A Comprehensive Formal Security Analysis of OAuth 2.0

Daniel Fett
University of Trier, Germany
fett@uni-trier.de

Ralf Küsters
University of Trier, Germany
kuesters@uni-trier.de

Guido Schmitz
University of Trier, Germany
schmitzg@uni-trier.de

ABSTRACT

The OAuth 2.0 protocol is one of the most widely deployed authorization/single sign-on (SSO) protocols and also serves as the foundation for the new SSO standard OpenID Connect. Despite the popularity of OAuth, so far analysis efforts were mostly targeted at finding bugs in specific implementations and were based on formal models which abstract from many web features or did not provide a formal treatment at all.

In this paper, we carry out the first extensive formal analysis of the OAuth 2.0 standard in an expressive web model. Our analysis aims at establishing strong authorization, authentication, and session integrity guarantees, for which we provide formal definitions. In our formal analysis, all four OAuth grant types (authorization code grant, implicit grant, resource owner password credentials grant, and the client credentials grant) are covered. They may even run simultaneously in the same and different relying parties and identity providers, where malicious relying parties, identity providers, and browsers are considered as well. Our modeling and analysis of the OAuth 2.0 standard assumes that security recommendations and best practices are followed in order to avoid obvious and known attacks.

When proving the security of OAuth in our model, we discovered four attacks which break the security of OAuth. The vulnerabilities can be exploited in practice and are present also in OpenID Connect.

We propose fixes for the identified vulnerabilities, and then, for the first time, actually prove the security of OAuth in an expressive web model. In particular, we show that the fixed version of OAuth (with security recommendations and best practices in place) provides the authorization, authentication, and session integrity properties we specify.

1. INTRODUCTION

The OAuth 2.0 authorization framework [20] defines a web-based protocol that allows a user to grant web sites access to her resources (data or services) at other web sites (*authorization*). The former web sites are called relying parties (RP) and the latter are called identity providers (IdP).[1] In practice, OAuth 2.0 is often used for *authentication* as well. That is, a user can log in at an RP using her identity managed by an IdP (single sign-on, SSO).

Authorization and SSO solutions have found widespread adoption in the web over the last years, with OAuth 2.0 being one of the most popular frameworks. OAuth 2.0, in the following often simply called *OAuth*,[2] is used by identity providers such as Amazon, Facebook, Google, Microsoft, Yahoo, GitHub, LinkedIn, StackExchange, and Dropbox. This enables billions of users to log in at millions of RPs or share their data with these [35], making OAuth one of the most used single sign-on systems on the web.

OAuth is also the foundation for the new single sign-on protocol OpenID Connect, which is already in use and actively supported by PayPal ("Log In with PayPal"), Google, and Microsoft, among others. Considering the broad industry support for OpenID Connect, a widespread adoption of OpenID Connect in the next years seems likely. OpenID Connect builds upon OAuth and provides clearly defined interfaces for user authentication and additional (optional) features, such as dynamic identity provider discovery and relying party registration, signing and encryption of messages, and logout.

In OAuth, the interactions between the user and her browser, the RP, and the IdP can be performed in four different flows, or *grant types*: authorization code grant, implicit grant, resource owner password credentials grant, and the client credentials grant (we refer to these as *modes* in the following). In addition, all of these modes provide further options.

The goal of this work is to provide an in-depth security analysis of OAuth. Analyzing the security of OAuth is a challenging task, on the one hand due to the various modes and options that OAuth provides, and on the other hand due to the inherent complexity of the web.

So far, most analysis efforts regarding the security of OAuth were targeted towards finding errors in specific implementations [6, 10, 25, 33, 34, 36, 38], rather than the comprehensive analysis of the standard itself. Probably the most detailed formal analysis carried out on OAuth so far is the one in [6]. However, none of the existing analysis efforts of OAuth account for all modes of OAuth running simultaneously, which may potentially introduce new security risks. In fact, many existing approaches analyze only the authorization code mode and the implicit mode of OAuth. Also, importantly, there are no analysis efforts that are based on a comprehensive formal web model (see below), which, however, is essential to rule

CCS'16, October 24 – 28, 2016, Vienna, Austria

DOI: http://dx.doi.org/10.1145/2976749.2978385

[1] Following the OAuth 2.0 terminology, IdPs are called *authorization servers* and *resource servers*, RPs are called *clients*, and users are called *resource owners*. Here, however, we stick to the more common terms mentioned above.

[2] Note that in this document, we consider only OAuth 2.0, which is very different to its predecessor, OAuth 1.0(a).

out security risks that arise when running the protocol in the context of common web technologies (see Section 6 for a more detailed discussion of related work).

Contributions of this Paper. We perform the first extensive formal analysis of the OAuth 2.0 standard for all four modes, which can even run simultaneously within the same and different RPs and IdPs, based on a comprehensive web model which covers large parts of how browsers and servers interact in real-world setups. Our analysis also covers the case of malicious IdPs, RPs, and browsers/users.

Formal model of OAuth. Our formal analysis of OAuth uses an expressive Dolev-Yao style model of the web infrastructure [14] proposed by Fett, Küsters, and Schmitz (FKS). The FKS model has already been used to analyze the security of the BrowserID single sign-on system [14, 15] as well as the security and privacy of the SPRESSO single sign-on system [16]. This web model is designed independently of a specific web application and closely mimics published (de-facto) standards and specifications for the web, for instance, the HTTP/1.1 and HTML5 standards and associated (proposed) standards. It is the most comprehensive web model to date. Among others, HTTP(S) requests and responses, including several headers, such as cookie, location, strict transport security (STS), and origin headers, are modeled. The model of web browsers captures the concepts of windows, documents, and iframes, including the complex navigation rules, as well as new technologies, such as web storage and web messaging (via postMessage). JavaScript is modeled in an abstract way by so-called *scripts* which can be sent around and, among others, can create iframes and initiate XMLHTTPRequests (XHRs). Browsers may be corrupted dynamically by the adversary.

Using the generic FKS model, we build a formal model of OAuth, closely following the OAuth 2.0 standard (RFC6749 [20]). Since this RFC does not fix all aspects of the protocol and in order to avoid known implementation attacks, we use the OAuth 2.0 security recommendations (RFC6819 [26]), additional RFCs and OAuth Working Group drafts (e.g., RFC7662 [30], [8]) and current web best practices (e.g., regarding session handling) to obtain a model of OAuth with state-of-the-art security features in place, while making as few assumptions as possible. Moreover, as mentioned above, our model includes RPs and IdPs that (simultaneously) support all four modes and can be dynamically corrupted by the adversary. Also, we model all configuration options of OAuth (see Section 2).

Formalization of security properties. Based on this model of OAuth, we provide three central security properties of OAuth: authorization, authentication, and session integrity, where session integrity in turn is concerned with both authorization and authentication.

Attacks on OAuth 2.0 and fixes. While trying to prove these properties, we discovered four attacks on OAuth. In the first attack, which breaks the authorization and authentication properties, IdPs inadvertently forward user credentials (i.e., username and password) to the RP or the attacker. In the second attack (IdP mix-up), a network attacker playing the role of an IdP can impersonate any victim. This severe attack, which again breaks the authorization and authentication properties, is caused by a logical flaw in the OAuth 2.0 protocol. Two further attacks allow an attacker to force a browser to be logged in under the attacker's name at an RP or force an RP to use a resource of the attacker instead of a resource of the user, breaking the session integrity property. We have verified all four attacks on actual implementations of OAuth and OpenID Connect. We present our attacks on OAuth in detail in Section 3. In our technical report [17], we show how the attacks can be exploited in OpenID Connect. We also show how the attacks can be fixed by changes that are easy to implement in new and existing deployments of OAuth and OpenID Connect.

We notified the respective working groups, who confirmed the attacks and that changes to the standards/recommendations are needed. The IdP mix-up attack already resulted in a draft of a new RFC [22].

Formal analysis of OAuth 2.0. Using our model of OAuth with the fixes in place, we then were able to prove that OAuth satisfies the mentioned security properties. This is the first proof which establishes central security properties of OAuth in a comprehensive and expressive web model (see also Section 6).

We emphasize that, as mentioned before, we model OAuth with security recommendations and best practices in place. As discussed in Section 5, implementations not following these recommendations and best practices may be vulnerable to attacks. In fact, many such attacks on specific implementations have been pointed out in the literature (e.g., [6, 10, 20, 25, 26, 36, 37]). Hence, our results also provide guidelines for secure OAuth implementations.

We moreover note that, while these results provide strong security guarantees for OAuth, they do not directly imply security of OpenID Connect because OpenID Connect adds specific details on top of OAuth. We leave a formal analysis of OpenID Connect to future work. The results obtained here can serve as a good foundation for such an analysis.

