SAAAM: Simple Authentication and Authorization in the Age of Mobile

Renamed from SimpleAuth

Francisco Corella, PhD
fcorella@pomcor.com

Karen Lewison, MD
kplewison@pomcor.com

Revised June 26, 2012

1 Motivation

In 2008 Facebook introduced Facebook Connect, a protocol that allowed a Web site or application to delegate authentication of a user to Facebook and to gain limited access to the user's Facebook account, without asking the user to reveal her Facebook password. Other top-tier social networks have followed suit and provided similar functionality. The combination of authentication and authorization pioneered by Facebook Connect is called *social login* [1, 2].

Social login is an important innovation, but it is not well supported by existing protocols. Existing protocols require prior registration of the client (the Web application or Web site that delegates authentication to the social network) with the server (the social network), which is a cumbersome procedure. (There are dynamic registration proposals, but they do not provide security against client impersonation.) Only top-tier social networks can take full advantage of the functionality because clients only register with top-tier social networks. Key use cases where the client and/or the server is a native application on a mobile device (connected to an online back-end over the Internet) are not supported. And the existing protocols are very complex and have security gaps.

In this paper we informally propose a new social login protocol, called *SAAAM* that addresses the shortcomings of existing social login protocols. SAAAM does not require prior client registration, and it supports native applications both as clients and as servers. It is also simpler than existing social login protocols: it uses a single token, it uses no XML or JSON data structures, it does not require the use of JavaScript for POST redirection, and it uses no cryptography besides TLS. Yet it provides strong security: we believe that SAAAM may be able to achieve levels of assurance 3 and 4 as defined by NIST in [3], when suitably profiled and combined with strong identity proofing and strong user authentication at the server.

In Section 2 we describe the shortcomings of the existing protocols in more detail. In Section 3 we explain the rationale used in the design of SAAAM. SAAAM has only one protocol flow, which is described in Section 4. Finally, in Section 5 we recapitulate the benefits to be gained from standardizing and implementing SAAAM. *Section 4 may be read first.*

2 Shortcomings of Existing Social Login Standards

Social login was initially implemented using proprietary protocols, but today it is mostly implemented using OAuth 2.0 [4], a protocol currently being standardized by the IETF. Another social login protocol, OpenID Connect [5], is being standardized by the OpenID Foundation. OpenID Connect is an extension of OAuth that incorporates features of the OpenID third-party login protocol. This section discusses shortcomings of OAuth and OpenID Connect that motivate the proposal of SAAAM as an alternative protocol for social login.

2.1 Few Social Networks Can Fully Benefit from Social Login

OAuth requires prior registration of the client with the server in order to establish a shared secret[1]. Consequently users of the client can only log in via one of the social networks that the client has registered with. Typically the client registers only with Facebook, or with Facebook and one or more of Twitter, Google+ and LinkedIn. Other social networks cannot hope that significant numbers of Web sites and Web applications will register with them as OAuth clients, and are thus deprived of the benefits of social login.

This lack of freedom of choice contrasts with a current trend toward social network diversification. Until recently, Facebook seemed destined to completely dominate the social network space. But that is changing. Facebook users have serious privacy concerns [6] and are afraid of using the social login feature of Facebook for privacy reasons[2]. Alternative social networks such as Tumblr and Pinterest are getting traction [7]. Hundreds of small social networks focused on particular user interests or targeting users in particular countries are appearing [8]. And there are platforms that allow end-users to set up their own social networks without programming [9, 10]. Allowing all of these social networks to fully benefit from social login is a major motivation of SAAAM.

[1] OAuth distinguishes between an authorization server and a resource server. But in social login these are best viewed as two endpoints of the same server, so we shall not make that distinction.

[2] Eric Sachs, Google Product Manager, reported at the last Internet Identity Workshop (IIW 14) that users participating in usability focus groups were afraid of logging in with Facebook or Google+ because "their friends would be spammed".

The need for prior registration has been recognized by others as a problem, and there are two proposals to obviate it by using instead *dynamic registration* (i.e. registration on the fly) for the existing social login protocols; but neither proposal is helpful. The OAuth Dynamic Client Registration Protocol [11] is a proposal to add dynamic registration to OAuth. It specifies several flows where the client communicates metadata to the server and obtains a client identifier and a shared secret; but the client does not authenticate when requesting the secret, and thus anybody can obtain a secret associated with a claimed client identity. The OpenID Connect Dynamic Client Registration [12] is a specification in the OpenID Connect Protocol Suite [13]. As in the OAuth dynamic registration proposal, the client sends metadata to the server and obtains the shared secret. This proposal recognizes the need for the client to authenticate in order to get the secret, but the only authentication method it proposes is an access token obtained "out of band". It is not clear how the client could obtain such a token.

2.2 Lack of Support for Native Applications

It has been said that we have entered the *Age of Mobile* [14]. More than half of Facebook users use smart phones or tablets to access Facebook [15]. Facebook has a native application that can be accessed directly by other native applications on the same mobile device. But OAuth does not support native clients securely, and does not consider the use of a native application in the role of authorization server (the role played by the Facebook mobile application).

Early versions of OAuth (around version 10) made an attempt at supporting native applications with a separate flow. Later versions have given up on that. The latest version has a section [4, Section 9] with suggestions on how to use the implicit grant flow and the authorization code grant flow to support native applications as clients. But neither flow is applicable.

The authorization code grant flow is not applicable because it would require the native application to store the secret shared by client and server in the user's device, exposing it to the user. The OAuth specification [4] recommends not using the shared secret when the authorization code grant flow is used by a native client. But then a malicious native application may be able to impersonate a legitimate client.

The implicit grant flow is not applicable because, as specified, it would require the native application to use JavaScript code to obtain the access token, which is included in the URL fragment; this is something that the native application cannot do. The implicit grant flow can be adapted for use by native applications, by having the native application read the fragment, without using any JavaScript code. But the implicit grant flow is not secure for social login, because it allows impersonation of the user by a malicious client, as explained below in Section 2.3. Recent discussions on the OAuth Mailing List have confirmed that some native applications that use OAuth for Facebook Login are vulnerable to attacks [16, 17], while others protect themselves using

non-standard and undocumented workarounds provided by Facebook [18, 19].

The section of the OAuth specification on native applications makes other suggestions that are impractical or insecure, including manual cut-and-paste of credentials (utterly unfeasible using the tiny keyboard of a small phone) and using a browser embedded in the native application (which would defeat a stated purpose of OAuth by revealing the user's password to the client).

The OpenID Connect specifications do not discuss native applications.

2.3 Security Vulnerabilities

OAuth is used for social login, but was originally designed as an authorization protocol, which allowed the user to grant access to her account at the server without revealing her server password [20]. When OAuth is used for social login today, authentication is a by-product of authorization: the client obtains the user's identity at the server by accessing the user's account.

In the OAuth flow for traditional Web applications, the "authorization code grant flow", the client obtains an authorization code, uses the authorization code to obtain an access token, and uses the access token to access the user's account. The client authenticates with the shared secret when it uses the authorization code to obtain the access token. But in the flow for browser-resident JavaScript applications, the "implicit grant flow", the browser-resident code cannot have the shared secret, because that would expose the secret to the user. In that flow there is no authorization code, and the client obtains the access token without authenticating to the server. But that allows the client, if malicious, to impersonate the user vis-a-vis another client by presenting the access token to the other client [21]. OpenID Connect eliminates this vulnerability by introducing a third authenticator, the *ID Token*, and making the ID Token specific to the client.

Another vulnerability of OAuth comes from the fact that it recommends but does not require TLS protection for the client callback endpoint specified by the `redirect_uri` parameter [4, Section 3.1.2.1]. The editor and chairs of the working group decided against requiring TLS following a message from Facebook saying that Facebook would not require TLS from its client developers [22]. Eventually, a TLS requirement for the callback endpoint was added for the authorization code grant flow in the case where OAuth is used for authentication (not only authorization), as a countermeasure againt user impersonation attacks. The requirement is tucked away in the security considerations section [4, Section 10.5]. But not using TLS in other cases allows other attacks at least as serious. In the authorization code grant flow, when OAuth is used only for authorization, not requiring TLS for the callback endpoint allows an attacker to intercept the authorization code sent to a victim user and present it to the client instead of the attacker's own authorization code, thus gaining access to the victim's protected resources. In the implicit grant flow, not requiring TLS allows a man-in-the-middle attack where the attacker replaces the client application's JavaScript code as it is downloaded to the user's browser, with the

attacker's own JavaScript code, thus gaining control of the client application. These are practical attacks that can be mounted, e.g., by using a rogue WiFi access point [23].

