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

The OAuth 2.0 protocol allows users to grant relying parties access to resources at identity providers. In addition to being used for this kind of authorization, OAuth is also often employed for authentication in single sign-on (SSO) systems. OAuth 2.0 is, in fact, one of the most widely used protocols in the web for these purposes, with companies such as Google, Facebook, or PayPal acting as identity providers and millions of websites connecting to these services as relying parties.

OAuth 2.0 is at the heart of *Facebook Login* and many other implementations, and also serves as the foundation for the upcoming SSO system 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 and authentication 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 and identity providers are considered as well.

While proving security, we found two previously unknown attacks on OAuth, which both break authorization and authentication in OAuth. The underlying vulnerabilities are present also in the new OpenID Connect standard and can be exploited in practice.

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 provides the authorization and authentication properties we specify.

Contents

1. Introduction

The OAuth 2.0 authorization framework [13] 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 Facebook, Google, Microsoft, Yahoo, GitHub, and Dropbox. This enables billions of users to log in at millions of RPs or share their data with these [27], 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 user logout.

OAuth defines a very complex protocol. 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, in most of these modes, depending on the configuration and prior setup of the RP and the IdP, further options within the different modes are provided.

Therefore, analyzing the security of OAuth is a complex task. So far, most analysis efforts were targeted towards finding errors in specific implementations [4, 6, 18, 26, 28], rather than the comprehensive analysis of the standard itself: none of the existing analysis efforts of OAuth account for multiple 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, which, however, is essential to rule out security risks that arise when running the protocol in the context of common web technologies.

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 and RPs.

Informal description of OAuth. As a basis for the model, we first provide a detailed description of the protocol flows in all four modes of OAuth.

Formal model of OAuth. Our formal analysis of OAuth uses an expressive Dolev-Yao style model of the web infrastructure [8] proposed by Fett, Küsters, and Schmitz. This model has already been used to analyze the security of the BrowserID single sign-on system [8, 10] as well as the security and privacy of the SPRESSO single sign-on system [11]. 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

[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.

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 cross-document messaging (postMessages). JavaScript is modeled in an abstract way by so-called scripting processes which can be sent around and, among others, can create iframes and initiate XMLHTTPRequests (XHRs). Browsers may be corrupted dynamically by the adversary.

Using this generic web model, we build a formal model of OAuth, closely following the OAuth 2.0 standard [13]. Since this standard does not fix all aspects of the protocol, we use the current OAuth 2.0 security recommendations (RFC6819 [19]) and current web best practices (e.g., regarding sesssion handling) to obtain a model of OAuth 2.0 with state-of-the-art security features in place, in order to avoid known implementation attacks. (Note that the security recommendations in RFC6819 cover many of the bugs found in earlier analysis efforts on implementations of OAuth.) 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 formal definitions of the security properties of OAuth. In particular, we state two separate properties: authorization and authentication.

New attacks on OAuth 2.0 and fixes. While trying to prove these properties, we discovered two previously unknown attacks on OAuth, which both break authorization as well as authentication. In the first attack, IdPs inadvertently forward user credentials (i.e., username and password) to the RP or the attacker. In the second attack, a network attacker can impersonate any victim. This severe attack is caused by a logical flaw in the OAuth 2.0 protocol and depends on the presence of malicious IdP. In practice, OAuth setups often allow for selected (and thus hopefully trustworthy) IdPs only. In these setups the attack would not apply. The attack, however, can be exploited in OpenID Connect, which, as mentioned, builds directly on OAuth. We have verified the attacks on an implementation of OAuth and OpenID Connect. We also notified the respective working groups, who confirmed the attacks and adopted the fixes we propose in this paper. We present our attacks on OAuth in Section 3 in detail. In Appendix A we show how the attacks can be exploited in OpenID Connect.

We show how both attacks can be fixed by changes that are easy to implement in new and existing deployments of OAuth and OpenID Connect.

Formal analysis of OAuth 2.0. We then prove that OAuth satisfies the authorization and authentication properties in a model of OAuth with the fixes in place. This is the first proof which establishes the security of OAuth in a comprehensive and expressive web model.

We 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 the OAuth 2.0 protocol. In Section 3 we present the attacks that we found during our analysis. An overview of the generic web 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. We show how the attacks can be applied to OpenID Connect in Appendix A. Full details, in particular regarding our modeling of OAuth 2.0 and the proof, can be found in the appendices B– G.

2. OAuth 2.0

In this section, we provide a detailed description of OAuth. We here describe in detail all four modes of OAuth.

OAuth was first intended for *authorization*, i.e., the 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 (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, being 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.

We now first provide some preliminary information regarding OAuth and then present the details of the four modes of OAuth.

2.1. Preliminaries

Endpoints. In OAuth, RPs and IdPs have to provide certain URIs to each other. The parties and services these URIs point to are called *endpoints*; often the URIs themselves are called endpoints. An IdP provides an *authorization endpoint* at which the user can authenticate to the IdP and authorize an RP to access her user data. The IdP also provides a *token endpoint* at which the RP can request access tokens. An RP provides one or more *redirection endpoints* to which the user's browser gets redirected by an IdP after the user authenticated to the IdP. The URIs of the endpoints are not fixed by the standard, but are communicated when RPs register at IdPs, as described below.

The OAuth standard [13] and the accompanying security recommendations [19] suggest that all endpoints use HTTPS. We follow this recommendation in our analysis of OAuth.

Registration. 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 to the RP a fixed OAuth client id and client secret.[4] The RP may later use the client secret to authenticate to the IdP. If the RP cannot keep the OAuth client secret confidential, e.g., if the RP is an in-browser app or a native application, the secret can be omitted. Note that the OAuth client id is public information. It is, for example, revealed to users in redirects issued by the RP.

Also, an RP registers one or more redirection endpoints at an IdP. As we will see below, in some OAuth modes, the IdP redirects the user's browser to one of these redirect URIs. If more than one redirect URI is registered, the RP must specify which redirect URI is to be used in each run of the OAuth protocol. For simplicity of presentation, we will assume that an RP always specifies its choice, although this can be omitted if there exits only one (fixed) redirect URI. Note that (depending on the

[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).

[4] Recall that in the terminology of the OAuth standard the term "client" stands for RP.

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 this case, the IdP checks if the specified redirect URI matches this pattern.

During the registration process, the (fixed) endpoints belonging to an IdP are configured at an RP as well.

Our analysis presented in Section 5 covers all the above mentioned options: absence and presence of client secrets, specified redirect URIs, and URI patterns.

Login Sessions. As mentioned before, in some OAuth modes, an RP redirects the user's browser to an IdP which later redirects the browser back to the RP. In order to prevent cross-site request forgery (CSRF) attacks, the RP typically establishes a session with the browser before the first redirect. The OAuth standard recommends that an RP selects the so-called *state* parameter and binds this value to the session, e.g., by choosing a fresh nonce and storing the nonce in the session state. When the user later gets redirected back to the RP, the *state* value must be identical. The intention is that this value should always be unknown to an attacker in order to prevent CSRF attacks. In our analysis, we follow the recommendation of using the *state* parameter.[5]

Further Recommendations and Options. The standard and the recommendations do not specify all implementation details. For example, the precise user interaction with an RP, formatting details of messages, and the authentication of the user to an IdP (e.g., user name and password or some other mechanism) are not covered. In our security analysis of OAuth we follow all OAuth security recommendations as well as common best practices for state-of-the-art web applications in order to avoid known attacks.

OAuth allows RPs to specify which *scope* of the user's data they are requesting access to at an IdP. The scopes themselves are not defined in the standard and are considered an implementation detail of IdPs. Therefore, in our description and analysis of OAuth, we omit the scope parameter and assume that the user always grants full access to her data at the IdP.

2.2. OAuth Modes

OAuth can run in the following four major modes (called *grant types* in the standard): authorization code mode, implicit mode, resource owner password credentials mode, and client credentials mode. While the first two modes are very commonly used and similar to each other, the latter two modes are used less often and are different to the other modes and to each other.

We now describe these modes in detail. Contrary to our analysis, for simplicity of presentation, we only consider a specific set of options in the following description. 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 redirects the user's browser to the IdP. The user authenticates to the IdP, e.g., by providing user name and password, and is redirected back to the RP along with an *authorization code* generated by the IdP. The RP then contacts the IdP with this authorization code (along with the client id and client secret) and receives an *access token*. The RP can then use the access token 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

[5] Note that the OAuth standard [13] as well as the accompanying security recommendations [19] do not specify the session mechanism for RPs. In our analysis we assume the usual session mechanism with session cookies following common best practices. For more details, see Section 5.1.

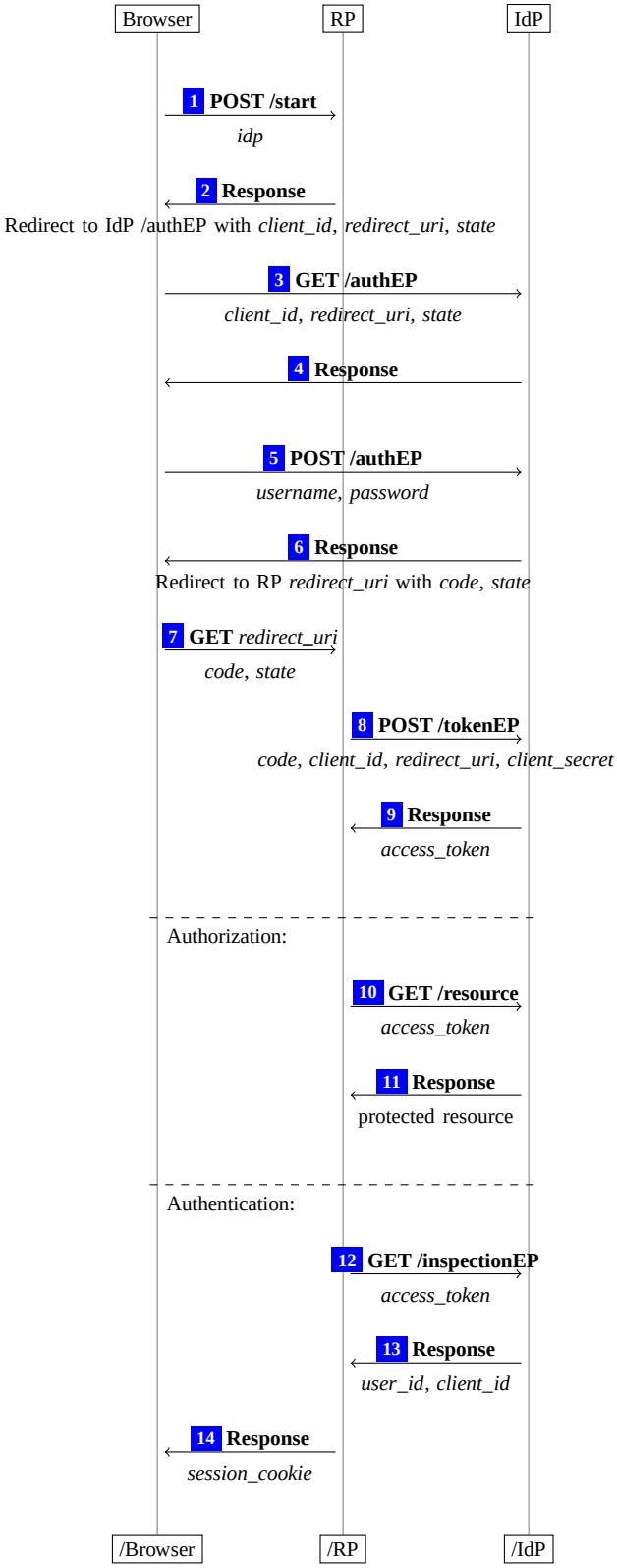

Figure 1. OAuth 2.0 authorization code mode. Note that data depicted below the arrows is either transferred in URL parameters, HTTP headers, or POST bodies. For details see [13].

button to select an IdP, resulting in request 1 being sent to the RP. The RP selects one of its redirect URIs *redirect_uri* (which will be used later in 7) and a value *state* (as described above). The RP then redirects the browser to the authorization endpoint at the IdP in 2 and 3 with its *client_id*, *redirect_uri*, and *state* appended as parameters to the URI.[6] 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 user's credentials are correct, the IdP creates a nonce *code* (the authorization code) and redirects the user's browser to RP's redirection endpoint *redirect_uri* in 6 and 7 with *code* and *state* appended as parameters to the redirection URI. Next, 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 correct 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 as illustrated in Steps 12 and 13. Along with the user id, the RP also receives its OAuth client id from the IdP. The RP checks if this client id is in fact the client id of the RP at the IdP. The RP then issues a session cookie to the user's browser as shown in 14.

We note that the IdP may also issue a so-called *refresh token* along with the access token in Step 9. The RP may use the refresh token along with its credentials to obtain a fresh access token. The refresh token can be considered as a long-living access token, while the access token itself is only valid for a limited time.[7]

For brevity of presentation, in the remaining modes, explained below, we only consider authorization. The last steps for both authorization and authentication are analogous to those presented here.

Implicit Mode. This mode is a simplified version of the authorization code mode: instead of providing an authorization code to an RP, an IdP directly delivers an access token to the RP (via the user's browser).

Step-by-Step Protocol Flow. We now provide a step-by-step description of the protocol flow (see also Figure 2). As in the authorization code mode, the user starts the OAuth flow, e.g., by clicking on a button to select an IdP, triggering the browser to send request 1 to the RP. The RP selects the redirect URI *redirect_uri* (which will be used later in 7) and a value *state*. The RP then redirects the browser with its *client_id*, *redirect_uri*, and *state* to the authorization endpoint at the IdP[8] in 2 and 3. The IdP prompts the user to enter her username and password in 4. The user's browser sends this information to the IdP in 5. If the user's credentials are correct, the IdP creates an access token *access_token* and redirects the user's browser to the RP's redirection endpoint *redirect_uri* in 6 and 7, where the IdP appends *access_token* and *state* to the fragment of the redirection URI. (Recall that a fragment is a special part of a URI indicated by the '#' symbol. When the browser opens a URI, the information in the fragment is not transferred to the server.) Hence, in Step 7 *access_token* and *state* are not transferred to the RP. To retrieve these values, the RP in 8 delivers a document containing JavaScript code. It retrieves *access_token* and *state* from the fragment and sends these to the RP in 9. The RP then checks if *state*

[6] Note that also a fixed string "`code`" indicating to the IdP that authorization code mode is used is appended as a parameter to the URI.

