet that does not return any web content but just redirects the user to other web page, or if the target web page does only include local resources, this should not be a problem. Sadly, and because nowadays most websites are funded by advertisements, it is quite common that all web pages of a site contain some kind of banner, text advertisement, or traffic analysis script from some third party.

Therefore, because of this combination of OpenID URL parameter encoding and HTTP Referer field, there is a potential privacy vulnerability in all OpenID Providers and Relying Parties where the web page generated from the Authentication Request/Reply URL contains any reference to a third party resource, as depicted[5] in Fig. 4.

In order to study the real impact of this potential vulnerability we have analysed multiple websites that support OpenID Identifiers (Relying Parties), as well as different OpenID Providers. We have found that this vulnerability affects all the studied Relying Parties and even one of OpenID Providers, which does employ a traffic analysis script in the login web page. Even worse, in many cases, the third parties the User OpenID Identity was leaked to were the same ones: a well-known Internet advertisement company and a free traffic analysis website (that even belongs to the same corporation). In fact, since the traffic analysis service was based in JavaScript, the OpenID Authentication URL appeared twice in the HTTP

[4] This field is misspelled in English. It should be spelled "Referrer" but as it appears as Referer in the HTTP specification, implementations use it as it is.

[5] Disclaimer: All user names, DNS domains and URLs that appear in this paper are fictitious, and they do not belong to any of the RPs and OPs studied by this work.

```
GET /openid-login?openid.mode=id_res
  &openid.identity=http://alice.myid.org
  &openid.return_to=http://www.example.org/openid-login
  &openid.response_nonce=2010-03-22T12:00
  &openid.assoc_handle=xxxxxxxxxxxxxxxx
  &openid.signed=mode,identity,return_to
  &openid.sig=yyyyyyyyyyyyyyyy
  HTTP/1.1
Host: www.example.com
```

Fig. 3 An example of an OpenID Authentication Reply HTTP message sent by the User Agent to the Relying Party (Step 7 of the OpenID Authentication process)

requests to this third party: Once as expected in the Referer field, but also encoded as a parameter of the requested traffic analysis URL itself.

Therefore we consider that this privacy vulnerability is very real and even widespread, thus it could be exploited by unscrupulous Advertisement, Audience Metering or Traffic Analysis companies to track all the websites accessed by a particular OpenID User, for instance in order to create user behaviour profiles for targeted advertising.