The TLS requirements of OpenID Connect for the callback endpoint are unclear.

Yet another vulnerability of OAuth is the fact that an attacker can initiate an OAuth flow, obtain an authorization code from the server, or an access token in the case of the implicit grant flow, and then, instead of sending the code or token to the client, cause a victim user to send it instead. This attack, which we call a *login-as-attacker* attack, can be easily mounted by tricking the victim into clicking on a seemingly unrelated button in a page provided by the attacker, causing submission of a form that targets an invisible frame. As a result, the victim is logged in to the attacker's account at the client, without realizing that it is not the victim's own account. Consequences of a victim user being logged in to an attacker's account without realizing that it is the wrong account are discussed in [24]; the victim may for example enter confidential information into the attacker's account.

The OAuth specification refers to this login-as-attacker attack [4, Section 10.12], but leaves it to implementors to cope with it. It suggests a countermeasure that does not seem applicable when OAuth is used for social login.[3] SAAAM, on the other hand, has protection against this attack built into the protocol.

2.4 Complexity

OAuth is a logically complex protocol because it mixes different approaches to security: sometimes an authorization code is used, sometimes it is not; sometimes the client authenticates to the server, sometimes it does not; sometimes the client obtains the access token through a direct connection to the server, sometimes it uses JavaScript code to extract it from a URL fragment. Complexity makes it difficult to analyze the security of the protocol, which may explain some of the vulnerabilities described above.

OpenID Connect inherits the logical complexity of OAuth and increases complexity by using an "ID token" in addition to the authorization code and the access token of OAuth, and by adding six specification documents to the already substantial collection of specification documents of OAuth.

Facebook uses a proprietary method to allow a native application running on a mobile device to offer social login via a Facebook native application installed on the same device. The method requires a Facebook SDK, which links Facebook code with the native application code. On Android, the native application has to register a public key with Facebook for authentication of the application to

[3] The suggestion is not clear, but it seems to involve including in the `state` parameter a hash of a session cookie used by the client to authenticate the user's browser. There may not be such a session cookie when OAuth is used for social login, because the user has not logged in yet to the client.

Facebook. Facebook does not support social login to a Web application that a user accesses through a mobile browser via a Facebook application installed in the same mobile device.

By contrast SAAAM is a simple protocol that provides strong security and supports all four combinations of native and Web applications in the roles of client and server with only minor variations in protocol flow.

3 Protocol Design and Security Considerations

3.1 Overview

The SAAAM protocol is a composite protocol consisting of a SAAAM Core protocol that specifies how a SAAAM client obtains an access token from a SAAAM server, and any number of usage protocols specifying different possible usages of the access token. One particular usage is social login, where the server is a social network and the client uses the access token to obtain user identity data relative to the social network, a.k.a. as user profile data, and to issue social network updates on behalf of the user.

Social login is the usage that motivates SAAAM, but a variety of other usages of an access token are possible. A very different usage could be to authenticate the user for a high-value transaction in order to authorize the transaction and create a log of the transaction that could not be repudiated by the user.

The flow of SAAAM is similar to that of several other protocols where a client delegates user authentication to a server and/or obtains delegated authorization from the user to access the server (Microsoft Passport [25], Windows Live [26], SAML Browser SSO Profile [27], Shibboleth [28], OpenID [29], OAuth [4], OpenIDConnect [5], etc.).

The client sends a request for an access token to an authorization endpoint of the server. Then the server authenticates the user, identifies the client to user, and obtains permission from the user to send the access token to the client. Then the server sends the access token to a callback endpoint of the client. In the social login usage, the client then uses the access token to obtain user profile data and to issue updates.

How the server authenticates the user is not specified by SAAAM, except for requiring that the server must not ask for a password[4] which would facilitate phishing attacks. If the user uses a password to log in to the server, the login to the server must have occurred before the client asks for the access token. Instead of a password, the user may use a TLS client certificate, as in the Dartmouth implementation of Shibboleth [30]. A TLS client certificate issued by the server would be easy to use, as the server could verify its validity without obtaining a certificate revocation list or a certificate status signed by a separate certificate authority. A protocol that would allow the server to issue a certificate to the

[4]We are referring here to a long term password; a one-time password would be OK.

user's browser securely and automatically would be a good complement to the SAAAM protocol.

The SAAAM client may be a native application running on a mobile device or a Web application that interacts with the user through a Web browser. We say that the client is a native client in the former case, an online client in the latter. In the social login usage, we assume that a native client is connected to an online back-end, which maintains user accounts. The case where the client is online includes the case where the client is a JavaScript application embedded in a Web page of the online client. In SAAAM there is no need for a separate flow to support this case.

The SAAAM server is not necessarily a single physical server. It is an abstraction that may realized by one or more physical or virtual (cloud) servers. It may also be realized by a front-end implemented as a native application running on a mobile device, connected over the Internet to an online back-end consisting of one or more physical or virtual servers. We then refer to the server as a native server; however, some communications between the client and a native server take place directly between the client and the online back-end of the server, instead of going through the native front-end of the server. We refer to a server that does not have a native front-end as an online server.

Besides the authorization endpoint, the server has usage-specific endpoints, used by the client with authorization provided by the access token.

In the social login usage, the server has a user-info endpoint where the client obtains user profile data, and an update-issuance endpoint where the client can issue updates on behalf of the user. In the case of a native server, those endpoints must be located on the online back-end of the server so that they can be accessed by online clients. An additional update-issuance endpoint may be located on the native front-end, for convenient access by a native client front-end.

But there must not be a user-info endpoint on the native front-end of the server. It might be tempting to provide a user-info endpoint on the server front-end, from which the client front-end could obtain user profile data, which it would then send to the client back-end, which would use it to locate or create a user account. But the client's back-end cannot trust its own front-end (nor the server front-end) to supply the user's identity (since front-ends are under user control). The client back-end must obtain the user's identity through a TLS connection to the user-info endpoint located on the server's back-end, after authenticating the server using the TLS certificate presented by the endpoint.

The URL of a usage endpoint may have a different hostname than the URL of the authorization endpoint, and may reside on a different physical or virtual server than the authorization endpoint. But the virtual or physical servers hosting the authorization endpoint and the usage endpoint should have access to a common back-end: an access token is a high-entropy random value that references an authorization record that specifies the scope of access provided

by the access token;[5] the authorization record should be stored in the common back-end. It is not impossible, though, to have a usage endpoint located on a remote server that cannot access the back-end containing the authorization record. The remote server can use the access token by sending it to a trusted proximal server that does have access to the back-end.

The authorization endpoint of an online server and the callback endpoint of an online client must be protected by TLS, i.e. the URLs of both endpoints must use the `https` scheme.

An online client sends the request for an access token to the authorization endpoint of an online server using HTTP redirection. A native client sends the request for an access token to an online server by asking the mobile operating system to point an external browser[6] to the authorization endpoint. A native client sends the request for an access token to a native server directly via the operating system. An online client sends the request for an access token to a native server by redirecting the user's mobile browser to a native URL of the native server, as explained below in Section 3.2.

Symmetrically, an online server sends the access token to the callback endpoint of an online client using HTTP redirection. A native server sends the access token to an online client by asking the mobile operating system to point an external browser to the callback endpoint. A native server sends the access token to a native client directly via the operating system. An online server sends the access token to a native client by redirecting the user's mobile browser to a native URL of the native client, as explained below in Section 3.2.

More details of the protocol flow can be found below in Section 4 and illustrations can be found in Figures 1-5.

3.2 Interapp Communications

Some mobile operating systems feature an interapp communication mechanism that allows native applications to communicate with each other using URLs internal to a mobile device, which we shall call *native URLs*. This feature is well documented in iOS [31]; it is less well documented, but available, in Android [32]; we do not know to what extent it is available in other mobile operating systems.[7]

A native application uses a special kind of URL, which we call a *native URL*, to send a message via the operating system of the mobile device to another native application residing on the same device. As in an HTTP GET request, the URL is both the address and the contents of the message. The URL identifies to the operating system the target application to which the message is being

[5] An access token can therefore be revoked by removing or invalidating the corresponding record. Hence there is no need to issue short-lived access tokens plus long-lived refresh tokens as OAuth does.

[6] I.e. a browser external to the client, rather than a browser embedded in the client.

