Discovering Concrete Attacks on Website Authorization by Formal Analysis *

Submission to special issue for
Dagstuhl seminar on Web Application Security

Chetan Bansal[1], Karthikeyan Bhargavan[2],
Antoine Delignat-Lavaud[2], and Sergio Maffeis[3]

[1]BITS Pilani-Goa
[2]INRIA Paris-Rocquencourt
[3]Imperial College London

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 attacks on websites. To ease the task of writing formal models in our framework, we present a model extraction tool that automatically translates programs written in PHP and JavaScript subsets to the applied pi-calculus.

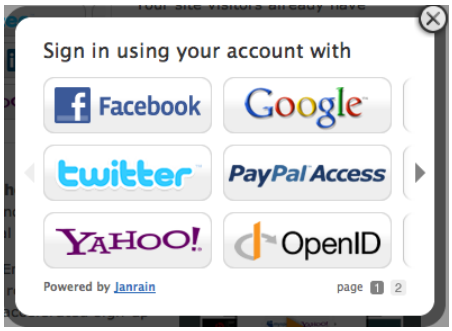
Our approach is validated by finding dozens of previously unknown vulnerabilities in popular websites such as Yahoo and WordPress, when they connect to social networks such as Twitter and Facebook.

*This paper is an extended and revised version of [12].

1. 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 [22] and other XML-based security protocols are primarily used for SOAP-based API access, for example, on Amazon and Microsoft Azure. OpenID [45] is used for light-weight user authentication, for example, on Google and PayPal. OAuth [27, 34] 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. A consensus seems to be emerging around the use of some variation or combination of the OpenID and OAuth protocols, and OAuth 2.0 [34] is currently the most widely supported protocol for API authorization, especially for REST, AJAX, and JSON-based websites. It is currently supported by Google, Facebook, Microsoft, and LinkedIn among others. OpenID Connect [41] 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.

The main novelty of OpenID and OAuth over traditional authentication and authorization protocols is that they do not require custom software: they are designed to be embedded in standard websites and executed by standard web browsers. All protocol messages are encoded as HTTP requests and responses and the protocol is mediated by the user's browser, so that the user may authorize or deny access to her data through a familiar interface. A typical implementation consists of server-side PHP or Python scripts running on each website and a client-side HTML and JavaScript component that communicates with these servers to execute the protocol.

Formal analysis against web attacks The security of web authorization protocols relies both on the design of the websites that deploy them and on browser-based mechanisms like cookies and JavaScript. Hence, in addition to the traditional *network attacker* who may hijack insecure HTTP connections, such protocols are also exposed to a new class of *web attackers* who may exploit website vulnerabilities such as Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), and open redirectors (OR) to partially compromise the client or server scripts running these protocols. Such web vulnerabilities are widely prevalent even on popular websites and have proved hard to eliminate [40]. Evaluat-

ing the robustness of new security mechanisms such as web authorization against web attackers requires a precise model of browsers, servers, and interactions between them.

Previous works on the formal analysis of single sign-on protocols [43, 44, 33, 16, 7], account for network attackers (e.g. as formalized by Dolev and Yao [26]) but do not model web attacks at the level of cookies and JavaScript. Web authorization protocols have also been subject to careful human analysis [37, 24], 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 [35], AVISPA [6] and ProVerif [17] have proved to be effective in the formal analysis of security protocols and to find network-based attacks. Can they also be used to automatically analyze web security mechanisms and find concrete web attacks?

Akhawe *et al.* [5] use Alloy to develop a general model of web components and use a SAT solver to find vulnerabilities in new security mechanisms including a Kerberos-based single sign-on solution. 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 extract models for various compositions of web security mechanisms and verify them against different attacker models, until she reaches a design that satisfies her specific security goals.

Our approach. We model various configurations of the OAuth 2.0 protocol in the applied pi-calculus [1, 2] and analyze them using ProVerif [17]. 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. In order to express realistic security goals for a web application, we show how to encode distributed authorization policies in the style of [25, 29, 28] in ProVerif. 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 reported these problems and helped fixing them.

We present a model extraction tool that generates applied pi-calculus models from website components written using carefully-chosen subsets of PHP and JavaScript. Model extraction frees the web security analyst from the more mundane (but error-prone) activity of encoding protocols in ProVerif syntax, allowing her to focus on the results of the analysis instead. As an example, we demonstrate an implementation of OAuth's user-agent flow that can be analyzed by our toolset; attacks found by ProVerif can then be mapped back to concrete attacks on our implementation. We advocate the use of our automated analysis framework when designing and prototyping the security-critical core of a new application. Once verified, this core can be easily extended into a complete website.

Contributions. The main contributions of this paper are the WebSpi library (Section 4), a formal analysis of OAuth 2.0 using WebSpi and ProVerif (Section 5), a description of new concrete website attacks found and confirmed by our formal analysis, and a

model extraction tool for PHP websites (Section 6). A shorter version of this paper was published in CSF 2012 [12]; a new technical contribution in this paper is the model extraction framework of Section 6. Full ProVerif scripts, including the WebSpi library, the OAuth 2.0 model, and formal attacks, are available online [11].

2. Social Sign-On and Social Sharing

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*.

2.1. Motivating Example: Authenticated Website Comments

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 as a WordPress, Facebook, or Twitter user (Figure 1-L).

When a 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 (Figure 1-R). `WordPress.com` would like access to Alice's basic information, in particular her name and email address, as proof of identity.

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, completing the *social sign-on* protocol. All subsequent actions that Alice performs at `WordPress.com`, such as commenting on a blog, are associated with her Facebook identity.

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 even 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.

2.2. 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.

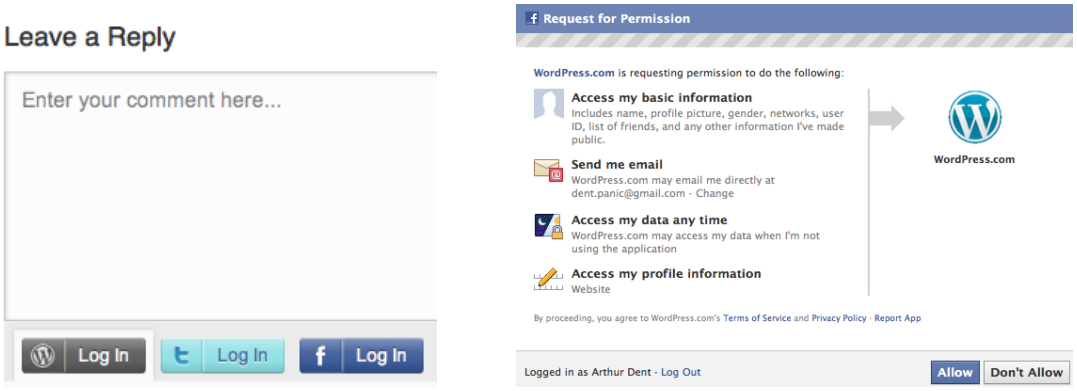


Figure 1: (L) *Log in with Facebook* on Wordpress; (R) Facebook requires authorization.

- 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 and authorization frameworks, and in order to achieve them the protocol relies on two secrets: Alice's password at Facebook and the access token issued by Facebook to WordPress.

What makes social sign-on more interesting than traditional authentication protocols is the need 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 users who can try to fool Facebook or WordPress by pretending to be Alice; and malicious websites who can try to fool Facebook or Alice by pretending to be WordPress. A *web attacker* may use any combination of these three attack vectors.

2.3. Web-based attacks

Network attacks are well understood, and can be mitigated by the systematic use of HTTPS [46], or more sophisticated cryptographic mechanisms. Many websites, such as Facebook (at the time of writing), do not even seek to protect against network attackers for normal browsing, allowing users to access their data 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 [40].

For example, various flavors 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* [13]: 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. A typical countermeasure for CSRF is to require every security-sensitive request to include a session-specific nonce that would be difficult for a malicious website to forge. This nonce can be embedded in the target URL or within a hidden form field. However, such mechanisms are still not widely deployed and CSRF attacks remain prevalent on the web, even on respected websites.

2.4. Social CSRF attacks

We now describe one of the new attacks we found in our formal analysis of OAuth in Section 5. This example shows how a CSRF attack on a 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 (Figure 2-L).

When she submits this form, the review is posted to CitySearch as a standard HTTP POST request (Figure 2-R); CitySearch subsequently reposts it on Alice's Facebook profile (after attaching its API access token) on the server side.

We found that the review form above is susceptible to a standard 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 into 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 an *automatic login* 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 into a new client.

Figure 2: (L) CitySearch review form; (R) Corresponding Post request.

Combining the two attacks, we built a demonstrative malicious website that, when visited by a user who has previously used Facebook login 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 (since the user signs in under her own name). Instead, the attacker uses CitySearch as a proxy to mount an indirect CSRF attack on Facebook, treating the API access token like a session cookie. We call this class of attacks *Social CSRF*.

2.5. 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.

In practice, 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 their 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 4), becomes a login CSRF vulnerability on all of its client websites.

2.6. 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 involves the exchange of at least nine HTTP(S) messages. It does not depend on the details of the underlying authorization protocol, but the other vulnerabilities in Section 5 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 web attacks such as CSRF. Conversely, a model that treats the browser and the user as one entity will miss CSRF attacks completely, since they rely on the browser performing actions not intended by the user. In Section 4, we present a web security library that includes a precise model of the browser as well as an attacker model that can be fine-tuned, enabling the discovery of new and interesting web attacks.

3. OAuth 2.0: Browser-based API Authorization

The goal of the OAuth 2.0 protocol [34] is to enable third party clients to obtain limited access, on behalf of a resource owner, to the API of a resource server. 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 2, Facebook is both the authorization server and resource server; we find that this is the most common configuration.

The first version of OAuth [27] was designed to unify previous 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 protocol specification defines several different *flows*, two of which directly apply to website applications. The protocol itself requires no direct use of cryptography, but 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.

