Discovering Concrete Attacks on Website Authorization by Formal Analysis

Chetan Bansal, Karthikeyan Bhargavan, Sergio Maffeis

Discovering Concrete Attacks on Website Authorization by Formal Analysis

Chetan Bansal
[†]*BITS Pilani-Goa*

Karthikeyan Bhargavan
[*]*INRIA Paris-Rocquencourt*

Sergio Maffeis
[‡]*Imperial College London*

***Abstract*—Social sign-on and social sharing are becoming an ever more popular feature of web applications. This success is largely due to the APIs and support offered by prominent social networks, such as Facebook, Twitter, and Google, on the basis of new open standards such as the OAuth 2.0 authorization protocol. A formal analysis of these protocols must account for malicious websites and common web application vulnerabilities, such as cross-site request forgery and open redirectors. We model several configurations of the OAuth 2.0 protocol in the applied pi-calculus and verify them using ProVerif. Our models rely on WebSpi, a new library for modeling web applications and web-based attackers that is designed to help discover concrete website attacks. Our approach is validated by finding dozens of previously unknown vulnerabilities in popular websites such as Yahoo and WordPress, when they connect to social networks such as Twitter and Facebook.**

I. Introduction

A growing number of websites now seek to use social networks to personalize each user's browsing experience. For example, using the social sign-on, social sharing, and social integration APIs provided by Facebook, a website can read and write social data about its visitors, without requiring them to establish a dedicated personal profile. Access to these APIs is mediated by an authorization protocol that ensures that only websites that a user has explicitly authorized may access her social data.

Web authorization protocols. After years of *ad hoc* authentication and authorization mechanisms for web APIs, a series of standards have emerged. SAML [14] and other XML-based security protocols (such as Cardspace) are primarily used for SOAP-based API access, for example, on Amazon and Microsoft Azure. OpenID [29] is used for light-weight user authentication, for example, on Google and PayPal. OAuth [20], [22] is used for REST-based API access to social APIs, for example, on Twitter and Facebook.

It is no longer uncommon to see websites supporting a variety of login options using different social networks. Consensus seems to be emerging around the use of some variation or combination of the OpenID and OAuth protocols [6], and OAuth 2.0 [22] is currently the most widely supported protocol for API authorization, especially for REST, AJAX, and JSON-based API access. It is currently supported by Google, Facebook, and Microsoft, among others. OpenID Connect is a proposal to build the next version of OpenID on top of OAuth 2.0, hence unifying API-based authentication and authorization in a single framework.

Formal analyses of web security. Web authorization protocols have been subject to careful human analysis [25], [17], which can detect some potential vulnerabilities. However, most practical vulnerabilities depend on specific deployment configurations that are too difficult to analyze systematically by hand. Automatic tools such as Alloy [24], AVISPA [1] and ProVerif [9] have proven to be effective in the formal analysis of security protocols. Unfortunately, these tools are geared towards low-level network attackers and it is often difficult to relate the formal counterexamples produced by such analyses to concrete website attacks.

Our approach. In this paper, inspired by [5], we define an automated framework to find web authorization vulnerabilities in a systematic way. We show how a protocol designer can model different protocol configurations and verify them against different attacker models, until she reaches a design that satisfies her specific security goals. After all, choosing the right adversary is fundamental to reason about composition of security mechanisms [18].

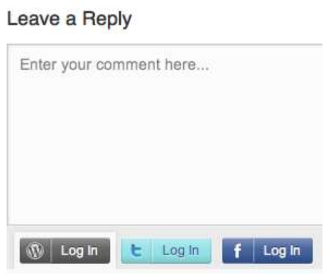
We model various configurations of the OAuth 2.0 protocol in the applied pi-calculus [3] and analyze them using ProVerif. Our models rely on a generic library, WebSpi, that defines the basic components (users, browsers, HTTP servers) needed to model web applications and their security policies. The library also defines an *operational* web attacker model so that attacks discovered by ProVerif can be mapped to concrete website actions closely corresponding to the actual PHP implementation of an exploit. The model developer can fine-tune the analysis by enabling and disabling different classes of attacks. The effectiveness of our approach is testified by the discovery of several previously unknown vulnerabilities involving some of the most popular web sites, including Facebook, Yahoo, and Twitter. We have reported these problems and helped fixing them.

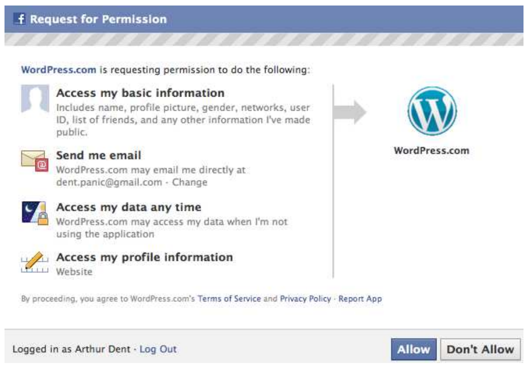
The main contributions of this paper are the WebSpi library, a formal analysis of OAuth 2.0 using WebSpi and ProVerif, and a description of new concrete website attacks found and confirmed by our formal analysis. Full ProVerif scripts, including the WebSpi library, the OAuth 2.0 model, and formal attacks, are available online [15].

II. Motivating Example: Social Sign-On

Social sign-on (or social login) is the use of a social network to login to a third-party website, without having to register at the website. It is a service provided by many social networks and authentication servers, using protocols such as OpenID (e.g. Google) and OAuth (e.g. Facebook). For clarity, we henceforth adopt OAuth terminology: a user who owns some data is called a *resource owner*, a website that holds user data and offers API access to it is called a *resource server*, and a third party that wishes to access this data is called a *client* or an *app*.

Consider `WordPress.com`, a website that hosts hundreds of thousands of active blogs with millions of visitors every day. A visitor may comment on a blog post only after authenticating herself by logging in as a WordPress, Facebook, or Twitter user. When visitor Alice clicks on "Log in with Facebook", an authorization protocol is set into motion where Alice is the resource owner, Facebook the resource server, and WordPress the client. Alice's browser is redirected to `Facebook.com` which pops up a window asking to allow `WordPress.com` to access her Facebook profile. `WordPress.com` would like access to Alice's basic information (her name and email address) as proof of identity.




Request for Permission

WordPress.com is requesting permission to do the following:

Access my basic information
Includes name, profile picture, gender, networks, user ID, list of friends, and any other information I've made public.

Send me email
WordPress.com may email me directly at dent.panic@gmail.com · Change

Access my data any time
WordPress.com may access my data when I'm not using the application

Access my profile information
Website

WordPress.com

By proceeding, you agree to WordPress.com's Terms of Service and Privacy Policy · Report App

Logged in as Arthur Dent · Log Out

If Alice authorizes this access, she is sent back to `WordPress.com` with an API access token that lets `WordPress.com` read her email address from Facebook and log her in. All subsequent actions that Alice performs at `WordPress.com`, such as commenting on a blog, are associated with her Facebook identity.

Social sharing. Some client websites also implement *social sharing*: reading and writing data on the resource owner's social network. For example, on `CitySearch.com`, a guide with restaurant and hotel recommendations, any review or comment written by a logged-in Facebook user is instantly cross-posted on her profile feed ('Wall') and shared with all her friends. Some websites go further: `Yahoo.com` acts as both client and resource server to provide deep *social integration* where the user's social information flows both ways, and may be used to enhance her experience on a variety of online services, such as web search and email.

Security goals. Let us first consider the informal security goals of the social sign-on interaction described above, from the viewpoint of Alice, WordPress and Facebook.

- Alice wants to ensure that her comments will appear under her own name; nobody else can publish comments in her name; no unauthorized website should gain access to her name and email address; even authorized websites should only have access to the information she decided to share.
- WordPress wants to ensure that the user trying to log in and post comments as Alice, is indeed Alice.
- Facebook wants to ensure that both the resource owner and client are who they say they are, and that it only releases data when authorized by the resource owner.

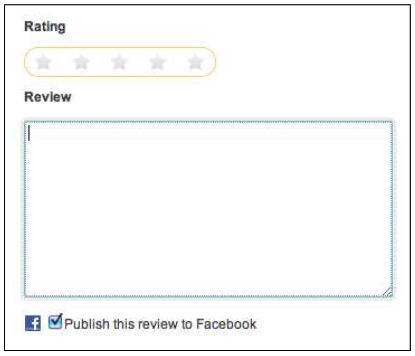
These security goals are fairly standard for three-party authentication. What makes social sign-on more interesting, is that it needs to enforce these goals under normal web conditions. For example, Alice may use the same browser to log-in on WordPress and, in another tab, visit an untrusted website, possibly over an insecure Wi-Fi network. In such a scenario, threats to Alice's security goals include: network attackers who can intercept and inject clear-text HTTP messages between Alice and WordPress; malicious websites who can try to fool Facebook or Alice by pretending to be WordPress; malicious users who can try to fool Facebook or WordPress by pretending to be Alice.

Web-based attacks. Network attacks are well understood, and can be mitigated by the systematic use of HTTPS [30], or more sophisticated cryptographic mechanisms. Many websites, such as Facebook, do not even seek to protect against network attackers, allowing users to browse over HTTP. They are more concerned about website- and browser-based attacks, such as Cross-Site Scripting (XSS), SQL Injection, Cross-Site Request Forgery (CSRF) and Open Redirectors.