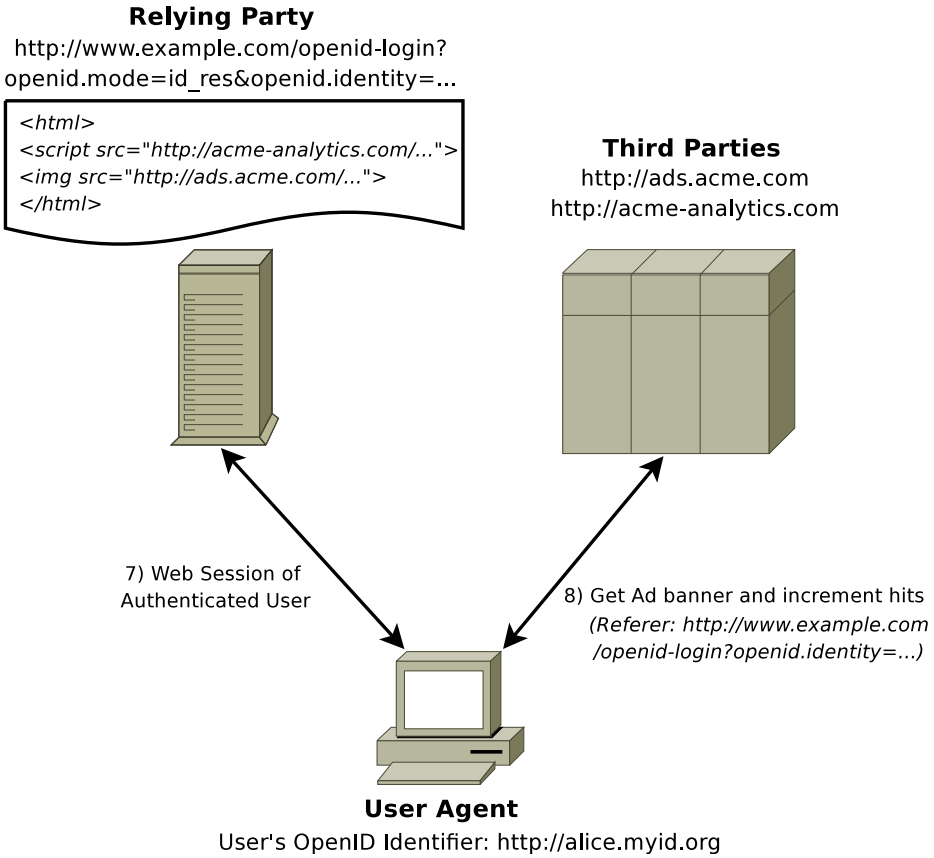

Fig. 4 User OpenID Identity being leaked to third parties

2.2 Possible countermeasures

Unfortunately, this vulnerability is not a bug or some kind of implementation issue. All the studied User Agents, Relying Parties and OpenID Providers do implement the OpenID Authentication and HTTP Protocols correctly. In fact we have tested major browsers (Internet Explorer, Firefox, Opera and Konqueror) with the default settings; over different Operating Systems (Windows XP and Linux) and we always have been able to reproduce this vulnerability. Therefore, we consider that this vulnerability is a design problem of the OpenID Authentication Protocol because of using URL parameters to exchange private information.

Moreover, as stated in the Security Considerations section of the HTTP 1.1 specification [4] (Subsection 15.1.3 "Encoding Sensitive Information in URI's"): "*Authors of services which use the HTTP protocol SHOULD NOT use GET based forms for the submission of sensitive data, because this will cause this data to be encoded in the Request-URI. Many existing servers, proxies, and user agents will log the request URI in some place where it might be visible to third parties. Servers can use POST-based form submission instead.*"

Therefore a solution to this vulnerability must be found by:

1) Redesigning the OpenID Authentication Protocol.
2) Disabling the HTTP Referer field in User Agents.
3) Recommending Relying Parties and OpenID Providers to do not include links to third parties in the web pages processing OpenID Authentication URLs.

Clearly, the first alternative is the optimal solution of the three, since the encoding of information as URL parameters done by the OpenID Authentication Protocol is the root cause of this privacy vulnerability. There are two alternative solutions for a new OpenID Authentication Protocol:

- To avoid exchanging information between the RP and OP by means of URL parameters, but using instead POST forms where data is carried in the body of the HTTP message.[6]
- To encrypt the whole OpenID Authentication URLs by leveraging the existing signing key or by generating a new encryption key during the security association phase. This way the User OpenID Identifier parameter will be meaningless, even if leaked to a third party.

Although clearly desirable, these solutions can only be applied in the long term. First, it is necessary to define a new OpenID version that deprecates URL parameters, and then wait for all OpenID Providers, and especially all existing Relying Parties, to migrate to the new protocol version. Therefore a short term solution is needed meanwhile.

The third alternative is probably the most complex to implement because requires many of the existing websites that employ OpenID to limit where third party resources can appear, to change its implementation and even its internal structure. Although this may be a plausible solution for the OpenID Providers currently affected by this vulnerability, it is a questionable solution for all the websites that already support OpenID. Nevertheless, for privacy-concerned sites, the suggested solution is to process the OpenID `openid.return_to` URL with an interstitial page that does not show any content (including from third parties) but just redirects the user to the personal page of the user or the home page of the site, only after processing the

[6] This HTTP POST mechanism has been already defined in the version 2.0 [8] of the OpenID Authentication Protocol, but none of the analysed RPs or OPs employ it. Probably this is because, in order to send a HTML form automatically, the User Agent must support and enable JavaScript.

complete OpenID `openid.return_to` URL and thus removing all OpenID-related data from the local redirection URL. The required OpenID user's data may be stored as a cookie or, better, with some kind of user session's data at the server.

Therefore the best short-term alternative seems to be the second one. Many privacy advocates have suggested in the past to get rid of the HTTP Referer field since it could expose the behaviour of the user to the web server. However the Referer field was finally included in the HTTP specification because it is a legitimate and useful tool for webmasters. Nevertheless some browsers are able to disable this field, by changing its default configuration (e.g. by setting `network.http.sendRefererHeader=0` in Firefox) or by enabling some kind of "Private Browsing" mode where visited web pages are not cached nor stored in the history list. However, as in the previous case, this is not a global solution as it requires all OpenID Users to change the default behaviour of their browsers.