[7] In our analysis, we do not explicitly consider refresh tokens. Instead we overapproximate access tokens in that they do not expire, and hence, we make these tokens long-living themselves. Note that refresh tokens are less powerful than such access tokens in the sense that refresh tokens are only useful in combination with the credentials of the appropriate RP.

[8] Note that also a fixed string "`token`" indicating to the IdP that implicit mode is used is appended as a parameter to the URI.

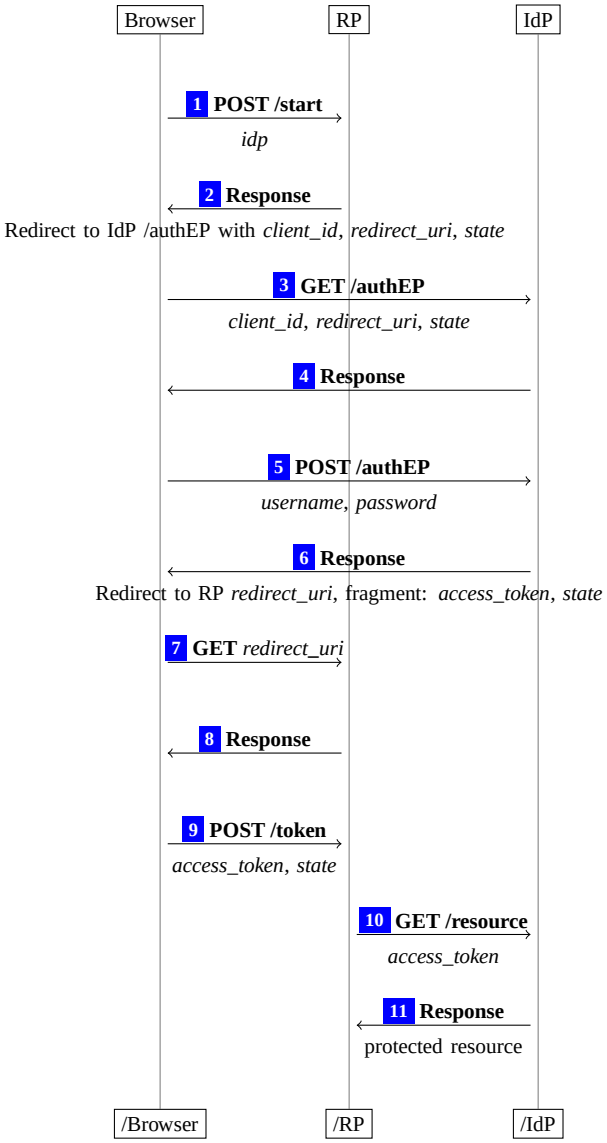

Figure 2. OAuth 2.0 implicit mode

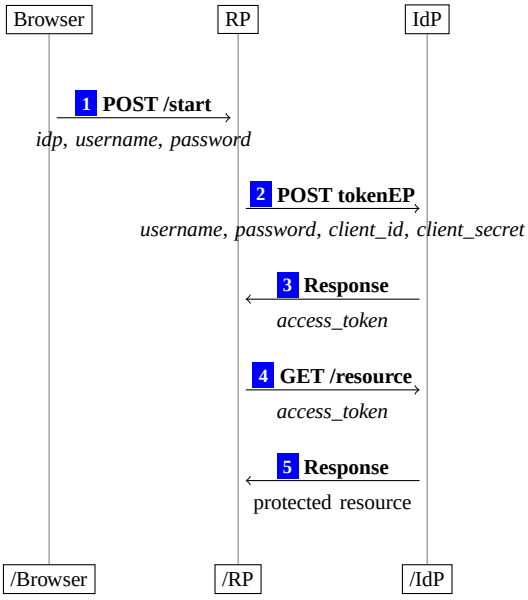

Figure 3. OAuth 2.0 resource owner password credentials mode

is the same as above. Just as in the authorization code mode, the RP can now use *access_token* for authorization (illustrated in Steps 10 and 11); authentication is analogous to Steps 12, 13, and 14 of Figure 1.

For authentication, note that the response from the IdP includes the RP's OAuth client id, which is also checked by the RP. This check prevents re-usage of access tokens across RPs in the OAuth implicit mode as explained in [29].

We note that in the implicit mode, an IdP cannot verify the identity of the receiver of the access token, as an RP does not authenticate itself to the IdP (using *client_secret*). Hence, this mode is more suitable for RPs that do not have access to a secure, long-lived storage (for a *client_secret*) such as in-browser applications.

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. The resource owner password credentials 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). In the following, we assume that the authorization/login process is started by the user using a web browser.

Step-by-Step Protocol Flow. We now provide a step-by-step description of the resource owner password credentials mode (see also Figure 3): The user provides her username and password for the IdP to the RP in 1. Now, the RP sends the username, the password, its *client_id* and *client_secret*[9] to the IdP in 2. The IdP then issues an access token *access_token* to the RP in 3.[10] Just as in the authorization code mode, the RP can now use *access_token* for authorization (illustrated in Steps 4 and 5) and authentication (as in Steps 12, 13, and 14 of Figure 1).

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 RP's own resources

[9] Note that in this mode, if an RP does not have an OAuth client secret for an IdP, the *client_secret* and *client_id* parameters are *both* omitted in this request. This option is also covered by our analysis.

[10] As in the authorization code mode, an IdP may also issue a refresh token to the RP here.

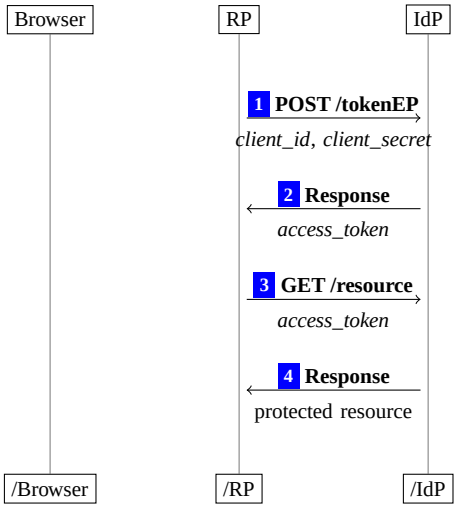

Figure 4. OAuth 2.0 client credentials mode

at an IdP or to access resources at an IdP the RP is authorized to by other means. For example, Facebook allows RPs to use the client credentials mode to obtain an access token to access reports of their advertisements' performance.

Step-by-Step Protocol Flow. The step-by-step description of the client credentials mode is as follows (see also Figure 4): First, the RP contacts the IdP with RP's *client_id* and *client_secret* in [1]. The IdP now issues an *access_token* in [2]. Just as in the authorization code mode, the RP can now use *access_token* for authorization (illustrated in Steps [3] and [4]). In contrast to the other modes presented above, the access token is not bound to a specific user account, but only to the RP.

3. Attacks on OAuth 2.0 and OpenID Connect

As mentioned in the introduction, while trying to prove the security of OAuth, we discovered two previously unknown attacks, which both apply to the authorization code mode as well as the implicit mode. We call these attacks *307 redirect attack* and *IdP mix-up attack*, respectively. In this section, we provide detailed descriptions of these attacks along with fixes which are easy to implement. Our formal analysis of OAuth (see Section 5) then shows that these fixes are indeed sufficient to establish the security of OAuth. As mentioned in the introduction, the attacks on OAuth also apply to OpenID Connect. Since in this paper the main focus is on OAuth, we only briefly present OpenID Connect and the attacks on this protocol in Section 3.3 and Appendix A. Figure 5 provides an overview of where the attacks apply. We have verified our attacks on an implementation of OAuth and OpenID Connect and reported both attacks to the respective working groups, who confirmed the attacks and adopted our fixes.

3.1. Attack: 307 Redirect

In this attack, 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.

Assumptions. We assume that (1) the user wants to log in at a malicious RP (or authorize a malicious RP to access protected resources). Besides providing some (harmless) services, this RP collects all data it receives. We further assume that (2) the IdP that is used for the login chooses the wrong HTTP status code (307) when redirecting the user's browser in Step [6] of the authorization code mode (Figure 1)

		OAuth		OpenID Connect		
		auth. code mode	implicit mode	auth. code mode	implicit mode	hybrid mode
307 Redirect Attack	authorization broken	yes	yes	yes	yes	yes
	authentication broken	yes	yes	yes	yes	yes
IdP Mix-Up Attack	authorization broken	yes*	yes	yes*		yes**
	authentication broken	yes	yes	yes		yes**

Figure 5. Attacks on OAuth 2.0 and OpenID Connect: Overview. The attacks are not applicable to resource owner password credentials mode or client credentials mode. (*): if OAuth client credentials are not used. (**): if OAuth client secrets are not used, and otherwise, if such credential are used, then either authorization or authentication is broken, depending on implementation details, see Appendix A.

or in Step 6 of the implicit mode (Figure 2). Finally, we assume that (3) 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 obviously reasonable. It cannot be assumed that all RPs where some users log in behave honestly.

Assumption (2). This assumption is reasonable because neither the OAuth standard [13] nor the OAuth security considerations [19] (nor the OpenID Connect standard [23]) 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 (3). 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`. (GitHub is not vulnerable to the attack presented here, as assumption (2) is not satisfied in this case.)

Attack. When the user uses OAuth (or OpenID Connect, see Section 3.3) to log in at this RP, the user 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, it 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 [12], only the 303 redirect is defined unambigiously to drop the body of an HTTP POST request. *All other* HTTP redirection status codes, including the most commonly used 302, leave the browser the option to preserve the POST request and the form data. In practice, browsers typically rewrite to a GET request, thereby dropping the form data, except for 307 redirects. Therefore, the OAuth standard should require 303 redirects for the steps mentioned above in order to fix this problem.

3.2. Attack: IdP Mix-Up

In this attack, 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

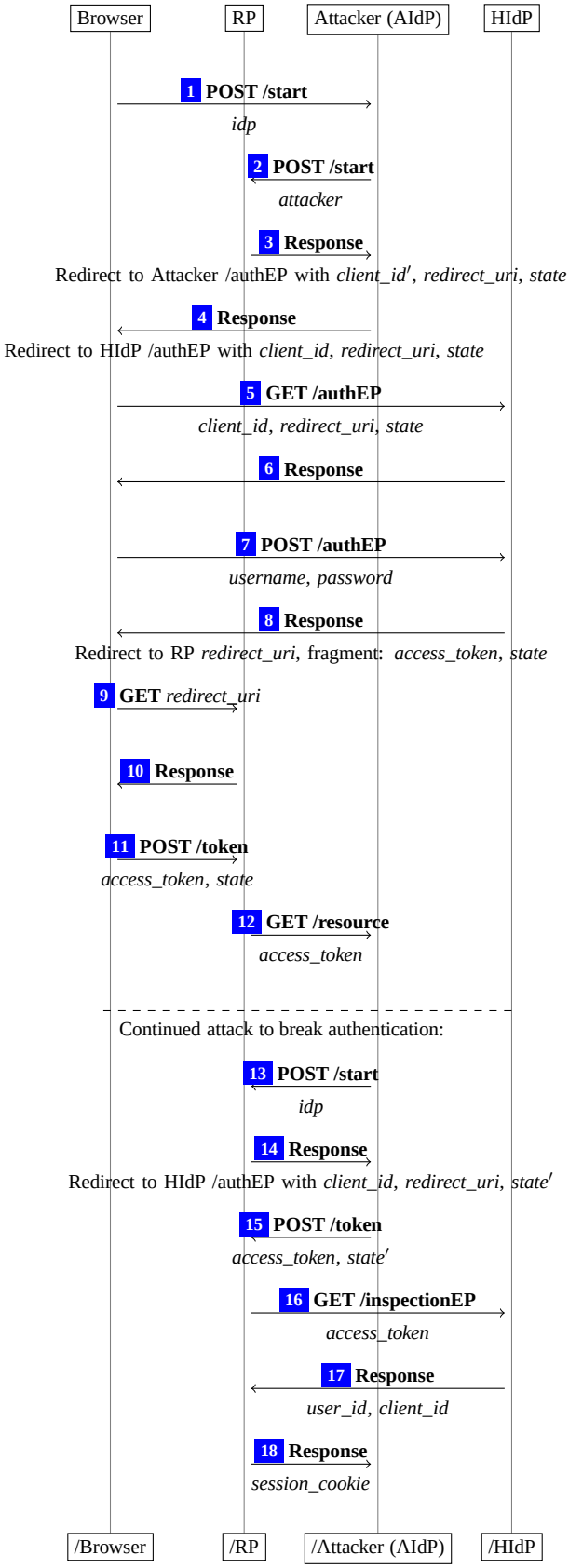

Figure 6. Attack on OAuth 2.0 implicit mode

to impersonate the user or access user data.

Just as the previous attack, the IdP mix-up attack applies to the authorization code mode and the implicit mode of OAuth. 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 while the user instead wishes to use her identity managed by an honest IdP. As a result, the RP sends the authorization code or the access token (depending on the OAuth mode) issued by the honest IdP to the attacker, who then can use these values to login at the RP under the user's identity (managed by the honest IdP) or access the user's protected resources at the honest IdP.

In what follows, we refer to the IdP run by the attacker by AIdP and the honest IdP by HIdP.

We present the attack in the implicit mode. The attack for the authorization code mode is very similar and is presented briefly at the end of this section.

Assumptions. For the IdP mix-up attack to work, we need two assumptions (see below for further discussion and explanation): (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 2),[11] and (2) an RP which allows users to log in with identities provided by (some) HIdP and identities provided by AIdP. 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 2.