For example, various flavours of CSRF are common on the web. When a user logs into a website, the server typically generates a fresh, unguessable, session identifier and returns it to the browser as a *cookie*. All subsequent requests from the browser to the website include this cookie, so that the website associates the new request with the logged-in session. However, if the website relies only on this cookie to authorize security-sensitive operations on behalf of the user, it is vulnerable to CSRF. A malicious website may fool the user's browser into sending a (cross-site) request to the vulnerable website (by using JavaScript, HTTP redirect, or by inviting the user to click on a link). The browser will then automatically forward the user's session cookie with this forged request, implicitly authorizing it without the knowledge of the user, and potentially compromising her security. A special case is called *login CSRF*: when a website's login form itself has a CSRF vulnerability, a malicious website can fool a user's browser into silently logging in to the website under the attacker's credentials, so that future user actions are credited to the attacker's account. The typical countermeasure for CSRF is to require in every security-sensitive request to the website a session-specific nonce that would be difficult for a malicious website to forge. The nonce can be embedded in the target URL or within a hidden form field. However, such mechanisms are particularly difficult to enforce in multi-party interactions such as social sign-on.

Social CSRF attacks. We now describe one of the new attacks we found thanks to our formal analysis of OAuth in Section V. This example shows how a CSRF attack on low-value client website `CitySearch.com` can be translated into an attack on its high-value resource server `Facebook.com`.

Suppose Alice clicks on the social login form on CitySearch to log in with her Facebook account. So, CitySearch obtains an API access token for Alice's Facebook profile. If Alice then wants to review a restaurant on CitySearch, she is presented with a form that also asks her if she would like her review to be posted on Facebook.



When she submits this form, the review is posted to CitySearch as a standard HTTP POST request; CitySearch subsequently reposts it on Alice's Facebook profile (using its API access token on the server side).

```
POST /rate/listing?listingId=628337570 HTTP/1.1
 Host: lasvegas.citysearch.com
 Content-Type: application/x-www-form-urlencoded
 Cookie: usrid=ab76fb...

 title=GREAT&rating=6&publishToFacebook=true&text=...
```

We found that the review form above is susceptible to a regular CSRF attack; the contents of the POST request do not contain any nonce, except for the cookie, which is automatically attached by the browser. So, if Alice were to go to an untrusted website while logged in to CitySearch, that website could post a review in Alice's name on CitySearch (and hence, also on Alice's Facebook profile.)

Moreover, CitySearch's social login form is also susceptible to a CSRF attack. So, if Alice has previously used social login on CitySearch, any website that Alice visits could submit this form to silently log in Alice on CitySearch via Facebook. Alice is not asked for permission since Facebook typically only asks a user for authorization the first time she logs in to a client. Combining the two attacks, we built a demonstrative malicious website that, when visited by a user who has previously used social sign-on on CitySearch, can automatically log her into CitySearch and post arbitrary reviews in her name both on CitySearch and Facebook. This is neither a regular CSRF attack on Facebook, nor a login CSRF attack on CitySearch (the user signs-in in her own name). We call this class of attack a *Social CSRF* attack.

Attack amplification. To understand the novelty of Social CSRF attacks, it is instructive to compare Alice's security before and after she used social sign-on on CitySearch. Before, Alice's reviews were subject to a CSRF attack, but only if she visited a malicious site at the same time as when she was logged into CitySearch. No website could log Alice automatically into CitySearch since it would require Alice's password. Moreover, no website would have been able to post a message on Alice's Facebook wall without her permission, because Facebook implements strong CSRF protections. But now, even if Alice uses social login once on CitySearch and never visits the site again, a website attacker will always be able to modify both Alice's Facebook wall and her CitySearch reviews.

Empirically, we find that social CSRF attacks are widespread, probably because websites have been encouraged to hastily integrate social login and social sharing without due consideration of the security implications. Social CSRFs pose a serious threat both to resource servers and clients, because these attacks can be amplified both ways. On one hand, as we have seen, a CSRF vulnerability in any Facebook client becomes a CSRF on Facebook. On the other hand, a login CSRF attack that we discovered on `twitter.com` (see Section IV), becomes a login CSRF vulnerability for all of its client websites.

Towards a systematic discovery of web-based attacks. The CitySearch vulnerability described above composes two different CSRF attacks, involves three websites and a browser, and consists of at least nine HTTP(S) connections. It does not depend on the details of the underlying authorization protocol, but the other vulnerabilities in Section V rely on specific weaknesses in OAuth 2.0 configurations. We found such attacks by a systematic formal analysis, and we believe at least some would have escaped a human protocol review.

Modeling web-based attackers offers new challenges compared to the attackers traditionally considered in formal cryptographic protocol analysis. For example, in a model that enables the attacker to control the network, websites such as CitySearch and Facebook are trivially insecure as most user data is sent over insecure HTTP. With such strong attacker models, we are unlikely to discover subtle website-based attacks such as CSRF. Conversely, a model that treats the browser and the user as one entity will miss CSRF attacks completely. In Section IV we present a web security library that allows us to fine-tune the attacker model, enabling the discovery of new and interesting web attacks.

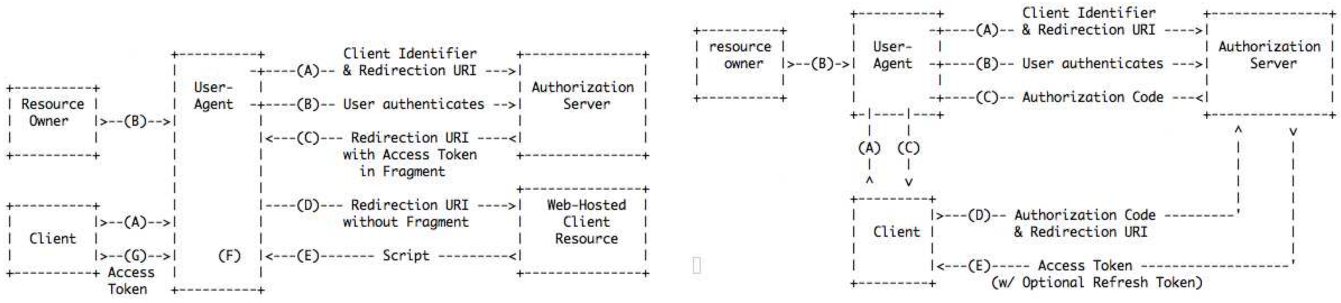

Resource Owner
User-Agent
Client Identifier & Redirection URI
Authorization Server
(A)
(B)
User authenticates
(C) Redirection URI with Access Token in Fragment
(D) Redirection URI without Fragment
Web-Hosted Client Resource
Client
(E) Script
(F)
(G)
Access Token
resource owner
User-Agent
(A) Client Identifier & Redirection URI
(B) User authenticates
(C) Authorization Code
Authorization Server
Client
(D) Authorization Code & Redirection URI
(E) Access Token (w/ Optional Refresh Token)

Figure 1. OAuth 2.0: User-Agent Flow (left) and Web Server Flow (right).

III. OAuth 2.0: Browser-based API Authorization

The aim of the OAuth 2.0 authorization protocol is to enable third party clients to obtain limited access, on behalf of a resource owner, to the API of a resource server [22]. The protocol involves five parties: a *resource server* that allows access to its resources over the web on receiving an access token issued by a trusted *authorization server*; a *resource owner* who owns data on the resource server, has login credentials at the authorization server, and uses a *user-agent* (browser) to access the web; a *client* website, that needs to access data at the resource server, and whose application credentials are registered at the authorization server. In the example of Section II, Facebook is both the authorization server and resource server; we find that this is a common configuration.

The first version of OAuth was designed to unify existing authorization mechanisms implemented by Twitter, Flickr, and Google. However, it was criticized as being website-centric, inflexible, and too complex. In particular, the cryptographic mechanisms used to protect authorization requests and responses were deemed too difficult for website developers to implement (correctly).

OAuth 2.0 is designed to address these shortcomings. The draft specification defines five different *flows* or protocol configurations, only two of which directly apply to website applications. The protocol itself requires no cryptographic mechanisms whatsoever and instead relies on transport layer security (HTTPS). Hence, it claims to be lightweight and flexible, and has fast emerged as the API authorization protocol of choice, supported by Microsoft, Google and Facebook, among others. We next describe the two website flows of OAuth 2.0, their security goals, and their typical implementations.

User-Agent Flow. The User-Agent flow, also called Implicit Grant flow, is meant to be used by client applications that can run JavaScript on the resource owner's user-agent. For example, it may be used by regular websites or by browser extensions. The authorization flow, adapted from the specification, is depicted on the left in Figure 1.

Let the resource server be located at the URL RS and its authorization server be located at AS. Let the resource owner RO have a username u and password p at AS. Let the client be located at URL C and have an application identifier id at AS. The message flow and relevant security events of the user-agent flow are as follows:

1) Login(RO,b,sid,AS,u): RO using browser b starts a login session sid at AS using credentials u,p.
2) SocialLogin(RO,b,sid',C,AS,RS): RO using b starts a social sign-on session sid' at C using AS for RS.
3) TokenRequest(C,b,AS,id,perms): C redirects b to AS requesting a token for id with access rights perms.
4) Authorize(RO,b,sid,C,perms): AS looks up id and asks RO for authorization; RO using browser b in session sid at AS authorizes C with perms.
5) TokenResponse(AS,b,C,token): AS redirects b back to C with an access token.
6) APIRequest(C,RS,token,getId()): C makes an API request getId() to RS with token.
7) APIResponse(RS,C,token,getId(),u): RS verifies token, accepts the API request and returns u to C.
8) SocialLoginAccept(C,sid',u,AS,RS): C accepts RO's social sign-on session sid' as u at AS for RS.
9) SocialLoginDone(RO,b,sid',C,u,AS,RS): RO is logged in to C in a browser session sid' associated with u at AS, granting access to RS.

These steps may be followed by any number of API calls from the client to the resource server, on behalf of the resource owner. Each step in this flow consists of one (or more) HTTP request-response exchanges. The specification requires that the AS *must* and the C *should* implement these exchanges over HTTPS. In the rest of this paper, we assume that all OAuth exchanges occur over HTTPS unless specified otherwise.