Structure of this Paper. In Section 2, we provide a detailed description of OAuth 2.0 using the authorization code mode as an example. In Section 3, we present the attacks that we found during our analysis. An overview of the FKS model we build upon in our analysis is provided in Section 4, with the formal analysis of OAuth presented in Section 5. Related work is discussed in Section 6. We conclude in Section 7. Full details, including how the attacks can be applied to OpenID Connect, further details on our model of OAuth, and our security proof, can be found in our technical report [17].

2. OAUTH 2.0

In this section, we provide a description of the OAuth authorization code mode, with the other three modes explained only briefly. In our technical report [17], we provide a detailed description of the remaining three modes (grant types).

OAuth was first intended for *authorization*, i.e., users authorize RPs to access user data (called *protected resources*) at IdPs. For example, a user can use OAuth to authorize services such as IFTTT[3] to access her (private) timeline on Facebook. In this case, IFTTT is the RP and Facebook the IdP.

Roughly speaking, in the most common modes, OAuth works as follows: If a user wants to authorize an RP to access some of the user's data at an IdP, the RP redirects the user (i.e., the user's browser) to the IdP, where the user authenticates and agrees to grant the RP access to some of her user data at the IdP. Then, along with some token (an *authorization code* or an *access token*) issued by the IdP, the user is redirected back to the RP. The RP can then use the token as a credential at the IdP to access the user's data at the IdP.

OAuth is also commonly used for *authentication*, although it was not designed with authentication in mind. A user can, for example, use her Facebook account, with Facebook being the IdP, to log in at the social network Pinterest (the RP). Typically, in order to log in, the user authorizes the RP to access a unique user identifier at the IdP. The RP then retrieves this identifier and considers this user to be logged in.

[3]IFTTT (*If This Then That*) is a web service which can be used to automate actions: IFTTT is triggered by user-defined events (e.g., Twitter messages) and carries out user-defined tasks (e.g., posting on the user's Facebook wall).

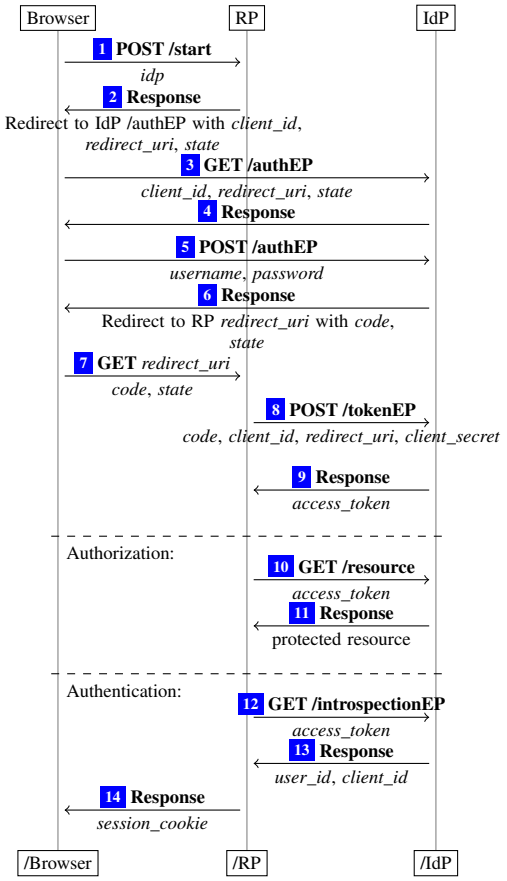

Figure 1: OAuth 2.0 authorization code mode. Note that data depicted below the arrows is either transferred in URI parameters, HTTP headers, or POST bodies.

Before an RP can interact with an IdP, the RP needs to be registered at the IdP. The details of the registration process are out of the scope of the OAuth protocol. In practice, this process is usually a manual task. During the registration process, the IdP assigns credentials to the RP: a public OAuth client id and (optionally) a client secret. (Recall that in the terminology of the OAuth standard the term "client" stands for RP.) The RP may later use the client secret (if issued) to authenticate to the IdP.

Also, an RP registers one or more *redirection endpoint* URIs (located at the RP) at an IdP. As we will see below, in some OAuth modes, the IdP redirects the user's browser to one of these URIs. Note that (depending on the implementation of an IdP) an RP may also register a pattern as a redirect URI and then specify the exact redirect URI during the OAuth run.

In all modes, OAuth provides several options, such as those mentioned above. For brevity of presentation (and in contrast to our analysis), in the following descriptions, we consider only a specific set of options. For example, we assume that an RP always provides a redirect URI and shares an OAuth client secret with the IdP.

Authorization Code Mode. When the user tries to authorize an RP to access her data at an IdP or to log in at an RP, the RP first redirects the user's browser to the IdP. The user then authenticates to the IdP, e.g., by providing her user name and password, and finally is redirected back to the RP along with an *authorization code* generated by the IdP. The RP can now contact the IdP with this authorization code (along with the client id and client secret) and receive an *access token*, which the RP in turn can use as a credential to access the user's protected resources at the IdP.

Step-by-Step Protocol Flow. In what follows, we describe the protocol flow of the authorization code mode step-by-step (see also Figure 1). First, the user starts the OAuth flow, e.g., by clicking on a button to select an IdP, resulting in request 1 being sent to the RP. The RP selects one of its redirection endpoint URIs *redirect_uri* (which will be used later in 7) and a value *state* (which will serve as a token to prevent CSRF attacks). The RP then redirects the browser to the so-called *authorization endpoint* URI at the IdP in 2 and 3 with its *client_id*, *redirect_uri*, and *state* appended as parameters to the URI. The IdP then prompts the user to provide her username and password in 4. The user's browser sends this information to the IdP in 5. If the credentials are correct, the IdP creates a nonce *code* (the authorization code) and redirects the user's browser to RP's redirection endpoint URI *redirect_uri* in 6 and 7 with *code* and *state* appended as parameters to the URI. If *state* is the same as above, the RP contacts the IdP in 8 and provides *code*, *client_id*, *client_secret*, and *redirect_uri*. Then the IdP checks whether this information is correct, i.e., it checks that *code* was issued for the RP identified by *client_id*, that *client_secret* is the secret for *client_id*, that *redirect_uri* coincides with the one in Step 2, and that *code* has not been redeemed before. If these checks are successful, the IdP issues an access token *access_token* in 9. Now, the RP can use *access_token* to access the user's protected resources at the IdP (authorization) or log in the user (authentication), as described next.

When OAuth is used for *authorization*, the RP uses the access token to view or manipulate the protected resource at the IdP (illustrated in Steps 10 and 11).

For *authentication*, the RP fetches a user id (which uniquely identifies the user at the IdP) using the access token, Steps 12 and 13. The RP then issues a session cookie to the user's browser as shown in 14.[4]

Tracking User Intention. Note that in order for an RP which supports multiple IdPs to process Step 7, the RP must know which IdP a user wanted to use for authorization. There are two different approaches to this used in practice: First, the RP can use different redirection URIs to distinguish different IdPs. We call this *naïve user intention tracking*. Second, the RP can store the user intention in a session after Step 1 and use this information later. We call this *explicit user intention tracking*. The same applies to the implicit mode of OAuth presented below.

Implicit Mode. This mode is similar to the authorization code mode, but instead of providing an authorization code, the IdP directly delivers an access token to the RP via the user's browser.

More specifically, in the implicit mode, Steps 1–5 (see Figure 1) are the same as in the authorization code mode. Instead of creating an authorization code, the IdP issues an access token right away and redirects the user's browser to RP's redirection endpoint with the access token contained in the fragment of the URI. (Recall that a fragment is a special part of a URI indicated by the '#' symbol.)

As fragments are not sent in HTTP requests, the access token is not immediately transferred when the browser contacts the RP. Instead, the RP needs to use a JavaScript to retrieve the contents of the fragment. Typically, such a JavaScript is sent in RP's answer at the redirection endpoint. Just as in the authorization code mode, the

[4] Authentication is not part of RFC6749, but this method for authentication is commonly used in practice, for example by Amazon, Facebook, LinkedIn, and StackExchange, and is also defined in OpenID Connect [31].

RP can now use the access token for authorization or authentication (analogously to Steps 10–14 of Figure 1).[5]

Resource Owner Password Credentials Mode. In this mode, the user gives her credentials for an IdP directly to an RP. The RP can then authenticate to the IdP on the user's behalf and retrieve an access token. This mode is intended for highly-trusted RPs, such as the operating system of the user's device or highly-privileged applications, or if the previous two modes are not possible to perform (e.g., for applications without a web browser).

Client Credentials Mode. In contrast to the modes shown above, this mode works without the user's interaction. Instead, it is started by an RP in order to fetch an access token to access the resources of RP at an IdP. For example, Facebook allows RPs to use the client credentials mode to obtain an access token to access reports of their advertisements' performance.