First, many RPs might not require this request to be HTTPS since no sensitive user data is transferred at this point of the protocol, and hence, an RP might not see the need for encrypted communication for this request. Note that the user only selects an IdP at this point; credentials are not sent. Moreover, the use of HTTPS for this step is not suggested by the OAuth security recommendations.

Second, even if an RP intends to use HTTPS also for the first request (as in our model), the attacker can intercept the initial (often unencrypted) request from the user to the RP and alter all links from HTTPS to HTTP and remove any redirections to HTTPS URLs (a technique known as SSL stripping). This interception would fail only if the user accesses the website of the RP using HTTPS from the very beginning (e.g., by entering the URL manually), if RP has set a Strict Transport Security (STS) header in an earlier communication with the browser and this is not circumvented by the attacker (see [25]), or if the RP domain would be included in a browser preloaded STS list [7]. It is, however, unrealistic to assume that all RPs are always protected in one of these ways.

We emphasize that 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).

We also note that when intercepting the response in Step 2 as discussed here, the attacker could also read the session identifier for the session that the user's browser has established with the RP. Our attack, however, is not based on this possibility and works even if the RP later (at a point where the protocol is protected by HTTPS) changes this session identifier as soon as the user is logged in (a best practice for session management).

Assumption (2). This assumption also conforms with the OAuth standard as RPs may use different IdPs. And hence, OAuth should provide security in this case. From the point of view of how OAuth is typically deployed in practice, this assumption might seem unrealistic at first: OAuth RP deployments typically consider only one or two selected (and hopefully trusted) IdPs. OAuth forms, however, the basis for OpenID Connect. There, using the dynamic client registration extension, RPs can dynamically establish an OAuth registration with any IdP. This enables the ad-hoc use of any IdP in OpenID Connect,

[11] Note that to manipulate these two messages the network attacker needs to only be able to intercept the communication between the user and the RP (e.g., using ARP spoofing in a wifi network). Importantly, he does not need to intercept messages exchanged between servers.

including malicious IdPs. See Section 3.3 and Appendix A for details on how the IdP mix-up attack can be carried out on OpenID Connect in practice.

Attack. We now describe the attack on the OAuth implicit mode in detail. As mentioned, a very similar attack also applies to the OAuth authorization code mode and both attacks even work if IdP supports just one of these two modes, rather than both or all four OAuth modes. Note that these two modes are the most common modes in practice.

Attack on Implicit Mode. The IdP mix-up attack for the implicit mode is depicted in Figure 6. Just as in the implicit mode, the attack starts when the user selects that she wants to log in using HIdP (Step 1 in Figure 6). Now, the attacker intercepts the request intended for the RP and modifies the content of this request by replacing HIdP by AIdP. 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.[12] (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, however, still assumes that the access token contained in this redirect is an access token issued by AIdP, rather than HIdP. The RP therefore now uses this access token to retrieve protected resources of the user (or the user id) at AIdP 12, rather than HIdP. This leaks the access token to the attacker who can now access protected resources of the user at IdP. This breaks the authorization property (see Section 5.2 below). (We note that at this point, the attacker might even provide false information about the user or her protected resources to the RP.)

To break authentication and impersonate the honest user, the attacker now starts a new login process (using his own browser) at the RP. In 13 he selects HIdP as the IdP for this login process. He receives a redirect to HIdP, which he skips.[13] The attacker now sends the access token *access_token* captured in Step 12 to the RP imitating a real login 15. The RP now uses this access token to retrieve the user id at HIdP 16 and receives the (honest) user's id as well as its own OAuth client id 17. Being convinced that the attacker owns the honest user's account, the RP issues a session cookie for this account to the attacker 18. As a result, the attacker is logged in at the RP under the honest user's id. This breaks the authentication property of OAuth (see Section 5.2 below).

Attack on Authorization Code Mode. A similar attack can be applied to the authorization code mode of OAuth. This attack is based on the same assumptions as the attack on the implicit mode. First, the attacker applies the same modifications to the request from the browser to the RP and its response to initiate an OAuth flow. Recall that in the authorization code mode, RP receives an authorization code instead of an access token. The RP then sends this authorization code to the token endpoint of AIdP (in order to request an access token). If the RP does not have a client secret registered at HIdP, the attacker can redeem this authorization code at HIdP in order to receive an access token to access the honest user's protected resources at HIdP. In this case, the authorization property of OAuth (see Section 5.2 below) is broken.

Alternatively, the attacker can break the authentication property for OAuth as follows: The attacker starts a new OAuth flow with the RP using his own browser and selecting HIdP as the IdP. He then receives a redirect from the RP to HIdP to authenticate himself. He omits the interaction with HIdP and sends a request to the RP's redirection endpoint containing the authorization code. As this authorization code has not been redeemed before at HIdP, the code is still valid. Hence, when the RP requests an access token with this authorization code at HIdP, HIdP issues such an access token to the RP. The RP

[12] As mentioned above, OAuth client ids are public information.

[13] Note that this redirect contains (besides a cookie for a new login session) a fresh state parameter, say $state'$. The attacker will use this information in subsequent requests to the RP.

retrieves user information with this access token at HIdP, receives the user identifier for the user under attack and considers the attacker to be logged in as this user. (Note that the attacker does not learn an access token in this case.)

Fix. A fundamental problem in the implicit mode of the OAuth standard (and similarly for the authorization code mode) is that the redirect in Steps 6 and 7 in Figure 2 does not contain information from where the redirect was initiated. And 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 in some form. More specifically, we propose that RPs provide a unique redirection endpoint for each IdP. Hence, the information which IdP redirected the browser to the RP is encoded in the request and the RP can detect a mismatch. Note that the communication at this point in the protocol is secured with HTTPS (when following the OAuth security recommendations) and cannot be manipulated by the attacker. We show in Section 5 that this fix is indeed sufficient. In addition, one could try to prevent the IdP mix-up directly at the beginning of the protocol. (Nonetheless, in any case the RP should be able to check consistency in Step 7.) We have not considered such additional countermeasures because, as we show, our fix is sufficient for the security of OAuth (as far as authorization and authentication goes).

3.3. Implications to OpenID Connect

Both, the 307 redirect attack and the IdP mix-up attack, can be applied to OpenID Connect as well.

OpenID Connect [23] is a standard for authentication built on top of the OAuth protocol. It was published only recently. Among others, OpenID Connect is already used in practice by PayPal, Google, and Microsoft. OpenID Connect uses the same concept for authentication as described above and supports retrieval of a unique user identifier and further meta data.

We here give a brief overview of OpenID Connect and how the attacks apply to this protocol. A detailed description can be found in Appendix A.

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.

OpenID Connect consists of a core protocol and (optional) extensions, such as *discovery* [24] and *dynamic client registration* [22]. Both extensions are used to fully automate the registration process between the relying party and the identity provider. This means that, using these extensions, any RP can register dynamically at any IdP. This is triggered, for example, when a user wishes to authenticate with an IdP that the RP is not registered at.

In addition to an access token, OpenID Connect uses a so-called *id token*. The id token is issued by the IdP along with the access token and contains a unique user identifier and additional meta data, such as the intended receiver (the RP) of the id token and the issuer of the id token (the IdP). The id token is (optionally) signed by the IdP.

307 Redirect Attack in OpenID Connect. This attack applies to OpenID Connect in exactly the same way as described above. The vulnerable steps are identical in OAuth and OpenID Connect.

IdP Mix-Up Attack in OpenID Connect. 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 attack 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 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).

For details on OpenID Connect and how the attacks can be executed in OpenID Connect see Appendix A.

3.4. Verification

We have verified both attacks on *mod_auth_openidc*, an implementation of an OpenID Connect (and therefore also OAuth) RP.

We notified the developers of *mod_auth_openidc* and reported both attacks to the OAuth and OpenID Connect working groups, who confirmed that changes to the standard (and recommendations) are needed to mitigate the attacks.

4. Generic Web Model

Our formal security analysis of OAuth is based on the general Dolev-Yao style web model proposed by Fett et al. in [8], with extensions presented in [11]. This web model is designed independently of a specific web application and closely mimics published (de-facto) standards and specifications for the web. Based on a general communication model, the model specifies web systems, which contain browsers, DNS servers, and web servers as well as web and network attackers.

Here, we only briefly recall this model (see the mentioned papers for a full description).

4.1. Communication Model

The main entities in the communication model are *atomic processes*, which are used to model web browsers, web servers, DNS servers as well as web and network attackers. Each atomic process listens to one or more (IP) addresses. A set of atomic processes forms what is called a *system*. Atomic processes can 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 the current "pool" of events and is delivered to one of the atomic processes that listens to the receiver address of that event. The atomic process can then process the event and output new events, which are added to the pool of events, and so on. More specifically, messages, processes, etc. are defined as follows.

Terms, Messages and Events. As usual in Dolev-Yao models (see, e.g., [1]), messages are expressed as formal terms over a signature. The signature Σ for the terms and messages considered in the web model contains, among others, constants (such as (IP) addresses, ASCII strings, and nonces), sequence and projection symbols, and further function symbols, including those for (a)symmetric encryption/decryption and digital signatures. Messages are defined to be ground terms (terms without variables). For example, $\mathsf{pub}(k)$ denotes the public key which belongs to the private key k. To provide another example of a message, in the web model, an HTTP request is represented as a ground term containing a nonce, a method (e.g., `GET` or `POST`), a domain name, a path, URL parameters, request headers (such as `Cookie`), and a message body. For instance, an HTTP `GET` request for the URL `http://ex.com/show?p=1` is modeled as the term

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

where headers and body are empty. An HTTPS request for r is of the form $\mathsf{enc_a}(\langle r, k'\rangle, \mathsf{pub}(k_{\text{ex.com}}))$, 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.

Events are terms of the form $\langle a, f, m\rangle$ where a and f are receiver/sender (IP) addresses, and m is a message, for example, an HTTP(S) message as above or a DNS request/response.

The *equational theory* associated with the signature Σ is defined as usual in Dolev-Yao models. The theory induces a congruence relation $\equiv$ on terms. It captures 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,

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

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

Atomic Processes, Systems and Runs. Atomic Dolev-Yao processes, systems, and runs of systems are defined as follows.

An *atomic Dolev-Yao (DY) process* is a tuple

$$p = (I^p, Z^p, R^p, s_0^p)$$

where I^p is the set of addresses the process listens to, Z^p is a set of states (formally, terms), $s_0^p \in Z^p$ is an initial state, and R^p is a relation that takes an event and a state as input and (non-deterministically) returns a new state and a sequence of events. This relation models a computation step of the process, which upon receiving an event in a given state non-deterministically moves to a new state and outputs a set of events. It is required that the events and states in the output can be computed (more formally, derived in the usual Dolev-Yao style) from the current input event and state. An atomic process may also create fresh nonces in a computation step.

The so-called *attacker process* is an atomic DY process which records all messages it receives and outputs all events it can possibly derive from its recorded messages. Hence, an attacker process is the maximally powerful DY process. It carries out all attacks any DY process could possibly perform and is parametrized by the set of sender addresses it may use. Attackers may corrupt other DY processes (e.g., a browser).

A *system* is a set of atomic processes. A *configuration* (S, E, N) of this system consists of the current states of all atomic processes in the system (S), the pool of waiting events (E), and the mentioned sequence of unused nonces (N).

A *run* of a system for an initial sequence of events E^0 is a sequence of configurations, where each configuration (except for the initial one) is obtained by delivering one of the waiting events of the preceding configuration to an atomic process p (which listens to the receiver address of the event), which in turn performs a computation step according to its relation R^p. The initial configuration consists of the initial states of the atomic processes, the sequence E^0, and an initial infinite sequence of unused nonces.

Scripting Processes. The web model also defines scripting processes, which model client-side scripting technologies, such as JavaScript.

A *scripting process* (or simply, a *script*) is defined similarly to a DY process. It is called by the browser in which it runs. The browser provides it with state information s, and the script then, according to its computation relation, outputs a term s', which represents the new internal state and some command which is interpreted by the browser (see also below). Again, it is required that a script's output is derivable from its input.

Similarly to an attacker process, the so-called *attacker script* R^{att} may output everything that is derivable from the input.

4.2. Web System

A web system formalizes the web infrastructure and web applications. Formally, a *web system* is a tuple

$$(\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$$

with the following components:

- The first component, $\mathcal{W}$, denotes a system (a set of DY processes as defined above) and contains honest processes, web attacker, and network attacker processes. While a web attacker can listen to and send messages from its own addresses only, a network attacker may listen to and spoof all addresses (and therefore is the maximally powerful attacker). Attackers may corrupt other parties. In the analysis of a concrete web system, we typically have one network attacker only and no web attackers (as they are subsumed by the network attacker), or one or more web attackers but then no network attacker. In analysis of OAuth we consider network attackers. Honest processes can either be web browsers, web servers, or DNS servers. The modeling of web servers heavily depends on the specific application. The web browser model, which is independent of a specific web application, is presented below.

- The second component, $\mathcal{S}$, is a finite set of scripts, including the attacker script R^{att}. In a concrete model, such as our OAuth model, the set $\mathcal{S} \setminus \{R^{\text{att}}\}$ describes the set of honest scripts used in the web application under consideration while malicious scripts are modeled by the "worst-case" malicious script, R^{att}.

- The third component, script, is an injective mapping from a script in $\mathcal{S}$ to its string representation $\mathsf{script}(s)$ (a constant in Σ) so that it can be part of a messages, e.g., an HTTP response.

- Finally, E^0 is a sequence of events, which always contains an infinite number of events of the form $\langle a, a, \texttt{TRIGGER}\rangle$ for every IP address a in the web system.

A *run* of the web system is a run of $\mathcal{W}$ initiated by E^0.

4.3. Web Browsers

We now sketch the model of a web browser. A web browser is modeled as a DY process (I^p, Z^p, R^p, s_0^p).

An honest browser is thought to be used by one honest user, who is modeled as part of the browser. User actions are modeled as non-deterministic actions of the web browser. For example, the browser itself non-deterministically follows the links in a web page. User data (i.e., passwords and identities) is stored in the initial state of the browser and is given to a web page when needed, similar to the AutoFill feature in browsers.