For example, in a user-agent flow between WordPress and Facebook, the TokenRequest(C,b,AS,id,perms) step is typically implemented as an HTTPS redirect from WordPress to a URI of the form: `https://www.facebook.com/dialog/permissions.request?app_id=id&perms=email`. The TokenResponse is also an HTTPS redirect back to WordPress, of the form: `https://public-api.wordpress.com/connect/?service=facebook#access_token=token`. Note that the access token is passed as a fragment URI.

JavaScript running on behalf of the client can extract the token and then pass it to the client when necessary.

Web Server Flow. The Web Server flow, also called Explicit Grant flow or Authorization Code flow, can be used by client websites wishing to implement a deeper social integration with the resource server by using server-side API calls. It requires that the client must have a security association with the authorization server, using for example an application secret. Moreover, it requires that the access token be retrieved on the server-side by the client. The motivation for this is two-fold. (i) It allows the authorization server to authenticate the client's token request using a secret that only the client and the server know. In contrast, the authorization server in the user-agent flow has no way to ensure that the client in fact wanted a token to be issued, it simply sends a token to the client's HTTPS endpoint. (ii) It prevents the access token from passing through the browser, and hence ensures that only the client application may access the resource server directly. In contrast, the access token in the user-agent flow may be leaked through referrer headers or browser history to malicious third-party JavaScript running on the client.

The authorization flow is depicted in Figure 1. Let the client at URL C and have both an application identifier id and a secret sec pre-registered at AS. The difference between the web server and user-agent flows begins after the SocialLogin step, and ends before the APIRequest step:

3) CodeRequest(C,b,AS,id,perms): C redirects b to AS requesting authorization for id with perms.
4) Authorize(RO,b,sid,C,perms): AS looks up id and asks RO for authorization; RO using browser b in session sid at AS authorizes C with perms.
5) CodeResponse(AS,b,C,code): AS redirects b back to C with an authorization code.
5.1) APITokenRequest(C,AS,code,id,sec): C makes an API request for an access token to AS with code, id, and sec.
5.2) APITokenResponse(AS,C,token): AS checks id and sec, verifies the code and returns a token to C.

Additional Protocol Parameters. In addition to the basic protocol flows outlined above, OAuth 2.0 enables several other optional features. Our models capture the following:
- *Redirection URI*: Whenever a client sends a message to the authorization server, it may optionally provide a redirect_uri parameter, where it wants the response to be sent. In particular, the TokenRequest and CodeRequest messages above may include this parameter, and if they do, then also the corresponding APITokenRequest must include it. The client may thus ask for the authorization server to redirect the browser to the same page (or state) from which the authorization request was issued. Since the security of OAuth crucially depends on the URI where codes and tokens are sent, the specification strongly advises that clients must register all their potential redirection URIs beforehand at the authorization server. If not, it predicts attacks where a malicious website may be able to acquire codes or tokens and break the security of the protocol. Indeed, our analysis found such attacks both in our model and in real websites. We call such attacks *Token Redirection* attacks.
- *State Parameter*: After the TokenRequest or CodeRequest steps above, the client waits for the authorization server to send a response. The client has no way of authenticating this response, so a malicious website can fool the resource owner into sending the client a different authorization code or access token (belonging to a different user.) This is a variation of the standard website login CSRF attack that we call a *Social Login CSRF* attack. To prevent this attack, the OAuth specification recommends that clients generate a nonce that is strongly bound to the resource owner's session at the client (say, by hashing a cookie). It should then pass this nonce as an additional state parameter in the CodeRequest or TokenRequest messages. The authorization server simply returns this parameter in its response, and by checking that the two are the same, the client can verify that the returned token or code is meant for the current session.

After incorporating the above parameters, the following protocol steps are modified as shown:

TokenRequest(C,b,AS,id,perms,state,redirect_uri)
TokenResponse(AS,b,redirect_uri,state,token)
CodeRequest(C,b,AS,id,perms,state,redirect_uri)
CodeResponse(AS,b,redirect_uri,state,code)
APITokenRequest(C,AS,code,id,sec,redirect_uri)
APITokenResponse(AS,C,token)

Our analysis does not cover other features of OAuth, such as refresh tokens, token and code expiry, the right use of permissions, or the other four flows described in the specification. We leave these features for future work.

A Threat Model for OAuth 2.0. The OAuth specification and documents commenting on it [25], [17] together provide an exhaustive list of potential threats to the protocol. We consider a subset of these threats.

The ultimate aim of the attackers we consider is to steal or modify the private information of an honest resource owner, for example by fooling honest or buggy clients, authorization servers, or resource owners into divulging this information. To this end, we consider: network based attackers who can sniff, intercept, and inject messages into insecure HTTP sessions; malicious websites that honest resource owners may browse to; malicious clients, resource owners, and authorization servers; honest clients with redirectors that may forward HTTP requests to malicious websites; honest clients and authorization servers with CSRF vulnerabilities. We do not explicitly consider attacks on the browser or operating system of honest participants; we treat such participants as *compromised*, that is, as fully controlled by an attacker.

Security Goals for OAuth 2.0. We describe the security goals for each participant by defining Datalog-like authorization policies [19] that must be satisfied at different stages of the protocol. The policy A :− B,C is read as "A if B and C".

The resource owner has completed successfully the social sign-on if it intended to sign-on to the client, if it agreed to authorize the client, and if the client and resource owner agree upon the user's social identity and session identifier:

```
SocialLoginDone(RO,b,sid',C,u,AS,RS) :−
   Login(RO,b,sid,AS,u),SocialLogin(RO,b,sid',C,AS,RS),
   Authorize(RO,b,sid,C,idPermission),
   Says(C,SocialLoginAccept(C,sid',u,AS,RS)).
```

The authorization server must ensure that a token is issued only to authorized clients. Its policy for the user-agent flow is written as:

```
TokenResponse(AS,b,C,state,token) :−
  ValidToken(token,AS,u,perms),Says(RO,Login(RO,b,sid,AS,u)),
  Authorize(RO,b,sid,C,perms),ValidClient(C,id,redirect_uri).
```

Note that we do not require that the TokenResponse is only issued in response to a valid TokenRequest: at this stage, the user-agent flow has not authenticated the client yet.

The web server flow policy of the authorization server is

```
APITokenResponse(AS,C,state,token) :−
  ValidToken(token,AS,u,perms),Says(RO,Login(RO,b,sid,AS,u)),
  Says(C,TokenRequest(C,b,AS,id,perms,state,redirect_uri)),
  Authorize(RO,b,sid,C,perms),ValidClient(C,id,redirect_uri).
```

From the viewpoint of the resource server, every API call must be authorized by a token from the authorization server.

```
APIResponse(RS,b,C,token,req,resp) :−
  ValidToken(token,AS,u,perms),Permitted(perms,req),
  Says(C,APIRequest(C,RS,token,req)).
```

Finally, from the viewpoint of the client, the social sign-on has completed successfully if it has correctly identified the resource owner currently visiting its page, and obtained an access token for the API accesses it requires.

```
SocialLoginAccept(C,sid',u,AS,RS) :−
  Says(RO,SocialLogin(RO,b,sid',C,AS,RS)),
  Says(AS,TokenResponse(AS,b,C,token)),
  Says(RS,APIResponse(RS,C,token,getId(),u)).
```

IV. The WebSpi library and its usage

Various calculi, starting from the spi calculus [4], have been remarkably successful as modeling languages for cryptographic protocols, thanks also to the emergence of automated verification tools that can analyze large protocol models. Following in this tradition, we model web security mechanisms in an applied pi-calculus, and verify them using ProVerif. We identify a set of idioms that are particularly useful in modeling web applications and web-based attackers, and offer them as a library, called WebSpi, available to other developers of web models. The syntax of the language used by ProVerif is summarized in Appendix A, and its authoritative definition is the ProVerif manual [11].

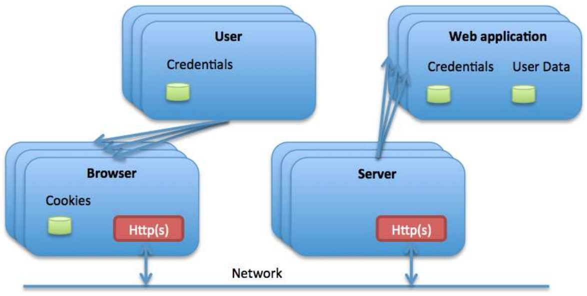


Figure 2. WebSpi architectural diagram.

A. *Principals, Browsers, and HTTP Servers*

WebSpi models consist of users who surf the Internet through web browsers, to interact with web applications that are hosted by web servers. Figure 2 gives a schematic representation of the model. Users and servers are the *principals*, or agents, of our model. Users hold credentials to authenticate with respect to a specific web application (identified by a host name and by a path) in the table credentials, whereas servers hold private and public keys to implement TLS secure connections in the table serverIdentities.

```
table credentials(Host,Path,Principal,Id,Secret).
table serverIdentities(Host,Principal,pubkey,privkey).
```

These tables are private to the model and represent a pre-existing distribution of secrets (passwords and keys). They are populated by the *process* CredentialFactory that provides an API for the attacker (explained later) to create an arbitrary population of principals and compromise some of them. The process WebSurfer models a generic user principal who is willing to browse the web to any public URL.

Browsers and web servers communicate using the HTTP(S) protocol over a *channel* net, and their core functionality is modeled by the processes HttpClient and HttpServer. The process HttpClient accepts an HTTP request and a URI (from a user, or a client-side web application) on the channel httpClientRequest and performs the desired GET or POST request-response exchange with the specified URI. If the response is an HTTP redirect, then it retries the exchange with the new URI. Otherwise, it returns the response on the private channel httpClientResponse. The HttpClient also maintains browser cookies for each browser and host in a table browserCookies(Browser,Host,Cookie) With every HTTP(S) request, it automatically attaches the cookies stored for the server host, and every response contains an updated cookie which the client should store in the table.

