# Formal analysis of Facebook Connect Single Sign-On authentication protocol

Marino Miculan Caterina Urban

Dept. of Mathematics and Computer Science, University of Udine, Italy.
marino.miculan@uniud.it, caterina.urban@gmail.com

**Abstract.** We present a formal analysis of the authentication protocol of *Facebook Connect*, the Single Sign-On service offered by the Facebook Platform which allows Facebook users to login to affiliated sites.
Formal specification and verification have been carried out using the specification language HLPSL and AVISPA, a state-of-the-art verification tool for security protocols. AVISPA has revealed two security flaws, one of which (previously unheard of, up to our knowledge) allows an intruder to impersonate a user at a service provider affiliated with Facebook. To address this problem, we propose a modification of the protocol, by adding a message authentication mechanism; this protocol has been verified with AVISPA to be safe from the masquerade attack. Finally, we sketch a JavaScript implementation of the modified protocol.

## 1 Introduction

Single Sign-On (SSO) protocols allow to establish a federated environment, where users can login once and access to services offered by different systems. This approach addresses the issue of having multiple user-names and passwords. The need for a federated environment is even stronger due to the diffusion of integrated web services: a website can aggregate contents and services from other sites, each of which may require a specific authentication (e.g., a page embedding a video from YouTube, a calendar from Google and a slideshow from Flickr).

Federated authentication mechanisms need one or more *identity providers*, that is, sites providing users identities registration and user authentication. Recently, *social networks* are being proposed as possible identity providers. In fact, users are keen to register to these sites, and to update regularly their profile; in this way social networks already gather lots of personal information about users, such as habits, tastes, friend networks, etc.—all data very valuable for third parties. Moreover, the friend network of a user can be seen as an implicit "web of trust" for that user's identity.

Among others, Facebook is emerging as the most used social network by worldwide monthly active users [5]. Remarkably, the development of third-party applications interacting with Facebook is quite simple, thanks to the *Facebook Platform*, a REST-like API to access Facebook services. A Facebook method call is made over the Internet by sending HTTP GET or POST requests to the Facebook API REST server. Third-party web applications (i.e., websites

affiliated with Facebook) can be implemented as Java servlets or PHP pages, but also stand-alone desktop applications, interacting with Facebook and third-party sites, can be developed (in C, C++, Java, JavaScript...).

Clearly, this raises the issue of user authentication before granting access to web services. To this end, the Facebook Platform has been extended with *Facebook Connect*[1], a Single Sign-On service that enables Facebook users to login to applications (e.g., affiliated websites) using their Facebook accounts, and at the same time these applications can access user's information on Facebook.

In this paper, we present a formal analysis of the authentication protocol of Facebook Connect for web applications. Formal specification and verification are carried out using the specification language HLPSL and AVISPA [1], a state-of-the-art verification tool for security protocols.

This work is useful for many reasons. Obviously, mechanized formal methods are very helpful to identify flaws in protocols, and to suggest corrections to fix these flaws. Moreover, protocols are seldom described in a precise, formal way; therefore, a HLPSL specification can be used as an accurate, authoritative definition of the protocol, where all important details must be spelled out. Indeed, many security protocols have been formally analyzed, using similar techniques and tools, see e.g. [3,4], and [2] for the analysis of Google Single Sign-On protocol.

The first problem we have to face is that Facebook Connect is a *service*, and not a protocol specification like SAML SSO [6] or OpenID [7]. In fact, a detailed protocol description has not been officially provided, because a programmer is supposed to use the API provided by Facebook Connect, and not to implement the authentication protocol from scratch.

To overcome this problem, in Section 2 we analyze the actual HTTP message exchange between a client, a server and Facebook, during authentication. From our analysis, in Section 3 we distill the first abstract formalization of the Facebook Connect protocol in Alice-Bob notation; the corresponding HLPSL specification is then obtained (almost) directly.

In Section 4, we inspect this formalization using AVISPA, which has identified two security issues. First, a legitimate request can be observed and replayed by an intruder; however, this issue is common to many HTTP-based services, since HTTP is stateless. But also a more serious *masquerade attack* is possible: an intruder can capture the session credential during a legitimate request, and use them to impersonate the client to access any resource of the website. This attack was previously unheard of, up to our knowledge.