Besides the user identities and passwords, the state of a web browser (modeled as a term) contains a tree of open windows and documents, lists of cookies, localStorage and sessionStorage data, a DNS server address, and other data.

In the browser state, the *windows* subterm is the most complex one. It contains a window subterm for every open window (of which there may be many at a time), and inside each window, a list of documents, which represent the history of documents that have been opened in that window, with one of these documents being active, i.e., this document is presented to the user and ready for interaction. A document contains a script loaded from a web server and represents one loaded HTML page. A document also contains a list of windows itself, modeling iframes. Scripts may, for example, navigate or create windows, send XHRs and postMessages, submit forms, set/change cookies, localStorage, and sessionStorage data, and create iframes. When activated, the browser provides a script with all data it has access to, such as a (limited) view on other documents and windows, certain cookies as well as localStorage and sessionStorage.

Figure 7 shows a brief overview of the browser relation R^p which defines how browsers behave. For example, when a TRIGGER message is delivered to the browser, the browser non-deterministically choses an *action*. If, for instance, this action is 1, then an active document is selected non-deterministically,

PROCESSING INPUT MESSAGE m

$m =$ `FULLCORRUPT`: *isCorrupted* := `FULLCORRUPT`

$m =$ `CLOSECORRUPT`: *isCorrupted* := `CLOSECORRUPT`

$m =$ `TRIGGER`: non-deterministically choose *action* from $\{1,2,3\}$

action = 1: Call script of some active document.
Outputs new state and *command*.

command = `HREF`: $\rightarrow$ *Initiate request*

command = `IFRAME`: Create subwindow, $\rightarrow$ *Initiate request*

command = `FORM`: $\rightarrow$ *Initiate request*

command = `SETSCRIPT`: Change script in given document.

command = `SETSCRIPTSTATE`: Change state of script in given document.

command = `XMLHTTPREQUEST`: $\rightarrow$ *Initiate request*

command = `BACK` or `FORWARD`: Navigate given window.

command = `CLOSE`: Close given window.

command = `POSTMESSAGE`: Send postMessage to specified document.

action = 2: $\rightarrow$ *Initiate request to some URL in new window*

action = 3: $\rightarrow$ *Reload some document*

$m =$ DNS response: send corresponding HTTP request

$m =$ HTTP(S) response: (decrypt,) find reference.

reference to window: create document in window

reference to document: add response body to document's script input

Figure 7. The basic structure of the web browser relation R^p with an extract of the most important processing steps, in the case that the browser is not already corrupted.

and its script is triggered. The script (with inputs as outlined above), can now output a command, for example, to follow a hyperlink (`HREF`). In this case, the browser will follow this link by first creating a new DNS request. Once a response to that DNS request arrives, the actual HTTP request (for the URL defined by the script) will be sent out. After a response to that HTTP request arrives, the browser creates a new document from the contents of the response. Complex navigation and security rules ensure that scripts can only manipulate specific aspects of the browser's state. Browsers can become corrupted, i.e., be taken over by web and network attackers. The browser model comprises two types of corruption: *close-corruption*, modeling that a browser is closed by the user, and hence, certain data is removed (e.g., session cookies and opened windows), before it is taken over by the attacker, and *full corruption*, where no data is removed in advance. Once corrupted, the browser behaves like an attacker process.

5. Analysis of OAuth 2.0

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 for authorization and authentication and state the main theorem, namely the security of OAuth w.r.t. these properties. We also sketch the proof of this theorem, with full details provided in Appendix G.

5.1. Model of OAuth 2.0

As mentioned above, our model for OAuth is based on the generic web model outlined in Section 4. We developed the OAuth model to adhere to RFC6749, the OAuth 2.0 standard, and follow the security considerations described in [19].

OAuth Options and Implementation Details. For the options and implementation details that are left open by the OAuth standard and the security considerations, we made the following design decisions:

OAuth Modes. As mentioned, the OAuth standard does not fix which subset of modes RPs and IdPs use and support. We therefore model that every RP, IdP, and browser may run any of the four OAuth modes, even simultaneously.

CSRF Protection. The *state* parameter is used with a freshly chosen nonce that is bound to the user's session in order to prevent Cross-Site Request Forgery attacks on the RP. This is recommended by the standard.

Redirection URLs. In a run of the protocol, the RP chooses a redirection URL explicitely or the IdP selects a redirection URL that was registered before. Redirection URLs can be defined using patterns. This covers all cases specified in the OAuth standard.

Client Secrets. Clients can, for a certain IdP, have a secret or not have a secret in our model. And hence, both options in the OAuth standard regarding client secrets are captured in our model

Scope. As described in Section 2.1, we assume that RPs always get full access to the user's data at the IdP.

HTTPS Endpoints. Following the security recommendations, all endpoint URLs use HTTPS. Obviously, IdPs or RPs do not register URLs that point to servers other than their own.

Session Mechanism. The OAuth standard and the OAuth security recommendations, as mentioned, do not prescribe a specific session mechanism to be used at an RP. Our model therefore includes a standard cookie-based session mechanism which follows best practices. For example, cookies are always set as session cookies (the cookie is discarded when the browser window is closed), with the attribute *HTTP only* (the cookie value is not accessible by JavaScript), and, for HTTPS connections, *secure* (the cookie value is only transmitted over HTTPS). After successful login at an RP, the RP starts a new session. In our model, this second session is modeled using a *service token* that is sent to the browser (see below).

Authentication to the IdP. User authentication to the IdP, which is not part of the OAuth standard, is performed using username and password. 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 (the user does not re-use her password for different IdPs).

Authentication to the RP. To log a user in, an RP typically requests a unique user identifier from an IdP, as already mentioned above. We follow this common practice also in our model.

General User Interaction. User interaction (entering the user identifier at the RP, user credentials at the IdP) is modeled as simple as possible (i.e., regular HTML forms).[14]

[14] Of course, this does not limit the possibilities for user interaction, as defined in the general web model. For example, the user can at any time navigate backwards or forward in her browser history, or navigate to any web page.

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

Protected Resources. 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 protected resources, where the identity part can be $\bot$, modeling a resource that is acquired in the client credentials mode and thus not bound to a user.

Generic Model for Session Mechanisms. 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 users 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) since the actual session mechanism used by the RP *after* a successful login is out of the scope of OAuth and our analysis. In our model, the service token is not stored by the browser.

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. (Formally, it is a mapping from passwords to sets of RPs, where every password belongs to a specific user.) In particular, whether or not an RP is trusted depends on the considered user. In our security properties, when, for example, 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 ($\mathcal{OWS}$). We model OAuth as a web system (in the sense of Section 4). We call a web system $\mathcal{OWS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ an *OAuth web system* if it is of the form described in what follows.

Outline. The system $\mathcal{W} = \mathsf{Hon} \cup \mathsf{Net}$ consists of a network attacker (in Net), a finite set B of web browsers, a finite set RP of web servers for the RPs, a finite set IDP of web servers for the IdPs, with $\mathsf{Hon} := \mathsf{B} \cup \mathsf{RP} \cup \mathsf{IDP}$. Recall that since we have a network attacker, we do not need to consider web attackers (as our network attacker subsumes all web attackers). The set $\mathcal{S}$ consists of the scripts *script_rp_index*, *script_rp_implicit*, and *script_idp_form*, with their respective string representations defined in script as `script_rp_index`, `script_rp_implicit`, and `script_idp_form`. The set E^0 is defined as described in Section 4.2. We now briefly sketch the models of RPs, IdPs, and the scripts, with more details provided in the appendix and full details provided in the appendix.

Relying Parties. Each relying party is a web server modeled as an atomic DY process following the description in Section 2 and the fixes discussed in Section 3. 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 login phase of a user, and the *service sessions* (modeled by a *service token* as described above).

When receiving the special message `CORRUPT` an RP can become corrupted. Similar to the definition of corruption for the browser, the RP then starts sending out all messages that are derivable from its state.

Identity Providers. Each IdP is a web server modeled as an atomic DY process following the description in Section 2 and the fixes discussed in Section 3. In particular, users can authenticate to an IdP with their credentials. Authenticated users can interact with the authorization endpoint of the IdP (e.g., to acquire an authorization code). Just as RPs, IdPs can become corrupted.

Scripts. The scripts which run in a user's browser (and their respective string represenations) are defined as follows:

script_rp_index. This script is loaded from an RP into a user's browser when the user visits RPs web site. It starts the authorization or login process.

script_rp_implicit. This script is loaded into the user's browser from an RP during an implicit mode flow to retrieve the data from the URL fragment. It extracts the access token and state from the fragment part of its own URL. The script then sends this information in the body of an HTTPS POST request to the RP.

script_idp_form. This script is loaded from an IdP into the user's browser and models the form where the user enters her credentials to authenticate to the IdP.

5.2. Authorization and Authentication Properties

Based on the formal OAuth model described above, we now formalize central security properties of OAuth, namely authorization and authentication (see Appendix F for full details). We note that these properties do not capture the possibility that an attacker forcefully logs in a user under the attacker's account. This is indeed often possible in web applications as long as cookies are used in the login process (see, e.g., [30]).

Authorization. Intuitively, authorization for $\mathcal{OWS}$ means that an attacker should not be able to obtain a protected resource available to some honest RP at an IdP for some user unless certain parties involved in the authorization process are corrupted.

More formally, we say that $\mathcal{OWS}$ is *secure w.r.t. authorization* if the following holds true: if at any point in a run of $\mathcal{OWS}$ an attacker can derive 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 $\mathcal{OWS}$ means that an attacker should not be able to login at an (honest) RP under the identity of a user unless certain parties involved in the login process are 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 $\mathcal{OWS}$ is *secure w.r.t. authentication* if the following holds true: if at any point in a run of $\mathcal{OWS}$ an attacker can derive 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.

5.3. Main Theorem

We prove the following theorem:

Theorem 1. Let $\mathcal{OWS}$ be an OAuth web system, then $\mathcal{OWS}$ is secure w.r.t. authorization and secure w.r.t. authentication.

Proof Outline (see Appendix G for the full proof). We first show three lemmas that apply to honest RPs and capture specific technical details. The first one says that, roughly speaking, messages transferred over HTTPS connections that were initiated by honest RPs cannot be read or altered by other parties. In particular, honest RPs do not leak the encryption keys to other parties. In the second lemma, we show that HTTP(S) messages which await DNS resolution in an honest RP's state are later sent out over the network without being altered in between. In the third lemma, we show that honest RPs never send

HTTP messages to other RPs or themselves, and that they only send HTTPS messages that other RPs cannot decrypt.

Authentication. We first prove the authentication property, which we do by contradiction. To this end, we show in three separate lemmas building on each other that (1) the attacker does not learn passwords of the user, (2) the attacker does not learn authorization codes that could be used to learn a relevant access token, and (3) that the attacker in fact does not learn an access token that could be used to retrieve a service token as described in the authentication property. We finally show that there is no other way for an attacker to get hold of a service token (of the kind described in the authentication property), and that therefore, the authentication property must hold true.

Authorization. As above, we assume that the authorization property does not hold and lead this to a contradication. The proof then builds upon lemmas shown in the authentication proof. We show that the attacker would need to know an access token to acquire a protected resource. If the protected resource is bound to a user (i.e., it was not issued in the client credentials mode), then (3) from above applies and shows that the attacker cannot learn such an access token, and thus cannot learn this protected resource. If the protected resource was not assigned to a user (i.e., it was issued in the client credentials mode), then we can show that the attacker would need to know client secrets to get the protected resource. We show, however, that it is not possible for the attacker to learn the necessary client secrets (which are always required in the client credentials mode). Therefore, whether it is a user-bound protected resource or not, the attacker cannot learn it, leading our assumption to a contradiction.

6. Related Work

As already mentioned in the introduction, security analysis of OAuth has so far concentrated on finding bugs in specific OAuth implementations, rather than on performing a detailed security analysis of the OAuth standard itself in a comprehensive web model.

In [4], Bansal, Bhargavan and Maffeis analyze the security of OAuth using the applied pi-calculus and the WebSpi library. They model settings of OAuth 2.0 that can be found in practice and assume the presence of common web vulnerabilities such as Cross-Site Request Forgery and open redirectors in RPs and IdPs. Using this method, they identify previously unknown attacks (mostly CSRF attacks) on the OAuth implementations of Facebook, Yahoo, and Twitter. The main goal of [4] was to find concrete vulnerabilities in implementations that can arise when other vulnerabilities in RPs and IdPs are present. Also, compared to our work, the WebSpi model used in [4] is less expressive and comprehensive, and the models of OAuth are more limited and aimed at specific implementations (for example, only one mode is considered at any time and malicious IdPs are not considered). While in [4] an automated tool is used to find the vulnerabilities, our work is based on manual proofs.

Wang et al. [29] 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 [21], Pai et al. analyze the security of OAuth in a (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 [17] in a formal analysis of the older OAuth 1.0 protocol (which, as mentioned in the introduction, is very different to OAuth 2.0).

Chari, Jutla, and Roy [5] 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 [28], Sun and Beznosov analyze the security of three IdPs and 96 RPs. In [18], Li and Mitchell

study the security of 10 IdPs and 60 RPs based in China. In [6, 26] practical evaluations on the security of OAuth implementaions of mobile apps are performed.

In [20], Mladenov et al. perform an informal analysis of the OpenID Connect extensions discovery and dynamic client registration and present an attack which assumes a CSRF vulnerability on the RP's side. On a high level, this attack resembles the IdP mix-up attack when applied to OpenID Connect. It exploits, however, a different flaw in the dynamic registration extension and is not applicable to OAuth 2.0. Mladenov et al. do not perform formal analysis and they do not consider OAuth itself, which is the focus of our work.

Note that many of the works listed here led to improved security recommendations for OAuth as listed in RFC6749 [13] and RFC6819 [19] which are thus incorporated in our model of OAuth.

Altogether there have been only very few analysis efforts for web applications and standards based on formal web models. Besides those listed above, important work includes [2, 3, 8, 10, 11, 16].