[7] It seems to be available for communication between Internet Explorer and other applications on desktop Windows [33].

sent, and the operating system delivers the URL to the target application, which parses it to extract parameters, usually encoded in a so-called query-string portion of the URL. An important aspect of this communication method is how it is used by a browser, which is itself a native application. A URL with scheme `http` or `https` (only `https` is used in SAAAM) is delivered to a browser and causes the browser to send an HTTP GET request targetting the URL over a network. When a browser receives an HTTP redirection response to a request that it has sent over a network, if the URL specified by the redirection is a native URL, the browser sends that URL to the native application identified by the URL, via the operating system.

SAAAM assumes that an interapp communication mechanism like the one featured in iOS and Androed is available. When the SAAAM server is native, the authorization endpoint is a native URL and an online client redirects the user's mobile browser to that native URL. When the SAAAM client is native, the callback endpoint is a native URL and an online server redirects the user's mobile browser to that native URL.

More specifically, SAAAM assumes that each native application has an identifier from which it is possible to construct a URL prefix such that all native URLs that target the application are derived by appending suffixes to that URL prefix. We shall refer to the URL prefix as the *root URL* of the application. For consistency, we shall also use the term *root URL* to refer to a prefix of a an ordinary URL that uses the `https` scheme, specifically to the prefix comprising the scheme, the colon, the two slashes, the domain name, and the slash following the domain, e.g. `https://example.com/`.

In iOS and Android, the identifier of a native application is a *custom scheme*, and the root URL is obtained by adding a colon and two slashes to the custom scheme. For example, the root URL of the Twitter native application could be `twitter://`, where `twitter` is a custom scheme. While the use of native URLs for interapp communication is a good idea, the use of custom schemes to that purpose is not the best approach. Custom schemes are not compliant with IETF standards because they are not registered at the IANA registry of URI schemes [34]. Also, the custom scheme of an application is registered by the application with the operating system at run time rather than at installation time, and it is not clear how Apple and Google prevent a rogue application from registering a custom scheme that seems to refer to a legitimate application with the purpose of masquerading as the legitimate application.

A better approach would be to use a single scheme for native URLs, such as `app`, registered with IANA [34], and to use DNS domain names as native application identifiers. The root URL of an application could then be, e.g., `app://example.com/`. The operating system provider would verify that the provider of a trusted application owns the domain name that the application uses as identifier in its root URL. We hope that this better approach will be eventually used by mobile operating systems, perhaps codified by a future interapp communication standard. For the time being, custom schemes are suitable in use cases where native applications installed in a mobile device can be trusted

not to masquerade as each other.

A widely implemented feature of interapp communications is the ability for a native application to point an external browser to a URL. Upon request by the application, the mobile operating system launches the browser if it is not already running, and causes the browser to send an HTTP GET request to the URL specified by the application.

This feature could be enhanced by having the browser add extension HTTP headers to the request, providing the name and icon that the mobile operating system uses to identify the native application to the user (if requested, or at least permitted, by the native application). In SAAAM, this would allow an online server to receive the name and icon of a native client when the client sends the request for the access token to the authorization endpoint, and use them to identify the client to the user. We hope that this feature will eventually become available, perhaps included in the future interapp communication standard envisioned above. Client identification to the user is further discussed below in Section 3.4.

3.3 Server Discovery

Server discovery is the process by which the client learns the authorization endpoint of the server that the user wishes to use. The client has the following options for server discovery. Multiple options can be combined.

User Supplies Server Identifier. The user types a server identifier into an input box. The box provides suggested completions of partial entries.

On a desktop or laptop, the server is an online server and the server identifier is a DNS domain name. The input box is presented by a browser, and suggests completions as usual, based on prior entries in input boxes with the same input name, displayed by other clients.

On a mobile device different input boxes are used to enter the server identifier of an online server and that of a native server. The identifier of an online server is a DNS domain name. The identifier of a native server is the native application identifier used to construct the root URL of the application as described above in Section 3.2 (a custom scheme in the current versions of iOS and Android). If the interapp communication standard envisioned above in Section 3.2 becomes a reality, the identifier of a native server will be a DNS domain name, as in the case of an online server. If the client is online, the two input boxes are presented by a mobile browser, and completions are suggested as usual. If the client is native, the input box for a native server identifier suggests completions of partial entries based on the set of the native front-ends of SAAAM servers installed on the mobile device.

Once the client has obtained a server identifier, it constructs the root URL of the server, defined above in Section 3.2, such as `example://` or `app://example.com/` for a native server, or `https://example.com/` for an online server. Then it adds the suffix

`.well-known/SAAAM/authorization` to construct a well-known URL of the authorization endpoint, where the path segment `SAAAM` is to be registered with IANA as specified in [35].

An online server is free to redirect requests sent to the well-known URL of the authorization endpoint to a more convenient URL chosen by the server, which may use a different hostname, e.g. it may redirect from `https://example.com/.well-known/SAAAM/authorization` to `https://api.example.com/authorization`.

Client Consults User Preferences. When SAAAM is used for social login, an online client could learn what server to use from the user's browser rather than from an interaction with the user. The browser could learn what SAAAM servers are available to the user, either by observing redirections to SAAAM authorization endpoints, which are well-known locations, or by explicit declaration of each server to the browser. (A future protocol-agnostic *identity provider declaration standard* could allow an identity provider to declare itself as a preferred identity provider for the user, and be entered into the user preferences facility, with user consent.) The browser could maintain a list of available SAAAM servers and a current preferred server, as part of a general user preferences facility. It could also keep track of an association between clients and servers, remembering what server is preferred for each client, and what server should be used for newly encountered clients. The browser may then provide the client with the server identifier of a preferred server and metadata about the server.

A native client could similarly learn what server to use from the operating system of the mobile device, based on a similar user preferences facility maintained by the operating system.

Client Remembers Previous Server Choices. A previously visited client can remember the SAAAM server that was used in previous visits. To that purpose an online client may use a cookie, or HTML5 persistent storage, and a native client on a mobile device may use persistent storage provided by the mobile operating system.

Online Client Redirects to Account Chooser An online client may use Account Chooser [36], which involves redirecting the browser to `accountchooser.com`. Account Chooser is a protocol-agnostic facility contributed by Google to the OpenID Foundation that allows a user to choose a user account at a third-party identity provider from a menu.

User Chooses Server from Menu The client may let the user select a server from a menu of servers known to the client. However, a major goal of SAAAM is to allow the user to use a server that is not known to the client. Therefore this method should not be used except in combination with other methods that provide freedom of choice.

3.4 Identifying the Client to the User

As the server asks the user for permission to send the access token to a client, it must identify the client to the user, in order to protect ther user against a client impersonation attack.

At first glance it may seem that a client impersonation attack is impractical because the user is directed to the server after interacting with the client, and therefore should have already verified the identity of the client.

But without identification of the client to the user by the server, client impersonation would be a practical attack.

To impersonate an online client, an attacker could use DNS spoofing (which is a practical attack [23]) to present to the user a page constructed by the attacker that purports to be a login page of the legitimate client. The login page presented by the attacker would have to be an `http` rather than an `https` page, but even if the user paid attention to the fact that it is not an `https` page, she might not find that fact suspicious if the login page was not asking for a password. A social login form in the attacker's page could then execute the SAAAM protocol with a callback endpoint controlled by the attacker.

To impersonate a native client, we conjecture that an attacker could be able to install a malicious native application that would initiate the SAAAM protocol while running in the background, causing the user to believe that the protocol was initiated by a legitimate application running in the foreground.

To identify a native client to the user, the SAAAM server displays to the user the application identifier included in the root URL of the native client, which is itself included in the URL of the callback endpoint. If the interapp communication standard envisioned above in Section 3.2 becomes a reality, the server will be able to display to the user the name and icon by which the application is identified on the mobile device. It should be noted that the client could report to the server its name and icon, without waiting for the standard; but displaying a self-asserted name and/or icon to the user would facilitate client impersonation instead of protecting against it.

When the client is online, the URL of the callback endpoint is an `https` URL and the SAAAM client presents a TLS-server certificate at the endpoint, which contains information about the client. The SAAAM server obtains the certificate by initiating a TLS connection to the callback endpoint, executing the TLS handshake and then immediately closing the connection.[8] It can thus display information contained in the certificate to the user. Notice, however, that not all information in a TLS-server certificate is verified by the certificate authority (CA) that signs the certificate. (In common cases the only verified

[8]This connection could be avoided if the browser sent the TLS certificate of the client to the server. The HTTP redirection mechanism could be extended to specify that upon a redirection of an `https` request the browser adds an HTTP header containing the TLS certificate that was presented by the redirecting host (if requested or at least permitted by the redirecting host). This would be the counterpart of the browser providing the name and icon of the native app that launches the browser.

information is ownership of the domain name of the server.) The SAAAM server should only display verified information, or indicate to the user what information has been verified and what information was self-asserted to the CA. The SAAAM server should also display the names of the CAs that have signed the chain of CA certificates that back the TLS-server certificate.