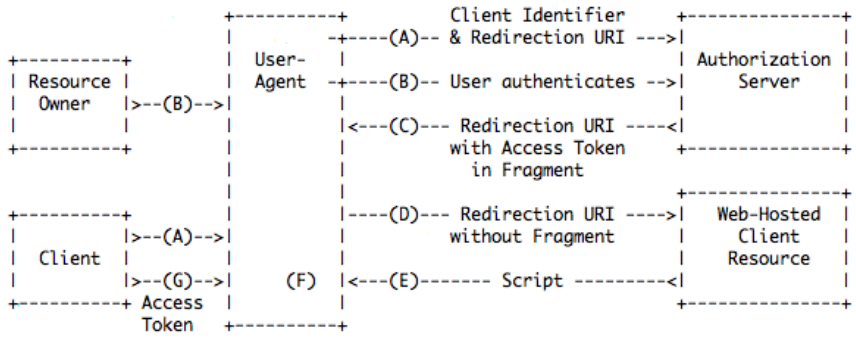


Figure 3: OAuth 2.0: User-Agent Flow (adapted from [34]).

3.1. 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 plugins.

The authorization flow diagram from the OAuth specification, is depicted in Figure 3. 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 at AS. Let the client be located at URL C and have an application identifier id at AS. The protocol steps of the user-agent flow are explained below based on the relevant security events:

1. SocialLogin(RO,b,sid1,C,AS,RS): RO using b starts a social sign-on session sid1 at C using AS for RS. For example, the user clicked a "Log in with..." link on the client web page.

2. TokenRequest(C,b,AS,id,perms): C instructs b to request AS a token for id with access rights perms. This is the redirection message (A) in the diagram above.

3. Login(RO,b,sid2,AS,u): RO using browser b starts a login session sid2 at AS with username u. This step is not necessary if RO was already logged in AS.

4. Authorize(RO,b,sid2,C,perms): AS looks up id and asks RO for authorization; RO using browser b in session sid2 at AS authorizes C with perms. Message (B) above is part of this step.

5. TokenResponse(AS,b,C,token): AS grants C a token for b. The token is sent via the redirection message (C) as a fragment identifier, which is not forwarded to RS in message (D). RS sends b a script in message (E), that b uses to retrieve the access token in step (F) and forward it to C with message (G).

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 on behalf of u the API request from C.

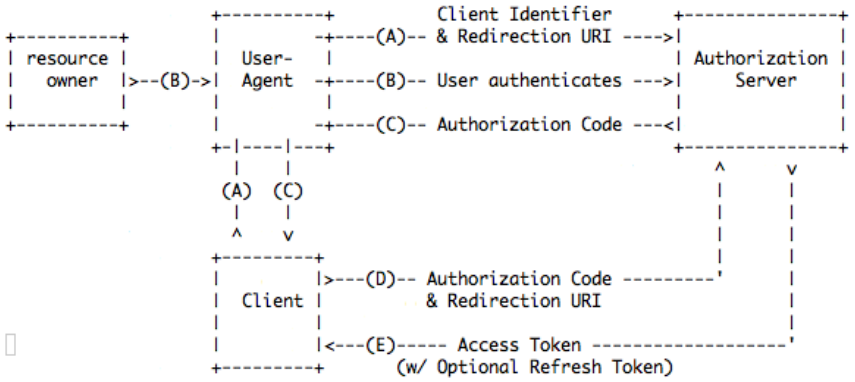


Figure 4: OAuth 2.0: Authorization Code Flow (adapted from [34]).

8. SocialLoginAccept(C,sid1,u,AS,RS): C accepts RO's social sign-on session sid1 as u at AS for RS.

9. SocialLoginDone(RO,b,sid2,C,u,AS,RS): RO is logged in to C in a browser session sid2 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. Several steps in this flow consist of (at least) one HTTP request-response exchange. 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.

As an example of the user-agent protocol flow, consider the social sign-on interaction between websites like Pinterest and Facebook; the TokenRequest(C,b,AS,id,perms) step is typically implemented as an HTTPS redirect from Pinterest 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 Pinterest, of the form: `https://pinterest.com/#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. In practice, these HTTP exchanges are implemented by a JavaScript SDK provided by Facebook and embedded into Pinterest, hence messages may have additional Facebook-specific parameters, but generally follow this pattern.

3.2. Authorization Code Flow

The Authorization Code flow, also called Explicit Grant flow or Web Server 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 a shared 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 in the Referer header, browser history, or by malicious third-party JavaScript running on the client.

The authorization flow diagram from the OAuth specification is depicted in Figure 4. Let the client at URL C have both an application identifier id and a secret sec pre-registered at AS.

1. SocialLogin(RO,b,sid1,C,AS,RS): RO using b starts a social sign-on session sid1 at C using AS for RS. For example, the user clicked a "Log in with..." link on the client web page.

2. CodeRequest(C,b,AS,id,perms): C instructs b to request AS a token for id with access rights perms. This is the redirection message (A) in the diagram above.

3. Login(RO,b,sid2,AS,u): RO using browser b starts a login session sid2 at AS with username u. This step is not necessary if RO was already logged in AS.

4. Authorize(RO,b,sid2,C,perms): AS looks up id and asks RO for authorization; RO using browser b in session sid2 at AS authorizes C with perms. Message (B) above is part of this step.

5. CodeResponse(AS,b,C,code): AS grants C a code for b. The code is sent to C via the redirection message (C).

6. APITokenRequest(C,AS,code,id,sec): with message (D), C makes an API request for an access token to AS with code, id, and sec.

7. APITokenResponse(AS,C,token): AS checks id and sec, verifies the code and returns a token to C with message (E).

Once the token is received by C in message (E), the Authorization Code flow continues with the steps 6-9 of the User-Agent flow described above. Note that also steps 1, 3 and 4 above are the same as in the User-Agent flow.

3.3. 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 the corresponding APITokenRequest must also 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 match, 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 protocol flows described in the specification. We leave these features for future work.

3.4. A Threat Model for OAuth 2.0

The OAuth specification [34] and a companion document describing its threat model [37] together provide an exhaustive list of potential threats to the protocol. We consider a subset of these threats in our formal analysis.

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 traffic; malicious clients, resource owners, and authorization servers; malicious websites that honest resource owners may browse to; and honest clients with specific web vulnerabilities, such as CSRF attacks, or redirectors that may forward

HTTP requests to malicious websites.

Other threats. We do not explicitly consider attacks on the browser or operating system of honest participants; instead, we treat such participants as *compromised*. This is equivalent to the worst case scenario, where an exploit lets the attacker completely take over the user machine. In this way, we err on the safe side. We assume that honest resource owners choose strong passwords and use secure web browsers. Attacks such as brute force password cracking, that become possible if these assumptions are released, are independent from the use of OAuth or other protocols. We focus on vulnerabilities in client websites, and we assume that honest authorization servers have no web vulnerabilities, otherwise all bets are off: there is little hope of achieving autorization goals if the authorization server is compromised.

3.5. Security Goals for OAuth 2.0

We describe the security goals for each participant by defining Datalog-like authorization policies [25] 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 RO (using browser b) in a session sid' with a client C has successfully completed the social sign-on with authorization server AS (and resource server RS) if it intended to sign into the client, if it agreed to authorize the client, and if the client and resource owner agree upon the user's social identity (u) for the current session (sid'):

```
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 says that a TokenResponse can only be sent to C if the resource owner has logged in and authorized the client.

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

Note that we do not require a TokenResponse to be only issued in response to a TokenRequest from the client: at this stage, the user-agent flow has not authenticated the client, and so cannot know whether the client intended to request a token.

The correponding policy for the authorization code flow is stronger:

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

```
  Says(C,TokenRequest(C,b,AS,id,perms,state,redirect_uri)),
  Says(RO,Authorize(RO,b,sid,C,perms)).
```

From the viewpoint of the resource server, every API call must be issued by an authorized client and accompanied by a token issued by 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)).
```

4. The WebSpi library and its usage

Various calculi, starting from the spi-calculus [3], 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 [1, 2], and verify them using ProVerif [17]. 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.

4.1. ProVerif

The ProVerif specification language is a variant of the applied pi-calculus, an operational model of communicating concurrent processes with a flexible sublanguage for describing data structures and functional computation. Below, we summarize the ProVerif specification language and its verification methodology, to the extent used in this paper. We refer the reader to [19, 17] for further details on ProVerif.

4.1.1. 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(M1,...,Mn). Constructors and destructors are particularly useful for cryptography, as described below. Messages may be sent on private or public channels or stored in tables. The matching operator is used by the processes for pattern matching and receiving messages described in Section 4.1.3.

Informally, pattern f(x,=g(y)) matches message f(M,g(N)) and(re-)binds variable x to term M if N equals the current value of variable y.

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

4.1.2. Cryptography

ProVerif models *symbolic* cryptography: cryptographic algorithms are treated as perfect black-boxes whose properties are abstractly encoded using constructors (introduced by the fun keyword) and destructors (introduced by the reduc keyword). As an example, 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 when necessary. Such primitives can be used for example to build detailed models of protocols like TLS [14].

4.1.3. Protocol Processes

The syntax of ProVerif's specification language, given below, is mostly standard compared to other process algebras. Messages may be sent and received on channels, or stored and retrieved from tables (which themselves are internally encoded by 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

4.1.4. 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

$$\text{query M1:T1,...,Mn:Tn; E(M1,...Mn)} \Longrightarrow \phi.$$

When the tool encounters such a query, it 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 [52], where an event e is split into two sub-events $begin_e$ and end_e. The goal is to show that if end_e is reachable then $begin_e$ must have been reached beforehand. The corresponding ProVerif query is

$$\text{query M1:T1,...,Mn:Tn; End(e,M1,...Mn)} \Longrightarrow \text{Begin(e,M1,...,Mn)}.$$

Correspondence queries naturally encode authentication goals, as noted in Section 4.1.5. Syntactic secrecy goals are encoded as reachability queries on the attacker's knowledge.

4.1.5. Distributed security policies

Since their introduction in the context of the spi-calculus [29], 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, and should precede the actual code used by the user to send the message. The Assume statement has no effect on the operation it precedes: its purpose it just to reflect it 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. Note that in the clause, → can be interpreted as logical implication, whereas in the query ⟹ represents the proof obligation described in Section 4.1.4.

As we have described above, assumptions are normally associated with process code performing some specific operation. If such code belongs to the process representing a particular principal in the system, then it can be desirable to associate the logical facts being assumed to the principal in question. To this end, we encode a standard Says modality, inspired by Binder [25, 4]. This modality makes it possible to distinguish a fact e that is true in general, from a fact that is true according to a specific principal p, written Says(p,e). Two axioms, which we encode below in ProVerif, characterize this modality: 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 because it is ready to say anything.