3. ATTACKS

As mentioned in the introduction, while trying to prove the security of OAuth based on the FKS web model and our OAuth model, we found four attacks on OAuth, which we call *307 redirect attack*, *IdP mix-up attack*, *state leak attack*, and *naïve RP session integrity attack*, respectively. In this section, we provide detailed descriptions of these attacks along with easily implementable fixes. Our formal analysis of OAuth (see Section 5) then shows that these fixes are indeed sufficient to establish the security of OAuth. The attacks also apply to OpenID Connect (see Section 3.5). Figure 2 provides an overview of where the attacks apply. We have verified our attacks on actual implementations of OAuth and OpenID Connect and reported the attacks to the respective working groups who confirmed the attacks (see Section 3.6).

3.1 307 Redirect Attack

In this attack, which breaks our authorization and authentication properties (see Section 5.2), the attacker (running a malicious RP) learns the user's credentials when the user logs in at an IdP that uses the wrong HTTP redirection status code. While the attack itself is based on a simple error, to the best of our knowledge, this is the first description of an attack of this kind.

Assumptions. The main assumptions are that (1) the IdP that is used for the login chooses the 307 HTTP status code when redirecting the user's browser back to the RP (Step 6 in Figure 1), and (2) the IdP redirects the user immediately after the user entered her credentials (i.e., in the response to the HTTP POST request that contains the form data sent by the user's browser).

Assumption (1). This assumption is reasonable because neither the OAuth standard [20] nor the OAuth security considerations [26] (nor the OpenID Connect standard [31]) specify the exact method of how to redirect. The OAuth standard rather explicitly permits any HTTP redirect:

> While the examples in this specification show the use of the HTTP 302 status code, any other method available via the user-agent to accomplish this redirection is allowed and is considered to be an implementation detail.

Assumption (2). This assumption is reasonable as many examples for redirects immediately after entering the user credentials can be found in practice, for example at github.com (where, however, assumption (1) is not satisfied.)

Attack. When a user uses the authorization code or implicit mode of OAuth to log in at a *malicious* RP, then she is redirected to the IdP and prompted to enter her credentials. The IdP then receives these credentials from the user's browser in a POST request. It checks the credentials and redirects the user's browser to the RP's redirection endpoint in the response to the POST request. Since the 307 status code is used for this redirection, the user's browser will send a POST request to RP that contains all form data from the previous request, including the user credentials. Since the RP is run by the attacker, he can use these credentials to impersonate the user.

Fix. Contrary to the current wording in the OAuth standard, the exact method of the redirect is not an implementation detail but essential for the security of OAuth. In the HTTP standard [18], only the 303 redirect is defined unambiguously to drop the body of an HTTP POST request. Therefore, the OAuth standard should require 303 redirects for the steps mentioned above in order to fix this problem.

3.2 IdP Mix-Up Attack

In this attack, which breaks our authorization and authentication properties (see Section 5.2), the attacker confuses an RP about which IdP the user chose at the beginning of the login/authorization process in order to acquire an authentication code or access token which can be used to impersonate the user or access user data.

This attack applies to the authorization code mode and the implicit mode of OAuth when explicit user intention tracking[6] is used by the RP. To launch the attack, the attacker manipulates the first request of the user such that the RP thinks that the user wants to use an identity managed by an IdP of the attacker (AIdP) while the user instead wishes to use her identity managed by an honest IdP (HIdP). As a result, the RP sends the authorization code or the access token issued by HIdP to the attacker. The attacker then can use this information to login at the RP under the user's identity (managed by HIdP) or access the user's protected resources at HIdP.

We here present the attack in the authorization code mode. In the implicit mode, the attack is very similar and is shown in detail in [17].

Assumptions. For the IdP mix-up attack to work, we need three assumptions that we further discuss below: (1) the presence of a network attacker who can manipulate the request in which the user sends her identity to the RP as well as the corresponding response to this request (see Steps 1 and 2 in Figure 1), (2) an RP which allows users to log in with identities provided by (some) HIdP and identities provided by AIdP, and (3) an RP that uses explicit user intention tracking and issues the same redirection URI to all IdPs.[7] We emphasize that we do not assume that the user sends any secret (such as passwords) over an unencrypted channel.

Assumption (1). It would be unrealistic to assume that a network attacker can never manipulate Steps 1 and 2 in Figure 1.

First, these messages are sent between the user and the RP, i.e., the attacker does not need to intercept server-to-server communication. He could, e.g., use ARP spoofing in a wifi network to mount the attack.

[5] The response from the IdP in Step 13 includes the RP's OAuth client id, which is checked by the RP when *authenticating* a user (cf. RFC7662 [30]). This check prevents re-use of access tokens across RPs in the OAuth implicit mode, as explained in [37]. This check is not needed for authorization.

[6] Recall the meaning of "user intention tracking" from Section 2.

[7] Alternatively, the attack would work if the RP issues different redirection URIs to different IdPs, but treats them as the same URI.

	attack on OAuth		applicable to OpenID Connect		
	auth code mode	implicit mode	auth code mode	implicit mode	hybrid mode
307 Redirect Attack	az + an	az + an	az + an	az + an	az + an
IdP Mix-Up Attack	az* + an	az + an	az* + an	–	az + an**
State Leak Attack	si	si	si	si	si
Naïve RP Session Integrity Attack	si	si	si	si	si

az: breaks authorization. **an:** breaks authentication. **si:** breaks session integrity. **–:** not applicable. * if client secrets are not used. ** restriction: if client secrets are used, either authorization or authentication is broken, depending on implementation details.

Figure 2: Overview of attacks on OAuth 2.0 and OpenID Connect.

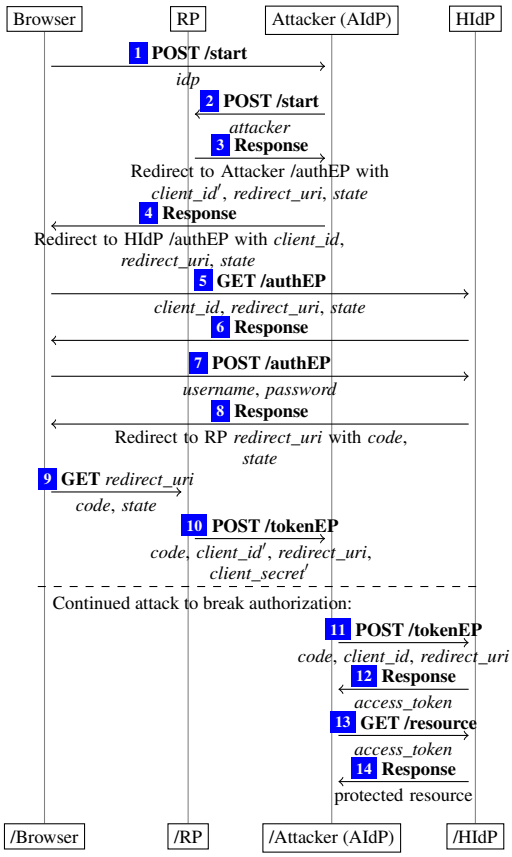

Figure 3: Attack on OAuth 2.0 authorization code mode

Second, the need for HTTPS for these steps is not obvious to users or RPs, and the use of HTTPS is not suggested by the OAuth security recommendations, since the user only selects an IdP at this point; credentials are not transferred.

Third, even if an RP intends to use HTTPS also for the first request (as in our model), it has to protect itself against TLS stripping by adding the RP domain to a browser preloaded Strict Transport Security (STS) list [11]. Other mitigations, such as the STS header, can be circumvented (see [32]), and do not work on the very first connection between the user's browser and RP. For example, when a user enters the address of an RP into her browser, browsers by default try unencrypted connections. It is therefore unrealistic to assume that all RPs are always protected against TLS stripping.

Our formal analysis presented in Section 5 shows that OAuth can be operated securely even if no HTTPS is used for the initial request (given that our fix, presented below, is applied).

Assumption (2). RPs may use different IdPs, some of which might be malicious, and hence, OAuth should provide security in this case. Using a technique called dynamic client registration, OAuth RPs can even allow the ad-hoc use of any IdP, including malicious ones. This is particularly relevant in OpenID Connect, where this technique was first implemented.

Assumption (3). Typically, RPs that use explicit user intention tracking do not register different redirection URIs for different IdPs, as in this case the RP records the IdP a user wants to authenticate with. In particular, for RPs that allow for dynamic registration, using the same URI is an obvious implementation choice. This is for example the case in the OAuth/OpenID Connect implementations *mod_auth_openidc* and *pyoidc* (see below).

Attack on Authorization Code Mode. We now describe the IdP Mix-Up attack on the OAuth authorization code mode. As mentioned, a very similar attack also applies to the implicit mode. Both attacks also work if IdP supports just one of these two modes.