The process HttpServer is the dual of HttpClient: it accepts HTTP(S) requests over net, forwards them to the server-side application over the private channel httpServerRequest, waits for a response on httpServerResponse, and returns it to the client. Server-side sessions (stored in a table serverSessions) are maintained by individual web applications.

These processes incorporate a simple model of anonymous HTTP(S) connections: each request to an HTTP URI is encrypted with a fresh symmetric key, that is in turn encrypted under the server's public key. The response is encrypted with the same symmetric key.

B. Distributed security policies

Since their introduction in the context of the spi calculus [21], Datalog-like security policies have proven to be an ideal tool to describe enforceable authorization and authentication policies for distributed security protocols. A program statement such as Assume(UserSends(u,m)) adds to a global knowledge base the fact that user u has sent message m. Such a statement should precede the actual code used by the user to send the message, and its purpose it to reflect the operation in the policy world. A program statement such as Expect(ServerAuthorizes(s,u,d)) instead means that at this point in the code, we must be able to prove that the server s is willing to authorize user u to retrieve data d. The main idea is that the Expect triggers a query on the security policy, using the facts known (and assumed) so far. In this paper, we adopt a similar style to express our policies and bind them to protocol code.

Using ProVerif's native support for predicates defined by Horn clauses, we embed the assumption of fact e by the code if Assume(e) then P, where Assume is declared as a *blocking* predicate, so that ProVerif treats Assume(e) as an atomic fact and adds it as a hypothesis in its proof derivations about P. Conversely, the expectation that e holds is written as event Expect(e). Security policies are defined as Horn clauses extending a predicate fact. In particular, the WebSpi library includes the generic clause forall e:Fact; Assume(e) →fact(e) that admits assumed facts, and a generic *security query* forall e:Fact; event(Expect(e)) ⟹fact(e) that requires every expected predicate to be provable from the policy and previously assumed facts.

Moreover, inspired by Binder [19], [2], we also encode a standard Says modality, axiomatised below:

```
forall p:Principal,e:Fact; fact(e) →fact(Says(p,e));
forall p:Principal,e:Fact; fact(Compromised(p)) →fact(Says(p,e)).
```

The two rules state that if a fact is true, it can be assumed to be said by any principal, and that if a principal is known to be compromised, denoted by the fact Compromised(p), then it cannot be trusted anymore and is ready to say anything.

Although predicates have existed in ProVerif for some time, to our knowledge, we are the first to use them to embed Datalog-like policies and use them for a real case study.

C. Modeling web applications using WebSpi

To model a web application using WebSpi, one typically writes three processes:

- a server-side (PHP-like) process representing the website, running on top of HttpServer;
- a client-side (JavaScript-like) process representing the web page, running on top of HttpClient;
- a user process representing the behaviour of a human who uses a browser to access the web application.

In some simple cases, the second and third process may be combined. In addition to messaging over HTTP(S), these processes may perform other actions, such as cryptography or database operations.

As an example, we show how to model and analyze the core functionality of a typical website login application (such as the Twitter login form above). We also consider enhancements such as CSRF protections on the form, and JavaScript that hashes the password. We illustrate our methodology using the login application, which is a building block of the OAuth models considered in this paper.

First, we model the login user as a process that first views the login form and then fills and submits it:

```
let LoginUserAgent() =
  in(httpClientResponse,(b:Browser,sid:Cookie,sp:Principal,
                                       u:Uri,d:bitstring));
  let up = principal(b) in
  let uri(proto,h,app) = ep(u) in
  if proto = https() then
  if loginForm = formTag(d) then
  get credentials(=h,=app,=p,uid,pwd) in
  if Assume(Login(up,b,sid,ep(u),uid)) then
  out(httpClientRequest,
  (b,sid,formAction(d),u,httpPost(loginFormReply(d,uid,pwd))));
  in(httpClientResponse,(=b,=sid,=sp,formAction(d),loginSuccess()));
  event Expect(ValidSession(up,b,sid,ep(u)))
```

First, the user-agent receives the login page from the server, parsed through the HttpClient process, as a message on the channel httpClientResponse. In particular, it receives the URL u of the web page, the login form d, and a session cookie sid (already stored in the browserCookies table). If the protocol used by the URL is HTTPS, if the form is indeed a login form, and if the user wishes to log in to the web application, she retrieves her password (from the credentials table) and submits the form, by sending an HTTP POST request to the URL specified in the form action, via the httpClientRequest channel. Hence, this model assumes a careful user who only releases her password to the right website, and does not fall victim to phishing attacks. (We consider careless users to be *compromised*, that is under the control of the attacker.) Finally, the user-agent waits for the loginSuccess() page to be loaded before proceeding with security sensitive actions.

Both the statements Assume(Login(up,b,sid,ep(u),uid)) and Expect(ValidSession(up,b,sid,ep(u))) are included in the security specification. The former states that the user up agrees to log in as the user uid at the web application ep(u), using the browser b in session sid. The latter demands that at this stage the user can be sure to be logged in to the right website. We model the server-side login application as follows:

```
let LoginApp(h:Host,app:Path) =
  in(httpServerRequest,(sp:Principal,u:Uri,req:HttpRequest,
                                    hs:Headers,requestId:bitstring));
  if url(https(),h,app) = ep(u) then
  let sid = cookie(hs) in
  let httpPost(loginFormReply(d,uId,pwd)) = req in
  get credentials(=h,=app,p,=uId,pwd) in
  event Expect(LoginAuthorized(sp,uId,ep(u),sid));
  insert serverSessions(h,app,sid,uId);
  out(httpServerResponse,
  (sp,u,httpOk(loginSuccess()),sid,requestId))
```

The server receives parsed web requests from HttpServer on channel httpServerRequest, which is shared between all server-side applications. It first checks that the request was addressed to itself and that it was received over HTTPS. It then parses the headers to extract the session cookie, and parses the request body to obtain the login form containing uId and pwd. It retrieves the credentials of the user uId and checks the validity of the password pwd to authenticate the user. If these checks succeed the application registers a new server session for the user by the command insert serverSessions(h,app,sid,uId); if any check fails, it silently rejects the request; otherwise it returns a page loginSuccess(). Before registering the session, the process issues its policy event Expect(LoginAuthorized(sp,u,ep(u),sid)): the server expects that the user uId logged-in on the session sid has been authenticated and authorized. For clarity, we write the policies for LoginAuthorized and ValidSession in Datalog style (in ProVerif syntax, they are written right-to-left as clauses that extend the fact predicate).

```
LoginAuthorized(sp,uId,e,sid) :− Server(sp,h),User(up,uId,h,app),
                                  endpointUri(e,https(),h,loginPath(app)),
                                  Says(up,Login(up,b,sid,e,uId)).
ValidSession(up,b,sid,e) :− Server(sp,h),User(up,x,h,app),
                            endpointUri(e,https(),h,loginPath(app)),
                            Says(sp,LoginAuthorized(sp,x,e,sid)).
```

The first policy states that the application denoted by the host h (owned by principal sp) and path app (as encoded in the endpoint e) can login over HTTPS the registered user with user-id uId into a session sid if the principal up owning x' credentials intended to do so (or if up was compromised). The second policy states that the browser session sid of user up at endpoint e is valid if the principal sp owning e has logged in the user over HTTPS under the right username.

These policies can be read as the standard *correspondence assertions* typically used to specify authentication properties in cryptographic protocols. However, using predicates, we can also encode more expressive authorization policies that would generally be difficult to write as ProVerif queries.

D. A customizable attacker model

We consider a standard symbolic active (Dolev-Yao) attacker who controls all public channels and some principals, but cannot guess secrets or access private channels. Furthermore, the attacker can create new data and can encrypt or decrypt any message for which it has obtained the cryptographic key, but it cannot break cryptography.

By default, all the channels, tables, and credentials used in WebSpi are private. We define a process AttackerProxy that mediates the attacker's access to these resources, based on a set a configuration flags. The attacker executes a command by sending a message on the *public* channel admin and if the current configuration allows it, the process executes the command and returns the result (if any) on the public channel result. The full list of commands that the attacker can send is listed in Table I. This API is designed to be *operational*: each command corresponds to a concrete attack that can be mounted on a real web interaction. We identify three categories of attacker capabalities enabled by this API:

Managing principals. The first two commands (enabled by the flag NetworkSetup) allow the attacker to set up an arbitrary population of user and server principals by populating the credentials and serverIdentities tables. If these commands are disabled, the model developer must create his own topology of users and servers. The third and fourth command (enabled by the flag MaliciousUsers, MaliciousServers) allow the attacker to obtain the credentials of a selected user or server.

Network attackers. The next two commands (enabled by the flag NetworkAttackers) allow the attacker to intercept and inject arbitrary messages into a connection between any two endpoints. Hence, the attacker can alter the cookies of an HTTP request, but cannot read the (decrypted) content of an HTTPS message.

Website attackers. The next four commands (enabled by UntrustedWebsites) give the attacker an API to build web applications and deploy them (on top of HttpServer) at a given endpoint, potentially on a honest server. This API gives the attacker fewer capabilities than he would have on a compromised server, but is more realistic, and allows us to discover interesting website based (PHP) attacks. The last two commands (enabled by UntrustedJavaScript) model the API provided on the client side, by HttpClient, to JavaScript downloaded from untrusted websites.

E. From ProVerif analysis to concrete web attacks

When analyzing a web application model built on top of WebSpi, the model developer may fine-tune the attacker model and run ProVerif to discover attacks of interest. ProVerif will either prove the model correct (with respect to its security goals), or fail to verify the model, or not terminate. If verification succeeds, the correctness theorem for ProVerif [10] guarantees that no attacks exist, at least among the (limited) class of attacks considered in the model.