3.5 Remembering the Client State

Before sending the initial request to the authorization endpoint, the SAAAM client creates a record containing protocol data and, more generally, data related to the current state of the client. When the access token comes back from the server, the client uses the record to remember its state at the time it had sent the request.

When the client is online it has two options: it may store it in the browser as a cookie;[9] or it may store the record internally and store a reference to the record in the browser as a cookie.[10] The first option is preferred, because the second option is vulnerable to a denial of service attack by storage exhaustion. When the client is native, it retains the record in memory and sends to the server a reference to the record, as a parameter, which the server returns with the access token. We use the term *state string*, or simply *state* to refer to the reference or the record that the client stores in the browser or sends to the server so that it can later remember its state at the time it sent the request.

It would be simpler to always send the state string to the server. However, storing the state string in the browser when the client is online serves to protect against two different attacks, as explained below.

3.6 Protecting the Access Token

The SAAAM server sends the access token to the callback endpoint of the client.

If the server and the client are both native, the access token is transmitted directly from the server to the client by the mobile operating sytem.

If the server is native and the client is online, the server asks the mobile operating system to use a mobile browser, launching it if necessary, to send an HTTP request to the callback endpoint of the client. The HTTP request conveys the access token from the browser to the client over a TLS connection, as a parameter added to the URL of the callback endpoint. (See below how the access token is protected against leakage from the browser.)

If the server and the client are both online, the server sends the access token to the client as a parameter of the URL of an HTTP redirect response. (Again,

[9] The value of the cookie could be an encoding or the record, e.g. in base64, or several cookies could be used, one for each data item in the record. It is not necessary to encrypt the record, since the record does not contain any client secrets that need to be protected against the user.

[10] A browser-resident JavaScript client has a third option: store the record in HTML5 persistent storage. This is equivalent to storing the record in a cookie.

see below how the access token is protected against leakage from the browser.) The token travels from the server to the user's browser in the HTTP response over a TLS connection. Then it travels from the browser to the client in the redirected HTTP request, over another TLS connection.

If the server is online and the client is native, the server sends the access token to the browser in an HTTP redirect to the native URL of the callback endpoint. The token travels from the server to the user's mobile browser over a TLS connection as in the previous case. Then it is transmitted from the browser to the client by the operating system.

Thus the access token is transmitted over secure connections, either directly from the server to the client, or via a browser, which is a trusted entity. However, when transmitted via a browser, the access token is embedded in the callback URL. (The callback URL is obtained by appending a query string with various parameters to the URL of the callback endpoint.) And when the client is online, in the absence of countermeasures, the access token could leak to an attacker through an HTTP "referer" header, if the Web page downloaded by the client in response to an HTTP request that targets the callback URL contains a link whose target is controlled by the attacker. To prevent this, the access token is encrypted by x-oring it with a one-time random key of same bit length as the access token.

The key is sent by the client to the server when the client initiates the social login, as a parameter added to the URL of the authorization endpoint, and it is stored by the client in the record that the client uses to remember its state, described above in Section 3.5. In the absence of countermeasures the key could similarly leak to the attacker through a "referer" header if the server is online and the Web page downloaded by the server in response to an HTTP request that targets the authorization URL contains a link whose target is controlled by an attacker. If both the key and the encrypted access token leak to the attacker, the attacker can x-or the key with the encrypted access token to recover the plaintext access token. An attacker could also obtain the key and the encrypted access token from the browser history by gaining access to the user's device.

The risk presented by a double leak or by the attacker gaining access to the user's device may be acceptable for the social login usage. If higher security is called for, the key itself can be encrypted under a key-encryption public key of the server, made available in a well-known location. (When using the identity-provider declaration standard envisioned above in Section 3.3, the public key could have been conveyed by the server to the user's browser or the operating system of the mobile device when the server declared itself to the browser as a SAAAM server, and could have been stored in the user preferences facility as server metadata, well before execution of the SAAAM protocol, thus saving the client's roundtrip to the well-known location.)

We can thus prevent an attacker from obtaining a plaintext access token and using it to access the user's account at the server. However, we also have to prevent an attacker from obtaining an *encrypted* access token. An encrypted access token may leak, and, if no precautions were taken, an attacker could

impersonate the user by presenting the encrypted access token to the client. This attack is prevented by the fact that an online client (the type of client that may leak the encrypted access token) stores the state string in the browser, instead of sending it to the server. To impersonate the client, the attacker needs to present both the leaked encrypted access token and a cookie header containing the state string. But the cookie header does not leak. As an additional countermeasure, for the sake of defense in depth, the user-info endpoint honors a given access token only once (whereas the update-issuance endpoint honors the same access token multiple times).

3.7 Need for TLS Protection of the Callback Endpoint

Since the access token is sent encrypted to the callback endpoint of an online client, it may seem that there is no need to provide TLS protection for the callback endpoint. But TLS protection is needed, at least for the social login usage. Otherwise an attacker could intercept the HTTP request carrying the encrypted access point from a legitimate user's browser to the client, and log in to the client as the legitimate user by sending the intercepted HTTP request to the client from the attacker's own browser. This is a practical attack that can be mounted, e.g., by using a rogue WiFi access point [23].

3.8 Preventing the Client from Impersonating the User

Without countermeasures, in the social login usage, an attacker could use a malicious client to obtain an access token providing access to a legitimate user's account at a social network, and use the access token to impersonate the legitimate user vis-a-vis a legitimate client.

The attack is as follows: the attacker sets up the malicious client and tricks the legitimate user into doing a SAAAM social login to the malicious client via the social network. The attacker thus obtains an access token for the legitimate user's account at the social network. The attacker has a user account at the social network, and initiates a SAAAM social login to the legitimate client via the social network, i.e. a SAAAM execution with the legitimate client in the role of SAAAM client and the social network in the role of SAAAM server. The client sends the access token encryption key to the server through the attacker's browser, and the attacker captures it. (We assume that the key is not encrypted under a public key of the server; otherwise the attack is not possible.) The server sends an encrypted access token to the client via the attacker's browser. This access token is for the attacker's account, but the attacker intercepts it at the browser and substitutes for it the access token for the legitimate user's account that it obtained earlier, encrypting it with the captured key. The legitimate client decrypts the access token received from the attacker's browser, presents it to the user-info endpoint of the server, receives the legitimate user's profile data, and logs in the attacker with the user's social network identity.

The attack is prevented by making the access token specific to the client, and having the client identify itself when using the access token, using as identifier the URL of the callback endpoint. The attack then fails because the server rejects the access token sent to the malicious client when presented by the legitimate client.

Two important points must be made:

1. To identify itself, the client *declares* its identity by presenting the identifier (the URL of the callback endpoint) when using the access token. It does not need to *prove* its identity.

2. The fact that the access token is specific to the client does not mean that it is a data structure comprising the client's identity (the URL of the callback endpoint). As stated above in Section 3.1, the access token is a reference to an authorization record kept by the server. Making the access token specific to the client means including the identifier in the record. When the client uses the access token, it sends along the identifier, and the server verifies that the identifier received from the client coincides with the one present in the record.

3.9 Preventing Login to Attacker's Account

Without countermeasures, in the social login usage, an attacker could lure a user (the victim) into logging in to the attacker's account at the client. Login-as-attacker attacks are often overlooked, but they have been studied in academia and they can have serious consequences [24], such as causing the victim to enter confidential information into the attacker's account.

In SAAAM, when the client is online and in the absence of countermeasures, the attack would proceed as follows. The attacker creates an account at the server and an account at the client that can be accessed by social login with the attacker's identity at the server. The attacker sets up a Web site and lures the victim into downloading a page from the site that contains a link whose target attribute specifies an invisible frame. When the victim clicks on the link, the site simulates a browser executing a SAAAM social login with the client via the server.[11] The simulated browser obtains the encrypted access token for the attacker's account at the server but does not send it to the client. Instead it redirects the victim's browser (in response to the user's click on the link) to the callback endpoint of the client, with the intercepted encrypted access token as parameter. (To be precise, it redirects the victim's browser to the same callback URL to which the simulated browser was redirected by the server.) This causes the client to log in the victim, without the victim realizing it because the login acknowledgement page goes to the invisible frame. If the victim was already logged in, the client will typically set a session cookie for a

[11] The simulated browser only implements the interactions with the client and the server. It does not interact with the user.

login session pertaining to the attacker's account, replacing the previous session cookie for a login session pertaining to the victim's account.