7. Conclusion

In this paper, we carried out the first 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 two previously unknown attacks on OAuth as well as OpenID connect, which builds on OAuth. We have verified both attacks on an actual implementation and reported both attacks, along with propositions for fixes, to the respective working groups for OAuth and OpenID Connect. The working groups confirmed the attacks and adopted our fixes.

With the fixes applied, we were able to prove strong authorization and authentication properties for OAuth 2.0. This, in particular, shows that OAuth, when implemented correctly, can provide authorization and authentication in practice even though it was not primarily designed for the latter. The fact that OAuth is one of the most widely deployed authorization and authentication systems in the web 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.

Acknowledgment. This work was in part funded by *Deutsche Forschungsgemeinschaft* (DFG) through Grant KU 1434/10-1.

References

[1] M. Abadi and C. Fournet. Mobile Values, New Names, and Secure Communication. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages (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 *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010*, pages 290–304. IEEE Computer Society, 2010.

[3] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage. In D. A. Basin and J. C. Mitchell, editors,

Principles of Security and Trust - Second International Conference, POST 2013, volume 7796 of *Lecture Notes in Computer Science*, pages 126–146. Springer, 2013.

[4] C. Bansal, K. Bhargavan, and S. Maffeis. Discovering Concrete Attacks on Website Authorization by Formal Analysis. In S. Chong, editor, *25th IEEE Computer Security Foundations Symposium, CSF 2012*, pages 247–262. IEEE Computer Society, 2012.

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

[6] E. Y. Chen, Y. Pei, S. Chen, Y. Tian, R. Kotcher, and P. Tague. OAuth Demystified for Mobile Application Developers. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14*, pages 892–903, 2014.

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

[8] D. Fett, R. Küsters, and G. Schmitz. An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System. In *35th IEEE Symposium on Security and Privacy (S&P 2014)*, pages 673–688. IEEE Computer Society, 2014.

[9] D. Fett, R. Küsters, and G. Schmitz. Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web. Technical Report arXiv:1411.7210, arXiv, 2014. http://arxiv.org/abs/1411.7210.

[10] D. Fett, R. Küsters, and G. Schmitz. Analyzing the BrowserID SSO System with Primary Identity Providers Using an Expressive Model of the Web. In *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part I*, Lecture Notes in Computer Science, pages 43–65. Springer, 2015.

[11] D. Fett, R. Küsters, and G. Schmitz. SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 1358–1369. ACM, 2015.

[12] 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.

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

[14] HTML5, W3C Recommendation. Oct. 28, 2014.

[15] P. Jones, G. Salgueiro, M. Jones, and J. Smarr. RFC7033 – WebFinger. IETF. Sep. 2013. https://tools.ietf.org/html/rfc7033.

[16] F. Kerschbaum. Simple Cross-Site Attack Prevention. In *Third International Conference on Security and Privacy in Communication Networks and the Workshops, SecureComm 2007*, pages 464–472. IEEE Computer Society, 2007.

[17] A. Kumar. Using automated model analysis for reasoning about security of web protocols. In *Proceedings of the 28th Annual Computer Security Applications Conference on - ACSAC'12*. Association for Computing Machinery (ACM), 2012.

[18] W. Li and C. J. Mitchell. Security issues in OAuth 2.0 SSO implementations. In *Information Security - 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings*, pages 529–541, 2014.

[19] 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.

[20] V. Mladenov, C. Mainka, J. Krautwald, F. Feldmann, and J. Schwenk. On the security of modern Single Sign-On Protocols: OpenID Connect 1.0. *CoRR*, abs/1508.04324, 2015.

[21] S. Pai, Y. Sharma, S. Kumar, R. M. Pai, and S. Singh. Formal Verification of OAuth 2.0 Using Alloy Framework. In *CSNT '11 Proceedings of the 2011 International Conference on Communication Systems and Network Technologies*, pages 655–659. Proceedings of the International Conference on Communication Systems and Network Technologies, 2011.

[22] N. Sakimura, J. Bradley, and M. Jones. OpenID Connect Dynamic Client Registration 1.0 incorporating errata set 1. OpenID Foundation. Nov. 8, 2014. http://openid.net/specs/openid-connect-registration-1_0.html.

[23] 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.

[24] N. Sakimura, J. Bradley, M. Jones, and E. Jay. OpenID Connect Discovery 1.0 incorporating errata set 1. OpenID Foundation. Nov. 8, 2014. http://openid.net/specs/openid-connect-discovery-1_0.html.

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

[26] M. Shehab and F. Mohsen. Towards Enhancing the Security of OAuth Implementations in Smart Phones. In *2014 IEEE International Conference on Mobile Services*. Institute of Electrical & Electronics Engineers (IEEE), jun 2014.

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

[28] S.-T. Sun and K. Beznosov. The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. In T. Yu, G. Danezis, and V. D. Gligor, editors, *ACM Conference on Computer and Communications Security, CCS'12*, pages 378–390. ACM, 2012.

[29] R. Wang, Y. Zhou, S. Chen, S. Qadeer, D. Evans, and Y. Gurevich. Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 399–314. USENIX Association, 2013.

[30] X. Zheng, J. Jiang, J. Liang, H. Duan, S. Chen, T. Wan, and N. Weaver. Cookies Lack Integrity: Real-World Implications. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 707–721, Washington, D.C., Aug. 2015. USENIX Association.

A. OpenID Connect and the Attacks on this Standard

We here provide a more detailed description of the OpenID Connect standard as well as on the two attacks, 307 redirect and IdP mix-up, on it.

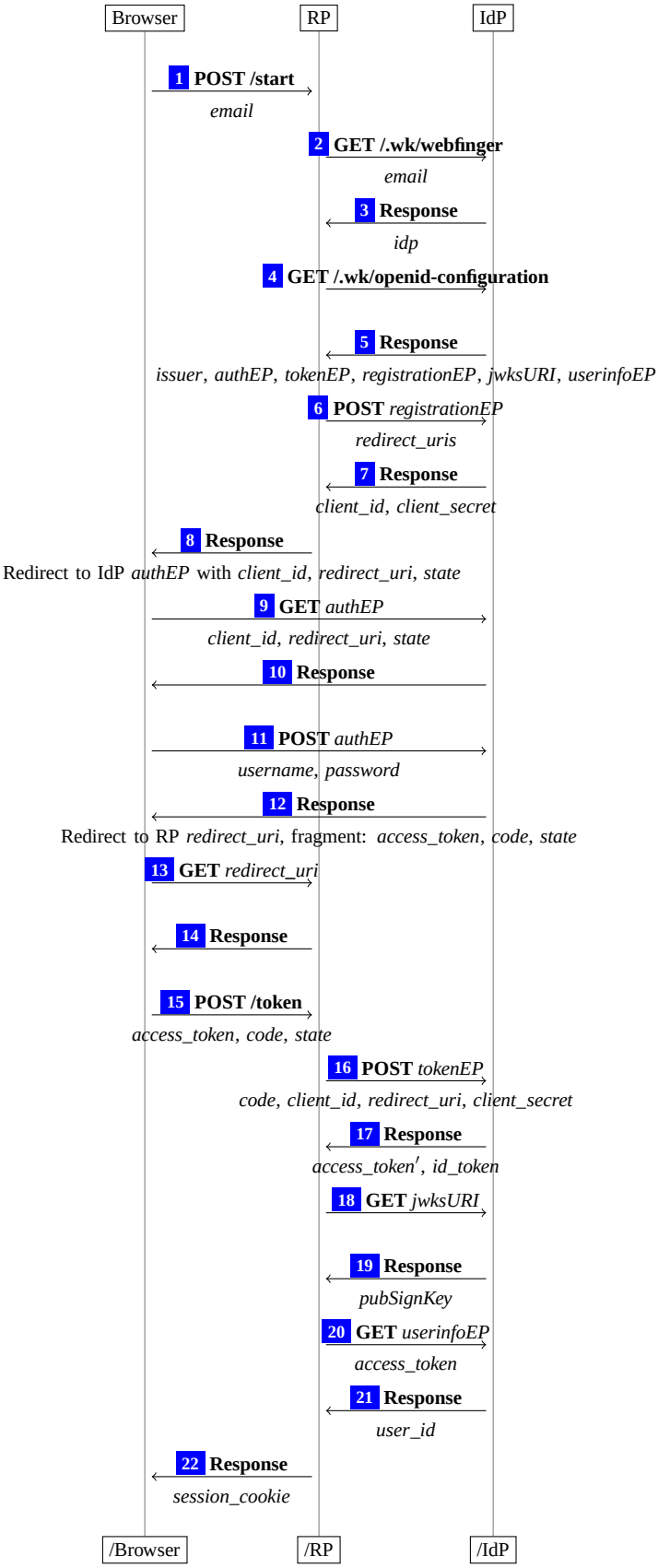

Figure 8. OpenID Connect 1.0 hybrid mode with discovery and dynamic client registration

A.1. Modes and Protocol Flow

OpenID Connect makes use of the OAuth authorization code mode and the implicit mode (both OAuth modes constitute an OpenID Connect mode), but also introduces a new *hybrid* mode, which combines both modes.

Overview. From a high-level perspective, first, the RP retrieves meta data about the IdP, such as the URLs of the IdP used in the protocol. This is the information that is "hard-wired" in the manual, out-of-band registration in a classic OAuth setup. Next, the RP automatically registers itself as an OAuth client at the IdP (using OpenID Connect dynamic client registration). Then, the OAuth protocol is started (using one of the modes mentioned above). In addition to an access token this (extended) run delivers a so-called *id token* to RP. The id token is issued by the IdP and contains a unique user identifier along with several meta data, such as the intended receiver (the RP) of the id token and the issuer of the id token (the IdP). The id token is (optionally) signed by the IdP. Finally, the RP can retrieve more meta data about the user at the *userinfo* endpoint at the IdP using the access token and consider the user to be logged in.

Step-by-Step Protocol Flow. In the step-by-step description below (see also Figure 8), we focus on the hybrid mode only. First, the user starts the login process by entering her email address[15] in her browser (at some web page of an RP), which sends the email address to the RP in 1.

Now, the RP uses the OpenID Connect Discovery protocol [24] to gain information about the IdP: The RP uses the WebFinger [15] mechanism to discover information about which IdP is responsible for this user. For this discovery, the RP contacts the server of the user's email domain (depicted as the same party as the IdP in the figure) in 2. The result of the WebFinger request in 3 contains the domain of the server responsible for the OpenID Connect configuration (the IdP). The configuration is requested from the IdP in 4 and returned in 5. The configuration contains meta data about the IdP, including all endpoints at the IdP. This concludes the OpenID Discovery in this login flow.

Next, if the RP is not registered at the IdP, the RP starts the OpenID Connect dynamic client registration [22] protocol: the RP contacts the IdP in 6 providing its redirect URIs. Now, the IdP issues an (OAuth) client id and (optionally) an (OAuth) client secret to the RP in 7. This concludes the OpenID Connect dynamic client registration.

Now, the core part of the OpenID Connect protocol (based on OAuth) starts: the RP redirects the user's browser to the IdP in 8. This redirect contains information that the hybrid mode is used and which tokens are requested. In this description, we assume that an authorization code and an access token are requested.[16] Also, this redirect contains the (OAuth) client id of the RP, a redirect URI and a state value. As in the OAuth flows, this data is sent to the IdP 9, the user authenticates to the IdP 10, 11, and the IdP redirects the user's browser back to the RP in 12 and 13 (using the redirect URI from the request in 9). This redirect contains an authorization code, an access token, and the state value in the fragment part of the URL.[17] Now, the RP in 14 sends a document containing JavaScript code which sends the parameters contained in the fragment back to the RP (in 15). If the state value matches, the RP contacts the IdP in 16 with the received authorization code, its (OAuth) client id, its (OAuth) client secret, and the redirect URI used to obtain the authorization code. The IdP sends a response with the same or a fresh access token and an id token to the RP in 17. Now, the RP retrieves the key that was used to sign the id token from the IdP in 18 and 19 and verifies the id token's signature. As the id token typically contains only a unique user identifier, but no other meta data about the user, RP requests this meta data (such as nickname, birthday, or address) from the IdP in 20 and 21 using one of the

[15] Note that OpenID Connect also allows other types of user identifiers, such as a personal URL.

[16] The Hybrid Flow allows to request several different combinations of authorization code, access token, and id token.

[17] Note that depending on the parameters in step 9, also an id token may be contained in the fragment part of the URL.

authorization tokens received before. Finally, the RP considers the user to be logged in and may set a session cookie at the user's browser in [22].

Note that the authorization code mode and the implicit mode are similar to the hybrid mode: Roughly speaking, the Steps [12]–[17] of the OpenID Connect hybrid mode are replaced by the corresponding steps of the OAuth authorization code or implicit mode, respectively. These OAuth modes are then extended with the transfer of an id token. In the authorization code mode, the id token is appended to the response [9] of Figure 1 and in the implicit mode, the id token is appended to the fragment of the redirect URI in [6] of Figure 2 (and later sent to the RP in Step [9]).

A.2. The 307 Redirect Attack

The 307 redirect attack presented in Section 3.1 can also be applied to OpenID Connect. Note that the critical part of OAuth, namely the redirect of the user's browser from the IdP to the RP after the authentication of the user to the IdP, is also present in OpenID Connect. Hence, if the IdP uses an HTTP status 307 redirect immediately after the user's browser has transferred the user's credentials to IdP in a POST request, the RP receives these credentials.

A.3. The IdP Mix-Up Attack

When applying the attack presented in Section 3.2 to OpenID Connect, the attacker needs to circumvent some additional security measures: In the implicit mode of OpenID Connect, an *id_token* (as described above) is sent along with *access_token* in the redirect from HIdP to the RP. As this redirect might use HTTPS, the attacker cannot inspect or modify the corresponding network messages. As mentioned above, the id token contains the domain of the issuer of both, the access token and the id token. Therefore, the RP can detect that the user did not use AIdP (which the RP redirected to).