When verification fails, ProVerif sometimes produces an

Managing principals	createServer(sp)	create a new server for principal sp
	createUser(up,h,p)	create a new user up for the app at path p on host h
	compromiseUser(id,h,p)	force user with login id on app p at h to reveal its password
	compromiseServer(h)	force principal of server hosted at h to reveal its secret key
Network attackers	injectMessage(e1,e2,m)	send message m to endpoint $e2$ as if it came from $e1$
	interceptMessage(e1,e2)	intercept a message from $e1$ to $e2$
Malicious websites	startUntrustedApp(h,p)	start a malicious application p at h
	getServerRequest(h,p)	intercept a request between the http module and app p at h
	sendServerResponse(h,p,u,r,c,m)	send m to u on behalf of h, p, with cookie c and HTTP response type r, from the server with principal sp
	httpRequestResponse(c,u,m)	send m to u and wait for response
Malicious JavaScript	getClientResponse(b,h,p)	intercept the response from browser b to app h, p
	sendClientRequest(b,h,p,c,u1,u2,m)	send m to h, p as if b clicked on $u1$ on a page from $u2$

Table I
A COMMAND API FOR THE ACTIVE WEB ATTACKER

attack trace, or else it provides a proof derivation that hints towards a potential attack. Because of the way our attacker model is designed, all attacker actions in traces and derivations appear as concrete commands and responses on the admin and result channels. This makes potential attacks remarkably easy to extract, and sometimes such attacks can be translated to real-world web attacks.

As an example, we analyze our WebSpi model of the login application against its two security policies, and explore its robustness against different categories of attackers. Our results are summarized at the beginning of Tables II and III.

If we only enable network attackers, malicious users, and malicious servers, ProVerif proves the model secure. Suppose we relax the LoginUserAgent process so that users may also login over HTTP; ProVerif then finds a network-based password-sniffing attack that breaks both policies.

If we also enable malicious websites, ProVerif finds a standard login CSRF attack. Our login forms, much like the Twitter login form, do not include any unguessable values. So a malicious website that also controls a malicious user Eve can fool an honest user Alice into logging in as Eve. Let us see how we can reconstruct this attack.

In this case, ProVerif produces a proof derivation, but not an attack trace. Such derivations can be very long, since they list all attempted attacks, ending in the successful one, and they explain how the attacker constructed each message. For our example, the derivation has 3568 steps. However, if we select just the messages on the admin and result channels, we end up with a derivation of 89 steps. Most of these steps are redundant commands towards the beginning of the derivation that are easy to identify and discard. Starting from the end, we can optimize the derivation by hand to finally obtain an attack in 7 steps (a time-consuming but rewarding process).

Next, we verify the attack by modeling the malicious website as a ProVerif process that uses the attacker API:

```
let TwitterAttack(twitterLoginUri:Uri,eveAppUri:App,
                  eveId:Id,evePwd:Secret) =
  (* Alice browses to Eve's website *)
  out (admin,getServerRequest(eveAppUri));
  in (result,(=getServerRequest(eveAppUri),
        (u:Uri,req:HttpRequest,hs:Params,corr:bitstring)));
  (* Eve redirects Alice to login as Eve@Twitter *)
  out(admin,sendServerResponse(eveAppUri,(u,
        httpOk(twitterLoginForm(twitterLoginUri,eveId,evePwd)),
        nullCookiePair(),corr))).
```

If ProVerif can find the attack again using just this attacker process, disabling all other attackers (by setting attacker mode to passive), then we say that the attack is concrete.

Finally, we attempt to execute the attack on a real website. We rewrite the process above as a PHP script and, indeed, we find that a login CSRF attack can be mounted on the Twitter login page. This attack was known to exist, but as we show in the following section, it can be used to build new login CSRF attacks on Twitter clients.

V. ANALYZING OAUTH 2.0 USING PROVERIF

In this section, we build a model of OAuth on top of the WebSpi library and analyze its security properties.

A. OAuth 2.0 model

Our model consists of an unbounded number of users and servers. Each user is willing to browse to any website (whether trusted or malicious) but only sends secret data to trusted sites. Each server may host one or more of the applications described below.

Login: As shown in Section IV, this application consists of a server process LoginApp and a corresponding user-agent process LoginUserAgent that together model form-based login for websites. In our model, both OAuth authorization servers and their client websites host login applications.

Data Server: An application that models resource servers. It includes a server process DataServerApp that offers an API with two functions: getData retrieves all the data for a particular user, and storeData stores new data for a user. We treat getId as a special case of getData where the caller is only interested in the user's identity. Users logged in locally on the resource server (through its LoginApp) may access their data through a browser, and their behaviour is modeled by a user-agent process DataServerUserAgent. OAuth clients may

Model	Lines	Verification Time
WebSpi Library	463	
Login Application	122	5s
Login with JavaScript Password Hash	124	5s
+ Data Server Application	131	41s
+ OAuth User-Agent Flow	180	1h12m
+ OAuth Web Server Flow	52	2h56m
Total (including attacks)	1245	

Table II
PROTOCOL MODELS VERIFIED WITH PROVERIF

remotely access data on behalf of their social login users, by presenting an access token.

OAuth Authorization (User−Agent Flow): A three-party social web application that models the user-agent flow of the OAuth protocol. The process OAuthImplicitServerApp models authorization servers, and the process OAuthUserAgent models resource owners. These processes closely follow the protocol flow described in Section III. The process OAuthImplicitClientApp models clients that offer social login; it offers a social login form for resource owners to click on to initiate social sign-on. When sign-on is completed, it provides the resource owner with additional forms to get and store data from the resource server. These additional data actions are not explicitly covered by the OAuth protocol, but are a natural consequence of its use.

OAuth Authorization (Web Server Flow): A three-party social web application that models the web server flow of the OAuth protocol, as described in Section III. The process OAuthExplicitClientApp models clients and OAuthExplicitServerApp models authorization servers.

We elide details of the ProVerif code for these applications, except to note that they are built on top of the library processes HttpClient and HttpServer, much like the login application, and implement message exchanges as described in the protocol. Each process includes Assume and Expect statements that track the security events of the protocol. For example, the OAuthUserAgent process assumes the predicate SocialLogin(RO,b,sid,C,AS,RS) before sending the social login form to the client; after login is completed it expects the predicate SocialLoginDone(RO,b,sid,C,u,AS,RS). We then encode the security goals of Section III as clauses defining such predicates. The full script is available online [15].

B. Results of the ProVerif analysis

We analyze the security of different configurations of our OAuth model using ProVerif. Table II summarizes our positive verification results. Each line lists a part of the model, the number of lines of ProVerif code, and the time taken to verify them. The most general model for which we were able to obtain positive results makes the following assumptions: network attackers, malicious resource owners and clients, untrusted websites and JavaScript are enabled; both OAuth explicit and implicit mode are enabled; but no HTTP redirectors are allowed on honest servers; no login or data CSRF attacks exist on honest apps; each client has exactly one authorization server; and every authorization server is honest. Under these conditions, ProVerif is unable to find any attacks, even considering an unbounded number of sessions. This should not be interpreted as a definitive proof of security, since we model only a subset of OAuth configurations and our attacker model is not complete.

Under other attacker configurations, ProVerif finds protocol traces that violate the security goals (attacks). Table III summarizes the attacks found by ProVerif. In each case, we were able to extract attacker processes (as we did for the login application of Section IV). In Appendix B we provide processes for some of these attacks, the full listings can be found online. These formal attacks led to our discovery of concrete, previously unknown attacks involving Facebook, Twitter, Yahoo, IMDB, Bitly and several other popular web sites. Table IV in Appendix B summarizes our website attacks. The rest of this section describes and discusses these attacks.

C. Social CSRF attacks against OAuth 2.0

We identify four conditions under which OAuth 2.0 deployments are vulnerable to Social CSRF attacks.

Automatic Login CSRF. As described in Section II, the social login form of the OAuth client CitySearch is not protected against CSRF. Hence, a malicious website can cause the resource owner to log in to CitySearch (through Facebook) even if she did not wish to. We call this an automatic login CSRF, and it is widespread among OAuth client websites.

ProVerif finds this attack on both OAuth flows, as a violation of the SocialLoginAccepted policy on our model. It demonstrates a trace where it is possible for the OAuth client process to execute the event SocialLoginAccepted even though this resource owner never previously executed SocialLogin.

Social Sharing CSRF. Again as described in Section II, the review forms on CitySearch are not protected against regular CSRF attacks. Hence, a malicious website can post arbitrary reviews in the name of an honest resource owner, and this form will be cross-posted on Facebook.

ProVerif finds this attack on both OAuth flows, as a violation of the APIRequest policy at the client. It demonstrates a trace where a malicious website causes the client process to send a storeData API request, even though the resource owner never asked for any data to be stored.

Social Login CSRF on stateless clients. OAuth clients like IMDB do not implement the optional *state* parameter, hence they are subject to a Social login CSRF attack, as predicted by the OAuth specification. ProVerif finds a trace that violates SocialLoginDone.

Social Login CSRF through AS Login CSRF. If an OAuth authorization server, such as Twitter, is vulnerable to a login CSRF, this vulnerability can be translated to a Social Login

CSRF attack on any of its clients. ProVerif again finds a violation of SocialLoginDone.

D. Token redirection attacks against OAuth 2.0

We identify three conditions under which OAuth 2.0 deployments are vulnerable to access token and authorization code redirection, leading to serious attacks such as unauthorized login on the client and resource theft on the resource server. All of these attacks involve manipulations of the redirect_uri, and rely on the existence of an open redirector on the client.

Unauthorized Login by Authentication Code Redirection. Many OAuth clients, such as WordPress and Yahoo, host HTTP redirectors on their website. Suppose the redirector's URI is an allowable redirect_uri for the client at the authorization server. Moreover, assume that the authorization server does not check that the same redirect_uri is used in both the CodeRequest and in the APITokenRequest. Then a malicious website can use this URI to login to the client impersonating an honest resource owner. ProVerif finds this attack as a violation of the SocialLoginAccepted policy.