The attack is prevented by the fact that an online client stores the state string in the browser, instead of sending it to the server. The attacker cannot set a cookie in the victim's browser that will be sent to the callback endpoint; more generally, it cannot cause the victim's browser to send the state string to the callback endpoint in a Cookie HTTP header.

The attack is not possible when the client is a native client rather than an online client because the attacker does not have access to the native client.

3.10 Using the Access Token

Each usage protocol specifies one or more usage endpoints, which implement usage-specific APIs. The client presents the access token along with the URL of the callback endpoint when making an API request at a usage endpoint. In the social login usage, as explained above in Section 3.1, the user-info endpoint of a native server is located on the online back-end; there must be an update-issuance endpoint on the back-end, and there may be an additional one on the front-end. The client presents the access token to an online endpoint by including it in an HTTP Authorization header, in the body of an HTTP POST request (which may be a URL-encoded body or a multipart body) or in the query portion of the URL of an HTTP GET request. Which method is used is specified by each usage protocol.

3.11 Usage-Specific Parameters

Besides specifying how the access token is presented, each usage protocol may specify additional request parameters that the client sends to the authorization endpoint of the server, and additional response parameters that the server sends to the callback endpoint of the client. For example, in the social-login usage protocol, the client sends to the server a parameter listing the user profile data items that the client wishes to retrieve at the user-info endpoint, and the server returns to the client the URL of the user-info endpoint, the URL of the update-issuance endpoint, and the length of time during which the access token will be honored at the endpoints.