In Section 5 we discuss how to avoid the subtlest of these flaws, that is the masquerade attack: we propose a correction to the protocol, by adding a message authentication mechanism based on a Diffie-Hellman key exchange. The fixed protocol has been formally verified using AVISPA, which has not found the masquerade attack. Moreover, we sketch also a JavaScript implementation, which can be actually used in websites.

Concluding remarks and directions for future work are in Section 6.

[1] `http://developers.facebook.com/news.php?blog=1\&story=174`

## 2 The Facebook Connect Authentication Protocol

In this section we describe the Facebook Connect authentication protocol. Actually, there is no official description of this protocol; hence, in order to understand it we have analyzed all incoming and outgoing HTTP traffic between the browser, the Facebook login server and an application server during the authentication process. In particular, we have considered *The Run Around*, a sample site created by Facebook to demonstrate the key features of Facebook Connect.

In order to allow users to login with their Facebook profile, an application must be *registered* with Facebook beforehand. At registration, Facebook assigns a unique *API key* (required for calling Facebook API methods), and an *API secret key* (to be kept secret between Facebook and the application).

The authentication flow consists of six HTTP requests-responses pairs, one of which uses a secure (i.e. TLS) channel. The data involved in each HTTP request-response transaction are request and response headers, sent and received cookies, query string parameters, POST data and response body. Cookies play a particularly important role, since they carry the information about the login status of the user. The message sequence chart is shown in Figure 1. The protocol begins with the user requesting the home page (GET `http://www.somethingtoputhere.com/therunaround/`); the server answers with an HTML page that includes a placeholder for the Facebook Connect login button (Figure 2). The page also contains a reference to the Facebook JavaScript Client Library, which provides access to the features of Facebook Platform.

After receiving this response, the browser initializes the Connect service by calling `FB.init`, and then pings Facebook for login status by executing the method `FB.Connect.get_status`. This returns one of three possible states:

**Not logged in:** the user is not logged in to Facebook;
**Not authorized:** the user is logged in to Facebook but has not yet connected with the website;
**Connected:** the user is logged in and has already connected with the website.

To check the login status, the `FB.Connect.get_status` method requests the page `http://www.facebook.com/extern/login_status.php` (second GET in Figure 1): the query string contains the Facebook Platform API key of the website and the cookies carry user's current session informations (if any). The response from Facebook does not contain any session information, since we consider the case when the user is not already logged in.

At this point, the user executes the `FB.Connect.requireSession` method by clicking on the Connect button. The browser requests the page `http://www.facebook.com/login.php` (third GET in Figure 1); the query string contains the parameter `return_session` set to 1, and the API key. The response opens a popup window, asking the user for permission and possibly user's Facebook credentials (email and password). At this step, the browser receives from Facebook the `lsd` cookie, which is a nonce identifying the authentication flow.

Then, user's credentials are sent as data of an HTTPS POST request to `https://login.facebook.com/login.php`, together with the API key, the cross-domain communications channel, and the `lsd` cookie.