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

Distributed authorization policies have already been used for typed-based verification of protocols in the applied pi-calculus [28]. To the best of our knowledge, we are the first to embed them on top of ProVerif predicates, thus driving the verification of realistic case studies.

4.1.6. Verification

ProVerif translates applied-pi processes into Horn clauses in order to perform automatic verification. The main soundness theorem in [18] 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, hence ProVerif does not always terminate. ProVerif uses conservative abstractions that let it analyze protocol instances for an unbounded number of participants, sessions, and attackers, but may report false positives. Hence, one needs to validate proof derivations and formal attack traces before accepting them as counterexamples on a model.

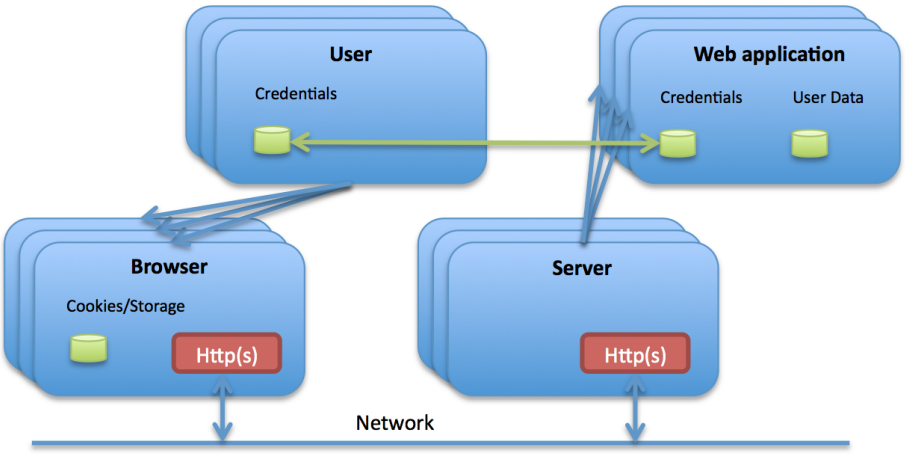


Figure 5: WebSpi architectural diagram.

4.2. WebSpi

WebSpi models consist of users who surf the Internet on web browsers, in order to interact with web applications that are hosted by web servers. A user can use multiple browsers, and a server can host multiple web applications. Figure 5 gives a schematic representation of the model.

4.2.1. Principals, HTTP Protocol, Browsers, and Servers

The agents in our model are called *principals*. They can play the role of users or owners of web applications. For example, the same principal may own two different web applications and be the user of a third one. This feature may help discovering flaws in applications that involve interaction between servers owned by different principals. Users hold credentials to authenticate with respect to a specific web application (identified by a host domain and subdomain) in the table credentials. Web applications hold private and public keys to implement TLS secure connections in the table serverIdentities.

```
table credentials(Host,Principal,Id,Credentials).
table serverIdentities(Host,Principal,pubkey,privkey,XdrFlag).
```

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.

Browsers and servers communicate using the HTTP(S) protocol over a channel net. Our model of HTTP(S) messages is fairly detailed. For example, the message

```
httpReq(uri(protocol,domainHost(subdomain,domain),path,params),
```

```
    headers(referrer,cookies,notajax()),
    httpGet())
```

denotes a regular (non-AJAX) `HTTP GET` request with cookies. Cookies can be associated to the root "" of a domain or to a specific path, and support `HttpOnly` and `Secure` attributes. The standardized, application-independent behavior of browsers and servers, which includes TLS connection and cookie handling, is modeled by the processes HttpClient and HttpServer. These processes incorporate a simple model of anonymous HTTP(S) connections: each request to an HTTPS 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.

HTTP Server. The process HttpServer simply handles the TLS connections on behalf of a web application (Section 4.2.2), and is reported below.

```
let HttpServer() =
  in(net,(b:Browser,o:Origin,m:bitstring));
  get serverIdentities(=originhost(o),pr,pk_P,sk_P,xdrp) in
  let (k:symkey,httpReq(u,hs,req)) = reqdec(o,m,sk_P) in
  if origin(u) = o then
  let corr = mkCorrelator(k) in
  out(httpServerRequest,(u,hs,req,corr));
  in(httpServerResponse,(=u,resp:HttpResponse,cookieOut:CookieSet,=corr));
  out(net,(o,b,respenc(o,httpResp(resp,cookieOut,xdrp),k))).
```

The HTTP(S) server accepts requests over channel net, from browser b, on behalf of the web application hosted from the destination origin o. If TLS is used, it decodes the message m to obtain the session key k and the actual request httpReq(u,hs,req). If the connection was plain HTTP, reqdec becomes the identity function on m.

Next, the server forwards the request to the corresponding web application on channel httpServerRequest, waiting for a response to encrypt (if necessary) and forward on the net back to b. Since the server may act on behalf of several applications, the token corr is used to correlate the right request/response pairs between the HTTP server process and the various web application processes. Server-side sessions, are maintained by individual web applications and are not visible at this stage.

HTTP Client. The process HttpClient is the core of the WebSpi library. It represent the behavior of a specific browser b, handling raw network requests and TLS encryption, offering to user processes an API for surfing the web, and modeling page origins and persistent state (cookies and local storage).

The browser API sends messages to the local channel internalRequest to a sub-process which handles network messages and TLS in a complementary fashion to the HTTP server process. This module also handles cookies, AJAX and cross-domain requests. The code below shows the final stages of handling an HTTP response.

```
(let httpOk(dataIn) = resp in
   if p = aboutBlank() then
```

```
      (let p1 = mkPage(u) in
       insert pageOrigin(p1,o,h,u);
       out (newPage(b),(p1,u,dataIn)))
    else
      (if aj = ajax() then
         (get pageOrigin(=p,oldorig,oldh,olduri) in
          if (foo = xdr() || oldorig = o) then
          out (ajaxResponse(b),(p,u,dataIn)))))
|(let httpRedirect(redir) = resp in
    out (internalRequest(b),(redir,httpGet(),ref,p,notajax())))
)
```

An `OK` response originated by clicking on a link, submitting a form, or editing the address bar to URI u, leads to the creation of a new page p1, a corresponding update in the page origin table, and a message on the newPage channel of browser b which corresponds to loading the HTML payload dataIn in the new page. An `OK` response originated by an AJAX call to the same origin oldorig, or to a server accepting cross-domain requests (flag xdr()) instead leaves the old page in place and creates a message on the ajaxResponse channel of b that makes the AJAX payload dataIn available to the page. A `Redirection` response, which is not allowed for an AJAX request, is handled by issuing a fresh request on the internalRequest channel.

The browser API includes commands browserRequest modelling a navigation request originating from the address bar or bookmarks (with an empty `Referer` header), pageClick modelling a navigation request or form submission from within a page (either caused by the user or by JavaScript), ajaxRequest to initiate an XMLHttpRequest and setCookieStorage to update the non-`HttpOnly` cookies from JavaScript.

Nondeterministically, the browser may reset the cookies and local storage for a given origin (modeling the user clearing the cookie cache) or release cookies/storage associated to a given origin to any page loaded from the same origin.[1] The second case is modeled by the code below.

```
(get pageOrigin(p,o,h,ref) in
    get cookies(=b,=originhost(o),=slash(),cs) in
    get cookies(=b,=originhost(o),=h,ch) in
    get storage(=b,=o,s) in
    out (getCookieStorage(b),(p,cookiePair(protocolCookie(domcookie(cs),o),
                                           protocolCookie(domcookie(ch),o)),s)))
```

Cookies are indexed by origin and by path (where slash() stands for the empty path).

Predefined processes. For convenience, the WebSpi library contains a number of predefined processes that implement useful patterns of behaviour on the web. To cite some representative examples, the HttpRedirector process provides a simple, predefined redi-

[1] In the case of cookies, we give access also to pages loaded from a sub-origin (we check for =originhost(o)), since a page from `sub.example.com` can upcast its origin to `example.com`, and read the corresponding cookies.

rection service. The process WebSurfer models a generic user principal who is willing to browse the web to any public URL. Process UntrustedApp implements a simple web application that can be compromised by the attacker, and is useful to test the robustness of a protocol with respect to compromise of third parties.

4.2.2. 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, which interfaces with HttpServer to handle network communications;
- a client-side (JavaScript-like) process representing the web page, which interfaces with the browsing API exposed by HttpClient;
- a user process representing the behavior of a human who uses a browser to access the web application, clicking on links, filling forms and so on.

In some simple cases, the second and third process may be combined. In addition to messaging over HTTP(S), client and server-side processes may perform for example cryptographic or database operations.

4.2.3. Example: login application

As an example, we show how to model and analyze the core functionality of a typical website login application, which is a building block of the OAuth models considered in this paper.

Login user process. Assume that user p, who controls browser b, has requested the login page of the web application at h. The process LoginUSerAgent below waits for the response on the newPage(b) channel, which b uses to forward the parsed HTTP response. We model a careful user, that checks that the protocol used is HTTPS and that the page came from the correct host h, avoiding phishing attacks. (We consider careless users to be *compromised*, that is under the control of the attacker.) If the page contains a form, the user retrieves her credentials and enters them in the loginFormReply which is embedded in a POST message to be forwarded to the browser on channel pageClick. If the credentials were the right ones for the user, the server will reply a second time (triggering a new instance of LoginUSerAgent) with a loginSuccess page, and the user is participating in a valid session.

```
let LoginUserAgent(b:Browser) =
 let p = principal(b) in
 in(newPage(b),(p1:Page, u:Uri, d:bitstring));
 let (=https(), h:Host, loginPath(app)) = (protocol(u),host(u),path(u)) in
 ((
   let loginForm = formTag(d) in
   get credentials(=h,=p,uId,pwd) in
```

```
    if assume(Login(p,b,h,uId)) then
      out(pageClick(b),(p1,u,httpPost(loginFormReply(d,uId,pwd))))
  )|(
    if loginSuccess() = d then
      event Expect(ValidSession(p,b,h))
  )).
```

Both the statements assume(Login(p,b,h,uId)) and Expect(ValidSession(p,b,h)) are part of the security specification. The former states that the user p intends to log in as the user uId at the web application h, using the browser b. The latter indicates that at this stage the user demands to be logged in to the right website.