Fortunately, there is an alternative, standard mechanism to clear the HTTP Referer field. The HTTP 1.1 specification [4] states that: "*Clients SHOULD NOT include a Referer header field in a (non-secure) HTTP request if the referring page was transferred with a secure protocol*". This statement was added to avoid any information -including URLs- that is exchanged with a secure web site, to be leaked to a different web site. The good news is that most advertisement and audience metering scripts are accessed employing HTTP to avoid overloading their servers. Therefore, if the User Agent employs HTTPS to connect to both, the Relaying Party and the OpenID Provider, when it is then redirected to the Relying Party using HTTPS, a standard-compliant User Agent should not include the OpenID Authentication Reply URL in the Referer field of any HTTP resource requested from that page. We have tested this behaviour in Firefox, and we have confirmed that when the OpenID Provider and the Relaying Party employ HTTPS, the User OpenID Identifier is never leaked in the Referer field to third parties using HTTP-retrieved scripts.

It should be noted that enabling HTTPS and TLS/SSL security protocols does not require changes to the web site but it can be enabled just by configuring the underlying web server. However it could add some overhead to the OpenID Providers and Relaying Parties servers, and also increase the total delay of the OpenID authentication process. However we consider that adding this additional security layer is a good idea by itself [10], in order to protect the Users credentials and personal information stored by the OpenID Providers, even without considering the discovered privacy vulnerability.

Therefore we recommend all OpenID Providers and Relaying Parties to switch from HTTP to HTTPS to authenticate their users or to implement an interstitial page to process the OpenID parameters, in order to mitigate this privacy vulnerability in the short term. Nevertheless we still consider that the usage of URL-encoded parameters by the current OpenID Authentication Protocol is flawed and should be redesigned or deprecated in the long term. Notice that the proposed solutions may solve the Referer leakage but do not guarantee that the OpenID Authentication URLs could not appear in other places like proxies, caches, history lists, or server logs, and thus being exposed to attackers or other third parties.

3 Facebook connect

Presently, Facebook is one of the most popular social network platforms. It has become so widespread that it can be regarded as a gigantic data base with information of over 800 million users.[7] This huge amount of users allows Facebook to develop and exploit a set of

[7] http://www.facebook.com/press/info.php?statistics

additional technologies with applications in very important areas such as access control and the exchange of personal information with third parties. One of these technologies is the Facebook-based single sign-on system known as Facebook Connect [2].

This single sign-on system is also one of the most used in the Internet at present, due to three main reasons:

1. Millions of users already have a Facebook account and many of them log in on a daily basis. An external web server can take advantage of this to authenticate their users through the social network without the need of registration and even of keeping any kind of user account information. Furthermore, the user gets freed from the tedious process of registering and memorising yet another username and credentials.
2. It is very simple for third-party applications to interoperate with Facebook. This is due to the Facebook Platform and an API that provides access to the services offered by this social network. It is a common practice for both web-based and stand-alone applications to implement the access by means of Java Servlets, PHP pages, C, C++, Java, Javascript or any other widespread programming language.
3. Facebook also provides "social characteristics" in those external pages (Facebook Social Plug-ins), like the now ubiquitous "I like" button, to post comments in the user's Facebook walls, or to easily share information. This is especially interesting for multimedia providers, since it enables website users to easily share multimedia content with their friends, by means of the hugely popular Facebook social website. Thus, improving its popularity and gaining additional visitors.

These factors have led to a current situation a where more than "*7 million apps and websites are integrated with Facebook*". Therefore the Facebook Connect platform provides a single sign-on service that allows Facebook users to authenticate to pre-registered applications by means of their social network accounts. At the same time, and subject to certain constraints, those applications can access user information stored in Facebook. Given its growing popularity in the Internet, we have decided to study Facebook Connect from the same perspective as we did with OpenID that is, by addressing aspects related to the potential leakage of private information.

Despite its popularity, Facebook Connect is a quite recent proposal. Therefore, there are few works in the literature that study the security or privacy aspects of its communication protocol. To the authors' knowledge, [6] is the most relevant work on this direction, although it only provides a brief description of the client-side authentication mode of the Facebook Connect service. Probably, this situation is due to the lack of authoritative information about Facebook Connect, since there is no formal specification of the protocol, other than the statement that Facebook Connect makes use of an extension of OAuth 2.0 [5], but just information for application developers [2]. Therefore, we have to resort to monitoring the exchange of HTTP requests between the user and the website being accessed and/or Facebook servers, in order to comprehend how this protocol really works and infer eventual privacy issues. Interestingly, the URLs we have found in these traces differ from the ones in Facebook examples, which seem to suggest that the Facebook Connect platform is still evolving.