An attacker could try to use the authorization code mode of OpenID Connect to mount a similar attack as described above. In this case, however, the attacker does not learn a valid access token for the user's account at HIdP if a client secret is used.

In the hybrid mode, however, an attacker can learn an access token and mount the attack as follows (see also Figure 9):

As above, the user first visits the RP. When the user sends her email address to the RP in order to login [1], the attacker manipulates the domain part of the email address, to be the domain of AIdP [2]. The RP then looks up the IdP to be used (which is now AIdP) using the WebFinger protocol in Steps [3] and [4]. The RP fetches the OpenID Connect configuration from the attacker ([5] and [6]). In this document, the attacker states that the authorization endpoint is located at HIdP while all other endpoints are located at the attacker. Using parameters not shown in the figures, the attacker can also state that this IdP does not support delivering an id token in the redirect and can state that no signatures are supported. Since no signatures need to be checked, also the key retrieval is skipped in the protocol.

After retrieving the OpenID configuration, the RP registers at AIdP, as the attacker uses a domain previously unknown to the RP. (If the domain was known to the RP, this step would be skipped.) The attacker issues the same *client_id* with which the RP is registered at HIdP ([7] and [8]). Now, the RP redirects the user's browser to HIdP in order to log in. After the user authenticated to HIdP, HIdP redirects the user's browser back to the RP. The fragment part of the URL contains an authorization code and an access token [14]. The RP then sends the authorization code to the attacker in [17].

If the RP does not have a client secret registered at HIdP, the attacker can redeem this authorization code at HIdP in order to receive an access token to access the honest user's protected resources at HIdP. This breaks the authorization of OpenID Connect (compare the OAuth authorization property in Section 5.2).

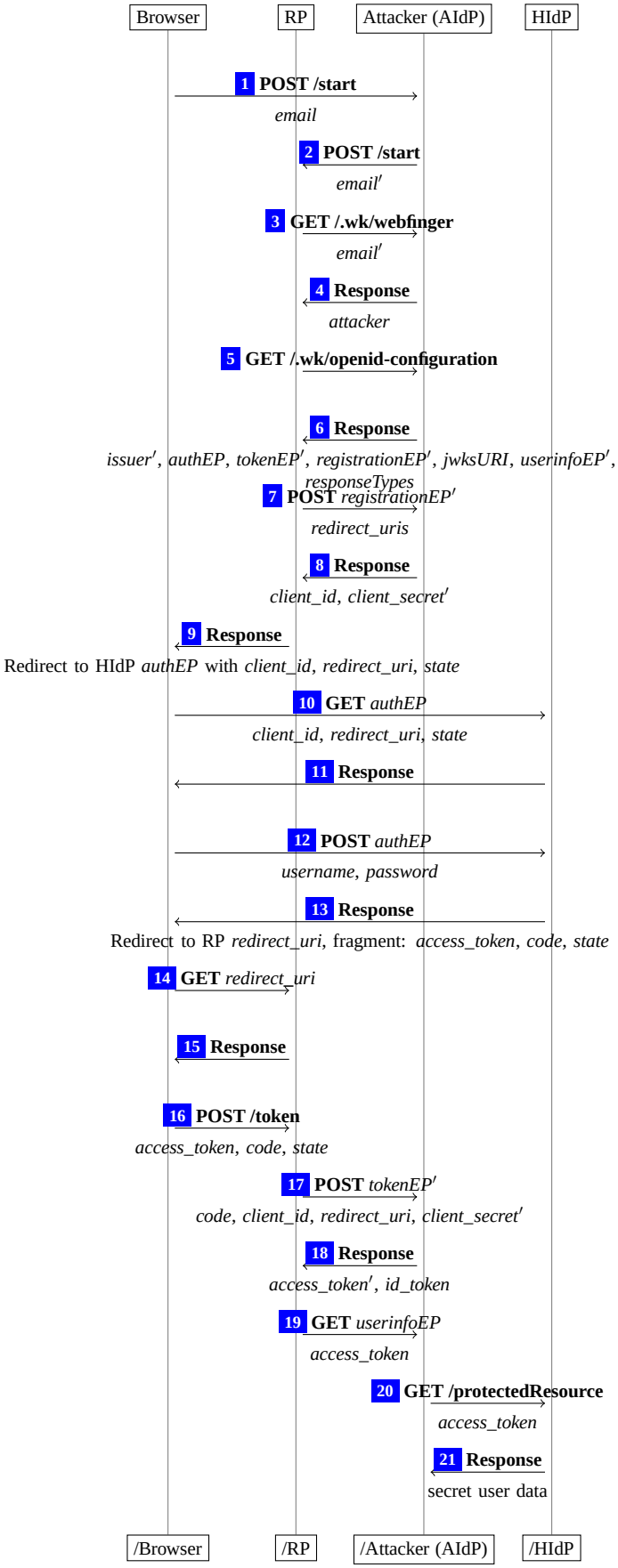

Figure 9. Attack on OpenID Connect 1.0 hybrid mode with discovery and dynamic client registration

Alternatively, the attacker responds to the RP with a faked access token and a faked id token [18] (which the attacker can create, because he controls all security settings for this id token, see Step [6]).

Next, the RP retrieves other meta information about the user from AIdP. The RP is now in possession of two access tokens. The OpenID Connect standard explicitly allows this situation, but fails to state which access token has to be used in subsequent requests. The RP can now chose either of the access tokens for the next steps, with different outcomes for the attacker:

First Access Token is Selected. In this case the access token originating from HIdP is selected by the RP and sent to the attacker [19]. (This behavior was observed by us in the real-world implementation mod_auth_openidc.)

Now the attacker can use this access token to access other protected resources of the user at HIdP. This breaks authorization for OpenID Connect (compare our OAuth authorization property in Section 5.2).

Second Access Token is Selected. In this case the access token originating from AIdP is selected. This means that the attacker does not learn a valid access token for HIdP. The attacker can, however, reuse the authorization code for HIdP, which he learned in [17] and which is still valid as it has not been redeemed at HIdP, yet. Using this method, the attacker can impersonate the honest user at the RP. To accomplish this, the attacker starts a new login flow at the RP with the user's email address. In Step [15] of Figure 8, he provides the authorization code he has learned along with some (invalid) access token and the state from his (new) login flow to the RP. The RP then requests an access token and an id token from HIdP with this (still valid) authorization code. The RP receives a valid access token and a valid id token (for the honest user) from HIdP. As the RP uses this valid access token in this case, all subsequent requests from the RP to HIdP are successful and the RP receives the user id of the honest user, the RP considers the attacker to be logged in as the honest user. This breaks the authentication of OpenID Connect (compare our OAuth authentication property in Section 5.2).

B. The Generic Web Model

In this section, we present the model of the web infrastructure as proposed in [8] and [9], along with the following changes and additions:

- We introduce a new header, `Authorization`, as a model for HTTP Basic Authentication.[18]
- Browsers now may have multiple passwords stored for a single origin; before, there was only one password for each origin.

B.1. Communication Model

We here present details and definitions on the basic concepts of the communication model.

Terms, Messages and Events. The signature Σ for the terms and messages considered in this work is the union of the following pairwise disjoint sets of function symbols:

- constants $C = \mathsf{IPs} \cup \mathbb{S} \cup \{\top, \bot, \diamond\}$ where the three sets are pairwise disjoint, $\mathbb{S}$ is interpreted to be the set of ASCII strings (including the empty string ε), and IPs is interpreted to be a set of (IP) addresses,
- function symbols for public keys, (a)symmetric encryption/decryption, and signatures: $\mathsf{pub}(\cdot)$, $\mathsf{enc_a}(\cdot,\cdot)$, $\mathsf{dec_a}(\cdot,\cdot)$, $\mathsf{enc_s}(\cdot,\cdot)$, $\mathsf{dec_s}(\cdot,\cdot)$, $\mathsf{sig}(\cdot,\cdot)$, $\mathsf{checksig}(\cdot,\cdot)$, and $\mathsf{extractmsg}(\cdot)$,

[18] Note that although the header is called "Authorization" (following RFC2617), this is a mechanism for authentication.

$$\mathsf{dec_a}(\mathsf{enc_a}(x,\mathsf{pub}(y)),y) = x \tag{1}$$
$$\mathsf{dec_s}(\mathsf{enc_s}(x,y),y) = x \tag{2}$$
$$\mathsf{checksig}(\mathsf{sig}(x,y),x,\mathsf{pub}(y)) = \top \tag{3}$$
$$\pi_i(\langle x_1,\ldots,x_n\rangle) = x_i \text{ if } 1 \le i \le n \tag{4}$$
$$\pi_j(\langle x_1,\ldots,x_n\rangle) = \diamond \text{ if } j \notin \{1,\ldots,n\} \tag{5}$$

Figure 10. Equational theory for Σ.

- n-ary sequences $\langle\rangle, \langle\cdot\rangle, \langle\cdot,\cdot\rangle, \langle\cdot,\cdot,\cdot\rangle$, etc., and
- projection symbols $\pi_i(\cdot)$ for all $i \in \mathbb{N}$.

For strings (elements in $\mathbb{S}$), we use a specific font. For example, `HTTPReq` and `HTTPResp` are strings. We denote by $\mathsf{Doms} \subseteq \mathbb{S}$ the set of domains, e.g., `example.com` $\in \mathsf{Doms}$. We denote by $\mathsf{Methods} \subseteq \mathbb{S}$ the set of methods used in HTTP requests, e.g., `GET`, `POST` $\in \mathsf{Methods}$.

The equational theory associated with the signature Σ is given in Figure 10.

Definition 1 (Nonces and Terms). By $X = \{x_0, x_1, \ldots\}$ we denote a set of variables and by $\mathcal{N}$ we denote an infinite set of constants (*nonces*) such that Σ, X, and $\mathcal{N}$ are pairwise disjoint. For $N \subseteq \mathcal{N}$, we define the set $\mathcal{T}_N(X)$ of *terms* over $\Sigma \cup N \cup X$ inductively as usual: (1) If $t \in N \cup X$, then t is a term. (2) If $f \in \Sigma$ is an n-ary function symbol in Σ for some $n \ge 0$ and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

By $\equiv$ we denote the congruence relation on $\mathcal{T}_{\mathcal{N}}(X)$ induced by the theory associated with Σ. For example, we have that $\pi_1(\mathsf{dec_a}(\mathsf{enc_a}(\langle \mathsf{a}, \mathsf{b}\rangle, \mathsf{pub}(k)), k)) \equiv \mathsf{a}$.

Definition 2 (Ground Terms, Messages, Placeholders, Protomessages). By $\mathcal{T}_N = \mathcal{T}_N(\emptyset)$, we denote the set of all terms over $\Sigma \cup N$ without variables, called *ground terms*. The set $\mathcal{M}$ of messages (over $\mathcal{N}$) is defined to be the set of ground terms $\mathcal{T}_{\mathcal{N}}$.

We define the set $V_{\text{process}} = \{\nu_1, \nu_2, \ldots\}$ of variables (called placeholders). The set $\mathcal{M}^{\nu} := \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$ is called the set of *protomessages*, i.e., messages that can contain placeholders.

Example 1. For example, $k \in \mathcal{N}$ and $\mathsf{pub}(k)$ are messages, where k typically models a private key and $\mathsf{pub}(k)$ the corresponding public key. For constants a, b, c and the nonce $k \in \mathcal{N}$, the message $\mathsf{enc_a}(\langle a, b, c\rangle, \mathsf{pub}(k))$ is interpreted to be the message $\langle a, b, c\rangle$ (the sequence of constants a, b, c) encrypted by the public key $\mathsf{pub}(k)$.

Definition 3 (Normal Form). Let t be a term. The *normal form* of t is acquired by reducing the function symbols from left to right as far as possible using the equational theory shown in Figure 10. For a term t, we denote its normal form as $t{\downarrow}$.

Definition 4 (Pattern Matching). Let $\mathit{pattern} \in \mathcal{T}_{\mathcal{N}}(\{*\})$ be a term containing the wildcard (variable $*$). We say that a term t *matches pattern* iff t can be acquired from *pattern* by replacing each occurrence of the wildcard with an arbitrary term (which may be different for each instance of the wildcard). We write $t \sim \mathit{pattern}$. For a sequence of patterns *patterns* we write $t \dot{\sim} \mathit{patterns}$ to denote that t matches at least one pattern in *patterns*.

For a term t' we write $t'|\mathit{pattern}$ to denote the term that is acquired from t' by removing all immediate subterms of t' that do not match *pattern*.

Example 2. For example, for a pattern $p = \langle \top, * \rangle$ we have that $\langle \top, 42 \rangle \sim p$, $\langle \bot, 42 \rangle \not\sim p$, and

$$\langle \langle \bot, \top \rangle, \langle \top, 23 \rangle, \langle \mathsf{a}, \mathsf{b} \rangle, \langle \top, \bot \rangle \rangle \,|\, p = \langle \langle \top, 23 \rangle, \langle \top, \bot \rangle \rangle \ .$$

Definition 5 (Variable Replacement). Let $N \subseteq \mathcal{N}$, $\tau \in \mathcal{T}_N(\{x_1, \ldots, x_n\})$, and $t_1, \ldots, t_n \in \mathcal{T}_N$.

By $\tau[t_1/x_1, \ldots, t_n/x_n]$ we denote the (ground) term obtained from τ by replacing all occurrences of x_i in τ by t_i, for all $i \in \{1, \ldots, n\}$.

Definition 6 (Events and Protoevents). An *event (over* IPs *and* $\mathcal{M}$*)* is a term of the form $\langle a, f, m \rangle$, for a, $f \in \mathsf{IPs}$ and $m \in \mathcal{M}$, where a is interpreted to be the receiver address and f is the sender address. We denote by $\mathcal{E}$ the set of all events. Events over IPs and $\mathcal{M}^\nu$ are called *protoevents* and are denoted $\mathcal{E}^\nu$. By $2^{\mathcal{E}\langle\rangle}$ (or $2^{\mathcal{E}^\nu\langle\rangle}$, respectively) we denote the set of all sequences of (proto)events, including the empty sequence (e.g., $\langle\rangle$, $\langle \langle a, f, m \rangle, \langle a', f', m' \rangle, \ldots \rangle$, etc.).