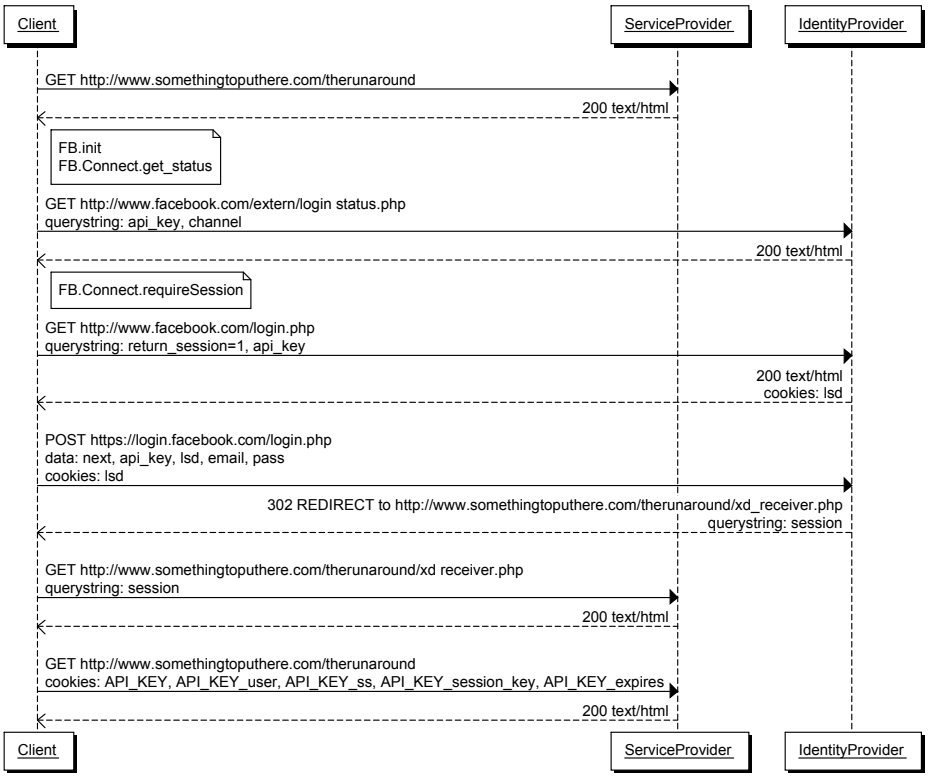


**Fig. 1.** Message sequence chart of the authentication flow

The response from Facebook contains user's new login status, which has to be stored in cookies for the application web server. There is a technical issue here: due to security restrictions imposed by the HTTP protocol, the response from `www.facebook.com` cannot modify cookies associated to another web server, as `www.somethingtoputhere.com`. To overcome this issue, the Client Library adopts a *Cross-Domain Communication* technique: the result from Facebook redirects the browser to a page of the application web server, called "cross-domain communication channel" (fourth GET in Figure 1, to `http://www.somethingtoputhere.com/therunaround/xd_receiver.php`), carrying user's login status on the query string. This page contains a JavaScript code which is executed by the browser: it decodes the session data contained in the query string and stores them in the cookies for `www.somethingtoputhere.com`. Let us describe briefly the meaning of these cookies, whose names begin with the API key value (denoted by *APIKEY*), with possibly a suffix.

*APIKEY*`_user:` currently logged in user's ID;
*APIKEY*`_session_key:` a value identifying the current session, required to make API calls;
*APIKEY*`_expires:` session's expire time. If it is 0, session does not expire;