Figure 5 shows the most significant steps of the process of login of a Facebook user on an external website. For this to happen, the website must register beforehand an application (the one providing the SSO process) in Facebook, which is assigned an API key and a unique application identifier required to interact with Facebook services. All the authentication process consists of a set of HTTP request-replies, parameters and cookies. The connection to

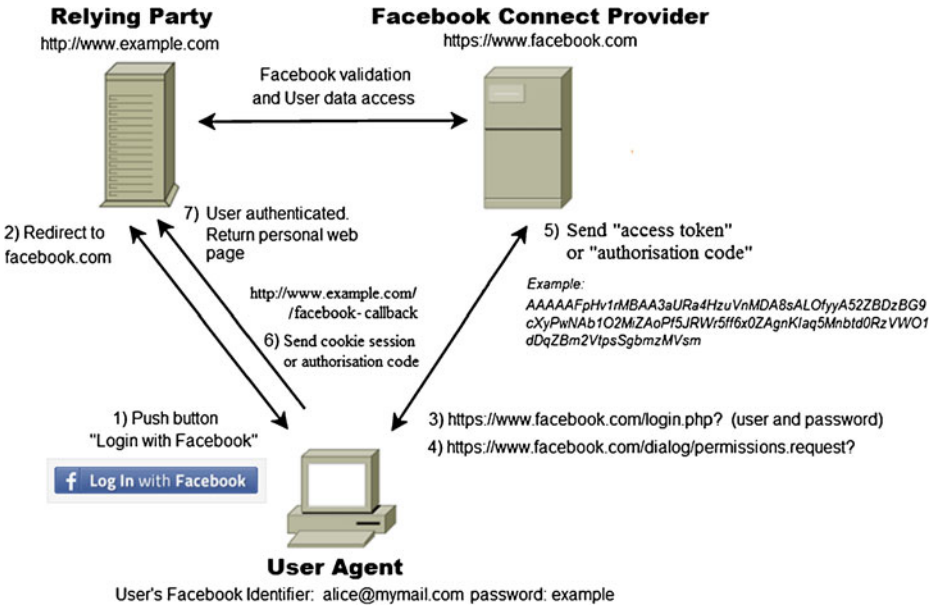

Fig. 5 The different phases of the Facebook Connect Authentication Protocol

the Identity Provider (i.e. Facebook) uses a secure channel (SSL), whereas the connection with the third-party website may not be secured.

Figure 5 depicts the two login modes supported by Facebook Connect. On the *client-side* mode, all calls to the Facebook API are performed by the client itself (by means of JavaScript code executed in the web browser) and user data is conveyed to the website by means of cookies. On the *server-side* mode, the website is the one communicating with the Facebook servers directly. In both cases an OAuth *access token* is required in order to call the Facebook API in name of the user. The difference is that on the *client-side* mode the user handles the access token to the website directly, whereas on the *server-side* mode the website first obtains an *authorization code* that later allows it to obtain the user's access token. In fact, the best way to know which mode is being employed is by means of the `response_type` parameter: client-side requests include a `response_type=token,signed_request` parameter, while server-side requests have a `response_type=code` value.

The remaining of this section explains how the two authentication modes of Facebook Connect work by using the example depicted in Fig. 5. Only the most relevant fields and parameters of the HTTP traffic exchanged with the website and Facebook servers are shown.

a) Client-side authentication.

1) The user connects to a web page that displays a "Login with Facebook" button.
2) When the user presses the button, the web browser connects to Facebook and a series of 302 HTTP redirections occur (Fig. 6):