Login server process. We model the server-side login application as follows:

```
let LoginApp(h:Host,app:Path) =
 in(httpServerRequest,(u:Uri,hs:Headers,req:HttpRequest,corr:bitstring));
 let uri(=https(),=h,=loginPath(app),q) = u in
 let c = getCookie(hs) in
 let cookiePair(sid,ch) = c in
 let httpPost(loginFormReply(d,uId,pwd)) = req in
 get credentials(=h,p,=uId,=pwd) in
 get serverIdentities(=h,sp,xx,yy,zz) in
 event Expect(LoginAuthorized(sp,h,uId,sid));
 insert serverSessions(h,sid,loggedIn(uId));
 out(httpServerResponse,(u,httpOk(loginSuccess()),c,corr))
```

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 the login application 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,sid,uId); if any check fails, it silently rejects the request; otherwise it returns a page loginSuccess().

Before registering the session, the policy event Expect(LoginAuthorized(sp,h,uId,sid)) is triggered to signal the user uId has logged in with the session sid on h.

Security goals. The security goals for the login protocol are written as policies that define when the predicates LoginAuthorized and ValidSession hold. For clarity, we write policies like in Datalog (in ProVerif syntax, they are written right-to-left as clauses that extend the fact predicate).

From the viewpoint of the server, the login protocol has a simple authentication goal: a user should be logged in only if the user intended to log in to that server in the first place. We can intuitively write this goal as a policy for LoginAuthorized:

```
LoginAuthorized(sp,h,uId,sid) :
      Server(sp,h),
```

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	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 $e2$
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
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 1: A command API for the active web attacker

```
    User(up,uId,h),
    Says(up,Login(up,b,h,uId))
```

where up and sp are respectively the user and server principals. The last line of the policy accounts for the possibility that the user may have been compromised (that is, her password may be known to that adversary.)

From the viewpoint of the browser, login has successfully completed if the server has logged the user in and both the browser and the server agree on the identity of the user:

```
ValidSession(up,b,h) :
    Server(sp,h),
    User(up,uId,h),
    Login(up,b,h,uId),
    Says(sp,LoginAuthorized(sp,h,uId,sid)).
```

These policies can be read as the standard *correspondence assertions* [52] typically used to specify authentication properties in cryptographic protocols. However, using predicates, we can also encode more intuitive authorization policies that would generally be difficult to encode as ProVerif queries.

4.2.4. 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 otherwise 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:

```
let AttackerProxy() =
```

```
in (pub,x:Command);
if commandEnabled(x) = true then
out(admin,x);
in (result,(=x,y:bitstring));
out(pub,y).
```

The full list of commands that the attacker can send is listed in Table 1. This API is designed to be *operational*: each command corresponds to a concrete attack that can be mounted on a real web interaction. It includes three categories of attacker capabilities:

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 flags 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.

Malicious websites. 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.

Malicious JavaScript. The last two commands (enabled by UntrustedJavaScript) provide the attacker with an API to access features from the browsers' HttpClient, to simulate some of the capabilities of JavaScript code downloaded from untrusted websites.

4.3. From ProVerif results to concrete web attacks

When analyzing a model in ProVerif, the tool will either prove the model correct (with respect to its security goals), or fail to verify the model, or not terminate.

4.3.1. Dealing with non-termination

When the verification of a script does not terminate (at least not within a reasonable amount of time) it is often the case that there is too much non-determinism in the model, and that messages of arbitrary complexity keep getting generated. To limit the number of cases when the analysis of a web application model built on top of WebSpi does not terminate, we have followed two approaches. First, we have taken care to use extensively constructors, destructors, and types, to give the most precise shape possible to messages, in particular abstracting away details of the HTML and HTTP formats. For example, in the login server process of Section 4.2.3, the HTTP message containing the HTML page

returned after successfully logging in is simply modelled by the term httpOk(loginSuccess()) (plus the additional headers transparently added by the browser's HTTP server module). Second, in order to fine tune the amount of non-determinism possible in each model, as described in Section 4.2.4, the security analyst may fine-tune the attacker model by setting various flags and then run ProVerif on different configurations. In this way, even though combining all of the possible attackers at once could lead to non-termination, it is possible to find attacks on subsets of attacker threats.

4.3.2. Guarantees and limitations for positive results

If verification succeeds, the correctness theorem for ProVerif [18] guarantees that no attacks exist, at least among the class of attacks considered in the model. However, the value of this positive result is limited because WebSpi, although expressive and extensible, is not a complete model of the web. For example, WebSpi does not cover many browser and server features, such as the treatment of advanced HTTP headers such as `Origin` and `ETag`. Hence, our main focus is on discovering attacks, which can be validated in the real world, rather than on providing positive guarantees, which may be violated in practice due to omissions from the model.

4.3.3. From verification failure to attack

When verification fails, ProVerif either produces an attack trace, or else it provides a proof derivation that points to a potential attack. Such proof derivations can be very long, since they list all attempted attacks, ending in the successful one, and contain details of how the attacker constructed each message.

In order to simplify the task of extracting an attack trace from such derivations, we have designed our attacker model so that all attacker actions in traces and derivations appear as concrete commands and responses on the admin and result channels. A simple filtering step therefore can drastically reduce the length of a derivation by excluding non-attacker actions. Parsing such a derivation from the end (which is the step that is guaranteed to have triggered the verification failure), the security analyst can manually optimize the derivation and obtain a succinct attacker process.

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.

The correspondence between concrete attacker processes and runnable PHP scripts is straightforward. The final step, in order to validate the attack against a real website, is to instantiate the constants in the model with actual web addresses and user credentials. Automated approaches for finding such data, based on recording network traces, have been considered for example in [30, 36].

4.3.4. Example: Login application

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 2 and 4.

If we only enable network attackers, malicious users, and malicious servers, ProVerif proves the model secure. Suppose we relax the LoginUserAgent process so that naive users may also agree to 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, the verification fails and ProVerif produces a proof derivation, but not an attack trace. The derivation has 3568 steps. However, after selecting onlythe messages on the admin and result channels, we end up with a derivation of 89 steps. Most of the steps towards the beginning of this derivation are redundant commands 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.