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:fb="http://www.facebook.com/2008/fbml">
<body>
 <fb:login-button></fb:login-button>
 <script
   src="http://static.ak.connect.facebook.com/js/api_lib/v0.4/Feature
Loader.js.php"
   type="text/javascript"></script>
 <script type="text/javascript">
   FB.init("YOUR_API_KEY_HERE", "xd_receiver.htm");
 </script>
</body>
</html>
```

**Fig. 2.** Basic Facebook Connect implementation

*APIKEY*`_ss`**:** a session-only secret key, which allows to make API calls from client-side applications;

*APIKEY***:** the signature generated by taking the hash of the string obtained by concatenating the other cookies and the API secret key.

Finally the browser reloads the page `http://www.somethingtoputhere.com/therunaround/`, this time providing the server with the cookies containing the signed session information. To make sure the data came from Facebook, the server performs the same hash encoding and checks the signature to make sure it matches. In this case, it proceeds to fulfil the request, showing the user the Facebook Connect login process is completed.

## 3 Protocol formalization

In this section we give a formal specification of the Facebook Connect protocol[2], first in Alice-Bob notation, and then in the specification language HLPSL. In particular, in the latter we formally specify the security properties to be verified.

### 3.1 Alice-Bob formalization

From HTTP traffic analysis described in Section 2, we can define a protocol formalization in Alice-Bob notation by abstracting from implementation-level, but still corresponding one-to-one to the steps in Figure 1. The protocol is shown in Figure 3; principals $C$, $SP$, and $IDP$ stand for the Client (i.e., the user or better the browser), the Service Provider (i.e., the website the user wants to authenticate for) and the Identity Provider (i.e., Facebook), respectively.

We briefly comment about some formalization choices. First, the secure channel (e.g. the SSL session) between $C$ and $IDP$ is modeled by symmetric encryption using the key *CIDPKey* (steps 7 and 8). This key is supposed to be generated

[2] Notice that we work on our reconstruction of the authentication protocol (Section 2), and not from official published specifications.

1. $C \rightarrow SP : ResourceReq$
2. $SP \rightarrow C : IDP, APIKey$
3. $C \rightarrow IDP : StatusReq$
4. $IDP \rightarrow C : NotLoggedIn$
5. $C \rightarrow IDP : SessionReq, APIKey$
6. $IDP \rightarrow C : Lsd$
7. $C \rightarrow IDP : \{APIKey, Credentials, Lsd\}_{CIDPKey}$
8. $IDP \rightarrow C : \{SP, \{Session\}_{SPKey}\}_{CIDPKey}$
9. $C \rightarrow SP : \{Session\}_{SPKey}$
10. $SP \rightarrow C : Session$
11. $C \rightarrow SP : ResourceReq, Session$
12. $SP \rightarrow C : Resource$

**Fig. 3.** Facebook Connect authentication protocol in Alice-Bob notation

1. $C \rightarrow SP : ResourceReq$
2. $SP \rightarrow C : IDP, APIKey$
3. $C \rightarrow IDP : SessionReq, APIKey$
4. $IDP \rightarrow C : Lsd$
5. $C \rightarrow IDP : \{APIKey, Credentials, Lsd\}_{CIDPKey}$
6. $IDP \rightarrow C : \{SP, Session\}_{CIDPKey}$
7. $C \rightarrow SP : ResourceReq, Session$
8. $SP \rightarrow C : Resource$

**Fig. 4.** Facebook Connect authentication protocol, short version

fresh by the client, and communicated to the Identity Provider through an external secure channel (which corresponds to the handshake protocol in SSL). In Section 3.2 we will see how this channel can be represented in HLPSL.

Another technical aspect concerns the modeling of the Cross-Domain Communication. As described in the previous section, the session credentials provided by IDP has to be "decoded" by some JavaScript code embedded in the "cross-domain communication channel" page, which resides on the Service Provider, before being stored in cookies. This mechanism is modeled by first sending the session credentials encrypted with a public key $SPKey$ (step 8); the client passes this message to the (cross-domain communication channel on) SP, which gives back the session credentials in clear (step 9).

In fact, one can notice that in this protocol there are several redundant steps, which can be actually omitted without altering the security analysis. First, we can assume the user not already logged in to the Identity Provider, when the authentication flow begins; hence, steps 3 and 4 are redundant and can be omitted. Secondly, the Cross-Domain Communication channel is not relevant either: the only important aspect is that the session credentials coming from the Identity Provider are eventually received by the Client (in order to be sent to the Service Provider in step 11). Thus, in step 8, the Identity Provider can send the session cookies directly to the Client, and steps 9, 10 can be omitted.

These simplifications allow for a shorter and simpler formalization, which is presented in Figure 4. It is important to notice that this formalization still captures the original protocol; in fact, we have formalized and analyzed also the longer version of Figure 3, obtaining the same results. Hence, for sake of simplicity, in the rest of the paper, we will consider the shorter formalization.

### 3.2 HLPSL formalization

Translating the protocol in Figure 4 into HLPSL is quite easy, since it is written in Alice-Bob notation. Full Facebook Connect HLPSL code can be obtained from `http://www.dimi.uniud.it/miculan/data/downloads/fbconnect.zip`.

A remark is due about how the key $CIDPKey$ can be shared secretly between C and the IDP. This is implemented by means of a variable `SSLKey` of type `(symmetric_key) set`, which can be seen as a "confidential shared memory" between C and IDP. Thus, the client invents a fresh key `CIDPKey` and sends the credentials encrypted with this key, saving it in the set:

```
role client (
    C,SP,IDP    : agent,
    SSLKey      : (symmetric_key) set, ... )
  transition ...
  3. State=2 /\ RCV(Lsd') =|> State':=3 /\ CIDPKey':=new()
     /\ SND({APIKey.credentials.Lsd'}_CIDPKey')
     /\ SSLKey':=cons(CIDPKey',SSLKey)
```

On the other hand, the identity provider retrieves the key from the set before using it for decrypting the message with the credentials:

```
role identityProvider (
    C,SP,IDP    : agent,
    SSLKey      : (symmetric_key) set, ... )
  transition ...
  2. State=1 /\ in(CIDPKey',SSLKey)
     /\ RCV({APIKey.credentials.Lsd}_CIDPKey') =|> ...
```

Thus, the shared variable `SSLKey` can be seen as an abstraction of the SSL handshake protocol, which is used for establishing the session key `CIDPKey`.

Finally, we describe how the security goals are specified.

First, the authentication goal `authentication_on sp_idp_sig` requests the Identity Provider to be authenticated with respect to the Service Provider, on session cookies signature. The related `request` *goal fact* is included in role `serviceProvider` in the transition where the signature is received from Client (see Figure 5). This can be read as "the Service Provider accepts the hash value and now relies on the guarantee that the Identity Provider exists and agrees with it on this value." The matching `witness` predicate is in role `identityProvider` as part of the transition where the signature is sent to the Client within session information (see Figure 6). This fact should be read "Identity Provider asserts that it wants to be the peer of Service Provider, agreeing on the hash value."

```
role serviceProvider ( ... )
  transition ...
  2. State=1 /\ RCV(resourcereq.Key'.Uid'.Expires'.Ss'.
        Hash(Expires'.Ss'.Key'.Uid'.APISecret)) =|>
     State':=2 /\ SND(resource)
     /\ request(SP,IDP,sp_idp_sig,Hash(Expires'.Ss'.Key'.Uid'.APISecret))
```

**Fig. 5.** Location of `request` *goal fact* related to `authentication_on sp_idp_sig`

```
role identityProvider ( ... )
  transition ...
  2. State=1 /\ in(CIDPKey',SSLKey)
     /\ RCV({APIKey.credentials.Lsd}_CIDPKey') =|>
     State':=2 /\ SSLKey':=delete(CIDPKey',SSLKey)
     /\ Key':=new() /\ Expires':=new() /\ Ss':=new()
     /\ Sig':=Hash(Expires'.Ss'.Key'.Uid.APISecret)
     /\ Session':=(Key'.Uid.Expires'.Ss'.Sig')
     /\ SND({SP.Session'}_CIDPKey')
     /\ witness(IDP,SP,sp_idp_sig,Sig')
```

**Fig. 6.** Location of `witness` *goal fact* related to `authentication_on sp_idp_sig`

Secondly, the secrecy goal `secrecy_of otherresourceid` asserts that other Service Provider resources, not requested by the Client, should be kept secret: no intruder should be able to obtain them.

In order to justify this goal, we need to add an extra transition in role `serviceProvider`, representing the fact that the Service Provider can fulfil other requests. Notice that according to the protocol, the Client is not supposed to make these requests. The related `secret` *goal fact* is included in role `serviceProvider` as part of this extra transition (see Figure 7).

## 4 Attacks on Facebook Connect

In this section we present and discuss the results obtained by analyzing the HLPSL formalization presented in Section 3 using AVISPA (and in particular the OFMC model checker). This analysis has shown that Facebook Connect authentication protocol is subject to a replay attack and a masquerade attack.

**Replay attack** This attack is found trying to achieve the `authentication_on sp_idp_sig` goal, in a scenario represented by an `environment` role with two parallel sessions between the three legitimate agents. The attack trace is shown as the first alternative in the Message Sequence Chart in Figure 8.