Consider as an example WordPress, which allows single sign-on with Facebook. Suppose the attacker has a blog on WordPress. For a fee, WordPress allows its members to forward all traffic sent to their blog to an external website. Hence, the attacker can set up an HTTP redirector at `eve.wordpress.com`.

Now, when a resource owner tries to log in to `someblog.wordpress.com` using Facebook, she is redirected to Facebook and then back with the authorization code to `someblog.wordpress.com/connect/?code=C`. However, Facebook is willing to redirect this code to any URL of the form `*.wordpress.com/*` because the domain registered for the WordPress app at Facebook is just `wordpress.com`. This enables an attack where a malicious website can obtain the authorization code of an honest resource owner by redirecting him to Facebook with the redirection uri `eve.wordpress.com`. Using this authorization code, the attacker can then log in to WordPress (on his own browser), thus breaking the primary authentication goal of social sign-on. We note that this attack is not prevented by using a state parameter at the client.

Resource Theft by Access Token Redirection. If an OAuth authorization server is willing to enter a user-agent flow with a client that has an HTTP redirector, then an attack similar to the previous one becomes possible, except that the malicious website is able to obtain the access token instead of the authorization code. It can then use this access token to directly access the resource server APIs to steal an honest user's resources. ProVerif finds this attack as a violation of the APIResponse policy.

For example, we found such an attack on Yahoo, since it offers an HTTP redirector as part of its search functionality. A malicious website can read the Facebook profile of any user who has in the past used social login on Yahoo. It is interesting to note that even though Yahoo itself never engages in the user-agent flow, Facebook is still willing to enter into a user-agent flow with a website pretending to be Yahoo, which leads to this attack.

Cross Social-Network Request Forgery. Suppose an OAuth client supports social login with multiple social networks, but it uses the same login endpoint for all networks. This is the case on many websites that use the JanRain or GigYa libraries to manage their social login. Then if one of the authorization servers is malicious, it can steal an honest resource owner's authorization code, access token, and resources at any of the other authorization servers, by confusing the OAuth client about which social network the user is logging in with. ProVerif finds the attack as a violation of the APITokenResponse policy.

E. Constructing concrete website attacks

Going from the formal counterexamples of ProVerif in Table III to the concrete website attacks of Table IV involved several steps. First we analysed the ProVerif traces to extract the short attacker processes of Appendix B, as illustrated in Section IV for the login application. Then we collected normal web traces using the TamperData extension for Firefox. By running a script on these traces, we collected client and authorization server login URIs, CSRF vulnerable forms, and client application identifiers. Using this data, we wrote website attackers in a combination of PHP and JavaScript and examined an arbitrary selection of OAuth 2.0 clients and authorization servers. We focused on websites on which we quickly found vulnerabilities. Other websites may also be vulnerable to these or related attacks.

F. Discussion

Some of the attacks described here were known (or predicted) in theory, but their existence in real websites were usually unknown before we reported them. We have notified Yahoo, Facebook and other websites mentioned in this paper, which have already adopted some of our suggested fixes.

Our attacks rely on weaknesses in OAuth clients or authorization servers, and we find that these *do* exist in practice. It is worth discussing why this may be the case.

CSRF attacks on websites are widespread and seem to be difficult to eradicate. We found a login CSRF attack on the front page of Twitter, a highly popular website, and it seems this vulnerability has been known for some time, and was not considered serious, except that it may now be used as a login CSRF attack on any Twitter client. Our analysis finds such flaws, and proposes a general rule-of-thumb: that *any* website action that leads to a social network action should be protected from CSRF.

Open redirectors in client websites are another known problem, although most of the focus on them is to prevent

Configuration	Time	Policy Violated	Attacks Found	Steps	Attack Process
Login over HTTP	12s	LoginAuthorized	Password Sniffing	1324	8 lines
Login form without CSRF protection	11s	ValidSession	Login CSRF	3568	12 lines
Data Server form update without CSRF protection	43	DataStoreAuthorized	Form CSRF	2360	11 lines
OAuth client login form without CSRF protection	4m	SocialLoginAccepted	Automatic Login CSRF	2879	11 lines
OAuth client data form without CSRF protection	13m	APIRequest	Social Sharing CSRF	11342	21 lines
OAuth auth server login form without CSRF protection	12m	SocialLoginAccepted	Social Login CSRF	13804	28 lines
OAuth implicit client without State	16m	SocialLoginDone	Social Login CSRF	25834	37 lines
OAuth implicit client with token redirector	20m	APIResponse	Resource Theft	23101	30 lines
OAuth explicit client with code redirector	23m	SocialLoginDone	Unauthorized Login	12452	34 lines
OAuth explicit client with multiple auth servers	17m	APITokenResponse	Cross Social-Network Request Forgery	19845	31 lines

The first three configurations correspond to normal website attacks and their effect on website security goals. The rest of the table shows OAuth attacks discovered by ProVerif. For each configuration, we name the security policy violation found by ProVerif, the number of steps in the ProVerif derivation, and the size of our attacker process.

Table III
FORMAL ATTACKS FOUND USING PROVERIF

phishing. Our attacks rely more generally on any redirector that may forward an OAuth token to a malicious website. We found three areas of concern. Search engines like Yahoo use redirection URLs for pages that they index. URL shortening services like Bitly necessarily offer a redirection service. Web hosting services such as WordPress offer potentially malicious clients access to their namespace. When integrating such websites with social networks, it becomes imperative to carefully delineate the part of the namespace that will be used for social login and to ensure there are no redirectors allowed in this namespace.

The incorrect treatment of redirection URIs at authorization servers enables many of our attacks. Contrarily to the OAuth 2.0 specification recommendations, Facebook does not require the registration of the full client redirection URI, because some clients may also use an OAuth flow where the redirection URI cannot be verified. Finding a way to rectify this problem while still supporting such clients is the subject of ongoing discussions.

Finally, a word of comparison between OAuth 2.0 and its competitors. OAuth 2.0 lacks request and response authentication, which leads to several of the issues found in this paper. Still, correct implementations of OAuth 2.0 do not suffer from these attacks. OAuth 1.0 featured both request and response authentication, but it was deemed too difficult to implement for widespread adoption (moreover, it was still vulnerable to session fixation attacks.) OpenID 2.0 features response authentication but not request authentication, which prevents some of the attacks found in this paper but not attacks like OpenID Realm Phishing [32].

VI. Related Work

Formal models of web browsing. Yoshihama *et al.* [34] present a browser security model that relies on information flow labels to enforce fine-grained access control, focusing on mashups. They describe the browser by means of a big-step operational semantics that models the evaluation of client-side scripts. The model includes multiple browser windows, the DOM, cookies and high-level HTTP requests. Some of the attacks we presented cannot be observed in that model. For example, CSRF attacks are prevented by construction. By contrast, since our goal is to analyze protocols and detect potential flaws, our browser model makes it possible to observe any sequence of events that can be triggered by a combination of web users, client side scripts and server-provided pages, including those leading to security violations.

Motivated by [34], Bohannon and Pierce [12] formalize the core of a web browser as an executable, small-step reactive semantics. The model gives a rather precise description of what happens within a browser, including DOM tags, user actions to navigate windows, and a core scripting language. Our formalization instead abstracts away from browser implementation details and focuses on web pages, client-side scripts and user behaviour. Both [34] and [12] focus on the *web script* security problem, that is how to preserve security for pages composed by scripts from different sources. The model does not encompass features such as HTML forms, redirection and `https` which are important in our case to describe more general security goals for web applications.

Akhawe *et al.* [5] propose a general model of web security, which consists of a discussion of important web concepts (browsers, servers and the network), a web threat model (with users and web, network and gadget attackers), and of two general web security goals: preserving existing applications invariants and preserving session integrity. They implement a subset of this general model in the Alloy protocol verifier [24]. Alloy lets user specify protocols in a declarative object-modeling syntax, and then verify bounded instances of such protocols by translation to a SAT solver. This formal subset of the web model is used on five different case studies, leading to the re-discovery of two known vulnerability and the discovery of three novel vulnerabilities.

Website	Role(s)	Preexisting Vulnerabilities			New Social CSRF Attacks			New Token Redirection Attacks		
		Login CSRF	Form CSRF	Token Redirector	Login CSRF	Automatic Login	Sharing CSRF	Resource Theft	Unauthorized Login	Cross Social-Network Request Forgery
Twitter	AS, RS	Yes								Yes
Facebook	AS, RS						Yes	Yes		Yes
Yahoo	Client			Yes				Yes	Yes	
WordPress	Client	Yes			Yes	Yes		Yes	Yes	
CitySearch	Client	Yes		Yes	Yes	Yes	Yes			
IndiaTimes	Client	Yes		Yes	Yes	Yes	Yes			
Bitly	Client				Yes	Yes		Yes		
IMDB	Client	Yes			Yes	Yes				
Posterous	Client				Yes			Yes		
Shopbot	Client	Yes			Yes	Yes				
JanRain	Client lib									Yes
GigYa	Client lib									Yes

The first section summarizes attacks on authorization servers, the second on OAuth clients, and the third on OAuth client libraries.
This is a representative selection of attacks found between June 2011 and February 2012. Most of these websites have since been fixed.

Table IV
CONCRETE OAUTH WEBSITE ATTACKS DERIVED FROM PROVERIF TRACES

Our work was most inspired by [5], with notable differences. We directly express our formal model in the variant of the applied pi-calculus, a formalism ideally suited to describe security protocols in an operational way, that is focusing on a high-level view of the actions performed by the various components of a web application. This approach reflects as closely as possible the intuition of the human designer (or analyzer) of the protocol, and helps us in the systematic reconstruction of attacks from formal traces. This language is also understood by the ProVerif protocol analysis tool, that is able to verify protocol instances of arbitrary size, as opposed to the bounded verification performed in Alloy.