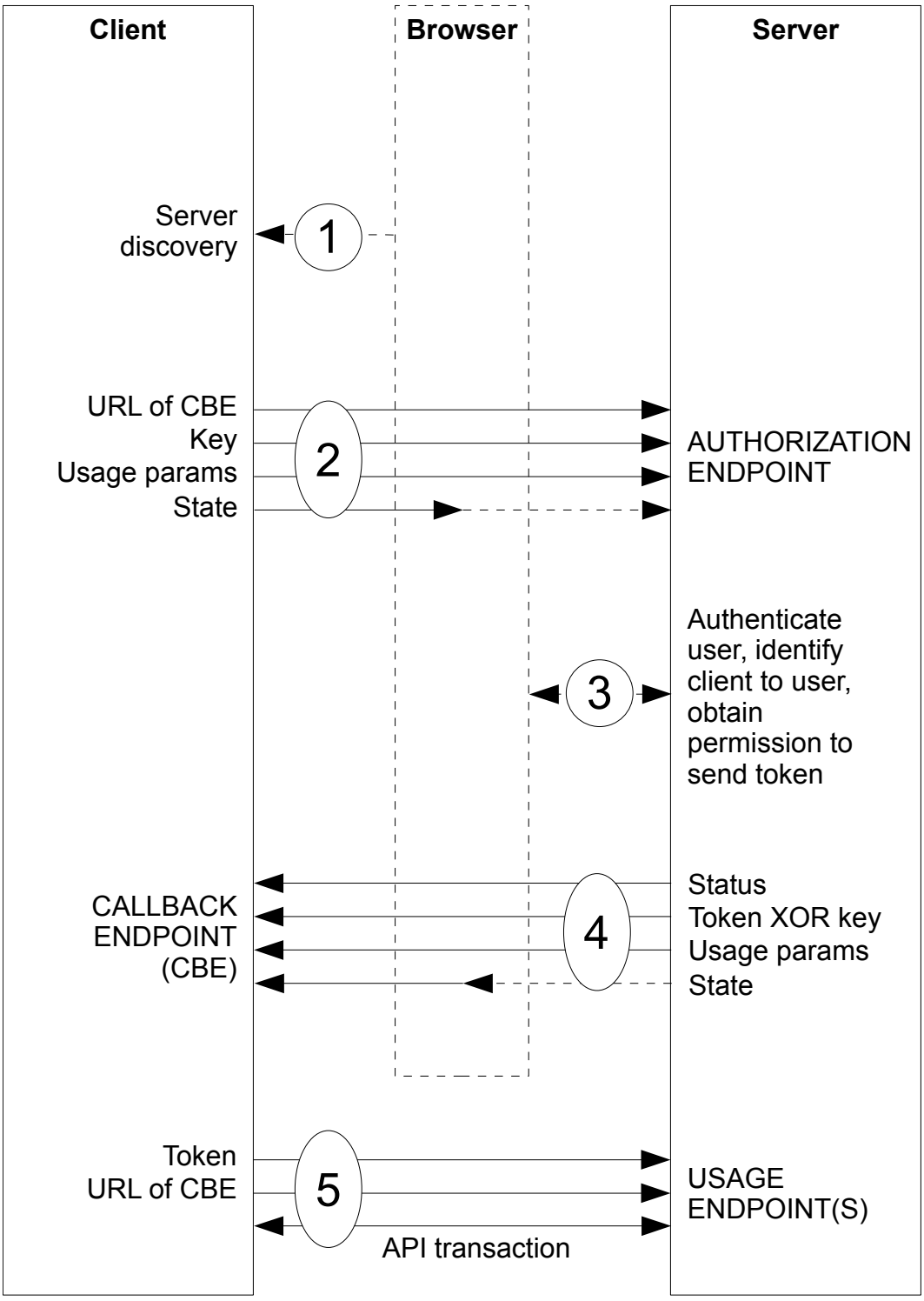

Figure 1. General Protocol Flow

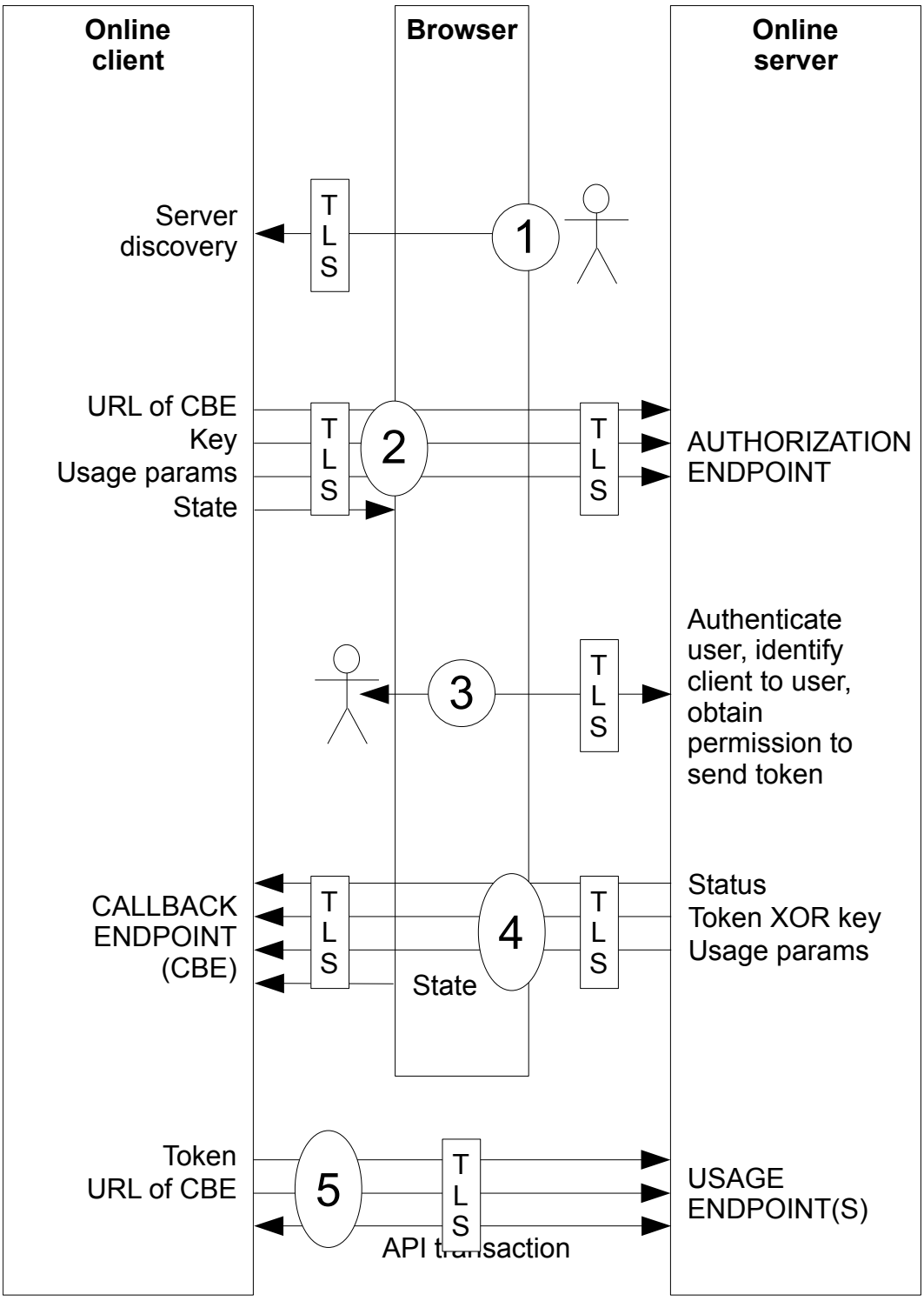

Figure 2. Online client, online server

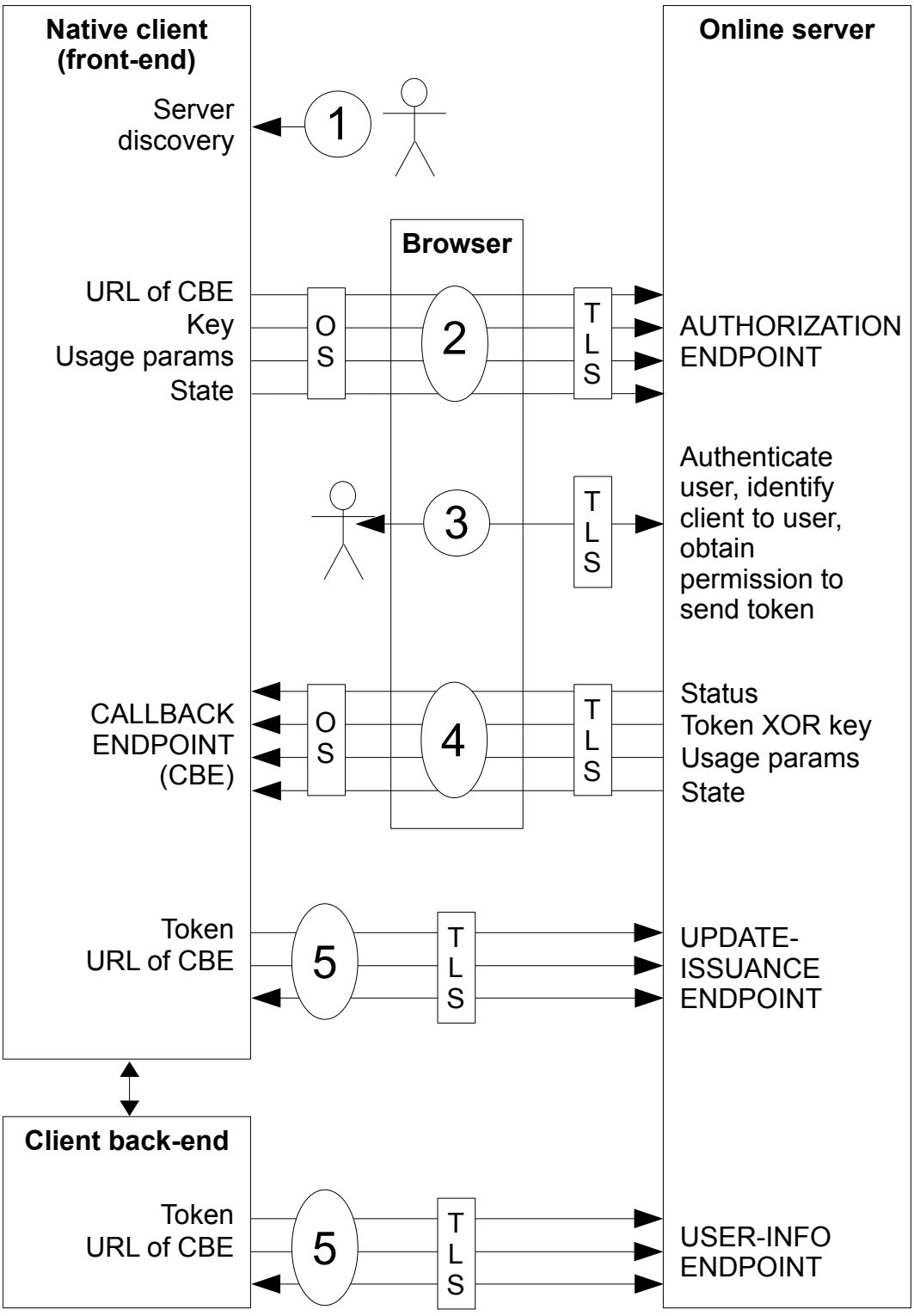

Figure 3. Social login with native client and online server

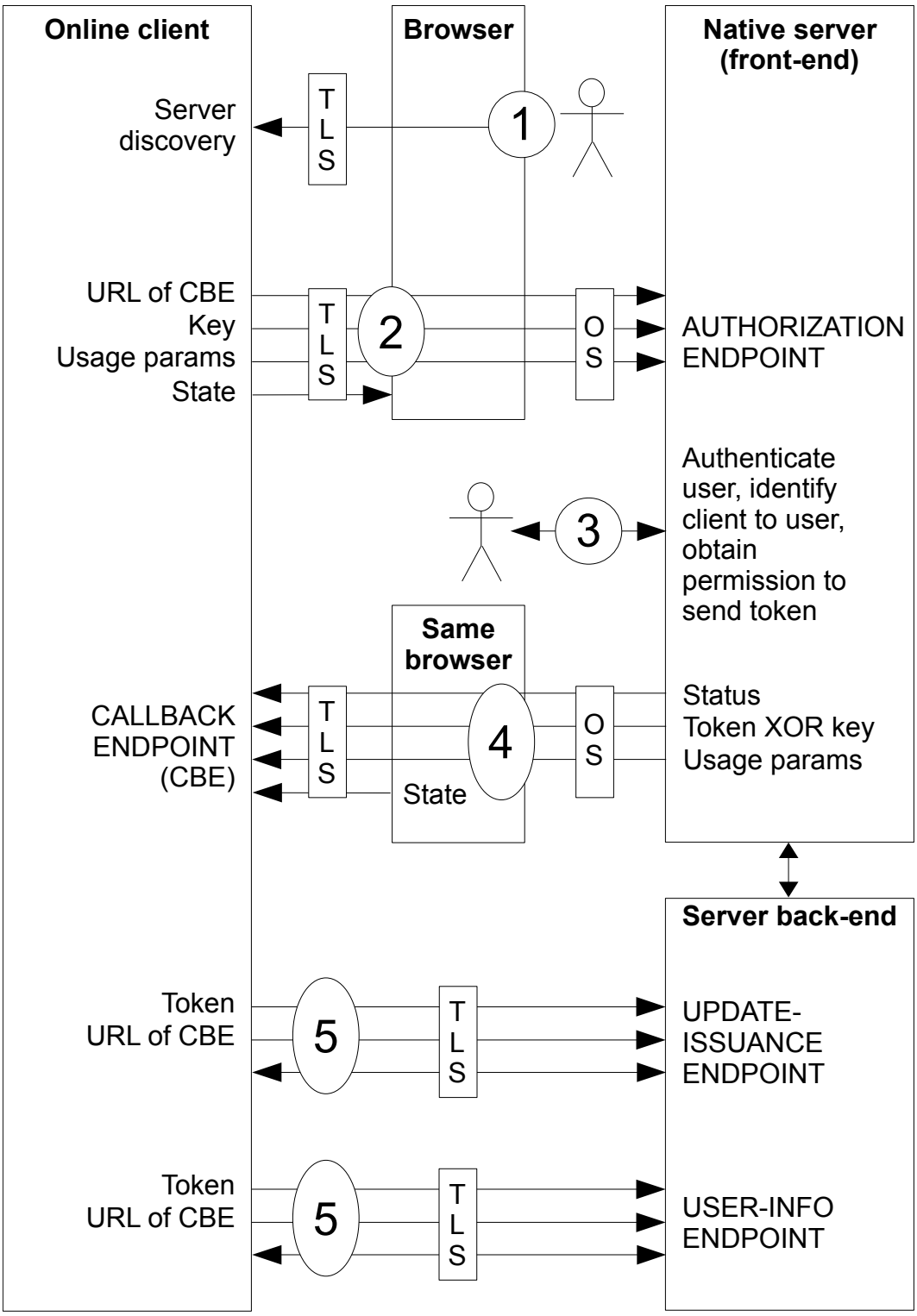

Figure 4. Social login with online client and native server

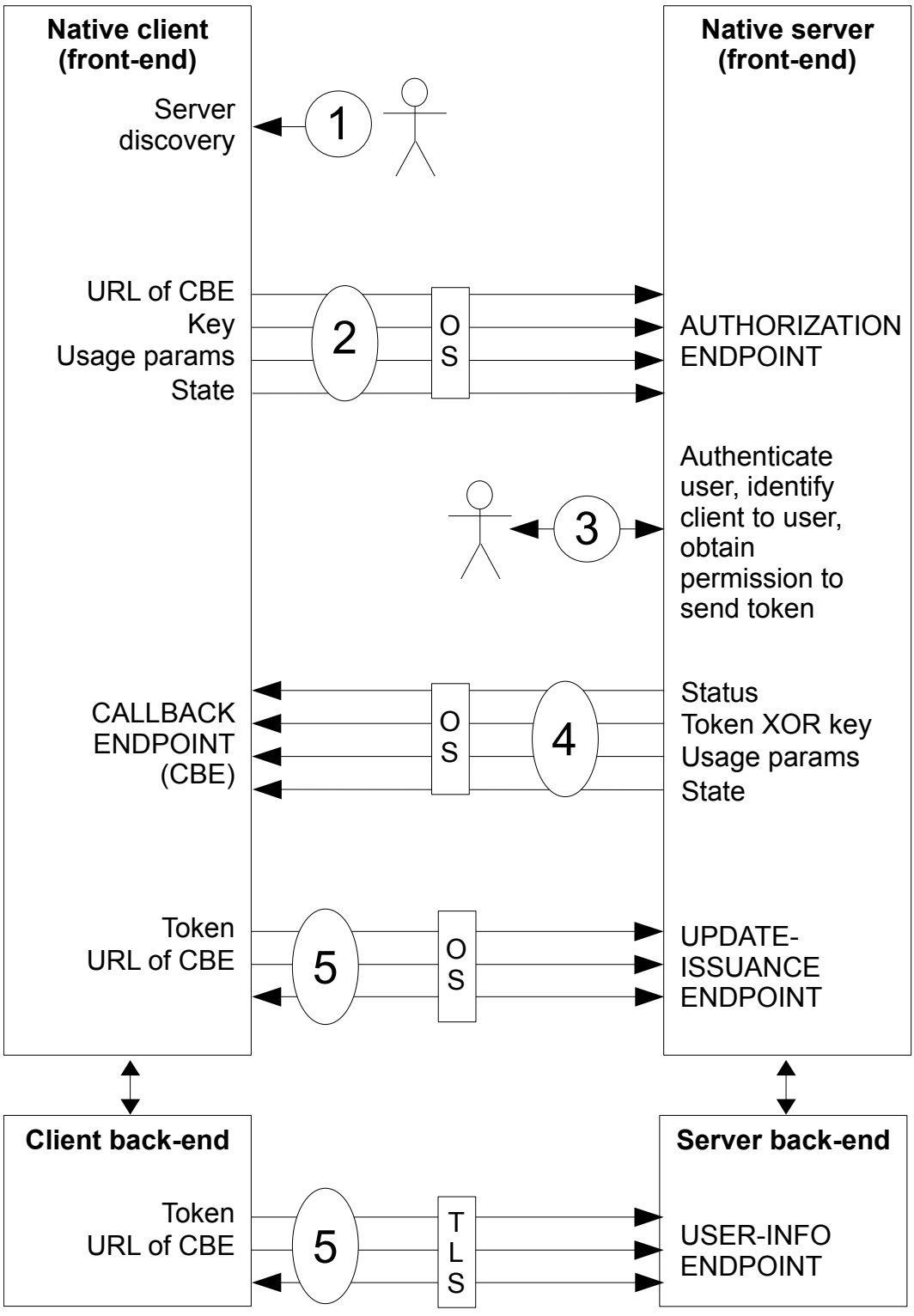

Figure 5. Social login with native client and native server

4 Protocol Flow

In this section we describe informally the protocol flow. Figure 1 illustrates the flow in general terms, and Figures 2-5 illustrate the four protocol variations corresponding to the four possible combinations of an online or native client with an online or native server. Figures 3, 4 and 5 are specific to the social login usage.

Step 1. The user initiates a social login, and the client discovers the URL of the authorization endpoint of the server that the user wants to use, as explained above in Section 3.3. When the client is online, the user interacts with the client via a browser. When the client is native, the user interacts directly with the user interface of the client.

Step 2. The client sends a request for an access token to the authorization endpoint of the server. The request includes the following parameters, some of them optional, in a query string that the client adds to the URL of the authorization endpoint to produce an authorization URL:

- The URL of the callback endpoint where the server will send the access token.
- A one-time key that the server will use to encrypt the access token by x-oring the key with the token. The key is a high-entropy random value, of same bit-length as the token.[12] For high-security usages, the client sends the key encrypted under a public key of the server, as discussed above in Section 3.6.

  The key is not strictly necessary for security in the case where both the client and the server are native (i.e. have native front-ends). However it is included for three reasons: (i) for defense in depth; (ii) to simplify client and server implementations by eliminating special cases; and (iii) to avoid intentional or inadvertant omissions of the key in cases where it is needed.
- Zero or more usage-specific parameters. In the social-login usage the client sends a parameter that lists the user profile data items that the client wishes to retrieve at the user-info endpoint.
- Optionally, a state string, described in Section 3.5, which the client uses to remember its transaction data, including the one-time key and the identity of the server discovered in Step 1, and possibly other data related to the state of the client at the time of step 2.

  If the client is native, it sends the state string to the server, which returns it in step 4. If the client is online, it stores the state string in the browser as the value of a cookie, and the browser returns it step 4.

[12] The length of the access token is to be specified by the SAAAM standard.

The state string may be a record or a reference to a record stored by the client. When the client is online, if the client stores the record internally, it is vulnerable to a denial-of-service attack by storage exhaustion; therefore the state string should be the record itself, so that the client does not have to store the record internally. When the client is native, that attack is not possible, and the state string should be a reference to a record stored by the client, to save bandwidth.