```
role serviceProvider ( ... )
  transition ...
  3. State=1 /\ RCV(otherresourcereq.Key'.Uid'.Expires'.Ss'.
        Hash(Expires'.Ss'.Key'.Uid'.APISecret)) =|>
     State':=2 /\ SND(otherresource)
     /\ secret(otherresource,otherresourceid,{SP})
     /\ request(SP,IDP,sp_idp_sig,Hash(Expires'.Ss'.Key'.Uid'.APISecret))
```

**Fig. 7.** Location of `secret` *goal fact* related to `secrecy_of otherresourceid`

This attack can be effectively carried out since the HTTP traffic between the Client and the Service Provider is not encrypted, and actually is common in many HTTP-based services: since HTTP (without SSL) is basically stateless, an intruder can always intercept a packet containing an HTTP request (e.g. using a packet-sniffer like Wireshark), and submit it again.

**Masquerade attack** As observed in the replay attack above, the intruder can acquire the HTTP cookies sent together with a legitimate query. Hence, he can represent them to the Service Provider to be authenticated as the Client to obtain illegitimately other resources (e.g., a user's mailbox, even if it has not been accessed before by the owner).

This attack is found when we require the `secrecy_of otherresourceid` goal to be satisfied. Intruder knows the alternative request *OtherResourceReq*, which can be used in place of *ResourceReq*. Indeed, in the sequence of events found by OFMC (shown as the second alternative in the Message Sequence Chart in Figure 8), the intruder replaces *ResourceReq* with *OtherResourceReq*, gaining the access to a resource which has not been requested by the Client.

## 5 Fixing the protocol

In this section, we discuss how to repair the security flaws described above.

The replay attack is due to the fact that, as in any unencrypted HTTP transaction, an intruder can intercept the traffic and submit it again subsequently. This kind of attacks can be prevented using well-known mechanisms based e.g. on timestamps and nonces. Alternatively, messages could be sent through a secure channel like a SSL session. However, it should be noticed that session credentials have a timeout, after which they cannot be reused for this kind of attacks.

On the other hand, the masquerade attack can also be prevented using SSL channels. However, one can be interested in preventing the masquerade attack without establishing an encrypted channel. To this end, we propose a small correction to the protocol which adds an authentication mechanism ensuring message integrity by means of a secret Diffie-Hellman key, without using SSL channels. The modified protocol, in Alice-Bob notation, is shown in Figure 9; the differences with respect to the protocol in Figure 4 are in boldface.

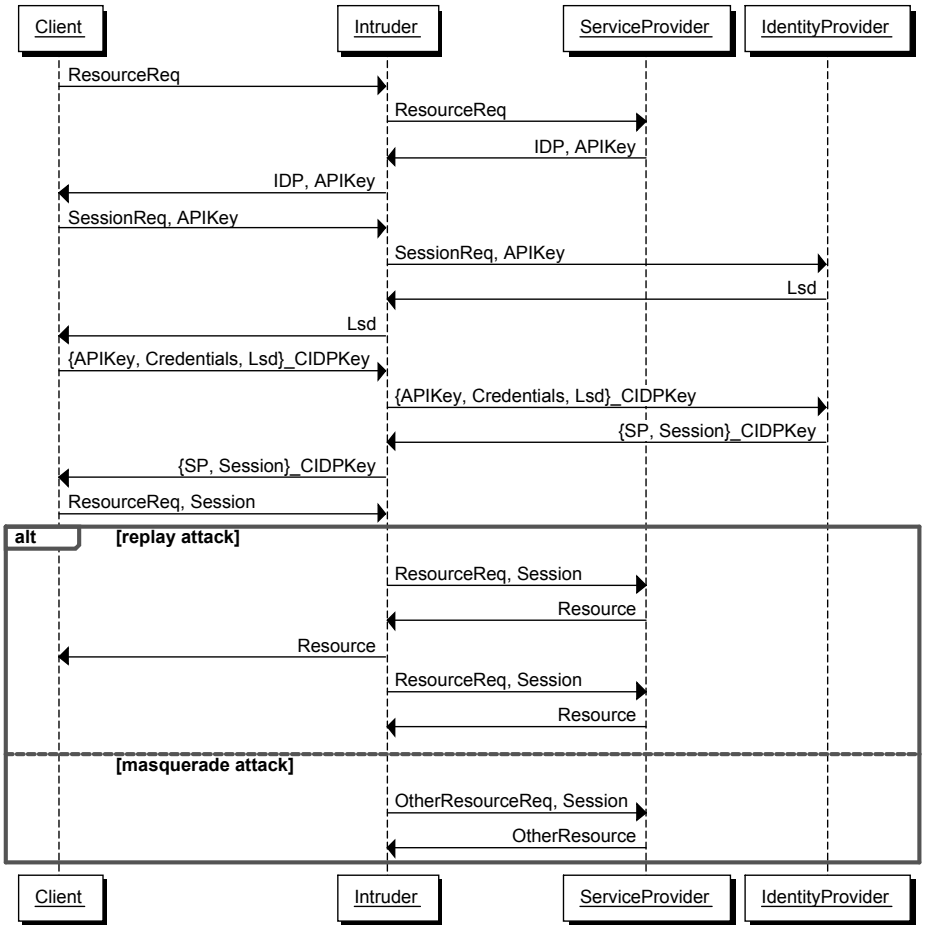


**Fig. 8.** Replay attack and masquerade attack

The protocol requires that after receiving an unauthenticated resource request, the Service Provider chooses a Diffie-Hellman base $A$, a fresh Diffie-Hellman secret key (not shown) and computes the corresponding public key $Ysp$; $A$ and $Ysp$ are sent to the client in step 2, encrypted with IDP public key (which exists because IDP establishes an SSL channel with C). This information is used during authentication, in steps 5, 6 and 7. In step 5 and 6, the IDP decrypts for the client the data $A, Ysp$; notice that this information is still protected by the SSL channel (the encryption with $CIDPKey$). At this point, the Client chooses his own secret key, computes the corresponding public key $Yc$ and the shared Diffie-Hellman secret key $CSPKey$. In step 7, the Client sends to the Service Provider his key $Yc$ and a "message authentication code" obtained by taking the hash of $ResourceReq$ and $CSPKey$. The Service Provider computes, in turn, the shared secret key using $Yc$, then the same hash encoding, and checks the authentication code received from the Client to make sure it matches.