   This series of requests first authenticate the application on the visited website into Facebook. If the application is already registered but the user is not yet authenticated (otherwise the cookie session would be present and the login would be directly OK) a pop-up window invites the user to input her login and password at Facebook (Fig. 7).

```
GET /dialog/oauth?app_key=API_KEY&app_id=WEBSITE_APP_ID
  &redirect_uri=http://www.example.com/facebook-callback
  &response_type=token,signed_request
  &scope=email,publish_actions
  HTTP/1.1
Host: www.facebook.com
…
GET /dialog/permissions.request?app_id=WEBSITE_APP_ID
  &next=http://www.example.com/facebook-callback
  &response_type=token,signed_request
  &perms=email,publish_actions
  HTTP/1.1
Host: www.facebook.com
…
GET /login.php?api_key=API_KEY
  &next=https://www.facebook.com/dialog/permissions.request?
    app_id=WEBSITE_APP_ID
    &redirect_uri= http://www.example.com/facebook-callback
    &response_type=token,signed_request
    &perms=email,publish_actions
  HTTP/1.1
Host: www.facebook.com
…
```

Fig. 6 HTTP requests following the pressing of "Login with Facebook" button

3) Once the user types in the login and password, and presses the "Log In" button, a new HTTP interaction takes place in order to obtain a token that will enable the web browser to build its authentication information for the web server where she is trying to authenticate (steps 3, 4, 5 y 6). The relevant HTTP message is shown in Fig. 8.

   In Fig. 8 the browser sends Facebook the authentication information (login, password). Next the application requests certain permissions to access personal information of the user or to perform some action on her behalf (see Fig. 9). The request in Fig. 9 is responded with an access token (valid only during certain time), and a signed request such as:

access_token=*AAAAAFpHv1rMBAA3aURa4HzuVnMDA8sALOfyyA52ZBDzBG9cXyPwN Ab1O2MiZAoPf5JRWr5ff6x0ZAgnKIaq5Mnbtd0RzVWO1dDqZBm2VtpsSgbmzMVsm*
&**expires_in**=6140 (seconds)
&**signed_request**=sJduS2StxFWvQIQg8CMz1sOoLmrmWbo384F7faH_QM8.eyJhbGdvcml0aG0iOiJITUFDLVNIQTI1NiIsImNvZGUiOiJBUUFpdGR0Q3hRQk0yRWplTTY3WWUzcm9NUzB4bVQ5ZGttR3M4R3FQRG5UZzJHUUduczNBaWFaclJ0ZF9TSDdscVZ6VDFhMzlfdlcyU3JQYTJyTk93WGR3dFhZOXp3QS1qM2d2ZDdITFhDZmZ2U290cy1lTUM5MmIyXzdzUURxS1lPTmxkS1BDYjM2UEhydWxKa1g1X2tVeFg0aXMzMTBwU2hOOHZFOFJ6NFBNd3ljREpUY3FFM3lRTTE3UWRxYjRfZ0UiLCJpc3N1ZWRfYXQiOjEzMjE5NzE0NjAsInVzZXJfaWQiOiIxMDcwMDAyNDY1In0

This response is important, as we shall see later, to complete the client-side authentication for the website. It is important to emphasize that the only authentication information obtained by the web browser is the `access_token` and the `signed_request` obtained by the response to the request in Fig. 9. This information is converted into a session cookie, which is the code to be used in

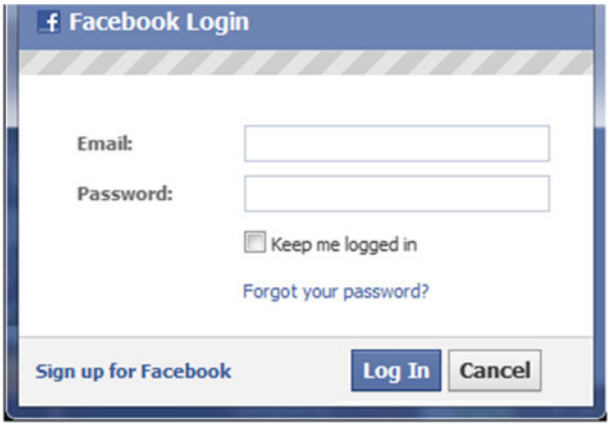

Fig. 7 Facebook Login Pop up

practice. For that purpose, the web agent makes use of the JavaScript SDK API defined by Facebook. In particular this API allows saving the authentication data into a cookie called `fbsr_APP_ID` that will be actually used as a session cookie.

Once the web browser knows the authentication information, it just needs to trigger the login mechanisms required by the visited web page. To this end, it issues a request (Fig. 10) to the web page dealing with Facebook Connect by means of a cookie.