The IdP mix-up attack for the authorization code mode is depicted in Figure 3. Just as in a regular flow, the attack starts when the user selects that she wants to log in using HIdP (Step 1 in Figure 3). Now, the attacker intercepts the request intended for the RP and modifies the content of this request by replacing HIdP by AIdP.[8] The response of the RP 3 (containing a redirect to AIdP) is then again intercepted and modified by the attacker such that it redirects the user to HIdP 4. The attacker also replaces the OAuth client id of the RP at AIdP with the client id of the RP at HIdP (which is public information). (Note that we assume that from this point on, in accordance with the OAuth security recommendations, the communication between the user's browser and HIdP and the RP is encrypted by using HTTPS, and thus, cannot be inspected or altered by the attacker.) The user then authenticates to HIdP and is redirected back to the RP 8. The RP thinks, due to Step 2 of the attack, that the nonce *code* contained in this redirect was issued by AIdP, rather than HIdP. The RP therefore now tries to redeem this nonce for an access token at AIdP 10, rather than HIdP. This leaks *code* to the attacker.

Breaking Authorization. If HIdP has not issued an OAuth client secret to RP during registration, the attacker can now redeem *code* for an access token at HIdP (in 11 and 12).[9] This access token allows the attacker to access protected resources of the user at HIdP. This breaks the authorization property (see Section 5.2). We note that at this point, the attacker might even provide false information

[8] At this point, the attacker could also read the session id for the user's session at RP. Our attack, however, is not based on this possibility and works even if the RP changes this session id as soon as the user is logged in and the connection is protected by HTTPS (a best practice for session management).

[9] In the case that RP has to provide a client secret, this would not work in this mode (see also Figure 2). Recall that in this mode, client secrets are optional.

about the user or her protected resources to the RP: he could issue a self-created access token which RP would then use to access such information at the attacker.

Breaking Authentication. To break the authentication property (see Section 5.2) and impersonate the honest user, the attacker, after obtaining *code* in Step 10, starts a new login process (using his own browser) at the RP. He selects HIdP as the IdP for this login process and receives a redirect to HIdP, which he ignores. This redirect contains a cookie for a new login session and a fresh state parameter. The attacker now sends *code* to the RP imitating a real login (using the cookie and fresh state value from the previous response). The RP then retrieves an access token at HIdP using *code* and uses this access token to fetch the (honest) user's id. Being convinced that the attacker owns the honest user's account, the RP issues a session cookie for this account to the attacker. As a result, the attacker is logged in at the RP under the honest user's id. (Note that the attacker does not learn an access token in this case.)

Variant. There is also a variant of the IdP mix-up attack that only requires a web attacker (which does not intercept and manipulate network messages). In this variant, the user wants to log in with AIdP, but is redirected by AIdP to log in at HIdP; a fact a vigilant user might detect.

In detail, the first four steps in Figure 3 are replaced by the following steps: First, the user starts a new OAuth flow with RP using AIdP. She is then redirected by RP to AIdP's authorization endpoint. Now, instead of prompting the user for her password, AIdP redirects the user to HIdP's authorization endpoint. (Note that, as above, in this step, the attacker uses the state value he received from the browser plus the client id of RP at HIdP.) From here on, the attack proceeds exactly as in Step 5 in Figure 3.

Related Attacks. An attack in the same class, *cross social-network request forgery*, was outlined by Bansal, Bhargavan, Delignat-Lavaud, and Maffeis in [6]. It applies to RPs with naïve user intention tracking (rather than explicit user intention tracking assumed in our IdP mix-up attack above) in combination with IdPs, such as Facebook, that only loosely check the redirect URI.[10] Our IdP mix-up attack works even if an IdP strictly checks redirect URIs. While the attack in [6] is described in the context of concrete social network implementations, our findings show that this class of attacks is not merely an implementation error, but a more general problem in the OAuth standard. This was confirmed by the IETF OAuth Working Group, who, as mentioned, are in the process of amending the OAuth standard according to our fixes (see Section 3.6).

Another attack with a similar outcome, called *Malicious Endpoints Attack*, leveraging the OpenID Connect Discovery mechanism and therefore limited to OpenID Connect, was described in [27]. This attack assumes a CSRF vulnerability on the RP's side.

Fix. A fundamental problem in the authorization code and implicit modes of the OAuth standard is a lack of reliable information in the redirect in Steps 6 and 7 in Figure 1 (even if HTTPS is used). The RP does not receive information from where the redirect was initiated (when explicit user intention tracking is used) or receives information that can easily be spoofed (when naïve user intention tracking is used with IdPs such as Facebook). Hence, the RP cannot check whether the information contained in the redirect stems from the IdP that was indicated in Step 1.

Our fix therefore is to include the identity of the IdP in the redirect URI in some form that cannot be influenced by the attacker, e.g., using a new URI parameter. Each IdP should add such a parameter to the redirect URI.[11] The RP can then check that the parameter contains the identity of the IdP it expects to receive the response from. (This could be used with either naïve or explicit user intention tracking, but to mitigate the *naïve RP session integrity attack* described below, we advise to use explicit user intention tracking only, see below.)

We show in Section 5 that this fix is indeed sufficient to mitigate the IdP mix-up attack (as well as the attacks pointed out in [6, 27]).

3.3 State Leak Attack

Using the state leak attack, an attacker can force a browser to be logged in under the attacker's name at an RP or force an RP to use a resource of the attacker instead of a resource of the user. This attack, which breaks our session integrity property (see Section 5.2), enables what is often called session swapping or login CSRF [7].

Attack. After the user has authenticated to the IdP in the authorization code mode, the user is redirected to RP (Step 7 in Figure 1). This request contains state and code as parameters. The response to this request (Step 14) can be a page containing a link to the attacker's website or some resource located at the attacker's website. When the user clicks the link or the resource is loaded, the user's browser sends a request to the attacker. This request contains a Referer header with the full URI of the page the user was redirected to, which in this case contains state and code.

As the state value is supposed to protect the browser's session against CSRF attacks, the attacker can now use the leaked state value to perform a CSRF attack against the victim. For example, he can redirect the victim's browser to the RP's redirection endpoint (again) and by this, overwrite the previously performed authorization. The user will then be logged in as the attacker.

Given the history of OAuth, leaks of sensitive data through the referrer header are not surprising. For example, the fact that the authorization code can leak through the Referer header was described as an attack (in a similar setting) in [21]. Since the authorization code is single-use only [20], it might already be redeemed by the time it is received by the attacker. State, however, is not limited to single use, making this attack easier to exploit in practice. Stealing the state value through the Referer header to break session integrity has not been reported as an attack before, as was confirmed by the IETF OAuth Working Group.

State Leak at IdPs. A variant of this attack exists if the login page at an IdP contains links to external resources. If the user visits this page to authenticate at the IdP and the browser follows links to external resources, the state is transferred in the Referer header. This variant is applicable to the authorization code mode and the implicit mode.

Fix. We suggest to limit state to a single use and to use the recently introduced *referrer policies* [13] to avoid leakage of the state (or code) to the attacker. Using referrer policies, a web server can instruct a web browser to (partially or completely) suppress the Referer header when the browser follows links in or loads resources for some web page. The Referer header can be blocked entirely, or it can, for example, be stripped down to the origin of the URI of the web page. Referrer policies are supported by all modern browsers.

Our OAuth model includes this fix (such that only the origin is permitted in the Referer header for links on web pages of RPs/IdPs) and our security proof shows its effectiveness (see Section 5). The

[10] Facebook, by default, only checks the origin of redirect URIs.

[11] The OAuth Working Group indeed created a draft for an RFC [22] that includes this fix, where this parameter is called *iss* (issuer).

fix also protects the authorization code from leaking as in the attack described in [21].

3.4 Naïve RP Session Integrity Attack

This attack again breaks the session integrity property for RPs, where here we assume an RP that uses *naïve user intention tracking*.[12] (Note that we may still assume that the OAuth state parameter is used, i.e., RP is not necessarily stateless.)

Attack. First, an attacker starts a session with HIdP (an honest IdP) to obtain an authorization code or access token for his own account. Next, when a user wants to log in at some RP using AIdP (an IdP controlled by the attacker), AIdP redirects the user back to the redirection URI of HIdP at RP. AIdP attaches to this redirection URI the state issued by RP, and the code or token obtained from HIdP. Now, since RP performs naïve user intention tracking only, the RP then believes that the user logged in at HIdP. Hence, the user is logged in at RP using the attacker's identity at HIdP or the RP accesses the attacker's resources at HIdP believing that these resources are owned by the user.

Fix. The fix against the IdP mix-up attack (described above) does not work in this case: Since RP does not track where the user wanted to log in, it has to rely on parameters in the redirection URI which the attacker can easily spoof. Instead, we propose to always use explicit user intention tracking.

3.5 Implications to OpenID Connect

OpenID Connect [31] is a standard for authentication built on top of the OAuth protocol. Among others, OpenID Connect is used by PayPal, Google, and Microsoft.