1. $C \rightarrow SP$ : $ResourceReq$
2. $SP \rightarrow C$ : $IDP, APIKey, \{\boldsymbol{A, Ysp}\}_{\boldsymbol{IDPKey}}$
3. $C \rightarrow IDP$ : $SessionReq, APIKey$
4. $IDP \rightarrow C$ : $Lsd$
5. $C \rightarrow IDP$ : $\{APIKey, Credentials, Lsd, \{\boldsymbol{A, Ysp}\}_{\boldsymbol{IDPKey}}\}_{CIDPKey}$
6. $IDP \rightarrow C$ : $\{SP, Session, \boldsymbol{A, Ysp}\}_{CIDPKey}$
7. $C \rightarrow SP$ : $ResourceReq, Session, \boldsymbol{Yc, Hash(ResourceReq.CSPKey)}$
8. $SP \rightarrow C$ : $Resource$

**Fig. 9.** Modified Facebook Connect protocol in Alice-Bob notation

In order to obtain another resource, the intruder should be able to compute the message authentication code for another request, say *OtherResourceReq*. This cannot happen, since there is no way for the intruder to know the secret key *CSPKey*. Notice also that, since $A$ and $Ysp$ are always sent encrypted, the man-in-the-middle attack to the Diffie-Hellman key exchange cannot be carried out.

Again, this protocol has been formalized in HLPSL; full code is available at `http://www.dimi.uniud.it/miculan/data/downloads/fbconnect.zip`.

Formal verification has been carried out using the OFMC model checker, since it supports modular exponentiation. The algebraic properties of modular exponentiation (needed for Diffie-Hellman key exchanges) are associativity, commutativity and identity (i.e. $x^1 = x$). The model checker has found no attacks against the `secrecy_of otherresourceid` goal. In other words, this security goal is satisfied for a bounded number of sessions, as specified in `environment` role. (Actually, the AVISPA TA4SP backend supports also unbounded scenarios, but it does not support sets nor exponentiation operations.)

*Implementation* Changing the authentication flow in Figure 1 according to the modified authentication protocol in Figure 9, requires a bit of care. First, the Service Provider should communicate the encrypted Diffie-Hellman base and his public key to the client at the first step, e.g. as a cookie, or along with the $APIKey$. These data is then passed to the Identity Provider (Facebook) in the POST request containing the credentials; the answer from the IDP contains $A$ and $Ysp$ in clear (but still covered by the SSL session), to be stored e.g. as a cookie. At this point, the Client can choose his secret key, and compute the corresponding public and shared keys; this can be performed by a JavaScript program contained in the `xd_receiver.php` page, which stores the values in JavaScript "session variables", like e.g. `sessvar.authdata.pubkey` and `sessvar.authdata.secret`.

A subtler modification is in step 7, where the Client has to compute the message authentication code and to send it along with the request and his public key. In practice, this means that each GET request from the Client to the Service Provider, in order to be authenticated, must also carry the hash code computed from the request itself. This "on-the-fly" hash generation can be performed by an `onclick` event handler, defined in the page header, and associated to each link by replacing a tag of the form `<a href="someurl">` with