Next, we encode 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))).
```

Since the model, together with this attacker process but disabling all other attackers (by setting attacker mode to passive), still fails to verify, then we know that this attack is concrete.

By translating the process above in PHP, and handpicking appropriate constants (internet address, user name, etc.) 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.

5. Analyzing OAuth 2.0 using ProVerif

Following the pattern of the Login example in the previous section, we build models for the two main protocol flows of OAuth using the WebSpi library, and analyze their security properties. While we do not report here the applied-pi calculus code listings of our model, the explanations below highlight in this font references to identifiers used in the code, to facilitate browsing it online [11].

5.1. OAuth 2.0 model

We consider an unbounded number of users and servers. Each user is willing to browse 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 4, 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 behavior is modeled by a user-agent process DataServerUserAgent. OAuth clients may remotely access data on behalf of their social login users, by presenting an access token.

OAuth Authorization (UserAgent 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 3. 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 (Authorization Code Flow): A three-party social web application that models the authorization code flow of the OAuth protocol, as described in Section 3. 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 3 as clauses defining such predicates. The full script is available online [11].

5.2. Results of the ProVerif analysis

We analyze the security of different configurations of our OAuth model using ProVerif. Table 2 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 verification results consists of OAuth in

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 Authorization Code Flow	52	2h56m
Total (including attacks)	1245	

Table 2: Protocol Models Verified with ProVerif

both explicit and implicit grant modes, exposed to network attackers, malicious resource owners and clients, untrusted websites and JavaScript. We assume that each client has exactly one authorization server, every authorization server is honest, all exchanges are over HTTPS, and no web vulnerabilities exists on honest servers, that is, clients and authorization servers do not host HTTP redirectors and protect all their forms against login and data CSRF attacks. Under these conditions, ProVerif is unable to find any attacks, even considering an unbounded number of sessions. These are encouraging results, although they should not be interpreted as definitive proof of security, since our web model is not complete.

As we varied some of the assumptions described above, ProVerif found protocol traces violating the security goals. Table 4 summarizes the configurations for which we found attacks in ProVerif. In each case, we were able to extract attacker processes (as we did for the login application of Section 4). In Appendix A we provide processes for some of these attacks.

These formal attacks led to our discovery of concrete, previously unknown attacks involving Facebook, Twitter, Yahoo, IMDB, Bitly and several other popular websites. We focused on websites on which we quickly found vulnerabilities. Other websites may also be vulnerable to these or related attacks. Table 5 in Appendix A summarizes our website attacks. The rest of this section describes and discusses these attacks.

Going from the formal counterexamples of ProVerif in Table 4 to the concrete website attacks of Table 5 involved several steps. First we analysed the ProVerif traces to extract the short attacker processes of Appendix A, as illustrated in Section 4 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. Many of these steps can be automated; for example, AUTHSCAN [30] shows how to heuristically extract concrete attacks from ProVerif counterexamples produced by WebSpi models.

5.3. Social CSRF attacks against OAuth 2.0

To begin with, we note that if any of the exchanges of our OAuth model is allowed over unencrypted HTTP, ProVerif immediately finds a network-based attack that breaks our security goals, either by stealing the password or access token, or by injecting a forged request or response. In the rest of this paper, we will ignore such well-known attack vectors and look for more interesting attacks. We first consider Social CSRF attacks.

To better understand these attacks, recall the typical OAuth protocol flow involves four forms where the user interacts with the protocol: the login form at the authorization server, the social login form ("Login with Facebook") at the client, the authorization form at the authorization server, and (potentially) a data entry (comment) form at the client. When the user submits (clicks on) any of these forms, an HTTP GET or POST request is sent to a form action URI, along with the parameters encoded in the form. If, however, there is no CSRF protection at this action URI, e.g. a session-specific secret token in the form parameters, a malicious website may directly send a user to the action URI without the user ever agreeing to submit the form, leading to various kinds of CSRF attacks that may break the user's authentication or authorization goals.

We identify several conditions under which OAuth 2.0 deployments are vulnerable to Social CSRF attacks. In our models, such attacks appear in two forms: either the network attacker injects an HTTP response which redirects the user to a carefully crafted URI, or a malicious website entices the user into clicking on a URL or submit button.

Automatic Login CSRF. Suppose the social login form has no CSRF protection. As described in Section 2, this is true for many OAuth clients, such as CitySearch. Then, a malicious website can effectively bypass the SocialLogin step of the protocol and directly redirect the user's browser to the TokenRequest or CodeRequest step. If the authorization server then silently authorizes the token release, say because the user is logged in and has previously authorized this client, then the protocol can proceed to completion without any interaction with the user. 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 clients (see Table 5).

In our model, ProVerif finds this attack on both OAuth flows, as a violation of the SocialLoginAccept policy on our model. It demonstrates a trace where it is possible for the OAuth client process to execute the event SocialLoginAccept even though this resource owner never previously executed SocialLogin. The trace also violate the user's SocialLoginDone policy. This is an interesting example of how a seemingly innocuous vulnerability in the client website can lead to the failure of the expected security goals of an honest user, who may simply have wished to remain anonymous.

Social Login CSRF through AS Login CSRF. Suppose the login form on the authorization server is not protected against login CSRF. This is the case for Twitter, as described in Section 4.3. In this case, a malicious website can bypass the Login step of the protocol and directly pass his own credentials to the login form's action URI. Hence, the user's browser will be silently logged into the attacker's Twitter account. Furthermore, if the user ever clicks on "Login with Twitter" on any client website, he will be logged into

that website also as the attacker, resulting in a *social login CSRF* attack. All future actions by the user (such as commenting on a blog) will be credited to the attacker's identity. We confirmed this attack on Twitter client websites such as WordPress.

In our model, ProVerif finds a violation of the user's SocialLoginDone policy; the browser completes social sign-on for one user but the client website has accepted the login (SocialLoginAccept) for a different user in the same session. This is an example where a flaw in a single authorization server can be amplified to affect all its clients.

Social Login CSRF on stateless clients. Suppose an OAuth client does not implement the optional *state* parameter. Then it is subject to a second kind of social login CSRF attack, as predicted by the OAuth specification. A malicious user (in concert with a malicious website) can inject his own TokenResponse in place of an honest resource owner's TokenResponse in step 5 of the user-agent flow, by redirecting the user to the corresponding URI. (In the authorization code flow, the malicious user injects her CodeResponse instead.) When the client receives this response, it has no way of knowing that it was issued for a different user in response to a different TokenRequest. Many OAuth clients, such as IMDB, do not implement the state parameter and are vulnerable to this attack.

ProVerif again finds a trace that violates SocialLoginDone; the browser and client have inconsistent views on the identity of the logged-in user.

Social Sharing CSRF. Once social sign-on is complete, the client has an access token that it can use to read, and sometimes write, user data on the resource server. Suppose a form that the client uses to accept user data is not protected against CSRF; then this vulnerability may be amplified to a CSRF attack on the resource server. For example, as described in Section 2, the review forms on CitySearch are not protected against regular CSRF attacks, and data entered in these forms is automatically cross-posted on Facebook. Hence, a malicious website can post arbitrary reviews in the name of an honest resource owner, and this form will be stored on Facebook, even though the resource owner never intended to fill in the form, and despite Facebook's own careful CSRF protections.

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.

5.4. Token stealing 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 user data theft on the resource server. All of these attacks rely on the existence of attacker-controlled URIs on the client website, and on the authorization server's willingness to issue authorization codes and access tokens to these URIs. The policy that an authorization server uses to match a given redirect_uri to a registered client is implementation-specific. For example, Facebook and Live identify clients by a domain, and are willing to issue tokens to any URI on that domain. This gives clients

maximum flexibility in terms of the pages where they can embed social login. Conversely, it substantially increases the attack surface. As we shall see, any untrusted page within this URI range can lead to serious attacks.

Unauthorized Login by Authentication Code Redirection. Suppose a client website hosts an HTTP Redirector that forwards all GET requests to an attacker's website. Then any browser that visits this URI will be forwarded to the attacker's webpage, and the browser will automatically also attach any parameters in the original URI, such as the authorization code, as a parameter to the redirected URI. We found such redirectors on multiple websites, including on WordPress as described below.

Notably, if the HTTP redirector is a valid redirect_uri for the authorization server, a malicious website can perform a triple-redirection attack to steal the authorization code: (1) it redirects the user to the authorization server to request a code (CodeRequest) but with the redirect_uri set to the HTTP redirector; (2) the authorization server redirects the browser to the redirect_uri with the authorization code as parameter (CodeResponse); (3) the redirector sends the browser to the attacker's website with the authorization code as parameter. Once it has obtained the authorization code, the website can impersonate the resource owner at the client website by using social login again but using its own browser. This time, when the client sends a CodeRequest, the malicious browser does not contact the authorization server; instead it returns the stolen authorization code in a CodeResponse. When the client subsequently verifies this code (using APITokenRequest) it will be given the identity of the honest resource owner, not the attacker, completely breaking the authentication goal of social sign-on.

In our model, ProVerif finds this attack as a violation of the SocialLoginAccept policy; the client completes social login for a user even though the user never executed Login with this browser; the browser in fact belongs to the attacker.

To see an example of the attack flow, consider the Facebook client WordPress. 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`. 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 client at Facebook is just `wordpress.com`. So, to execute our attack, a malicious website redirects the honest resource owner to Facebook with the redirection URI `eve.wordpress.com`, and the authorization code will be redirected back to the website. We note that this attack is not prevented by using a state parameter at the client, since the real client never sees the authorization code.

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 one above becomes possible, except that the malicious website is able to directly obtain the access token instead of the authorization code, again using a triple redirection attack. It can then use this access token to access the resource server APIs to steal an honest resource owner's data.

ProVerif finds this attack as a violation of the APIResponse policy; since the access token has been stolen, the resource owner can no longer reliably authenticate that it is only releasing user data to the authorized client.

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 (it only uses authorization codes), Facebook is still willing to enter into a user-agent flow with a website that pretends to be Yahoo, which leads to this attack.

Code and Token Theft by Hosted User Content. A simpler variation of the above authentication code and access token stealing attacks occurs on client websites that host user-controlled content on URIs that can be valid redirect_uris at the authorization server. For example, Dropbox allows user content at `dl.dropbox.com`, and its registered domain at Facebook is just `dropbox.com`. Hence, any user can upload an HTML file to `dl.dropbox.com` and by using the URI for this page as redirect_uri steal the Facebook access token for any other Dropbox user.

A special case of this attack is a Cross Site Scripting on the client website, whereby an attacker can inject JavaScript into a trusted page. Such a script can steal the access token for the current user from a variety of sources, including by starting a new OAuth user-agent flow with the authorization server.

ProVerif finds these attacks as violations of the APIResponse policy, when we enable UntrustedJavaScript or UntrustedWebsite on the client.

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. So far, we have assumed that all authorization servers are honest, but in this case, if one of the authorization servers is malicious, it can steal an honest resource owner's authorization code and access token at any of the other authorization servers, by confusing the OAuth client about which social network the user is logging in with.

For example, the JanRain website itself supports login with a number of OAuth providers, including Live, Facebook, LinkedIn, and SalesForce, but uses the same domain `login.janrain.com` to host all its redirect_uris. If any of these providers wanted to steal (say) a JanRain user's Facebook access token (or authorization code), it could redirect the user to Facebook's authorization server but with its own redirect_uri on JanRain. When JanRain receives the token (or code) response at this URI, it would assume that the token came from the malicious authorization provider and send back the token (or code) with any subsequent APIRequest (or TokenRequest) to the malicious provider.

In our model, if we enable multiple, potentially malicious, authorization servers, ProVerif finds the above attack as a violation of the APITokenResponse policy on the authorization code flow, and as a violation of APIResponse policy on the user-agent flow.

5.5. Discussion

Many of the attacks described in this Section were known (or predicted) in theory, but their existence in real websites was usually unknown before we reported them. We notified all the websites mentioned in this paper, and most have since been fixed.

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 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 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.

More generally, websites that integrate OAuth 2.0 should use separate subdomains for their security-critical pages that may have access to authorization codes and access tokens. For example, Yahoo now uses `login.yahoo.com` as a dedicated sub-domain for login-related activities. Pages on this domain can be carefully vetted to be free of web vulnerabilities, even if it may be hard to fully trust the much larger `yahoo.com` domain.

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, possibly in order to support a greater variety of clients, but also because modern browsers only enforce client-side protections at the Origin level. Finding a way to protect tokens from malicious pages in the same domain remains an open problem and the subject of ongoing research.

5.6. Beyond OAuth

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 [32]. 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 [51]. Despite its shortcomings, OAuth 2.0 seems to be gaining traction, and upcoming single sign-on protocols such as OpenID Connect [41] are

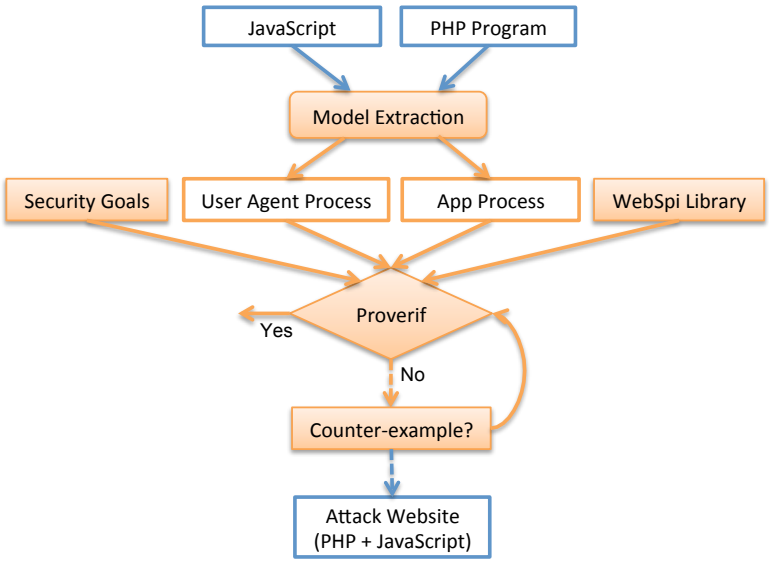


Figure 6: Model extraction and verification framework

being built on top of it. Formal analyses of such composite protocols are an interesting topic for future work.

6. Generating WebSpi Models from Concrete Websites

We acknowledge the difficulty for most web programmers to write WebSpi models of their website manually. To address this issue, we propose a framework for extracting user-agent and server-app processes directly from the security-sensitive part of carefully-written websites. Our model extraction framework, depicted in Figure 6, consists of three components:

- a subset of PHP equipped with a standard web library and its ProVerif counterpart;
- a subset of JavaScript equipped with a library for cryptography and secure communications and its ProVerif counterpart;
- automatic translations from these PHP and JavaScript subsets to the applied pi-calculus.

The generated processes may then be composed with the WebSpi library and automatically verified against the web application security goals. To support our translation, we extended the WebSpi model with a more realistic treatment of JavaScript that allowed multiple processes to share the same heap.

We focus on demonstrating the effectiveness of our translations rather than their soundness. At their core, they follow Milner's famous "functions as processes" encod-

ing of the lambda calculus into the pi calculus [39]. Translations similar to ours have previously been defined (and proved sound) for F# [15] and Java [9].

6.1. Syntax of Target PHP Subset

The syntax of the PHP subset that we translate to ProVerif is given below. Roughly speaking, a program written in that subset looks like a binary tree of `if` statements, whose leaves are either `echo`, `die` or `redirect` statements (similar to how WebSpi's application processes can return httpOk, httpError or httpRedirect).

⟨*program*⟩ ::= '`<?`'
('`require`' @string'`;`')* ⟨*statement*⟩

⟨*statement*⟩ ::= ε
| '`if(`' ⟨*if_expr*⟩ '`)`' ⟨*statement*⟩
('`else`' ⟨*statement*⟩)?
| '`{`' (⟨*expr*⟩ '`;`')* ⟨*statement*⟩ '`}`'
| '`echo sprintf(`' @string ('`,`' ⟨*expr*⟩)* '`);`'
| '`die(`' @string '`);`'
| '`redirect(`' ⟨*expr*⟩ '`)`'

⟨*if_expr*⟩ ::= ⟨*if_condition*⟩
| '`isset(`' ⟨*parameter_list*⟩ '`)`'
| '`get_table(`' (⟨*qvar*⟩ '`,`')* '`)`'
| '`sscanf(`' @variable '`,`' @string
('`,`' @variable)* '`)==`' @number

⟨*if_condition*⟩ ::= '`!`' ⟨*if_condition*⟩
| ⟨*if_condition*⟩ '`&&`' ⟨*if_condition*⟩
| ⟨*if_condition*⟩ '`||`' ⟨*if_condition*⟩
| ⟨*expr*⟩ '`===`' ⟨*expr*⟩

⟨*parameter_list*⟩ ::=
| ('`$_GET[`' @string '`],`')*
| ('`$_POST[`' @string '`],`')*

⟨*qvar*⟩ ::= @variable | '`&`' @variable

⟨*expr*⟩ ::=
| ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩
| @variable '`=`' ⟨*expr*⟩
| ⟨*if_condition*⟩ '`?`' ⟨*expr*⟩ '`:`' ⟨*expr*⟩
| @label '`(`' (⟨*expr*⟩ '`,`')* '`)`'
| '`$_`' ('`GET`' | '`POST`') ['`]`' @string '`]`'
| @variable | @string | @number

⟨*op*⟩ ::= ['`+`' '`-`' '`*`' '`/`' '`%`' '`<<`' '`>>`' '`&`' '`|`' '`^`' '`.`']

There are four kinds of `if` statements: normal conditions, parameter checking with `isset`, database lookups with the library function `get_table` and template parsing with `sscanf`. There is no support for functions, objects, arrays (besides those containing input parameters) or any kind of loop; while very limited compared to normal PHP, this subset is still expressive enough to build meaningful applications, provided operations that require actual computation (such as cryptographic primitives) are treated as calls to functions defined either in PHP's standard library or in an included file.

To demonstrate its usefulness, we implemented an example login provider for OAuth's implicit mode in this subset. The source code of the authorization handler is given in Table 7.

6.2. Translating PHP into ProVerif

At a high level, we require each PHP script to handle a single query path, for instance, `login.php` is translated into the process LoginServerApp, with a path constructor loginPath (corresponding to queries to `/login.php`). Before any other operation, the host and path of the script must be matched against incoming requests. Thus, a server process starts

PHP Source	Translation
`echo sprintf(T, `$e_1,\ldots,e_n$`)`	fun dataConst_T(bitstring,...,bitstring):bitstring [data]. out(httpServerResponse, (url,httpOk(dataConst_T($[\![e_1]\!],\ldots,[\![e_n]\!]$)),cookie_jar, corr));
`die(M)`	out(httpServerResponse, (url,httpError(),cookie_jar, corr));
`redirect(`e`)`	out(httpServerResponse, (url, httpRedirect(parseUri($[\![e]\!]$))),nullCookie(), corr);
`if(isset($_GET[`e_1`],..,$_GET[`e_n`]))`	fun P_get_params(bitstring,..,bitstring):Params [data]. let P_get_params(get_e_1,get_e_n) = query_string in
`if(isset($_POST[`e_1`],..,$_POST[`e_n`]))`	fun P_post_params(bitstring,..,bitstring):Params [data]. let httpPost(P_post_params(post_e_1, ..., post_e_n)) = method in
`if(get_table(T, $`v_i`,.. &$`w_j`, ..)`$_{i,j}$`)`	get T(var_v_i, ..., =var_w_j, ...) in
`if(sscanf(`s`, T, &$`e_1`,..,&$`e_n`)==`n`)`	let dataConst_T($[\![e_1]\!],\ldots,[\![e_n]\!]$)=$[\![s]\!]$ in
`session_start()`	if protocol(url) = https() then let cookiePair(session_cookie,path_cookie) = cookie_jar in if secure(session_cookie) <> nullCookie() then [...]
`embed_script(`S`,DOM,event)`	let $[P,S]$UserAgent(b:Browser) = $[\![S]\!]_{JS}$ free script_S:bitstring. [...] script_S
`f(`e_1`, .., `e_n`)`	f($[\![e_1]\!],\ldots,[\![e_n]\!]$)
`$_GET['a'], $_POST['a']`	get_a, post_a
`$_SESSION['a']`	get serverSessions(=host, =session_cookie, sessionPair(=str_a, session_a)) in [...] session_a
`$x = `e	let var_x = $[\![e]\!]$ in
e `+` f`, `$e.f$`...`	add($[\![e]\!]$, $[\![f]\!]$), concat($[\![e]\!]$,$[\![f]\!]$) ...

Table 3: Overview of the translation from PHP to ProVerif

with the following preamble, which also introduces free names (headers, method, protocol, query_string, cookie_jar) required for the translation:

```
fun loginPath(Path):Path [data].
let LoginServerApp(host:Host, app:Path) =
  in(httpServerRequest, (url:Uri, headers:Headers, method:HttpRequest, corr:bitstring));
  let uri(protocol, =host, =loginPath(app), query_string) = url in
  let cookie_jar = getCookie(headers) in P.
```

Writing a script in this subset is very similar to writing a ProVerif process; the main elements of the translation are given in Table 3.

For error handling purposes, many operations such as reading a database, accessing parameters or parsing a template are performed within atomic `if` statements. Any missing `else` branch is implicitly treated as `else die("")`; and translated to an httpError().

Before using session variables, the session cookie must be verified with a call to `session_start`. To simulate the actual behavior of this function, the three checks in the translation of `session_start` have else branches that will create a session cookie (if missing) and redirect the user to the same page (if required, over HTTPS). This behavior is only faithful if cookies are enabled on the client.

A typical script will first verify that its required parameters (either $_GET or $_POST, or a combination of both) are present, perform access control (based on the user's session), perform some operations based on the input (such as looking up a database) and return either an HTML result, represented by a data constructor that depends on all the dynamic values embedded in the page, or an error message, or a redirection.

In most cases, the parameters of a script come from an HTML form or link. In theory, if instead of using data constructors (`sprintf`) to represent HTML output, we directly parsed the HTML output to extract all forms, their method, target action and field names, we could generate User Agent processes to model the submission of forms. However, for now, we submit forms in JavaScript, using accesses to the DOM `document.forms[i].field`, which we translate to reading from a user_input channel to construct the parameters of the form submission.

Constants are converted to symbolic names by hashing, to get consistent names between PHP and JavaScript. Similarly, the name of a data constructor depends on the hash of its template. We use `sscanf` to reconstruct serialized messages between PHP and JavaScript, and translate it to a pattern match on the data constructor for the hashed template. The template may follow a standard serialization format such as JSON.

We also use a library function `get_table("t", $x, ..., &$y)` to perform database queries. This function works exactly like the ProVerif construct get t(=x, ..., y): the variables that are not passed by reference are used to construct the `WHERE` clause of the SQL query (the column names are retrieved from the table schema), while the variables passed by reference are filled with the result of the query (if multiple rows are selected, the first one is used).

To illustrate the translation on a concrete example, we provide the main section of the ProVerif translation (excluding the preamble and declarations) of the authorization handler from Figure 7 in Figure 8.