All four attacks can be applied to OpenID Connect as well. We here outline OpenID Connect and how the attacks apply to this protocol. A detailed description can be found in [17].

OpenID Connect extends OAuth in several ways, e.g., by additional security measures. OpenID Connect defines an *authorization code mode*, an *implicit mode*, and a *hybrid mode*. The former two are based on the corresponding OAuth modes and the latter is a combination of the two modes.

307 Redirect, State Leak, Naïve RP Session Integrity Attacks. All three attacks apply to OpenID Connect in exactly the same way as described above. The vulnerable steps are identical.

IdP Mix-Up Attack. In OpenID Connect, the mix-up attack applies to the authorization code mode and the hybrid mode. In the authorization code mode, the attack is very similar to the one on the OAuth authorization code mode. In the hybrid mode, the attack is more complicated as additional security measures have to be circumvented by the attacker. In particular, it must be ensured that the RP does not detect that the issuer of the id token, a signed cryptographic document used in OpenID Connect, is not the honest IdP. Interestingly, in the hybrid mode, depending on an implementation detail of the RP, either authorization or authentication is broken (or both if no client secret is used).

3.6 Verification and Disclosure

We verified the IdP mix-up and 307 redirect attacks on the Apache web server module *mod_auth_openidc*, an implementation of an OpenID Connect (and therefore also OAuth) RP. We also verified the IdP mix-up attack on the python implementation *pyoidc*. We verified the state leak attack on the current version of the Facebook PHP SDK and the naïve RP session integrity attack on nytimes.com.[13]

We reported all attacks to the OAuth and OpenID Connect working groups who confirmed the attacks. The OAuth working group invited us to present our findings to them and prepared a draft for an RFC that mitigates the IdP mix-up attack (using the fix described in Section 3.2) [22]. Fixes regarding the other attacks are currently under discussion. We also notified nytimes.com, Facebook, and the developers of *mod_auth_openidc* and *pyoidc*.

4. FKS MODEL

Our formal security analysis of OAuth is based on a slightly extended version (see Section 5.1) of the FKS model, a general Dolev-Yao (DY) style web model proposed by Fett et al. in [14, 16]. This model is designed independently of a specific web application and closely mimics published (de-facto) standards and specifications for the web, for example, the HTTP/1.1 and HTML5 standards and associated (proposed) standards. The FKS model defines a general communication model, and, based on it, web systems consisting of web browsers, DNS servers, and web servers as well as web and network attackers. Here, we only briefly recall the FKS model (see [14, 16] for a full description, comparison with other models, and a discussion of its limitations); see also [17].

Communication Model. The main entities in the model are *(atomic) processes*, which are used to model browsers, servers, and attackers. Each process listens to one or more (IP) addresses. Processes communicate via *events*, which consist of a message as well as a receiver and a sender address. In every step of a run, one event is chosen non-deterministically from a "pool" of waiting events and is delivered to one of the processes that listens to the event's receiver address. The process can then handle the event and output new events, which are added to the pool of events, and so on.

As usual in DY models (see, e.g., [1]), messages are expressed as formal terms over a signature Σ. The signature contains constants (for (IP) addresses, strings, nonces) as well as sequence, projection, and function symbols (e.g., for encryption/decryption and signatures). For example, in the web model, an HTTP request is represented as a term r containing a nonce, an HTTP method, a domain name, a path, URI parameters, headers, and a message body. For example, a request for the URI http://example.com/s?p=1 is represented as

$$r := \langle \mathtt{HTTPReq}, n_1, \mathtt{GET}, \mathtt{example.com}, /\mathtt{s}, \langle\langle \mathtt{p}, 1\rangle\rangle, \langle\rangle, \langle\rangle\rangle$$

where the body and the headers are empty. An HTTPS request for r is of the form $\mathsf{enc_a}(\langle r, k'\rangle, \mathsf{pub}(k_{\text{example.com}}))$, where k' is a fresh symmetric key (a nonce) generated by the sender of the request (typically a browser); the responder is supposed to use this key to encrypt the response.

The *equational theory* associated with Σ is defined as usual in DY models. The theory induces a congruence relation $\equiv$ on terms, capturing the meaning of the function symbols in Σ. For instance, the equation in the equational theory which captures asymmetric decryption is $\mathsf{dec_a}(\mathsf{enc_a}(x, \mathsf{pub}(y)), y) = x$. With this, we have that, for example,

$$\mathsf{dec_a}(\mathsf{enc_a}(\langle r, k'\rangle, \mathsf{pub}(k_{\text{example.com}})), k_{\text{example.com}}) \equiv \langle r, k'\rangle$$

i.e., these two terms are equivalent w.r.t. the equational theory.

A *(DY) process* consists of a set of addresses the process listens to, a set of states (terms), an initial state, and a relation that takes an

[12] Recall the meaning of "naïve user intention tracking" from Section 2.

[13] *mod_auth_openidc* and nytimes.com are not susceptible to the state leak attack since after the login/authorization, the user is immediately redirected to another web page at the same RP.

event and a state as input and (non-deterministically) returns a new state and a sequence of events. The relation models a computation step of the process. It is required that the output can be computed (more formally, derived in the usual DY style) from the input event and the state.

The so-called *attacker process* is a DY process which records all messages it receives and outputs all events it can possibly derive from its recorded messages. Hence, an attacker process carries out all attacks any DY process could possibly perform. Attackers can corrupt other parties.

A *script* models JavaScript running in a browser. Scripts are defined similarly to DY processes. When triggered by a browser, a script is provided with state information. The script then outputs a term representing a new internal state and a command to be interpreted by the browser (see also the specification of browsers below). Similarly to an attacker process, the so-called *attacker script* may output everything that is derivable from the input.

A *system* is a set of processes. A *configuration* of this system consists of the states of all processes in the system, the pool of waiting events, and a sequence of unused nonces. Systems induce *runs*, i.e., sequences of configurations, where each configuration is obtained by delivering one of the waiting events of the preceding configuration to a process, which then performs a computation step.

A *web system* formalizes the web infrastructure and web applications. It contains a system consisting of honest and attacker processes. Honest processes can be web browsers, web servers, or DNS servers. Attackers can be either *web attackers* (who can listen to and send messages from their own addresses only) or *network attackers* (who may listen to and spoof all addresses and therefore are the most powerful attackers). A web system further contains a set of scripts (comprising honest scripts and the attacker script).

In our analysis of OAuth, we consider either one network attacker or a set of web attackers (see Section 5). In our OAuth model, we need to specify only the behavior of servers and scripts. These are not defined by the FKS model since they depend on the specific application, unless they are corrupt or become corrupted in which case they behave like attacker processes and attacker scripts; browsers are specified by the FKS model (see below). The modeling of OAuth servers and scripts is outlined in Section 5.1 and defined in detail in [17].

Web Browsers. An honest browser is thought to be used by one honest user, who is modeled as part of the browser. User actions, such as following a link, are modeled as non-deterministic actions of the web browser. User credentials are stored in the initial state of the browser and are given to selected web pages when needed. Besides user credentials, the state of a web browser contains (among others) a tree of windows and documents, cookies, and web storage data (localStorage and sessionStorage).

A *window* inside a browser contains a set of *documents* (one being active at any time), modeling the history of documents presented in this window. Each represents one loaded web page and contains (among others) a script and a list of subwindows (modeling iframes). The script, when triggered by the browser, is provided with all data it has access to, such as a (limited) view on other documents and windows, certain cookies, and web storage data. Scripts then output a command and a new state. This way, scripts can navigate or create windows, send XHRs and postMessages, submit forms, set/change cookies and web storage data, and create iframes. Navigation and security rules ensure that scripts can manipulate only specific aspects of the browser's state, according to the web standards.

A browser can output messages on the network of different types, namely DNS and HTTP(S) requests as well as XHRs, and it processes the responses. Several HTTP(S) headers are modeled, including, for example, cookie, location, strict transport security (STS), and origin headers. A browser, at any time, can also receive a so-called trigger message upon which the browser non-deterministically chooses an action, for instance, to trigger a script in some document. The script now outputs a command, as described above, which is then further processed by the browser. Browsers can also become corrupted, i.e., be taken over by web and network attackers. Once corrupted, a browser behaves like an attacker process.

5. ANALYSIS

We now present our security analysis of OAuth (with the fixes mentioned in Section 3 applied). We first present our model of OAuth. We then formalize the security properties and state the main theorem, namely the security of OAuth w.r.t. these properties. We provide full details of the model and our proof in the technical report [17].

5.1 Model

As mentioned above, our model for OAuth is based on the FKS model outlined in Section 4. For the analysis, we extended the model to include HTTP Basic Authentication [19] and Referrer Policies [13] (the Referer header itself was already part of the model). We developed the OAuth model to adhere to RFC6749, the OAuth 2.0 standard, and follow the security considerations described in [26].