```
<html>
<head>
[... auxiliary libs omitted ...]
<script type="text/javascript">
[... auxiliary functions omitted ...]
function auth_url(url)
  {
    /* compute the hmac */
    hmac = hex_sha1(url + sessvars.authdata.secret);
    /* store the hmac in a cookie */
    createCookie("fbs_hmac", hmac + "-" + sessvar.authdata.pubkey, 0);
    /* then move to the requested url */
    window.location=url;
  }
</script>
</head>

<body>
[...]
This is an <a href="#" onclick="auth_url('somepage.html')">authenticated
request</a>.
[...]
</body>
</html>
```

**Fig. 10.** JavaScript implementation of a URL authentication mechanism

`<a href="#" onclick="auth_url('someurl')">` (see Figure 10). When the user clicks on the link, instead of following it immediately the browser executes the function `authenticate_url('someurl')`; this computes the authentication code of the requested URL, saves the result in a cookie, and then proceeds to load the requested URL. The resulting request provides the Service Provider with the cookie containing the authentication code (and the public key); then the Service Provider can perform the relevant checks.

## 6 Conclusions

In this paper, we have presented a formal analysis of the Facebook Connect authentication protocol. We have given a protocol formalization first in Alice-Bob notation, and then in HLPSL. Using AVISPA, we have found two security flaws, including a masquerade attack: an intruder can intercept cookies carrying user session information, and reuse them to illegitimately access resources on websites. In order to fix this problem, we have corrected the protocol by adding a message authentication mechanism which prevents an intruder to reuse cookies. This solution has been formally verified using AVISPA; finally, we have briefly described how this correction can be implemented in JavaScript.

In our opinion, the message authentication mechanism we have presented in this paper, can be applied in other situations where cookies are used to convey session credentials, and as such masquerade attacks are possible.

On the other hand, it is not easy to distinguish a replay attack from a iteration of the same request by the legitimate user (like, e.g. a page reload). One can apply known authentication mechanisms, e.g. based on sequence counters, to avoid this vulnerability. We leave as a possible future work the specification and verification of an authentication protocol extending that in Figure 9, taking into account these mechanisms.

One aspect of our formalization is that we have used a shared set variable for representing the SSL handshake and session key exchange. At the moment, not all AVISPA backends support these variables. In fact, very recently there have been promising developments in the AVANTSSAR project (the successor of AVISPA), in order to support channels beside Dolev-Yao ones: in the new specification language (called HLPSL++) it will be possible to describe precisely the security features of a SSL-like channel. We plan to adapt our formalizations to the new platform, as soon as it will be available.

## References


1. A. Armando, D. A. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuéllar, P. H. Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The AVISPA tool for the automated validation of internet security protocols and applications. In K. Etessami and S. K. Rajamani, editors, *Proc. CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285. Springer, 2005.
2. A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and M. L. Tobarra. Formal analysis of SAML 2.0 web browser single sign-on: breaking the SAML-based single sign-on for Google Apps. In V. Shmatikov, editor, *Proc. FMSE*, pages 1–10. 2008.
3. AVISPA Project. Deliverable d6.1: List of selected problems. Technical report, `http://avispa-project.org`, 2005.
4. J. L. Hernandez-Ardieta, A. I. González-Tablas Ferreres, and B. Ramos. Formal validation of OFEPSP+ with AVISPA. In P. Degano and L. Viganò, editors, *Proc. ARSPA-WITS*, volume 5511 of *Lecture Notes in Computer Science*, pages 124–137. Springer-Verlag, 2009.
5. A. Kazeniac. Social networks: Facebook takes over top spot, Twitter climbs. Available at `http://blog.compete.com/2009/02/09/facebook-myspace-twitter-social-network/`, Feb. 2009. Compete.com.
6. OASIS. Profiles for the OASIS Security Assertion Markup Language (SAML) v2.0. Available at `http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf`, Mar. 2005.
7. OpenID Foundation. Openid authentication 2.0. Available at `http://openid.net/specs/openid-authentication-2_0.html`, Dec. 2007.