When the client is online, the state string must be stored in the browser (instead of being sent to the server), to protect against a user impersonation attack with a leaked encrypted access token, explained in Section 3.6, and against a login-as-attacker attack, explained in Section 3.9.

If the client and the server are online, the request is sent by HTTP redirection. If the client is native and the server is online, the client asks the operating system to point the browser to the authorization URL. If the client is online and the server is native, the client redirects the browser to the authorization URL, which is a native URL, causing the operating system to pass the URL to the server; the server remembers which browser it receives the URL from. If the client and the server are both native, the URL is transmitted from the client to the server by the operating system.

Step 3. When the server receives the request, it authenticates the user without asking for a password, as discussed above in Section 3.1. Then it identifies the client to the user as discussed in Section 3.4 and asks the user for permission to send the access token. The interaction with the user is mediated by a browser external to the client if the server is an online server. If the server is native, the interaction is conducted by the native front-end of the server.

Step 4. The server sends a response to the request for the access token to the callback endpoint of the client. The response includes the following parameters, some of them optional, in a query string that the server adds to the URL of the callback endpoint to produce a callback URL:

- A status indicating that the request succeeded, or why it failed.
- The access token x-ored with the one-time key.
- Zero or more usage-specific parameters. In the social-login usage the server sends as parameters the URL of the user-info endpoint, the URL of the update-issuance endpoint, and the length of time during which the access token will be honored at the endpoints.
- The state string that the client, if native, sent in Step 2. If the client is online, the state string was stored in the browser as a cookie, and the browser returns it in a Cookie HTTP header that the browser adds to the server's response.

If the client and the server are online, the response is sent by HTTP redirection. If the client is native and the server is online, the server redirects the browser to the callback URL, which is a native URL, causing the operating system to pass the URL to client. If the client is online and the server is native, the server asks the operating system to point the same browser from which it received the request in Step 2 to the callback URL. If the client and the server are both native, the URL is transmitted from the server to the client by the operating system.

Step 5. This step is repeated any number of times. Each time, the client presents the access token to a usage-specific endpoint to seek authorization for a transaction made available by an API exposed by the endpoint. In the social-login usage the server has a user-info endpoint where the client obtains user profile data, and an update-issuance endpoint where the client can issue updates on behalf of the user.