Design. Our comprehensive model of OAuth includes all configuration options of OAuth and makes as few assumptions as possible in order to strengthen our security results:

OAuth Modes. Every RP and IdP may run any of the four OAuth modes, even simultaneously.

Corruption. RPs, IdPs, and browsers can be corrupted by the attacker at any time.

Redirection URIs. RP chooses redirection URIs explicitly or the IdP selects a redirection URI that was registered before. Redirection URIs can contain patterns. This covers all cases specified in the OAuth standard. We allow that IdPs do not strictly check the redirection URIs, and instead apply loose checking, i.e., only the origin is checked (this is the default for Facebook, for example). This only strengthens the security guarantees we prove.

Client Secrets. Just as in the OAuth standard, RPs can, for a certain IdP, have a secret or not have a secret in our model.

Usage of HTTP and HTTPS. Users may visit HTTP and HTTPS URIs (e.g., for RPs) and parties are not required to use Strict-Transport-Security (STS), although we still recommend STS in practice (for example, to reduce the risk of password eavesdropping). Again, this only strengthens our results.

General User Interaction. As usual in the FKS model, the user can at any time navigate backwards or forward in her browser history, navigate to any web page, open multiple windows, start simultaneous login flows using different or the same IdPs, etc. Web pages at RPs can contain regular links to arbitrary external web sites.

Authentication at IdP. User authentication at the IdP, which is out of the scope of OAuth, is performed using username and password.

Session Mechanism at RP. OAuth does not prescribe a specific session mechanism to be used at an RP. Our model therefore includes a standard cookie-based session mechanism (as suggested in [8]).

Attack Mitigations. To prove the security properties of OAuth, our model includes the fixes against the new attacks presented in

Section 3 as well as standard mitigations against known attacks. Altogether this offers clear implementation guidelines, without which OAuth would be insecure:

Honest Parties. RPs and IdPs, as long as they are honest, do not include (untrusted) third-party JavaScript on their websites, do not contain open redirectors, and do not have Cross-Site Scripting vulnerabilities. Otherwise, access tokens and authorization codes can be stolen in various ways, as described, among others, in [6, 20, 26, 36].

CSRF Protection. The *state* parameter is used with a nonce that is bound to the user's session (see [8]) to prevent CSRF vulnerabilities on the RP redirection endpoint. Omitting or incorrectly using this parameter can lead to attacks described in [6, 20, 25, 26, 36].

More specifically, a new state nonce is freshly chosen for each login attempt. Otherwise, the following attack is applicable: First, a user starts an OAuth flow at some RP using a malicious IdP. The IdP learns the state value that is used in the current user session. Then, as soon as the user starts a new OAuth flow with the same RP and an honest IdP, the malicious IdP can use the known state value to mount a CSRF attack, breaking the session integrity property.[14]

We also model CSRF protection for some URIs as follows: For RPs, we model origin header checking[15] (1) at the URI where the OAuth flow is started (for the implicit and authorization code mode), (2) at the password login for the resource owner password credentials mode, and (3) at the URI to which the JavaScript posts the access token in the implicit mode. For IdPs, we do the same at the URI to which the username and password pairs are posted. The CSRF protection of these four URIs is out of the scope of OAuth and therefore, we follow good web development practices by checking the origin header. Without this or similar CSRF protection, IdPs and RPs would be vulnerable to CSRF attacks described in [6, 36].

Referrer Policy and Status Codes. RPs and IdPs use the Referrer Policy [13] to specify that Referer headers on links from any of their web pages may not contain more than the origin of the respective page. Otherwise, RPs or IdPs would be vulnerable to the state leak attack described in Section 3.3 and the code leak attack described in [21]. IdPs use 303 redirects following our fix described in Section 3.1.

HTTPS Endpoints. All endpoint URIs use HTTPS to protect against attackers eavesdropping on tokens or manipulating messages (see, e.g., [26, 36]). Obviously, IdPs or RPs do not register URIs that point to servers other than their own. (Otherwise, access tokens or authorization codes can be stolen trivially.)

Session Cookies. Cookies are always set with the *secure* attribute, ensuring that the cookie value is only transmitted over HTTPS. Otherwise, a network attacker could read cookie values by eavesdropping on non-HTTPS connections to RPs. After successful login at an RP, the RP creates a fresh session id for that user. Otherwise, a network attacker could set a login session cookie that is bound to a known state value into the user's browser (see [39]), lure the user into logging in at the corresponding RP, and then use the session cookie to access the user's data at the RP (*session fixation*, see [28]).

[14]Note that in this attack, the state value does not leak unintentionally (in contrast to the state leak attack). Also note that this attack and the mitigation we describe here, while not surprising, do not seem to have been explicitly documented so far. For example, nytimes.com is vulnerable also to this attack.

[15]The origin header is added to certain HTTP(S) requests by browsers to declare the origin of the document that caused the request. For example, when a user submits a form loaded from the URI http://a/form and this form is sent to http://b/path then the browser will add the origin header http://a in the request to b. All modern browsers support origin headers. See [12] for details.

Authentication to the IdP. It is assumed that the user only ever sends her password over an encrypted channel and only to the IdP this password was chosen for (or to trusted RPs, as mentioned above). (The user also does not re-use her password for different IdPs.) Otherwise, a malicious IdP would be able to use the account of the user at an honest IdP.

Authentication using Access Tokens. When an RP sends an access token to the introspection endpoint of an IdP for authentication (Step 12 in Figure 1), the IdP returns the user identifier and the client id for which the access token was issued (Step 13). The RP must check that the returned client id is its own, otherwise a malicious RP could impersonate an honest user at an honest RP (see [20, 37]). We therefore require this check.

User Intention Tracking. We use explicit user intention tracking. Otherwise, the attack described in Section 3.4 can be applied.

Concepts Used in Our Model. In our model and the security properties, we use the following concepts:

Protected Resources. Closely following RFC6749 [20], OAuth protected resources are an abstract concept for any resource an RP could use at an IdP after successful authorization. For example, if Facebook gives access to the friends list of a user to an RP, this would be considered a protected resource. In our model, there is a mapping from (IdP, RP, identity) to nonces (which model protected resources). In this mapping, the identity part can be $\perp$, modeling a resource that is acquired in the client credentials mode and thus not bound to a user.

Service Tokens. When OAuth is used for authentication, we assume that after successful login, the RP sends a *service token* to the browser. The intuition is that with this service token a user can use the services of the RP. The service token consists of a nonce, the user's identifier, and the domain of the IdP which was used in the login process. The service token is a generic model for any session mechanism the RP could use to track the user's login status (e.g., a cookie). We note that the actual session mechanism used by the RP *after* a successful login is out of the scope of OAuth, which is why we use the generic concept of a service token. In our model, the service token is delivered by an RP to a browser as a cookie.

Trusted RPs. In our model, among others, a browser can choose to launch the resource owner password credentials mode with any RP, causing this RP to know the password of the user. RPs, however, can become corrupted and thus leak the password to the attacker. Therefore, to define the security properties, we define the concept of *trusted RPs*. Intuitively, this is a set of RPs a user entrusts with her password. In particular, whether an RP is trusted depends on the user. In our security properties, when we state that an adversary should not be able to impersonate a user u in a run, we would assume that all trusted RPs of u have not become corrupted in this run.

OAuth Web System with a Network Attacker. We model OAuth as a class of web systems (in the sense of Section 4) that can contain an unbounded finite number of RPs, IdPs, and browsers. We call a web system $\mathcal{OWS}^n$ an *OAuth web system with a network attacker* if it is of the form described in what follows.

Outline. The system consists of a network attacker, a finite set of web browsers, a finite set of web servers for the RPs, and a finite set of web servers for the IdPs. Recall that in $\mathcal{OWS}^n$, since we have a network attacker, we do not need to consider web attackers (as our network attacker subsumes all web attackers). The set of scripts consists of the three scripts *script_rp_index*, *script_rp_implicit*, and *script_idp_form*. We now briefly sketch RPs, IdPs, and the scripts, with full details provided in our technical report [17].

Relying Parties. Each RP is a web server modeled as an atomic DY process following the description in Section 2, including all OAuth modes, as well as the fixes and mitigations discussed before. The RP can either (at any time) launch a client credentials mode flow or wait for users to start any of the other flows. RP manages two kinds of sessions: The *login sessions*, which are used only during the user login phase, and the *service sessions* (modeled by a *service token* as described above). When receiving a special message, an RP can become corrupted and then behaves like an attacker process.

Identity Providers. Each IdP is a web server modeled as an atomic DY process following the description in Section 2, again including all OAuth modes, as well as the fixes and mitigations discussed before. Users can authenticate to an IdP with their credentials. Just as RPs, IdPs can become corrupted at any time.