Atomic Processes, Systems and Runs.

An atomic process takes its current state and an event as input, and then (non-deterministically) outputs a new state and a set of events.

Definition 7 (Generic Atomic Processes and Systems). A *(generic) atomic process* is a tuple

$$p = (I^p, Z^p, R^p, s_0^p)$$

where $I^p \subseteq \mathsf{IPs}$, $Z^p \in \mathcal{T}_\mathcal{N}$ is a set of states, $R^p \subseteq (\mathcal{E} \times Z^p) \times (2^{\mathcal{E}^\nu\langle\rangle} \times \mathcal{T}_\mathcal{N}(V_{\text{process}}))$ (input event and old state map to sequence of output events and new state), and $s_0^p \in Z^p$ is the initial state of p. For any new state s and any sequence of nonces $(\eta_1, \eta_2, \ldots)$ we demand that $s[\eta_1/\nu_1, \eta_2/\nu_2, \ldots] \in Z^p$. A *system* $\mathcal{P}$ is a (possibly infinite) set of atomic processes.

Definition 8 (Configurations). A *configuration of a system* $\mathcal{P}$ is a tuple (S, E, N) where the state of the system S maps every atomic process $p \in \mathcal{P}$ to its current state $S(p) \in Z^p$, the sequence of waiting events E is an infinite sequence[19] $(e_1, e_2, \ldots)$ of events waiting to be delivered, and N is an infinite sequence of nonces $(n_1, n_2, \ldots)$.

Definition 9 (Concatenating sequences). For a term $a = \langle a_1, \ldots, a_i \rangle$ and a sequence $b = (b_1, b_2, \ldots)$, we define the *concatenation* as $a \cdot b := (a_1, \ldots, a_i, b_1, b_2, \ldots)$.

Definition 10 (Subtracting from Sequences). For a sequence X and a set or sequence Y we define $X \setminus Y$ to be the sequence X where for each element in Y, a non-deterministically chosen occurence of that element in X is removed.

Definition 11 (Processing Steps). A *processing step of the system* $\mathcal{P}$ is of the form

$$(S, E, N) \xrightarrow[p \to E_{\text{out}}]{e_{\text{in}} \to p} (S', E', N')$$

where

1. (S, E, N) and (S', E', N') are configurations of $\mathcal{P}$,
2. $e_{\text{in}} = \langle a, f, m \rangle \in E$ is an event,
3. $p \in \mathcal{P}$ is a process,

[19] Here: Not in the sense of terms as defined earlier.

4. E_{out} is a sequence (term) of events

such that there exists

1. a sequence (term) $E^{\nu}_{\text{out}} \subseteq 2^{\mathcal{E}^{\nu}\langle\rangle}$ of protoevents,
2. a term $s^{\nu} \in \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$,
3. a sequence $(v_1, v_2, \ldots, v_i)$ of all placeholders appearing in E^{ν}_{out} (ordered lexicographically),
4. a sequence $N^{\nu} = (\eta_1, \eta_2, \ldots, \eta_i)$ of the first i elements in N

with

1. $((e_{\text{in}}, S(p)), (E^{\nu}_{\text{out}}, s^{\nu})) \in R^p$ and $a \in I^p$,
2. $E_{\text{out}} = E^{\nu}_{\text{out}}[m_1/v_1, \ldots, m_i/v_i]$
3. $S'(p) = s^{\nu}[m_1/v_1, \ldots, m_i/v_i]$ and $S'(p') = S(p')$ for all $p' \neq p$
4. $E' = E_{\text{out}} \cdot (E \setminus \{e_{\text{in}}\})$
5. $N' = N \setminus N^{\nu}$

We may omit the superscript and/or subscript of the arrow.

Intuitively, for a processing step, we select one of the processes in $\mathcal{P}$, and call it with one of the events in the list of waiting events E. In its output (new state and output events), we replace any occurences of placeholders ν_x by "fresh" nonces from N (which we then remove from N). The output events are then prepended to the list of waiting events, and the state of the process is reflected in the new configuration.

Definition 12 (Runs). Let $\mathcal{P}$ be a system, E^0 be sequence of events, and N^0 be a sequence of nonces. A *run ρ of a system $\mathcal{P}$ initiated by E^0 with nonces N^0* is a finite sequence of configurations $((S^0, E^0, N^0), \ldots, (S^n, E^n, N^n))$ or an infinite sequence of configurations $((S^0, E^0, N^0), \ldots)$ such that $S^0(p) = s_0^p$ for all $p \in \mathcal{P}$ and $(S^i, E^i, N^i) \to (S^{i+1}, E^{i+1}, N^{i+1})$ for all $0 \leq i < n$ (finite run) or for all $i \geq 0$ (infinite run).

We denote the state $S^n(p)$ of a process p at the end of a run ρ by $\rho(p)$.

Usually, we will initiate runs with a set E^0 containing infinite trigger events of the form $\langle a, a, \texttt{TRIGGER}\rangle$ for each $a \in \mathsf{IPs}$, interleaved by address.

Atomic Dolev-Yao Processes. We next define atomic Dolev-Yao processes, for which we require that the messages and states that they output can be computed (more formally, derived) from the current input event and state. For this purpose, we first define what it means to derive a message from given messages.

Definition 13 (Deriving Terms). Let M be a set of ground terms. We say that *a term m can be derived from M with placeholders V* if there exist $n \geq 0$, $m_1, \ldots, m_n \in M$, and $\tau \in \mathcal{T}_{\emptyset}(\{x_1, \ldots, x_n\} \cup V)$ such that $m \equiv \tau[m_1/x_1, \ldots, m_n/x_n]$. We denote by $d_V(M)$ the set of all messages that can be derived from M with variables V.

For example, $a \in d_{\{\}}(\{\mathsf{enc_a}(\langle a, b, c\rangle, \mathsf{pub}(k)), k\})$.

Definition 14 (Atomic Dolev-Yao Process). An *atomic Dolev-Yao process (or simply, a DY process)* is a tuple $p = (I^p, Z^p, R^p, s_0^p)$ such that (I^p, Z^p, R^p, s_0^p) is an atomic process and (1) $Z^p \subseteq \mathcal{T}_{\mathcal{N}}$ (and hence, $s_0^p \in \mathcal{T}_{\mathcal{N}}$), and (2) for all events $e \in \mathcal{E}$, sequences of protoevents E, $s \in \mathcal{T}_{\mathcal{N}}$, $s' \in \mathcal{T}_{\mathcal{N}}(V_{\text{process}})$, with $((e, s), (E, s')) \in R^p$ it holds true that $E, s' \in d_{V_{\text{process}}}(\{e, s\})$.

Definition 15 (Atomic Attacker Process). An *(atomic) attacker process for a set of sender addresses* $A \subseteq \mathsf{IPs}$ is an atomic DY process $p = (I, Z, R, s_0)$ such that for all events e, and $s \in \mathcal{T}_{\mathcal{N}}$ we have that $((e,s),(E,s')) \in R$ iff $s' = \langle e, E, s\rangle$ and $E = \langle\langle a_1, f_1, m_1\rangle, \ldots, \langle a_n, f_n, m_n\rangle\rangle$ with $n \in \mathbb{N}$, $a_1, \ldots, a_n \in \mathsf{IPs}$, $f_0, \ldots, f_n \in A$, $m_1, \ldots, m_n \in d_{V_{\text{process}}}(\{e,s\})$.

B.2. Scripting Processes

We define scripting processes, which model client-side scripting technologies, such as JavaScript. Scripting processes are defined similarly to DY processes.

Definition 16 (Placeholders for Scripting Processes). By $V_{\text{script}} = \{\lambda_1, \ldots\}$ we denote an infinite set of variables used in scripting processes.

Definition 17 (Scripting Processes). A *scripting process* (or simply, a *script*) is a relation $R \subseteq \mathcal{T}_{\mathcal{N}} \times \mathcal{T}_{\mathcal{N}}(V_{\text{script}})$ such that for all $s \in \mathcal{T}_{\mathcal{N}}$, $s' \in \mathcal{T}_{\mathcal{N}}(V_{\text{script}})$ with $(s,s') \in R$ it follows that $s' \in d_{V_{\text{script}}}(s)$.

A script is called by the browser which provides it with state information (such as the script's last state and limited information about the browser's state) s. The script then outputs a term s', which represents the new internal state and some command which is interpreted by the browser. The term s' may contain variables $\lambda_1, \ldots$ which the browser will replace by (otherwise unused) placeholders $\nu_1, \ldots$ which will be replaced by nonces once the browser DY process finishes (effectively providing the script with a way to get "fresh" nonces).

Similarly to an attacker process, we define the *attacker script* R^{att}:

Definition 18 (Attacker Script). The attacker script R^{att} outputs everything that is derivable from the input, i.e., $R^{\text{att}} = \{(s,s') \mid s \in \mathcal{T}_{\mathcal{N}}, s' \in d_{V_{\text{script}}}(s)\}$.

B.3. Web System

The web infrastructure and web applications are formalized by what is called a web system. A web system contains, among others, a (possibly infinite) set of DY processes, modeling web browsers, web servers, DNS servers, and attackers (which may corrupt other entities, such as browsers).

Definition 19. A *web system* $\mathcal{WS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ is a tuple with its components defined as follows:

The first component, $\mathcal{W}$, denotes a system (a set of DY processes) and is partitioned into the sets Hon, Web, and Net of honest, web attacker, and network attacker processes, respectively.

Every $p \in \mathsf{Web} \cup \mathsf{Net}$ is an attacker process for some set of sender addresses $A \subseteq \mathsf{IPs}$. For a web attacker $p \in \mathsf{Web}$, we require its set of addresses I^p to be disjoint from the set of addresses of all other web attackers and honest processes, i.e., $I^p \cap I^{p'} = \emptyset$ for all $p' \in \mathsf{Hon} \cup \mathsf{Web}$. Hence, a web attacker cannot listen to traffic intended for other processes. Also, we require that $A = I^p$, i.e., a web attacker can only use sender addresses it owns. Conversely, a network attacker may listen to all addresses (i.e., no restrictions on I^p) and may spoof all addresses (i.e., the set A may be IPs).

Every $p \in \mathsf{Hon}$ is a DY process which models either a *web server*, a *web browser*, or a *DNS server*, as further described in the following subsections. Just as for web attackers, we require that p does not spoof sender addresses and that its set of addresses I^p is disjoint from those of other honest processes and the web attackers.

The second component, $\mathcal{S}$, is a finite set of scripts such that $R^{\text{att}} \in \mathcal{S}$. The third component, script, is an injective mapping from $\mathcal{S}$ to $\mathbb{S}$, i.e., by script every $s \in \mathcal{S}$ is assigned its string representation $\mathsf{script}(s)$.

Finally, E^0 is an (infinite) sequence of events, containing an infinite number of events of the form $\langle a, a, \texttt{TRIGGER}\rangle$ for every $a \in \bigcup_{p \in \mathcal{W}} I^p$.

A *run* of $\mathcal{WS}$ is a run of $\mathcal{W}$ initiated by E^0.

C. Message and Data Formats

We now provide some more details about data and message formats that are needed for the formal treatment of the web model and the analysis of BrowserID presented in the rest of the appendix.

C.1. Notations

Definition 20 (Sequence Notations). For a sequence $t = \langle t_1, \ldots, t_n \rangle$ and a set s we use $t \subset^{\langle\rangle} s$ to say that $t_1, \ldots, t_n \in s$. We define $x \in^{\langle\rangle} t \iff \exists i : t_i = x$. We write $t +^{\langle\rangle} y$ to denote the sequence $\langle t_1, \ldots, t_n, y \rangle$. For a finite set M with $M = \{m_1, \ldots, m_n\}$ we use $\langle M \rangle$ to denote the term of the form $\langle m_1, \ldots, m_n \rangle$. (The order of the elements does not matter; one is chosen arbitrarily.)

Definition 21. A *dictionary over X and Y* is a term of the form

$$\langle \langle k_1, v_1 \rangle, \ldots, \langle k_n, v_n \rangle \rangle$$

where $k_1, \ldots, k_n \in X$, $v_1, \ldots, v_n \in Y$, and the keys $k_1, \ldots, k_n$ are unique, i.e., $\forall i \neq j : k_i \neq k_j$. We call every term $\langle k_i, v_i \rangle$, $i \in \{1, \ldots, n\}$, an *element* of the dictionary with key k_i and value v_i. We often write $[k_1 : v_1, \ldots, k_i : v_i, \ldots, k_n : v_n]$ instead of $\langle \langle k_1, v_1 \rangle, \ldots, \langle k_n, v_n \rangle \rangle$. We denote the set of all dictionaries over X and Y by $[X \times Y]$.

We note that the empty dictionary is equivalent to the empty sequence, i.e., $[] = \langle\rangle$. Figure 11 shows the short notation for dictionary operations that will be used when describing the browser atomic process. For a dictionary $z = [k_1 : v_1, k_2 : v_2, \ldots, k_n : v_n]$ we write $k \in z$ to say that there exists i such that $k = k_i$. We write $z[k_j] := v_j$ to extract elements. If $k \notin z$, we set $z[k] := \langle\rangle$.

$$[k_1 : v_1, \ldots, k_i : v_i, \ldots, k_n : v_n]\,[k_i] = v_i \tag{6}$$

$$[k_1 : v_1, \ldots, k_{i-1} : v_{i-1}, k_i : v_i, k_{i+1} : v_{i+1} \ldots, k_n : v_n] - k_i = [k_1 : v_1, \ldots, k_{i-1} : v_{i-1}, k_{i+1} : v_{i+1} \ldots, k_n : v_n] \tag{7}$$

Figure 11. Dictionary operators with $1 \leq i \leq n$.

Given a term $t = \langle t_1, \ldots, t_n \rangle$, we can refer to any subterm using a sequence of integers. The subterm is determined by repeated application of the projection π_i for the integers i in the sequence. We call such a sequence a *pointer*:

Definition 22. A *pointer* is a sequence of non-negative integers. We write $\tau.\overline{p}$ for the application of the pointer $\overline{p}$ to the term τ. This operator is applied from left to right. For pointers consisting of a single integer, we may omit the sequence braces for brevity.