The value of the cookie is exactly the same as the one in the last response; in particular is the value contained in the *signed_request* field. Further analysis of this value shows more information, since it is possible to decode a part of it. Skipping the first 256 bits, and from the dot ('.') character, the information is simply encoded in Base64. From the sample cookie above we could see the following JavaScript Object Notation (JSON) blob:

```
{"algorithm":"HMAC-SHA256",
"code":"AQAitdtCxQBM2EjeM67Ye3roMS0xmT9dkmGs8GqPDnTg2GQGns3AiaZrRt
d_SH7lqVzT1a39_vW2SrPa2rNOwXdwtXY9zwA-j3gvd7HLXCffvSots-
eMC92b2_7sQDqKYONldKPCb36PHrulJkX5_kUxX4is310pShN8vE8Rz4PMwycDJTcq
E3yQM17Qdqb4_gE", "issued_at":1321971460, "user_id":USER_ID}
```

where the information stored in code, is the authentication code required to access.

7) Finally the authentication information is validated. The cookie belongs to the holder and it is signed by Facebook. Once authenticated, the user is redirected to a personalised web page according to its identity.

```
POST /login.php?
  next=http://www.facebook.com/dialog/permissions.request?
    app_id=WEBSITE_APP_ID
    &redirect_uri=http://www.example.com/facebook-callback
    &response_type=token,signed_request
    &perms=email,publish_actions
  HTTP/1.1
Host: www.facebook.com

email=FACEBOOK_USER_EMAIL
pass=FACEBOOK_USER_PASSWORD
```

Fig. 8 Login request upon pressing the log in button

```
GET /dialog/permissions.request?app_id=WEBSITE_APP_ID
  &redirect_uri=http://www.example.com/facebook-callback
  &response_type=token,signd_request
  &perms=email,publish_actions
  HTTP/1.1
Host: www.facebook.com
```

Fig. 9 Access token request following login

b) Server-side authentication

Although both authentication processes are quite similar, there is a main difference with the previous client-side mode, because in this case the authentication information is not sent to the website inside a cookie, but the authentication code is directly appended to the website callback URL. Therefore, instead of requesting an access token plus a singed request, server-side Facebook Connect requests include a `response_type=code` parameter.

In order to illustrate how this authentication code is generated, Figs. 11, 12 and 13, show the steps 3), 4) and 5) of Fig. 5 in the server-side mode.

Finally, and once the request shown in Fig. 13 is processed by the website, the Facebook user is fully authenticated (step 7) of Fig. 5) and the website is able to fetch the user information from Facebook, for instance to create a personalized web page for the user.

After the detailed review of the Facebook Connect single sign-on platform, in the next section we address the question of potential information leakage to third parties, in the same way we did with OpenID.

3.1 Privacy, security and countermeasures

There exist intrinsic privacy issues in this type of third-party provided single sign-on mechanisms and Facebook Connect is not an exception. The most straightforward one is that the identity provider, Facebook in this case, may keep track of all the visited websites. The impact of this may be reduced by using several SSO systems (which was one of the key design decisions of OpenID), but, in practice, the current trend in Internet is toward using one or just very few SSO providers (Facebook, Twitter or Google), due to the convenience for both users and websites (a larger community of users will be able to authenticate).

However, when talking of privacy it is also important to pay attention to the flow of information from the identity provider to trusted third parties. For example, a web page may not have the same privacy policies as Facebook, and supply, perhaps unknowingly (as in the case of OpenID), private information to third parties, enabling for instance the disclosure of

```
POST /facebook-callback HTTP/1.1
Host: www.example.com
Cookie: fbsr_APP_ID=sJduS2StxFWvQIQg8CMz1sOoLmrmWbo384F7faH_Q
M8.eyJhbGdvcml0aG0iOiJITUFDLVNIQTI1NiIsImNvZGUiOiJBUUFpdGR0Q3hRQk0
yRWplTTY3WWUzcm9NUzB4bVQ5ZGttR3M4R3FQRG5UZzJHUUduczNBaWFaclJ0ZF9TS
DdscVZ6VDFhMzlfdlcyU3JQYTJyTk93WGR3dFhZOXp3QS1qM2d2ZDdITFhDZmZ2U29
0cy11lTUM5MmIyXzdzUURxS1lPTmxkS1BDYjM2UEhydWxKa1g1X2tVeFg0aXMzMTBwU
2hOOHZFOFJ6NFBNd3ljREpUY3FFM3lRTTE3UWRxYjRfZ0UiLCJpc3N1ZWRfYXQiOjE
zMjE5NzE0NjAsInVzZXJfaWQiOiIxMDcwMDAyNDY1In0
…
```