Scripts. The scripts which run in a user's browser are defined as follows: The script *script_rp_index* is loaded from an RP into a user's browser when the user visits the RP's web site. It starts the authorization or login process. The script *script_rp_implicit* is loaded into the user's browser from an RP during an implicit mode flow to retrieve the data from the URI fragment. It extracts the access token and state from the fragment part of its own URI. The script then sends this information in the body of an HTTPS POST request to the RP. The script *script_idp_form* is loaded from an IdP into the user's browser for user authentication at the IdP.

OAuth Web System with Web Attackers. In addition to $\mathcal{OWS}^n$, we also consider a class of web systems where the network attacker is replaced by an unbounded finite set of web attackers. We denote such a system by $\mathcal{OWS}^w$ and call it an *OAuth web system with web attackers*, Such web systems are used to analyze session integrity, see below.

Limitations of Our OAuth Model. While our model of OAuth is very comprehensive, a few aspects of OAuth were not taken into consideration in our analysis:

We do not model *expiration* of access tokens and session ids. Also, IdPs may issue so-called *refresh tokens* in Step 9 of Figure 1. In practice, an RP may use such a (long-living) refresh token to obtain a new (short-lived) access token. In our model, we overapproximate this by not expiring access tokens. We also do not model *revocation* of access tokens and *user log out*.

OAuth IdPs support controlling the *scope* of resources made available to an RP. For example, a Facebook user can grant a third party the right to read her user profile but deny access to her friends list. The scope is a property of the access token, but handled internally by the IdP with its implementation, details, and semantics highly dependent on the IdP. We therefore model that RPs always get full access to the user's data at the IdP.

In practice, IdPs can send *error messages* (mostly static strings) to RPs. We do not model these.

Limitations of the underlying FKS model are discussed in [14].

5.2 Security Properties

Based on the formal OAuth model described above, we now formulate central security properties of OAuth, namely authorization, authentication, and session integrity (see our technical report [17] for the full formal definitions).

Authorization. Intuitively, authorization for $\mathcal{OWS}^n$ means that an attacker should not be able to obtain or use a protected resource available to some honest RP at an IdP for some user unless, roughly speaking, the user's browser or the IdP is corrupted.

More formally, we say that $\mathcal{OWS}^n$ is *secure w.r.t. authorization* if the following holds true: if at any point in a run of $\mathcal{OWS}^n$ an attacker can obtain a protected resource available to some honest RP r at an IdP i for some user u, then the IdP i is corrupt or, if $u \neq \bot$, we have that the browser of u or at least one of the trusted RPs of u must be corrupted. Recall that if $u = \bot$, then the resource was acquired in the client credentials mode, and hence, is not bound to a user.

Authentication. Intuitively, authentication for $\mathcal{OWS}^n$ means that an attacker should not be able to login at an (honest) RP under the identity of a user unless, roughly speaking, the IdP involved or the user's browser is corrupted. As explained above, being logged in at an RP under some user identity means to have obtained a service token for this identity from the RP.

More formally, we say that $\mathcal{OWS}^n$ is *secure w.r.t. authentication* if the following holds true: if at any point in a run of $\mathcal{OWS}^n$ an attacker can obtain the service token that was issued by an honest RP using some IdP i for a user u, then the IdP i, the browser of u, or at least one of the trusted RPs of u must be corrupted.

Session Integrity. Intuitively, session integrity (for authorization) means that (a) an RP should only be authorized to access some resources of a user when the user actually expressed the wish to start an OAuth flow before, and (b) if a user expressed the wish to start an OAuth flow using some honest IdP and a specific identity, then the OAuth flow is never completed with a different identity (in the same session); similarly for authentication.

More formally, we say that $\mathcal{OWS}^w$ is *secure w.r.t. session integrity for authorization* if the following holds true: (a) if in a run $\mathcal{OWS}^w$ an OAuth login flow is completed with a user's browser, then this user started an OAuth flow. (b) If in addition we assume that the IdP that is used in the completed flow is honest, then the flow was completed for the same identity for which the OAuth flow was started by the user. We say that the OAuth flow was completed (for some identity v) iff the RP gets access to a protected resource (of v).

We say that $\mathcal{OWS}^w$ is *secure w.r.t. session integrity for authentication* if the following holds true: (a) if in a run ρ of $\mathcal{OWS}^w$ a user is logged in with some identity v, then the user started an OAuth flow. (b) If in addition the IdP that is used in that flow is honest, then the user is logged in under exactly the same identity for which the OAuth flow was started by the user.

We note that for session integrity, as opposed to authorization and authentication, we use the web attacker as an adversary. The rationale behind this is that a *network* attacker can always forcefully log in a user under his own account (by setting cookies from non-secure to secure origins [39]), thereby defeating existing CSRF defenses in OAuth (most importantly, the state parameter). This is a common problem in the session management of web applications, independently of OAuth. This is why we restrict our analysis of session integrity to web attackers since otherwise session integrity would trivially be broken. We note, however, that more robust solutions for session integrity are conceivable (e.g., using JavaScript and HTML5 features such as web messaging and web storage). While some proprietary approaches exist, such approaches are less common and typically do not conform to the OAuth standard.

Main Theorem. We prove the following theorem (see [17] for the proof):

Theorem 1. Let $\mathcal{OWS}^n$ be an OAuth web system with a network attacker, then $\mathcal{OWS}^n$ is secure w.r.t. authorization and secure w.r.t. authentication. Let $\mathcal{OWS}^w$ be an OAuth web system with web attackers, then $\mathcal{OWS}^w$ is secure w.r.t. session integrity for authorization and authentication.

Note that this trivially implies that authentication and authorization properties are satisfied also if web attackers are considered.

5.3 Discussion of Results

Our results show that the OAuth standard is secure, i.e., provides strong authentication, authorization, and session integrity properties, when (1) fixed according to our proposal and (2) when adhering to the OAuth security recommendations and best practices, as explained in Section 5.1. Depending on individual implementation choices, (2) is potentially not satisfied in all practical scenarios. For example, RPs might run untrusted JavaScript on their websites. Nevertheless, our security results, for the first time, give precise implementation guidelines for OAuth to be secure and also clearly show that if these guidelines are not followed, then the security of OAuth cannot be guaranteed.

6. RELATED WORK

We focus on work closely related to OAuth 2.0 or formal security analysis of web standards and web applications.

The work closest to our work is the already mentioned work by Bansal, Bhargavan, Delignat-Lavaud, and Maffeis [6]. Bansal et al. analyze the security of OAuth using the applied pi-calculus and the WebSpi library, along with the protocol analysis tool ProVerif. They model various settings of OAuth 2.0, often assuming the presence of common web implementation flaws resulting in, for example, CSRF and open redirectors in RPs and IdPs. They identify previously unknown attacks on the OAuth implementations of Facebook, Yahoo, Twitter, and many other websites. Compared to our work, the WebSpi model used in [6] is less expressive and comprehensive (see also the discussion in [14]), and the models of OAuth they employ are more limited.[16] As pointed out by Bansal et al., the main focus of their work is to discover attacks on OAuth, rather than proving security. They have some positive results, which, however, are based on their more limited model. In addition, in order to prove these results further restrictions are assumed, e.g., they consider only one IdP per RP and all IdPs are assumed to be honest.

Wang et al. [37] present a systematic approach to find implicit assumptions in SDKs (e.g., the Facebook PHP SDK) used for authentication and authorization, including SDKs that implement OAuth 2.0.

In [29], Pai et al. analyze the security of OAuth in a very limited model that does not incorporate generic web features. They show that using their approach, based on the Alloy finite-state model checker, known weaknesses can be found. The same tool is used by Kumar [24] in a formal analysis of the older OAuth 1.0 protocol (which, as mentioned, is very different to OAuth 2.0).

Chari, Jutla, and Roy [9] analyze the security of the authorization code mode in the universally composability model, again without considering web features, such as semantics of HTTP status codes, details of cookies, or window structures inside a browser.

Besides these formal approaches, empirical studies were conducted on deployed OAuth implementations. In [36], Sun and Beznosov analyze the security of three IdPs and 96 RPs. In [25], Li and Mitchell study the security of 10 IdPs and 60 RPs based in China. In [38], Yang et al. perform an automated analysis of 4 OAuth IdPs and 500 RPs. Shernan et al. [34] evaluate the lack of CSRF protection in various OAuth deployments. In [10, 33], practical evaluations on the security of OAuth implementations of mobile apps are performed.

[16]For example, only two OAuth modes are considered, the model is monotonic (e.g., cookies can only be added, but not deleted or modified), fixed bounded number of cookies per request, no precise handling of windows, documents, and iframes, no web messaging, omission of headers, such as origin. We note that while OAuth does not make use of all web features, taking such features into account is important to make positive security results more meaningful.