Methods of sending the access token to the usage endpoint are described above in Section 3.10. In the social-login usage, the client sends the URL of the callback endpoint along with the access token, both to the user-info endpoint and to the update-issuance endpoint. Sending the URL of the callback endpoint prevents user impersonation by a malicious client, as discussed above in Section 3.8. Sending it to the update-issuance endpoint protects a client against issuing updates that would be attributed to a different client.

Figures 3, 4 and 5 are specific to the social login usage. Figure 3 shows a native client accessing the update-issuance endpoint of an online server through the client's native front-end, but accessing the user-info endpoint through the client's online back-end over a TLS connection, using the TLS certificate presented by the endpoint to authenticate the server; the client back-end cannot trust its own front-end to authenticate the user, because the front-end is under control of the user. Figure 4 shows an online client accessing the user-info and update-issuance endpoints on the online back-end of a native server over TLS connections, with server authentication. Figure 5 shows the native front-end of a native client accessing the update-issuance endpoint on the native front-end of a native server, while the online back-end of the client accesses the user-info endpoint on the online back-end of the server over a TLS connection with server authentication.

5 Benefits

Standardization and implementation of SAAAM would provide many benefits:

- It would facilitate the implementation of social login, due to its simplicity.
- It would make social login available to all social networks, not just the top-tier ones, by freeing social login clients from the prior registration requirement of OAuth.

- It would eliminate the security vulnerabilities of existing social login protocols. In particular, it would provide secure social login functionality for Web 2.0 Javascript clients.

- It would provide secure social login functionality for native clients, on iOS, Android, and any other mobile operating systems that implement interapp communications using native URLs. A user of a native client would thus be able to log in with a social identity of her choice without compromising security. Social login is particularly useful on a smart phone because it is difficult to type a password on the tiny keyboard displayed on the touch screen of the phone.

- It would allow native clients to implement social login via a native front-end of a social network running on the same mobile device, without having to integrate social network code into the client code (as must be done today to take advantage of the Facebook apps for iOS and Android), on iOS, Android, and any mobile operating systems that implement interapp communications using native URLs.

- It would allow Web clients accessed through a mobile browser to implement social login via a native front-end of a social network running on the same mobile device, saving data bandwidth and latency, on iOS and any mobile operating systems that implement interapp communications using native URLs. This functionality does not exist today.

Furthermore, when combined with user authentication at the server using a TLS client certificate issued by the server itself, SAAAM would allow Web and native applications to rely on public key certificates for user authentication, without having to deal with certificates themselves. The combination would provide an easy-to-implement alternative to password-based authentication, and would enable social login at high levels of assurance for the first time.

References

[1] Janrain. Social Login. At http://www.janrain.com/products/engage/social-login.

[2] eMarketer. Facebook Becomes Top Choice for Social Sign-In, April 28, 2011. At http://www.emarketer.com/Article.aspx?R=1008364.

[3] W. E. Burr, D. F. Dodson, and W. T. Polk. Electronic Authentication Guideline, December 2011. Version 1.0.2. Available at http://csrc.nist.gov/publications/nistpubs/800-63/SP800-63V1_0_2.pdf.

[4] E. Hammer-Lahav, D. Recordon, and D. Hardt. The the oauth 2.0 authorization protocol draft-ietf-oauth-v2-28, June 19, 2012. Available at http://tools.ietf.org/html/draft-ietf-oauth-v2-28.

[5] Don Thibeau. OpenIDs Second Act: OpenID Connect, May 20, 2011. At http://openid.net/2011/05/20/openids-second-act-openid-connect/.

[6] eMarketer. Security Concerns Challenge Facebook Commerce, May 20, 2012. At http://www.emarketer.com/Article.aspx?R=1009080.

[7] eMarketer. Smaller Social Sites See Significant Boost in Usage, January 11, 2012. At http://www.emarketer.com/Article.aspx?R=1008776.

[8] Wikipedia. List of Social Networking Websites. At http://en.wikipedia.org/wiki/List_of_social_networking_websites.

[9] Buddy Press. A Social Network in a Box. WordPress plug-in for building a social network. At http://buddypress.org/.

[10] MonkDev. Cobblestone Community Network. At http://www.monkdevelopment.com/products/cobblestone-community-network/.

[11] T. Hardjono, M. Machulak, E. Maler, and C. Scholtz. OAuth Dynamic Client Registration Protocol draft-ietf-oauth-dyn-reg-00, April 2012. Available at http://tools.ietf.org/html/draft-ietf-oauth-dyn-reg-00.

[12] N. Sakimura, J. Bradley, and M. Jones. OpenID Connect Dynamic Client Registration 1.0 - draft 11, May 2012. Available at http://openid.net/specs/openid-connect-registration-1_0.html.

[13] OpenID Foundation. Welcome to OpenID Connect. At http://openid.net/connect/.

[14] Hamish McKenzie. Web 2.0 Is Over, All Hail the Age of Mobile, April 27, 2012. At http://gigaom.com/mobile/facebook-just-revealed-its-kryptonite-mobile/.

[15] Kevin Fitchard. Facebook Just Revealed Its Kryptonite: Mobile, February 2, 2012. At http://gigaom.com/mobile/facebook-just-revealed-its-kryptonite-mobile/.

[16] Rui Wang. [OAUTH-WG] Report an authentication issue, 14 Jun 2012 10:18:16 -0700. http://www.ietf.org/mail-archive/web/oauth/current/msg09270.html.

[17] Nat Sakimura. Re: [OAUTH-WG] Report an authentication issue, 15 Jun 2012 05:50:50 +0900. http://www.ietf.org/mail-archive/web/oauth/current/msg09273.html.

[18] Nat Sakimura. Re: [OAUTH-WG] Report an authentication issue, 16 Jun 2012 09:16:42 +0900. http://www.ietf.org/mail-archive/web/oauth/current/msg09316.html.

[19] Nov Matake. Re: [OAUTH-WG] Report an authentication issue, 16 Jun 2012 12:22:23 +0900. http://www.ietf.org/mail-archive/web/oauth/current/msg09321.html.

[20] Eran Hammer-Lahav. Valet Key for the Web, May 11 2010. At http://hueniverse.com/oauth/guide/intro/.

[21] John Bradley. The problem with OAuth for Authentication, January 28, 2012. At http://www.thread-safe.com/2012/01/problem-with-oauth-for-authentication.html.

[22] David Recordon. Message to the OAuth Working Group, available at http://www.ietf.org/mail-archive/web/oauth/current/msg05953.html.

[23] P. Moceri and T. Ruths. Cafe Cracks: Attacks on Unsecured Wireless Networks. At http://www1.cse.wustl.edu/~jain/cse571-07/cafecrack.htm.

[24] 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*, 2007. Available at http://www.adambarth.com/papers/2008/barth-jackson-mitchell-b.pdf.

[25] David P. Kormann and Aviel D. Rubin. Risks of the Passport Single Signon Protocol. *Computer Networks*, 33:51–58, 2000. Available at http://avirubin.com/passport.html.

[26] Microsoft. Windows live. At http://home.live.com/.

[27] J. Hughes et al. Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0. OASIS Standard, March 2005. Available at http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf.

[28] Tom Scavo and Scott Cantor. Shibboleth Architecture Technical Overview, Working Draft 02, June 2005. Available at http://shibboleth.internet2.edu/docs/draft-mace-shibboleth-tech-overview-latest.pdf.

[29] OpenID Foundation. OpenID Authentication 2.0 Final, December 5, 2007. At http://openid.net/specs/openid-authentication-2_0.html.

[30] Darmouth College PKI Lab. Using PKI Authentication with Shibboleth, May 2003. Available at http://www.dartmouth.edu/~pkilab/pages/ShibbAuthwithPKI.html.

[31] Apple Inc. iOS Programming Guide: Communicating with other Apps; Implementing Custom URL Schemes. At http://developer.apple.com/library/ios/#documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/AdvancedAppTricks/AdvancedAppTricks.html#//apple_ref/doc/uid/TP40007072-CH7-SW2.

[32] Google. IntentFilter. At http://developer.android.com/reference/android/content/IntentFilter.html.

[33] MSDN. Registering an Application to a URL Protocol, April 2011. At http://msdn.microsoft.com/en-us/library/aa767914%28v=VS.85%29.aspx.

[34] IANA. URI Schemes. At http://www.iana.org/assignments/uri-schemes.html.

[35] M. Nottingham and E. Hammer-Lahav. Defining Well-Known Uniform Resource Identifiers (URIs). Available at http://tools.ietf.org/html/rfc5785.

[36] Google. Account Chooser. At http://http://openid.net/wg/ac/.