Fig. 10 Session authentication request sent to the client-side Facebook Connect authentication callback page

```
POST /login.php?
  next=http://www.facebook.com/dialog/permissions.request?
    app_id=WEBSITE_APP_ID
    &redirect_uri=http://www.example.com/facebook-callback
    &response_type=code
    &perms=email,publish_actions
  HTTP/1.1
Host: www.facebook.com

email=FACEBOOK_USER_EMAIL
pass=FACEBOOK_USER_PASSWORD
```

Fig. 11 Login request upon pressing the log in button

user credentials to unauthorized applications or user information (in most cases not required to interact with the site) [3].

Unlike OpenID, Facebook Connect features mechanisms that prevent the vulnerability identified in OpenID, including, among others, the fact that the connection to the identity provider makes use of HTTPS and anonymous authentication codes. But let us analyse in detail the two Facebook Connect authentication modes, with respect to the leakage of the user identifier to third parties.

a) Client-side authentication.

   The only risk to privacy would appear in the (unsecure) communication between the web browser and the website being accessed. However, since the authentication credentials are sent inside a cookie instead of as a URL parameter, such information should not be unwillingly leaked to third parties by standard means, unless some form of attack is performed, such as sniffing the communications link of the user (e.g. in an unprotected WiFi hotspot). In that case, the Facebook user identifier would be disclosed to the attacker, since it is carried inside the signed request in an encoded but utterly unencrypted form, and the attacker could also employ the whole cookie (until it expires) to impersonate the user in the website.

   Obviously this attack could be easily thwarted by employing also HTTPS in the communication between the user and the website, as proposed for OpenID.

b) Server-side authentication.

   In this mode, Facebook sends the authentication code to the website as a URL-encoded parameter. Therefore, it would be possible that this information could be leaked to third parties (as in the OpenID case), sniffed by an attacker (if the website does not employ HTTPS), or just registered at the web server log.

   However, although the authentication code could be also employed to impersonate the user, there is a major difference with respect to the OpenID case: the authentication code seems to be just a random bit string that changes in each transaction or after some time and thus cannot be easily traced back to the user by a third party. Even so,

```
GET /dialog/permissions.request?app_id=WEBSITE_APP_ID
  &redirect_uri=http://www.example.com/facebook-callback
  &response_type=code
  &perms=email,publish_actions
  HTTP/1.1
Host: www.facebook.com
```

Fig. 12 Code request following login

```
GET /facebook-callback?code=AQAitdtCxQBM2EjeM67Ye3roMS0xmT9dkmGs8G
qPDnTg2GQGns3AiaZrRtd_SH7lqVzT1a39_vW2SrPa2rNOwXdwtXY9zwA-j3gvd7HL
XCffvSots-eMC92b2_7sQDqKYONldKPCb36PHrulJkX5_kUxX4is310pShN8vE8Rz4
PMwycDJTcqE3yQM17Qdqb4_gE
Host: www.example.com
```

Fig. 13 Authentication code sent to the server-side Facebook Connect callback page

Facebook explicitly recommends employing an interstitial web page to process the authentication code, without any script or frame from third parties.

4 Conclusions

OpenID and Facebook Connect are increasingly popular Single Sign-On (SSO) mechanisms that enable a user to have a single identity across the whole web. From a security point of view, SSO platforms can become a powerful tool to enhance the security of public websites, since user credentials and personal information are only stored at the Identity Provider, which is chosen and trusted by the end user. Therefore user data is much easier to protect because Identity Providers are able to implement advanced authentication mechanisms, such as one-time password tokens, SSL certificates, or smart cards.

However, among other privacy problems derived from using a unique user identifier, the OpenID Authentication Protocol is subject to a privacy vulnerability, where a third party like a web advertisement or audience metering agency is able to obtain the OpenID identity of a registered user, for instance with the purpose of tracking all the OpenID web sites visited by each individual user. This vulnerability is caused by the use of URL parameters in order to exchange information between the OpenID Providers and the Relying Parties. Then the (potentially unique) OpenID identifier of a user can be leaked to a third party by means of the HTTP Referer header. This is not a bug or an implementation problem, but a design one. As a matter of fact, after studying real Relying Parties and OpenID Providers, we have found that most of them (unless protected by HTTPS) are affected by this widespread vulnerability.