In [27], Mladenov et al. perform an informal analysis of OpenID Connect. They present several attacks related to discovery and dynamic client registration, which are extensions of OpenID Connect; see also the discussion in Section 3.2 (related attacks) concerning their malicious endpoint attack.

Note that many of the works listed here led to improved security recommendations for OAuth as listed in RFC6749 [20] and RFC6819 [26]. These are already taken into account in our model and analysis of OAuth.

More generally, there have been only very few analysis efforts for web applications and standards based on formal web models so far. Work outside of the context of OAuth includes [2–5, 14–16, 23].

7. CONCLUSION

In this paper, we carried out the first extensive formal analysis of OAuth 2.0 based on a comprehensive and expressive web model. Our analysis, which aimed at the standard itself, rather than specific OAuth implementations and deployments, comprises all modes (grant types) of OAuth and available options and also takes malicious RPs and IdPs as well as corrupted browsers/users into account. The generic web model underlying our model of OAuth and its analysis is the most comprehensive web model to date.

Our in-depth analysis revealed four attacks on OAuth as well as OpenID connect, which builds on OAuth. We verified the attacks, proposed fixes, and reported the attacks and our fixes to the working groups for OAuth and OpenID Connect. The working groups confirmed the attacks. Fixes to the standard and recommendations are currently under discussion or already incorporated in a draft for a new RFC [22].

With the fixes applied, we were able to prove strong authorization, authentication, and session integrity properties for OAuth 2.0. Our security analysis assumes that OAuth security recommendations and certain best practices are followed. We show that otherwise the security of OAuth cannot be guaranteed. By this, we also provide clear guidelines for implementations. The fact that OAuth is one of the most widely deployed authorization and authentication systems in the web and the basis for other protocols makes our analysis particularly relevant.

As for future work, our formal analysis of OAuth offers a good starting point for the formal analysis of OpenID Connect, and hence, such an analysis is an obvious next step for our research.

8. ACKNOWLEDGEMENTS

This work was partially supported by *Deutsche Forschungsgemeinschaft* (DFG) through Grant KU 1434/10-1.

9. REFERENCES

[1] M. Abadi and C. Fournet. Mobile Values, New Names, and Secure Communication. In *POPL 2001*, pages 104–115. ACM Press, 2001.

[2] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a Formal Foundation of Web Security. In *CSF 2010*, pages 290–304. IEEE Computer Society, 2010.

[3] A. Armando, R. Carbone, L. Compagna, J. 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. Elsevier, 2013.

[4] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and M. L. Tobarra. Formal Analysis of SAML 2.0 Web Browser Single Sign-on: Breaking the SAML-based Single Sign-on for Google Apps. In *FMSE 2008*, pages 1–10. ACM, 2008.

[5] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage. In *POST 2013*, volume 7796 of *LNCS*, pages 126–146. Springer, 2013.

[6] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Discovering Concrete Attacks on Website Authorization by Formal Analysis. *Journal of Computer Security*, 22(4):601–657, 2014. IOS Press, 2014.

[7] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *CCS 2008*, pages 75–88. ACM, 2008.

[8] J. Bradley, T. Lodderstedt, and H. Zandbelt. Encoding claims in the OAuth 2 state parameter using a JWT – draft-bradley-oauth-jwt-encoded-state-05. IETF. Dec. 2015. https://tools.ietf.org/html/draft-bradley-oauth-jwt-encoded-state-05.

[9] S. Chari, C. S. Jutla, and A. Roy. Universally Composable Security Analysis of OAuth v2.0. *IACR Cryptology ePrint Archive*, 2011:526, 2011.

[10] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague. OAuth Demystified for Mobile Application Developers. In *CCS 2014*, pages 892–903, 2014.

[11] Chromium Project. HSTS Preload Submission. https://hstspreload.appspot.com/.

[12] Cross-Origin Resource Sharing - W3C Recommendation 16 January 2014. http://www.w3.org/TR/2014/REC-cors-20140116/.

[13] J. Eisinger and E. Stark. Referrer Policy – Editor's Draft, 28 March 2016. W3C. Mar. 2016. https://w3c.github.io/webappsec-referrer-policy/.

[14] D. Fett, R. Küsters, and G. Schmitz. An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System. In *S&P 2014*, pages 673–688. IEEE Computer Society, 2014.

[15] D. Fett, R. Küsters, and G. Schmitz. Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web. In *ESORICS 2015*, volume 9326 of *LNCS*, pages 43–65. Springer, 2015.

[16] D. Fett, R. Küsters, and G. Schmitz. SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web. In *CCS 2015*, pages 1358–1369. ACM, 2015.

[17] D. Fett, R. Küsters, and G. Schmitz. A Comprehensive Formal Security Analysis of OAuth 2.0. Technical Report arXiv:1601.01229, arXiv, 2016. Available at http://arxiv.org/abs/1601.01229.

[18] R. Fielding (ed.) and J. Reschke (ed.). RFC7231 – Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. IETF. Jun. 2014. https://tools.ietf.org/html/rfc7231.

[19] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. RFC2617 – HTTP Authentication: Basic and Digest Access Authentication. IETF. Jun. 1999. https://tools.ietf.org/html/rfc2617.

[20] D. Hardt (ed.). RFC6749 – The OAuth 2.0 Authorization Framework. IETF. Oct. 2012. https://tools.ietf.org/html/rfc6749.

[21] E. Homakov. How I hacked Github again, 7 February 2014. http://homakov.blogspot.de/2014/02/how-i-hacked-github-again.html.

[22] M. Jones, J. Bradley, and N. Sakimura. OAuth 2.0 Mix-Up Mitigation – draft-ietf-oauth-mix-up-mitigation-01. IETF. Jul. 2016. https://tools.ietf.org/html/draft-ietf-oauth-mix-up-mitigation-01.

[23] F. Kerschbaum. Simple Cross-Site Attack Prevention. In *SecureComm 2007*, pages 464–472. IEEE Computer Society, 2007.

[24] A. Kumar. Using automated model analysis for reasoning about security of web protocols. In *ACSAC 2012*. ACM, 2012.

[25] W. Li and C. J. Mitchell. Security issues in OAuth 2.0 SSO implementations. In *ISC 2014*, volume 8783 of *LNCS*, pages 529–541, 2014. Springer, 2014.

[26] T. Lodderstedt (ed.), M. McGloin, and P. Hunt. RFC6819 – OAuth 2.0 Threat Model and Security Considerations. IETF. Jan. 2013. https://tools.ietf.org/html/rfc6819.

[27] V. Mladenov, C. Mainka, J. Krautwald, F. Feldmann, and J. Schwenk. On the security of modern Single Sign-On Protocols: Second-Order Vulnerabilities in OpenID Connect. *CoRR*, abs/1508.04324v2, 2016.

[28] Open Web Application Security Project (OWASP). Session fixation. https://www.owasp.org/index.php/Session_Fixation.

[29] S. Pai, Y. Sharma, S. Kumar, R. M. Pai, and S. Singh. Formal Verification of OAuth 2.0 Using Alloy Framework. In *CSNT 2011*, pages 655–659. IEEE, 2011.

[30] J. Richer (ed.). RFC7662 – OAuth 2.0 Token Introspection. IETF. Oct. 2015. https://tools.ietf.org/html/rfc7662.

[31] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore. OpenID Connect Core 1.0 incorporating errata set 1. OpenID Foundation. Nov. 8, 2014. http://openid.net/specs/openid-connect-core-1_0.html.

[32] J. Selvi. Bypassing HTTP Strict Transport Security. In *Blackhat (Europe) 2014*, 2014.

[33] M. Shehab and F. Mohsen. Towards Enhancing the Security of OAuth Implementations in Smart Phones. In *IEEE MS 2014*. IEEE, 2014.

[34] E. Shernan, H. Carter, D. Tian, P. Traynor, and K. R. B. Butler. More Guidelines Than Rules: CSRF Vulnerabilities from Noncompliant OAuth 2.0 Implementations. In *DIMVA 2015*, volume 9148 of *LNCS*, pages 239–260. Springer, 2015.

[35] SimilarTech. Facebook Connect Market Share and Web Usage Statistics. Last visited Nov. 7, 2015. https://www.similartech.com/technologies/facebook-connect.

[36] S.-T. Sun and K. Beznosov. The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. In *CCS 2012*, pages 378–390. ACM, 2012.

[37] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In *USENIX Security 2013*, pages 399–314. USENIX Association, 2013.

[38] R. Yang, G. Li, W. C. Lau, K. Zhang, and P. Hu. Model-based Security Testing: An Empirical Study on OAuth 2.0 Implementations. In *AsiaCCS 2016*, pages 651–662. ACM, 2016.

[39] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, T. Wan, and N. Weaver. Cookies Lack Integrity: Real-World Implications. In *USENIX Security 2015)*, pages 707–721, 2015. USENIX Association, 2015.