```
<?
require "lib.php";
session_start();

// Check parameters
if(isset($_GET['response_type'],$_GET['client_id'],$_GET['redirect_uri'])) {
 // Check response type
 if($_GET['response_type'] == "token") {
  // Check client id
  if(get_table("clients", $_GET['client_id'], &$client_key)) {
   // Is user logged in?
   if($_SESSION['is_logged'] == "yes") {
    // Is the client authorized already?
    if(get_table("user_auth", $_SESSION['username'], $_GET['client_id'], &$token)) {
     redirect(sprintf("%s#token=%s", $_GET['redirect_uri'], $token));
    } else { // Must authorize client
     $auth_code = hmac($_SESSION['username'], $client_key);
     if(isset($_POST['auth_code'])) {
      if($_POST['auth_code'] == $client_key) {
       insert_table("user_auth", $_SESSION['username'], $_GET['client_id'], gen_token());
       redirect(my_url());
      } else die("Invalid authorization key");
     } else {
      echo sprintf('<!DOCTYPE html><html><head>%s</head><body>
<h1>Do you want to authorize the application %s?</h1>
<form action="%s"><input type="hidden" name="auth_code" value="%s" />
<input type="submit" value="Authorize" /></form>
<a href="/">Go back home</a></body></html>',
       embed_script("var k = document.forms[0].auth_code;
post(my_url(),'auth_code='+k);","documents.forms[0]","submit"),
       my_url(), $_GET['client_id'], $client_key);
     }
    }
   } else redirect(sprintf("/login.php?redirect=%s",my_url()));
  } else die("Invalid client ID");
 } else die("Invalid response type parameter.");
} else die("Missing token request parameters.");
```

Figure 7: OAuth authorization script in PHP

```
let AuthServerApp(host:Host, app:Path) =
(...)
let Auth_get_params(get_redirect_uri,get_client_id,get_response_type)=query_string in
 if (get_response_type) = (str_15919241) then
  get clients(=get_client_id,var_client_key) in
   get serverSessions(=host, =session_cookie, sessionPair(str_1035747747,val_1)) in
   if (val_1) = (str_45715) then
    get serverSessions(=host, =session_cookie, sessionPair(str_737338002,val_2)) in
    get user_auth(=val_2,=get_client_id,var_token) in
     out(httpServerResponse, (url,httpRedirect(
      parseUri(dataConst_898097875(get_redirect_uri,var_token))
     )),nullCookie(), corr))
    else
     get serverSessions(=host, =session_cookie, sessionPair(str_737338002,val_3)) in
     let var_auth_code = hmac(val_3,var_client_key) in
     let httpPost(Auth_params(post_auth_code)) = method in
      if (post_auth_code) = (var_client_key) then
       get serverSessions(=host, =session_cookie, sessionPair(str_737338002,val_4)) in
       out(httpServerResponse, (url,httpRedirect(url),nullCookie(), corr))
      else
       out(httpServerResponse, (url,httpError(),cookie_jar, corr));
     else
      out(httpServerResponse, (url,httpOk(
       dataConst_470293626(script_0,serializeUri(url),get_client_id,var_client_key)
      ),cookie_jar, corr))
  else
   out(httpServerResponse, (url,httpRedirect(
    parseUri(dataConst_425409740(serializeUri(url)))
   )),nullCookie(), corr))
```

Figure 8: ProVerif (partial) translation of the script in Figure 7

6.3. Translating Client-Side JavaScript

Our JavaScript subset, whose syntax is given below, is more expressive than its PHP counterpart, with support for functions, objects and arrays.

⟨*statement*⟩ ::= ⟨*expression*⟩
  | '`if(`' ⟨*expression*⟩ '`)`' ⟨*statement*⟩
    ('`else`' ⟨*statement*⟩)?
  | '`{`' (⟨*statement*⟩ '`;`')* '`}`'

⟨*expression*⟩ ::= ⟨*literal*⟩
  | ⟨*expression*⟩ ⟨*binop*⟩ ⟨*expression*⟩
  | ⟨*lhs_expression*⟩ '`(`' (⟨*expression*⟩ '`,`')* '`)`'
  | ⟨*lhs_expression*⟩ '`=`' ⟨*expression*⟩
  | ⟨*lhs_expression*⟩

⟨*lhs_expression*⟩ ::= @identifier
  | ⟨*lhs_expression*⟩ '`[`' @number'`]`'
  | ⟨*lhs_expression*⟩ '`.`' @identifier

⟨*literal*⟩ ::= ⟨*function*⟩
  | '`{`' (@identifier '`:`' ⟨*expression*⟩ '`,`')* '`}`'
  | '`[`' (⟨*expression*⟩ '`,`')* '`]`'
  | @number | @string | @boolean

⟨*function*⟩ ::=
    '`function(`' (@identifier '`,`')*'`){`'
    ('`var`' (@identifier '`=`' ⟨*expression*⟩'`,`')+)?
    (⟨*statement*⟩ '`;`')*
    ('`return`' ⟨*expression*⟩?)? '`}`'

⟨*binop*⟩ ::= ['`+`' '`-`' '`*`' '`/`' '`%`' '`<<`' '`>>`' '`&`' '`|`' '`^`'
    '`==`' '`!=`' '`>`' '`<`' '`<=`' '`>=`' '`||`' '`&&`']

Since JavaScript does not natively support features such as `sscanf` or encryption, we provide them through a library. We distinguish such library functions with a special prefix, to distinguish them from global variables the script may try to access.

Our translation from JavaScript to ProVerif reflects the shared memory model of the browser. A single heap table in the browser stores pairs of locations and values on an origin (rather than page) basis, to reflect the ability of same-origin pages to read each other's variables. Because JavaScript is asynchronous and event driven, we support the translation of functions and closures. We intend each embedded script to correspond to the handler for one event (e.g. the page loading, a form being submitted, a link being clicked). Thus, the `embed_script` library function accepts a script `S`, a target DOM element d and event name e, which is used to generate the concrete script: d`.addEventListener(`e`, function(){S})`.

To illustrate the translation, we give in Figure 9 the login form event submission handler and its user agent process translation. This script simply reads the username and password entered in the login form, computes a login secret based on the username, password and salt and sends the result along with the username as a POST query to the login script. If we wanted to include a CSRF token, it would be set in the data constructor of the login form and accessible to the user agent within the variable `d`.

6.4. Limitations

The main limitation of our approach is the requirement to use the very limited subset of PHP in order to allow automated translation. Thus, this approach is not viable for the analysis of large deployed websites. However, it is useful for designing new web applications. The initial prototype can use our restricted subsets to implement the core features and protocols, which in most cases are relatively simple. The functionality of the application is now modeled in PHP and JavaScript, and can be directly tested.

```
embed_script("
  var u = document.forms[0].username;
  var p = document.forms[0].password;
  _lib.post(document.location,
    _lib.sprintf("%s|%s", u, _lib.hmac(p, 'e0f3...'+u)))",
  "document.forms[0]", "submit");
```