In order to mitigate this real privacy risk, we analyse the possible solution space and propose a number of countermeasures that could be applied. From all the proposed solutions, we consider that the OpenID Authentication Protocol should be redesigned in the long term in order to avoid exchanging URL-encoded parameters. Meanwhile, we recommend that the OpenID Foundation should declare the use of HTTPS as mandatory for all OpenID Providers, and should recommend existing Relaying Parties to enable HTTPS when processing the OpenID URL. Alternatively, Relaying Parties may develop an interstitial return page to only process the OpenID parameters and then redirect the User to the appropriate webpage.

Facebook Connect, on the other hand, does not suffer from this vulnerability in its current form since it employs HTTPS by default, and the authentication information is either temporal and anonymous (i.e. the authentication code in the server-side mode) or exchanged by means of HTTP cookies (in the client-based mode), and thus it does not appear in the visited URLs. However the centralization and huge popularity of Facebook (almost everyone has a Facebook account nowadays), is both its main selling point, but also its main drawback from the privacy point of view. In other words, a private enterprise such as Facebook is able to know, not only personal information of ours and our friends, but also all the Facebook Connect-enabled web sites we visit each day, in real time, and what is more

relevant: the security of the access to all our content servers is directly determined by the security of Facebook authentication system.

Therefore, the challenge for the design of a SSO system that fully preserves navigation privacy, even from the identity manager itself if this was feasible, and where the user can decide what entity manages its identity is still open.

Acknowledgements The work presented in this paper has been funded by the INDECT project (Ref 218086) of the 7th EU Framework Programme.

References

1. Escola R. Diffie-Hellman Key Agreement Method. IETF RFC 2631. June 1999.
2. Facebook Connect. http://developers.facebook.com/docs/authentication. Accessed 28 November 2011
3. Felt A, Evans D. Privacy protection for social networking platforms. In proceedings of the Workshop on Web 2.0 Security and Privacy. 2008.
4. Fielding R, Gettys J, Mogul J, Frystyk H, Masinter L, Leach P, Berners-Lee T. Hypertext Transfer Protocol – HTTP/1.1. June 1999.
5. Hammer-Lahav E, Recordon D, Hardt D. The OAuth 2.0 Authorization Protocol <draft-ietf-oauth-v2-22>. Internet-Draft, September 2011.
6. Miculan M, Urban C. Formal analysis of Facebook Connect single sign-on authentication protocol. 37th Conference on Current Trends in Theory and Practice of Computer Science, Slovakia, January 22-28, 2011.
7. OpenID Foundation website. http://openid.net. Accessed 28 November 2011.
8. OpenID Foundation. OpenID Authentication 2.0 - Final. December 2007.
9. Recordon D, Fitzpatrick B. OpenID Authentication 1.1. May 2006.
10. Sovis P, Kohlar F, Schwenk J. Security Analysis of OpenID. In proceedings of the Information Security Solutions Europe (ISSE'10) Conference. 2010.

Manuel Urueña received his M.Sc. degree in Computer Science from Universidad Politécnica de Madrid (Spain) in 2001 and his Ph.D. degree in Telecommunications from Universidad Carlos III de Madrid (Spain) in 2005. At present, he is an assistant professor in Telematics engineering at Universidad Carlos III de Madrid. His research activities range from P2P systems, through load balancing and service discovery protocols, to Optical networks. He has been involved in several international and national research projects related with these topics, including the EU IST GCAP and the EU SEC INDECT projects.

Alfonso Muñoz received his M.Sc. degree and Ph.D in telecommunications engineering from Technical University of Madrid (Spain) in 2006 and 2010, respectively. At present, he is a researcher in the ADSCOM group at the Carlos III University of Madrid (UC3M). His topics of interest are information security, computational linguistic and computer networks. He has written several articles about information security in forums and national and international conferences. Currently his researches are about cryptography and steganography.

David Larrabeiti is a full professor of switching and networking architectures at Universidad Carlos III de Madrid (UC3M). He got his M.Sc. and Ph.D. in telecommunications engineering from Universidad Politécnica de Madrid in 1991 and 1996, respectively. From 1998 to 2006 he was an associate professor at UC3M and led a number of international research projects. His research interests include the design of the future Internet infrastructure, ultra-broadband multimedia transport, and homeland security applications. He is the coordinator of EU INDECT project at UC3M.