Unbounded verification becomes important for flexible protocols such as OAuth 2.0, that even in the simplest case involve five heterogeneous principals and eight HTTP exchanges. In general, one may even construct OAuth configurations with a chain of authorization servers, say signing-on to a website with a Yahoo account, and signing-on to Yahoo with Facebook. For such extensible protocols, it becomes difficult to find a precise bound on the protocol model that would suffice to discover potential attacks.

Formal analysis of social sign-on. Early single sign-on protocols were often formally analyzed [28], [23], [8], but were not deployed widely enough to expose the kinds of website attacks (such as CSRF) discussed in this paper.

Pai *et al.* [27] adopt a Knowledge Flow Analysis approach [31] to formalize the specification of OAuth 2.0 in predicate logics, a formalism similar to our Datalog-like policies. They directly translate and analyze their logical specification in Alloy, rediscovering a previously known protocol flaw. Our ProVerif models are more operational, closer to a web programmer's intuition. Our analysis with respect to different classes of attackers is able to discover a larger number of potential protocol abuses.

Chari *et al.* [16] analyze the *authorization code* mode of OAuth 2.0 in the Universal Composability Security Framework [13]. They model a slightly revised version of the protocol that assumes that both client and servers use TLS and mandates some additional checks. This model is proven secure by a simulation argument, and is refined into an HTTPS-based implementation.

Miculan and Urban [26] model the Facebook Connect protocol for single sign-on using the HLPSL specification language and AVISPA. Due to the lack of a specification of the protocol, which is offered as a service by Facebook, they infer a model of Facebook Connect in HLPSL by observing the messages effectively exchanged during valid protocol runs. Using AVISPA, they identify a replay attack and a masquerade attack for which they propose and verify a fix.

VII. CONCLUSIONS

We present a security analysis of the OAuth 2.0 protocol, using ProVerif, extended with the WebSpi library that formalizes web users, applications and attackers. Our analysis establishes both positive and negative security results, and the design of our library makes it easy to translate formal counterexamples into concrete attacks on websites. The effectiveness of the approach is validated by the discovery of several vulnerabilities in leading websites that use the OAuth 2.0 protocol. Expert human reviewers would have been able to find these attacks on a case-by-case basis. Our contribution is to make this discovery systematic, and partially automated. The models presented here do not cover some other common attacks, such as XSS, SQL Injection, and DNS rebinding. In future work, we plan to extend WebSpi in order to capture also these attacks, and verify more web security mechanisms and protocols.

Acknowledgments. Bhargavan is supported by the ERC Starting Grant CRYSP. Maffeis is supported by EPSRC grant EP/I004246/1. Part of this work was done while Bansal and Maffeis were visiting the Microsoft-INRIA Joint Centre and INRIA Paris-Rocquencourt.

References

[1] AVISPA. http://http://avispa-project.org/.

[2] M. Abadi and B.T. Loo. Towards a declarative language and system for secure networking. In *Proceedings of the 3rd USENIX international workshop on Networking meets databases*, page 2. USENIX Association, 2007.

[3] Martín Abadi and Cédric Fournet. Private authentication. *Theoretical Computer Science*, 322(3):427–476, 2004.

[4] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computing*, 148(1):1–70, 1999.

[5] D. Akhawe, A. Barth, P.E. Lam, J. Mitchell, and D. Song. Towards a formal foundation of web security. In *2010 23rd IEEE Computer Security Foundations Symposium*, pages 290–304. IEEE, 2010.

[6] D. Balfanz, B. de Medeiros, D. Recordon, J. Smarr, and A. Tom. OpenID OAuth Extension. Internet Draft, 2009.

[7] Karthikeyan Bhargavan, Cédric Fournet, Ricardo Corin, and Eugen Zălinescu. Verified cryptographic implementations for tls. *ACM Trans. Inf. Syst. Secur.*, 15(1):3:1–3:32, March 2012.

[8] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Nikhil Swamy. Verified implementations of the information card federated identity-management protocol. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, ASIACCS '08, pages 123–135, New York, NY, USA, 2008. ACM.

[9] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *CSFW*, pages 82–96, 2001.

[10] B. Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, 2009.

[11] Bruno Blanchet and Ben Smyth. *ProVerif: Automatic Cryptographic Protocol Verier, User Manual and Tutorial.* http://www.proverif.ens.fr/manual.pdf.

[12] A. Bohannon and B. C. Pierce. Featherweight Firefox. *Proceedings of the 2010 USENIX conference on Web*, 2010.

[13] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[14] S. Cantor, J. Kemp, R. Philpott, and E. Maler. Assertions and protocols for the oasis security assertion markup language (saml) v2.0, 2005.

[15] C.Bansal, K. Bhargavan, and S. Maffeis. WebSpi and web application models. http://prosecco.gforge.inria.fr/webspi/, 2011.

[16] S. Chari, C. S. Jutla, and A. Roy. Universally composable security analysis of oauth v2.0. *IACR Cryptology ePrint Archive*, 2011:526, 2011.

[17] F. Corella and K. Lewison. Security Analysis of Double Redirection Protocols. Pomcor Technical Report, 2011.

[18] A. Datta, J. Franklin, D. Garg, L. Jia, and D. Kaynar. On adversary models and compositional security. *IEEE Security & Privacy*, 9(3):26–32, 2011.

[19] J. DeTreville. Binder, a logic-based security language. In *IEEE Symposium on Security and Privacy*, pages 105–113, 2002.

[20] E. Hammer-Lahav. The OAuth 1.0 Protocol. IETF RFC 5849, 2010.

[21] C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.

[22] E. Hammer-Lahav, D. Recordon, and D. Hardt. The OAuth 2.0 Authorization Protocol. IETF Internet Draft, 2011.

[23] Steffen M. Hansen, Jakob Skriver, and Hanne Riis Nielson. Using static analysis to validate the saml single sign-on protocol. In *Proceedings of the 2005 workshop on Issues in the theory of security*, WITS '05, pages 27–40, New York, NY, USA, 2005. ACM.

[24] D. Jackson. Alloy: A logical modelling language. In *ZB*, page 1, 2003.

[25] T Lodderstedt, M Mcgloin, and P Hunt. OAuth 2.0 Threat Model and Security Considerations. IETF Internet Draft, 2011.

[26] M. Miculan and C. Urban. Formal analysis of Facebook Connect Single Sign-On authentication protocol. In *SofSem 2011, Proceedings of Student Research Forum*, pages 99–116, 2011.

[27] S. Pai, Y. Sharma, S. Kumar, R. M. Pai, and S. Singh. Formal Verification of OAuth 2.0 Using Alloy Framework. *2011 International Conference on Communication Systems and Network Technologies*, pages 655–659, June 2011.

[28] Birgit Pfitzmann and Michael Waidner. Analysis of liberty single-sign-on with enabled clients. *IEEE Internet Computing*, 7(6):38–44, 2003.

[29] D. Recordon and D. Reed. OpenID 2 . 0 : A Platform for User-Centric Identity Management. *Discovery*, pages 11–15, 2006.

[30] E. Rescorla. HTTP Over TLS. IETF RCF 2818, 2000.

[31] E. Torlak, M. van Dijk, B. Gassend, D. Jackson, and S. Devadas. Knowledge flow analysis for security protocols. MIT Technical Report MIT-CSAIL-TR-2005-066, 2006.

[32] OpenID Wiki. Phishing Brainstorm. http://wiki.openid.net/w/page/12995216/OpenID_Phishing_Brainstorm, 2009.

[33] T.Y.C. Woo and S.S. Lam. A semantic model for authentication protocols. In *IEEE Symposium on Security and Privacy*, pages 178–194, 1993.

[34] S. Yoshihama, T. Tateishi, N. Tabuchi, and T. Matsumoto. Information-Flow-Based Access Control for Web Browsers. *IEICE Transactions on Information and Systems*, E92-D(5):836–850, 2009.

APPENDIX

A. *ProVerif*

In this Appendix we describe the ProVerif specification language, and summarize its verification methodology, to the extent used in this paper. We refer the reader to [11], [9] for details on the ProVerif syntax and verification method.

The ProVerif specification language is a variant of the applied pi-calculus [3], an operational model of communicating concurrent processes with a flexible sublanguage for describing data structures and functional computation.

Messages. Basic types are channels, bitstrings or user-defined. Atomic messages, typically ranged over by $a, b, c, h, k, ...$ are tokens of basic types. Messages can be composed by pairing (M, N) or by applying n-ary data constructors and destructors $f(M_1, ..., M_n)$. Constructors and destructors are particularly useful for cryptography, as described below. Pattern matching is extensively used to parse messages. Messages may be sent on private or public channels or stored in tables.

M,N,X	::=	message
	a	channel,key,data,...
	x	variable
	(M,N)	pair
	f(M1,...,Mn)	constructor or destructor f applied to $M1, ..., Mn$
	=M	matching operator

Cryptography. ProVerif models *symbolic* cryptography: cryptographic algorithms are treated as perfect black-boxes whose properties are abstractly encoded using constructors and destructors. Consider authenticated encryption:

```
fun aenc(bitstring,symkey): bitstring.
reduc forall b:bitstring,k:symkey; adec(aenc(b,k),k) = b.
```

Given a bit-string b and a symmetric key k, the term aenc(b,k) stands for the bitstring obtained by encrypting b under k. The destructor adec, given an authenticated encryption and the original symmetric key, evaluates to the original bit-string b.

ProVerif constructors are collision-free (one-one) functions and are only reversible if equipped with a corresponding destructor. Hence, MACs and hashes are modeled as irreversible constructors, and asymmetric cryptography is modeled using public and private keys:

```
fun hash(bitstring) : bitstring.
fun pk(privkey):pubkey.
fun wrap(symkey,pubkey): bitstring.
reduc forall k:symkey,dk:privkey; unwrap(wrap(k,pk(dk)),dk) = k.
fun sign(bitstring,privkey): bitstring.
reduc forall b:bitstring,sk:privkey; verify(sign(b,sk),pk(sk)) = b.
```

These and other standard cryptographic operations are part of the ProVerif library. Users can define other primitives where necessary. Such primitives can be used for example to build detailed models of protocols like TLS [7].