```
let LoginUserAgent(h:Host,b:Browser) =
  in(newPage(b),(pg:Page,u:Uri,d:bitstring));
  let uri(=https(),=h,loginPath(app),=nullParams()) = u in
  new var_u:Memloc;
  get user_input(=u,=number_zero,=in_username,str_1) in
  insert heap(origin(u),var_u,mem_string(str_1));
  new var_v:Memloc;
  get user_input(=u,=number_zero,=in_password,str_2) in
  insert heap(origin(u),var_p,mem_string(str_2));
  get heap(origin(u),=var_u,mem_string(val_1)) in
  get heap(origin(u),=var_p,mem_string(val_2)) in
  out(pageClick(b),(p1,u,httpPost(
    dataConst_5656244(val_1,hmac(val_2,concat(str_780069777,val_1)))
  ))).
```

Figure 9: Login form handler and its translation

Moreover, thanks to the automated translation the security of the design can be analyzed in WebSpi against various attacker models. Once the core application has been tested and verified, it can be extended using all the features of PHP and JavaScript into a fully featured website.

Another, more technical limitation of the translation is that, since ProVerif uses only symbolic equality, a program such as `if(1+1===2) echo "a"; else echo "b";` cannot be faithfully translated. We work around this problem by ensuring compared values rely on a combination of input parameters, constants and symbolically safe operations (such as concatenation). Developers should be aware of the various issues related to parsing malleable formats, such as JSON objects, URLs or query parameters.

Finally, even though model extraction is automatic, it is still up to the programmer to specify his intended security goals and interpret the result of the verification.

7. Related Work

A number of other works present attacks on single sign-on and web authorization mechanisms like OAuth 2.0 [47, 50, 48]. These attacks are similar to the ones discovered in our work and provide further evidence for the need for a systematic formal security analysis of such mechanisms that accounts for the precise details of the browser and common web vulnerabilities. In this Section, we review previous work related to formal models of web browsing and formal analysis of web authorization protocols similar to OAuth 2.0.

7.1. Formal models of web browsing

Gross *et al.* [31] model the communication behavior of web browsers as automata and use these state machines to prove the security of a password-based authentication protocol by hand. Their model does not cover cookies or scripts and hence does not cover most of the website attacks discussed in this paper.

Yoshihama *et al.* [53] 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 [53], Bohannon and Pierce [20] 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 behavior. Both [53] and [20] 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 [35]. 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. 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.

More recently, Bai *et al.* [30] present AUTHSCAN, an end-to-end tool to recover (and verify) authentication protocol specifications from their implementations. AUTHSCAN is composed of three modules. The first module extracts a protocol model by testing against an existing implementation. This is the main focus of this work. We do not attempt to extract models form protocol traces, but instead we provide an automated translation when the (PHP) source code is available, and resort to manual model extraction when the source code is not available. The second module, parametric in an attacker model and a set of security properties, verifies the protocol model using either ProVerif, PAT or AVISPA. The authors mostly use ProVerif, with a strict subset of our WebSpi attacker model [12]. This is a testament to the usefulness of WebSpi as a general-purpose web-protocol analysis library. The third module aims to confirm attacks discovered by the formal analysis instantiating the attack with the real-world data (IP addresses, credentials) used for testing. We also reconstruct concrete attacks from ProVerif traces, but we leave it to future work to make this process fully automatic. Unfortunately at the time of submission the implementation of AUTHSCAN is still not publicly available[2], so we cannot compare more closely our attack reconstruction techniques.

7.2. Formal analysis of web authorization

Early single-sign-on protocols, such as Passport, Liberty, Shibboleth, and CardSpace were often formally analyzed [43, 44, 33, 16], but these analyses mainly covered their cryptographic design against standard network-based adversaries, and do not account for the website attacks (such as CSRF) discussed in this paper.

Pai *et al.* [42] adopt a Knowledge Flow Analysis approach [49] 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 the intuition of a web programmer. Our analysis, parametric with respect to different classes of attackers, is able to discover a larger number of potential protocol abuses.

Chari *et al.* [23] analyze the *authorization code* mode of OAuth 2.0 in the Universal Composability Security Framework [21]. 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 [38] 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

[2]Personal communication with the authors of [30], 28/2/2013.

runs. Using AVISPA, they identify a replay attack and a masquerade attack for which they propose and verify a fix.

The AUTHSCAN tool [30] described above is validated by analyzing single-sign-on web protocols, including Mozilla's BrowserID and Facebook Connect, and discovering several fresh vulnerabilities. In particular, AUTHSCAN finds a vulnerability in Facebook Connect because it infers from observed traces that one particular token-bearing message in not sent over HTTPS, but is instead sent over HTTP. Our analysis did not discover this particular attack because we decided to model Facebook as using HTTPS in all the token-bearing communications. The kind of vulnerabilities we discovered tend to concern flaws in the design of a bug-free implementation, whereas recovering models from traces seems also able to discover lower-level "implementation bugs".

Armando *et al.* [7] verify in the SATMC model checker a formal model of the SAML Single-Sign-On protocol, discovering a new man-in-the-middle attack on the variant used by Google Apps. Their approach is similar to ours: they build a formal model of the protocol and discover possible attacks via automatic verification. Also their attacks need to be validated on actual deployments. Recent related work part of the SPaCIoS EU project [36] develops techniques to automate both the extraction of models form protocol traces and the validation of attack traces using real web addresses, cookies and protocol messages.

More recently, Armando *et al.* [8] extended their previous work to consider also OpenID and additional deployment scenarios for SAML SSO (where different messages may belong to different SSL connections). This led to the discovery of an authentication flaw that affected both SAML SSO and Open ID. Exploiting this problem, a malicious service provider could force a user to access protected resources without their explicit permission. They also discovered a cross-site scripting attack that made the exploit possible on Google Apps. The main idea of the exploit is that when a client engages in the SAML SSO protocol with a malicious service provider, the latter can start the same protocol with the service provider target of the attack, obtaining authentication obligations bound to a resource on the target server, that the client will discharge while thinking to be discharging obligations relating to the malicious provider. As noted in [8], this flaw can be used as a launching pad for CSRF attacks, if the malicious provider crafts a redirection URI for the client that triggers a CSRF attack on the target server (when the server is susceptible to CSRF). In this way, the attacker is silently forcing the client to have side effects on her data on the target server. This bears some similarity to our *social CSRF* attack, although our attack is more general because it rests on a weaker hypothesis. In the case of social CSRF in fact, the victim of the attack (Facebook) does not need to suffer from a CSRF vulnerability. Instead, to exploit the attack, it is sufficient to find a CSRF on a lower-value, non-malicious intermediary (CitySearch) that participates in the OAuth protocol.

8. Conclusions

In this paper, we presented 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.

This paper also introduced the WebSpi library. WebSpi is a general purpose library that can be used for modeling and analyzing any security protocol implemented on top of standardized web browsers, servers, and the HTTP protocol. To ease the task of writing formal models in our framework, we presented a model extraction tool that automatically translates programs written in PHP and JavaScript subsets to the applied pi-calculus.

We did not try to encompass in this work all possible web interactions and languages. Experiments with other case studies [10] suggest that our model can be suitably extended to address challenges posed by different classes of web applications. We hope that others will benefit from our analysis framework, and contribute to its extension to capture ever more features of the web.

References

[1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 104–115, 2001.

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

[3] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[4] M. Abadi and B.T. Loo. Towards a declarative language and system for secure networking. In *USENIX international workshop on Networking meets databases*. USENIX Association, 2007.

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

[6] A. Armando, D. A. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuéllar, P. Hankes Drielsma, P. 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 *International Conference on Computer Aided Verification*, pages 281–285, 2005.

[7] A. Armando, R. Carbone, L. Compagna, J. Cuellar, and L. Tobarra Abad. Formal analysis of SAML 2.0 web browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps. In *ACM Workshop on Formal Methods in Security Engineering*. ACM Press, 2008.

[8] A. Armando, R. Carbone, L. Compagna, J.o Cuéllar, G. Pellegrino, and A. Sorniotti. An authentication flaw in browser-based Single Sign-On protocols: Impact and remediations. *Computers & Security*, 33:41–58, 2013.

[9] M. Avalle, A. Pironti, D. Pozza, and R. Sisto. JavaSPI: A framework for security protocol implementation. *International Journal of Secure Software Engineering*, 2:34–48, 2011.

[10] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Keys to the cloud: Formal analysis and concrete attacks on encrypted web storage. In *Conference on Principles of Security and Trust*, 2013.

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

[12] C. Bansal, K. Bhargavan, and S. Maffeis. Discovering concrete attacks on website authorization by formal analysis. In *IEEE Computer Security Foundations Symposium*, pages 247–262. IEEE, 2012.

[13] A. Barth, C. Jackson, and J.C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 75–88. ACM, 2008.

[14] K. Bhargavan, C. Fournet, R. Corin, and E. Zălinescu. Verified cryptographic implementations for TLS. *ACM Transactions on Information and System Security*, 15(1):3:1–3:32, 2012.

[15] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *IEEE Computer Security Foundations Workshop*, pages 139–152, 2006.

[16] K. Bhargavan, C. Fournet, A.D. Gordon, and N. Swamy. Verified implementations of the information card federated identity-management protocol. In *ACM symposium on Information, computer and communications security*, pages 123–135. ACM, 2008.

[17] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *IEEE Computer Security Foundations Workshop*, pages 82–96, 2001.

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

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

[20] A. Bohannon and B. C. Pierce. Featherweight Firefox. *USENIX conference on Web*, 2010.

[21] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Symposium on Foundations of Computer Science*, pages 136–145, 2001.

[22] S. Cantor, J. Kemp, R. Philpott, and E. Maler. Assertions and protocols for the OASIS Security Assertion Markup Language (SAML) v2.0, 2005.

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

[24] F. Corella and K. Lewison. Security analysis of double redirection protocols. Pomcor Technical Report, 2011.

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

[26] D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT–29(2):198–208, 1983.

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

[28] C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization in distributed systems. In *IEEE Computer Security Foundations Symposium*, pages 31–48, 2007.

[29] C. Fournet, A. D. Gordon, and S. Maffeis. A type discipline for authorization policies. *ACM Transactions on Programming Languages and Systems*, 29(5), 2007.

[30] J. Lei G. Bai, G. Meng, S. Venkatraman, J. Sun P. Saxena, Y. Liu, and J. Dong. AUTHSCAN: Automatic extraction of web authentication protocols from implementations. In *Networks and Distributed Systems Security Symposium*, 2013.

[31] T. Groß, B. Pfitzmann, and A. Sadeghi. Browser model for security analysis of browser-based protocols. In *Eropean Symposium on Research in Computer Security*, pages 489–508, 2005.

[32] E. Hammer-Lahav. OAuth Security Advisory: 2009.1 - Session Fixation Attack, 2009.

[33] S. Hansen, J. Skriver, and H.R. Nielson. Using static analysis to validate the SAML Single Sign-On protocol. In *Workshop on Issues in the theory of security*, pages 27–40. ACM, 2005.

[34] D. Hardt. The OAuth 2.0 Authorization Framework. IETF RFC 6749, 2012.

[35] D. Jackson. Alloy: A logical modelling language. In *International Conference of B and Z Users*, page 1, 2003.

[36] L. Viganó *et al.* Spacios: Secure provision and consumption in the internet of services. `http://www.spacios.eu`.

[37] T. Lodderstedt, M. McGloin, and P. Hunt. OAuth 2.0 threat model and security considerations. IETF RFC 6819, 2013.

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

[39] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[40] Open Web Application Security Project (OWASP). OWASP Top Ten Project. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`, 2010.

[41] OpenID Foundation. OpenID Connect Standard 1.0. `http://openid.net/connect/`, 2011.

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

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

[44] B. Pfitzmann and M. Waidner. Federated identity-management protocols. In *Security Protocols Workshop*, pages 153–174, 2005.

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

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

[47] J. Somorovsky, A. Mayer, A. Worth, J. Schwenk, M. Kampmann, and M. Jensen. On breaking SAML: Be whoever you want to be. In *Workshop on Offensive Technologies*, 2012.

[48] S. Sun and K. Beznosov. The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems. In *ACM conference on Computer and communications security*, pages 378–390, 2012.

[49] 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.

[50] R. Wang, S. Chen, and X. Wang. Signing me onto your accounts through Facebook and Google: A traffic-guided security study of commercially deployed Single Sign-On web services. In *IEEE Symposium on Security and Privacy*, pages 365–379, 2012.

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

[52] 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.

[53] 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.

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 4: Formal Attacks found using ProVerif

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 5: Concrete OAuth Website Attacks derived from ProVerif Traces

A. Example ProVerif Attacks on OAuth 2.0 Websites

A.1. Automatic Login and Social Sharing CSRF (CitySearch, 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))).
```

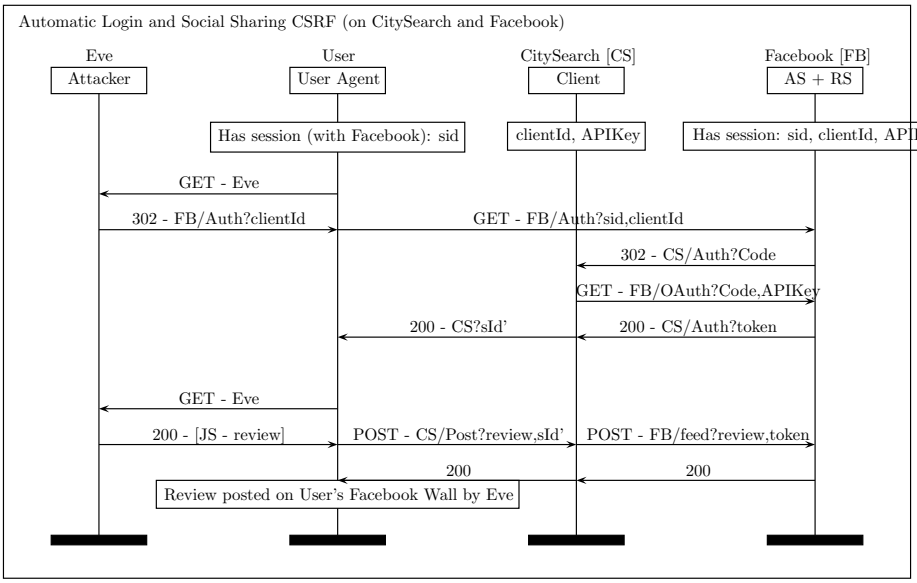
A.2. Social Login CSRF (IMDB, Facebook)

```
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);
```



1

Figure 10: Automatic Login and Social Sharing CSRF (CitySearch, Facebook)

```
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).
```

A.3. Resource Theft by Access Token Redirection (Yahoo, 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)))).
```

A.4. Unauthorized Social Login by Auth Code Redirection (WordPress, 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).
```