Protocol Processes.. The syntax of the applied-pi processes used in ProVerif (reported below) is mostly standard from process algebra. Messages may be sent and received on channels, or stored and retrieved from tables (which are formally just private channels). Fresh messages (such as nonces) are generated using new. Pattern matching is used to parse messages in let, but also when receiving messages from channels or tables. Predicates p(M) are invoked in conditionals (boolean conditions M=N are a special case). Finally, processes can be run in parallel, and even replicated.

P,Q	::=	process
	out(a,M);P	send M on channel a
	in(a,X);P	receive message in X
	insert(t,M);P	insert M into table t
	get(t,X) in P	retrieve table entry in X
	new a;P	fresh name with scope P
	event e(M1,...,Mn);P	insert event in trace
	let X=M in P	pattern matching
	if p(M) then P else Q	conditional statement
	P|Q	run P and Q in parallel
	!P	run unbounded number of copies of P in parallel

Security Queries. The command event e(M1,...,Mn) inserts an *event* e(M1,...,Mn) in the trace of the process being executed. Such events form the basis of the verification model of ProVerif. A script in fact contains processes and *queries* of the form $\forall M_1, ...M_k. e(M_1, ...M_k) \Rightarrow \phi$. The tool tries to prove that whenever the event e is reachable, the formula ϕ is true (ϕ can contain conjunctions or disjunctions).

A common case is that of correspondence assertions [33], where $\phi = e'(M_1, ...M_k)$ and the goal is to show that if e is reachable then e' must have been reached beforehand. Correspondence queries naturally encode authentication goals, as noted in Section IV-B. Syntactic secrecy goals are encoded as reachability queries on the attacker's knowledge.

Verification. ProVerif translates applied-pi processes into Horn clauses in order to perform automatic verification. The main soundness theorem in [10] guarantees that if ProVerif says that a query is true for a given script, then it is in fact the case that the query is true on all traces of the applied-pi processes defined in the script in parallel with any other arbitrary attacker processes.

If a query is false, ProVerif produces a proof derivation that shows how an attacker may be able to trigger an event that violates the query. In some cases, ProVerif can even extract a step-by-step attack trace.

General cryptographic protocol verification is undecidable, and hence ProVerif cannot always terminate. ProVerif uses conservative abstractions that let it analyze protocol instances for an unbounded number of participants, sessions, and attackers, but may potentially report false positives. Hence, one needs to validate proof derivations and formal attack traces before accepting them as counterexamples.

B. Example ProVerif Attacks on OAuth 2.0 Websites

Automatic Login and Social Sharing CSRF (on CitySearch and Facebook).

```
let CitysearchFacebookAttack(csSocialLoginUri:Uri,
                    csReviewSubmitUri:Uri,eveAppUri:App) =
  (* Alice browses to Eve's website*)
  out (admin,getServerRequest(eveAppUri));
  in (result,(=getServerRequest(eveAppUri),
              (u:Uri,req1:HttpRequest,
               hs1:Params,corr1:bitstring)));
  (* Eve redirects Alice to automatically login at Citysearch *)
  out(admin,sendServerResponse(eveAppUri,
              (u,httpRedirect(csSocialLoginUri),
               nullCookiePair(),corr1)));
  out (admin,getServerRequest(eveAppUri));
  (* Alice browses again to Eve's website *)
  in (result,(=getServerRequest(eveAppUri),
              (=u,req2:HttpRequest,
               hs2:Params,corr2:bitstring)));
  (* Eve redirects Alice to post Eve's review at Citysearch & Facebook *)
  new myReview:bitstring;
  out(admin,sendServerResponse(eveAppUri,(u,
              httpOk(csReviewForm(csReviewSubmitUri,myReview)),
              nullCookiePair(),corr2))).
```

Social Login CSRF (on IMDB using Facebook).

```
let IMDBAttack(facebookLoginUri:Uri,facebookOAuthUri:Uri,
               imdbSocialLoginUri:Uri,
               eveAppUri:App,eveId:Id,evePwd:Secret) =

  (* Eve logs in to Facebook *)
  let C1 = httpRequestResponse(nullCookiePair(),
              facebookLoginUri,httpGet()) in
  out (admin,C1);
  in (result,(=C1,(sid:Cookie,sp:Principal,httpOk(form1))));
  let C2 = httpRequestResponse(sid,facebookLoginUri,
              httpPost(loginFormReply(form1,eveId,evePwd))) in
  out (admin,C2);
  in (result,(=C2,(=sid,=sp,httpOk(loginSuccess()))));

  (* Eve authorize IMDB as a Client for Eve@Facebook *)
  let C3 = httpRequestResponse(sid,facebookOAuthUri,httpGet()) in
  out (admin,C3);
  in (result,(=C3,(=sid,=sp,httpOk(form2))));
  let C4 = httpRequestResponse(sid,facebookOAuthUri,
              httpPost(oauthFormReply(form2))) in
  out (admin,C4);

  (* Eve intercepts her Authorization Code for IMDB *)
  let C5 = httpRequestResponse(nullCookiePair(),
              imdbSocialLoginUri,httpGet()) in
  out (admin,C5);
  in (result,(=C5,(sid':Cookie,sp':Principal,httpRedirect(fb))));
  let C6 = httpRequestResponse(sid,fb,httpGet()) in
  out (admin,C6);
  in (result,(=C6,(=sid,=sp,httpRedirect(im))));

  (* Alice browses to Eve's website *)
  let C7 = getServerRequest(eveAppUri) in
  out (admin,C7);
  in (result,(=C7,(u:Uri,req:HttpRequest,
                  hs:Params,corr:bitstring)));
  (* Eve redirects Alice to login to IMDB using Eve's Authorization Code *)
  let C8 = sendServerResponse(eveAppUri,
              (u,httpRedirect(im),
               nullCookiePair(),corr)) in
    out(admin,C8).
```

Resource Theft by Access Token Redirection (on Yahoo and Facebook).

```
let YahooFacebookAttack(facebookOAuthUri:Uri,
           facebookGraphAPI:Uri,eveAppUri:App,
           yahoo_app_id:Id, yahoo_eve_redirector:Uri) =
  (* Alice browses to Eve's website *)
  let C1 = getServerRequest(eveAppUri) in
  out (admin,C1);
  in (result,(=C1,(u1:Uri,req1:HttpRequest,
                   hs1:Params,corr1:bitstring)));
  (* Eve redirects Alice to Facebook's OAuth Server
     using redirect_uri=yahoo_eve_redirector *)
  new state:Cookie;
  let authUri = uri(ep(facebookOAuthUri),
                   oauthRequest(yahoo_app_id,state,
                                ep(yahoo_eve_redirector))) in
  let C2 = sendServerResponse(eveAppUri,
              (u1,httpRedirect(authUri),
               nullCookiePair(),corr1)) in
  out(admin,C2);
  (* Alice is redirected to yahoo_eve_redirector with
     her access token for Yahoo, which redirects her back to Eve *)
  let C3 = getServerRequest(eveAppUri) in
  out (admin,C3);
  in (result,(=C3,(u2:Uri,req2:HttpRequest,
                   hs2:Params,corr2:bitstring)));
  let oauthToken(=state,token) = params(u2) in
  (* Eve uses Alice's access token to steal her Facebook data *)
  let dataUri = uri(ep(facebookGraphAPI),oauthDataRequest(token)) in
  let C4 = httpRequestResponse(nullCookiePair(),dataUri,httpGet()) in
  out (admin,C4);
  in (result,(=C4,(sid:Cookie,sp:Principal,httpOk(data)))).
```

Unauthorized Social Login by Auth Code Redirection (on WordPress and Facebook).

```
let WordpressFacebookAttack(wpSocialLoginUri:Uri,
           eveAppUri:App, wp_app_id:Id,wp_eve_redirector:Uri) =
  (* Eve starts to "Login with Facebook" on Wordpress *)
  let C1 = httpRequestResponse(nullCookiePair(),
              wpSocialLoginUri,httpGet()) in
  out (admin,C1);
  (* Eve intercepts the authorization request to Facebook
     and modifies redirect_uri to wp_eve_redirector *)
  in (result,(=C1,(sid:Cookie,sp:Principal,httpRedirect(fb))));
  let oauthRequest(app_id,state,redirect_ep) = params(fb) in
  let newParams = oauthRequest(app_id,state,ep(wp_eve_redirector)) in
  let newUri = uri(ep(fb),newParams) in
  (* Alice browses to Eve's website *)
  let C2 = getServerRequest(eveAppUri) in
  out (admin,C2);
  in (result,(=C2,(u1:Uri,req1:HttpRequest,
                   hs1:Params,corr1:bitstring)));
  (* Eve redirects Alice to modified Facebook authrization URI *)
  let C3 = sendServerResponse(eveAppUri,
              (u1,httpRedirect(newUri),
               nullCookiePair(),corr1)) in
  out(admin,C2);
  (* Alice is redirected to wp_eve_redirector with
     her access code for Wordpress, which redirects her back to Eve *)
  let C4 = getServerRequest(eveAppUri) in
  out (admin,C4);
  in (result,(=C4,(u2:Uri,req2:HttpRequest,
                   hs2:Params,corr2:bitstring)));
  let oauthCode(=app_id,=state,as,code) = params(u2) in
  (* Eve logs into Wordpress using this code pretending to
     respond to the original authorization request *)
  let loginUri = uri(redirect_ep,oauthCode(app_id,state,as,code)) in
  let C5 = httpRequestResponse(nullCookiePair(),loginUri,httpGet()) in
  out (admin,C5).
```