Example 3. For the term $\tau = \langle a, b, \langle c, d, \langle e, f \rangle \rangle \rangle$ and the pointer $\overline{p} = \langle 3, 1 \rangle$, the subterm of τ at the position $\overline{p}$ is $c = \pi_1(\pi_3(\tau))$. Also, $\tau.3.\langle 3,1 \rangle = \tau.3.\overline{p} = \tau.3.3.1 = e$.

To improve readability, we try to avoid writing, e.g., $o.2$ or $\pi_2(o)$ in this document. Instead, we will use the names of the components of a sequence that is of a defined form as pointers that point to the corresponding subterms. E.g., if an *Origin* term is defined as $\langle \mathit{host}, \mathit{protocol} \rangle$ and o is an Origin term, then we can write $o.\texttt{protocol}$ instead of $\pi_2(o)$ or $o.2$. See also Example 4.

C.2. URLs

Definition 23. A *URL* is a term of the form

$$\langle \texttt{URL}, \mathit{protocol}, \mathit{host}, \mathit{path}, \mathit{parameters}, \mathit{fragment} \rangle$$

with $\mathit{protocol} \in \{\texttt{P}, \texttt{S}\}$ (for **p**lain (HTTP) and **s**ecure (HTTPS)), $\mathit{host} \in \mathsf{Doms}$, $\mathit{path} \in \mathbb{S}$, $\mathit{parameters} \in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$, and $\mathit{fragment} \in \mathcal{T}_{\mathcal{N}}$. The set of all valid URLs is URLs.

The *fragment* part of a URL can be omitted when writing the URL. Its value is then defined to be $\bot$.

Example 4. For the URL $u = \langle \texttt{URL}, a, b, c, d \rangle$, $u.\texttt{protocol} = a$. If, in the algorithm described later, we say $u.\texttt{path} := e$ then $u = \langle \texttt{URL}, a, b, c, e \rangle$ afterwards.

C.3. Origins

Definition 24. An *origin* is a term of the form $\langle \mathit{host}, \mathit{protocol} \rangle$ with $\mathit{host} \in \mathsf{Doms}$ and $\mathit{protocol} \in \{\texttt{P}, \texttt{S}\}$. We write $\mathsf{Origins}$ for the set of all origins.

Example 5. For example, $\langle \texttt{FOO}, \texttt{S} \rangle$ is the HTTPS origin for the domain `FOO`, while $\langle \texttt{BAR}, \texttt{P} \rangle$ is the HTTP origin for the domain `BAR`.

C.4. Cookies

Definition 25. A *cookie* is a term of the form $\langle \mathit{name}, \mathit{content} \rangle$ where $\mathit{name} \in \mathcal{T}_{\mathcal{N}}$, and *content* is a term of the form $\langle \mathit{value}, \mathit{secure}, \mathit{session}, \mathit{httpOnly} \rangle$ where $\mathit{value} \in \mathcal{T}_{\mathcal{N}}$, *secure*, *session*, *httpOnly* $\in \{\top, \bot\}$. We write $\mathsf{Cookies}$ for the set of all cookies and $\mathsf{Cookies}^{\nu}$ for the set of all cookies where names and values are defined over $\mathcal{T}_{\mathcal{N}}(V)$.

If the *secure* attribute of a cookie is set, the browser will not transfer this cookie over unencrypted HTTP connections. If the *session* flag is set, this cookie will be deleted as soon as the browser is closed. The *httpOnly* attribute controls whether JavaScript has access to this cookie.

Note that cookies of the form described here are only contained in HTTP(S) requests. In responses, only the components *name* and *value* are transferred as a pairing of the form $\langle \mathit{name}, \mathit{value} \rangle$.

C.5. HTTP Messages

Definition 26. An *HTTP request* is a term of the form shown in (8). An *HTTP response* is a term of the form shown in (9).

$$\langle \texttt{HTTPReq}, \mathit{nonce}, \mathit{method}, \mathit{host}, \mathit{path}, \mathit{parameters}, \mathit{headers}, \mathit{body} \rangle \tag{8}$$

$$\langle \texttt{HTTPResp}, \mathit{nonce}, \mathit{status}, \mathit{headers}, \mathit{body} \rangle \tag{9}$$

The components are defined as follows:

- $\mathit{nonce} \in \mathcal{N}$ serves to map each response to the corresponding request
- $\mathit{method} \in \mathsf{Methods}$ is one of the HTTP methods.
- $\mathit{host} \in \mathsf{Doms}$ is the host name in the HOST header of HTTP/1.1.
- $\mathit{path} \in \mathbb{S}$ is a string indicating the requested resource at the server side

- *status* $\in \mathbb{S}$ is the HTTP status code (i.e., a number between 100 and 505, as defined by the HTTP standard)
- *parameters* $\in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$ contains URL parameters
- *headers* $\in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$, containing request/response headers. The dictionary elements are terms of one of the following forms:
  - $\langle \texttt{Origin}, o \rangle$ where o is an origin,
  - $\langle \texttt{Set-Cookie}, c \rangle$ where c is a sequence of cookies,
  - $\langle \texttt{Cookie}, c \rangle$ where $c \in, [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$ (note that in this header, only names and values of cookies are transferred),
  - $\langle \texttt{Location}, l \rangle$ where $l \in \mathsf{URLs}$,
  - $\langle \texttt{Referer}, r \rangle$ where $r \in \mathsf{URLs}$,
  - $\langle \texttt{Strict-Transport-Security}, \top \rangle$,
  - $\langle \texttt{Authorization}, \langle u, p \rangle \rangle$ where $u,\ p \in \mathbb{S}$,
- *body* $\in \mathcal{T}_{\mathcal{N}}$ in requests and responses.

We write $\mathsf{HTTPRequests}$/$\mathsf{HTTPResponses}$ for the set of all HTTP requests or responses, respectively.

Example 6 (HTTP Request and Response).

$$r := \langle \texttt{HTTPReq}, n_1, \texttt{POST}, \texttt{example.com}, \texttt{/show}, \langle\langle \texttt{index}, \texttt{1} \rangle\rangle, \\ [\texttt{Origin} : \langle \texttt{example.com}, \texttt{S} \rangle], \langle \texttt{foo}, \texttt{bar} \rangle \rangle \tag{10}$$

$$s := \langle \texttt{HTTPResp}, n_1, 200, \langle\langle \texttt{Set-Cookie}, \langle\langle \texttt{SID}, \langle n_2, \bot, \bot, \top \rangle\rangle\rangle\rangle, \langle \texttt{somescript}, x \rangle \rangle \tag{11}$$

An HTTP `GET` request for the URL `http://example.com/show?index=1` is shown in (10), with an Origin header and a body that contains $\langle \texttt{foo}, \texttt{bar} \rangle$. A possible response is shown in (11), which contains an httpOnly cookie with name `SID` and value n_2 as well as the string representation `somescript` of the scripting process $\mathsf{script}^{-1}(\texttt{somescript})$ (which should be an element of $\mathcal{S}$) and its initial state x.

Encrypted HTTP Messages.. For HTTPS, requests are encrypted using the public key of the server. Such a request contains an (ephemeral) symmetric key chosen by the client that issued the request. The server is supported to encrypt the response using the symmetric key.

Definition 27. An *encrypted HTTP request* is of the form $\mathsf{enc}_{\mathsf{a}}(\langle m, k' \rangle, k)$, where $k,\ k' \in \mathcal{N}$ and $m \in \mathsf{HTTPRequests}$. The corresponding *encrypted HTTP response* would be of the form $\mathsf{enc}_{\mathsf{s}}(m', k')$, where $m' \in \mathsf{HTTPResponses}$. We call the sets of all encrypted HTTP requests and responses $\mathsf{HTTPSRequests}$ or $\mathsf{HTTPSResponses}$, respectively.

Example 7.

$$\mathsf{enc}_{\mathsf{a}}(\langle r, k' \rangle, \mathsf{pub}(k_{\text{example.com}})) \tag{12}$$

$$\mathsf{enc}_{\mathsf{s}}(s, k') \tag{13}$$

The term (12) shows an encrypted request (with r as in (10)). It is encrypted using the public key $\mathsf{pub}(k_{\text{example.com}})$. The term (13) is a response (with s as in (11)). It is encrypted symmetrically using the (symmetric) key k' that was sent in the request (12).

C.6. DNS Messages

Definition 28. A *DNS request* is a term of the form $\langle \texttt{DNSResolve}, domain, n \rangle$ where $domain \in \mathsf{Doms}$, $n \in \mathcal{N}$. We call the set of all DNS requests DNSRequests.

Definition 29. A *DNS response* is a term of the form $\langle \texttt{DNSResolved}, domain, result, n \rangle$ with $domain \in \mathsf{Doms}$, $result \in \mathsf{IPs}$, $n \in \mathcal{N}$. We call the set of all DNS responses DNSResponses.

DNS servers are supposed to include the nonce they received in a DNS request in the DNS response that they send back so that the party which issued the request can match it with the request.

C.7. DNS Servers

Here, we consider a flat DNS model in which DNS queries are answered directly by one DNS server and always with the same address for a domain. A full (hierarchical) DNS system with recursive DNS resolution, DNS caches, etc. could also be modeled to cover certain attacks on the DNS system itself.

Definition 30. A *DNS server d* (in a flat DNS model) is modeled in a straightforward way as an atomic DY process $(I^d, \{s_0^d\}, R^d, s_0^d)$. It has a finite set of addresses I^d and its initial (and only) state s_0^d encodes a mapping from domain names to addresses of the form

$$s_0^d = \langle \langle \texttt{domain}_1, a_1 \rangle, \langle \texttt{domain}_2, a_2 \rangle, \ldots \rangle \ .$$

DNS queries are answered according to this table (otherwise ignored).

D. Detailed Description of the Browser Model

Following the informal description of the browser model in Section 4.3, we now present a formal model. We start by introducing some notation and terminology.

D.1. Notation and Terminology (Web Browser State)

Before we can define the state of a web browser, we first have to define windows and documents.

Definition 31. A *window* is a term of the form $w = \langle nonce, documents, opener \rangle$ with $nonce \in \mathcal{N}$, $documents \subset^{\langle\rangle}$ Documents (defined below), $opener \in \mathcal{N} \cup \{\bot\}$ where $d.\texttt{active} = \top$ for exactly one $d \in^{\langle\rangle} documents$ if *documents* is not empty (we then call d the *active document of* w). We write Windows for the set of all windows. We write $w.\texttt{activedocument}$ to denote the active document inside window w if it exists and $\langle\rangle$ else.

We will refer to the window nonce as *(window) reference*.

The documents contained in a window term to the left of the active document are the previously viewed documents (available to the user via the "back" button) and the documents in the window term to the right of the currently active document are documents available via the "forward" button.

A window a may have opened a top-level window b (i.e., a window term which is not a subterm of a document term). In this case, the *opener* part of the term b is the nonce of a, i.e., $b.\texttt{opener} = a.\texttt{nonce}$.

Definition 32. A *document d* is a term of the form

$$\langle nonce, location, referrer, script, scriptstate, scriptinputs, subwindows, active \rangle$$

where *nonce* $\in \mathcal{N}$, *location* $\in$ URLs, *referrer* $\in$ URLs $\cup \{\bot\}$, *script* $\in \mathcal{T}_{\mathcal{N}}$, *scriptstate* $\in \mathcal{T}_{\mathcal{N}}$, *scriptinputs* $\in \mathcal{T}_{\mathcal{N}}$, *subwindows* $\subset^{\langle\rangle}$ Windows, *active* $\in \{\top, \bot\}$. A *limited document* is a term of the form $\langle$*nonce*, *subwindows*$\rangle$ with *nonce*, *subwindows* as above. A window $w \in^{\langle\rangle}$ *subwindows* is called a *subwindow* (of d). We write Documents for the set of all documents. For a document term d we write d.`origin` to denote the origin of the document, i.e., the term $\langle d$.`location.host`, d.`location.protocol`$\rangle \in$ Origins.

We will refer to the document nonce as *(document) reference*.

We can now define the set of states of web browsers. Note that we use the dictionary notation that we introduced in Definition 21.

Definition 33. The *set of states Z^p of a web browser atomic process p* consists of the terms of the form

$$\langle \mathit{windows}, \mathit{ids}, \mathit{secrets}, \mathit{cookies}, \mathit{localStorage}, \mathit{sessionStorage}, \mathit{keyMapping}, \\ \mathit{sts}, \mathit{DNSaddress}, \mathit{pendingDNS}, \mathit{pendingRequests}, \mathit{isCorrupted} \rangle$$

where

- *windows* $\subset^{\langle\rangle}$ Windows,
- *ids* $\subset^{\langle\rangle} \mathcal{T}_{\mathcal{N}}$,
- *secrets* $\in \left[\text{Origins} \times \mathcal{T}_{\mathcal{N}}\right]$,
- *cookies* is a dictionary over Doms and sequences of Cookies,
- *localStorage* $\in \left[\text{Origins} \times \mathcal{T}_{\mathcal{N}}\right]$,
- *sessionStorage* $\in \left[OR \times \mathcal{T}_{\mathcal{N}}\right]$ for $OR := \{\langle o, r\rangle \mid o \in \text{Origins}, r \in \mathcal{N}\}$,
- *keyMapping* $\in \left[\text{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$,
- *sts* $\subset^{\langle\rangle}$ Doms,
- *DNSaddress* $\in$ IPs,
- *pendingDNS* $\in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$,
- *pendingRequests* $\in \mathcal{T}_{\mathcal{N}}$,
- and *isCorrupted* $\in \{\bot, \texttt{FULLCORRUPT}, \texttt{CLOSECORRUPT}\}$.

Definition 34. For two window terms w and w' we write $w \xrightarrow{\text{childof}} w'$ if

$$w \in^{\langle\rangle} w'.\texttt{activedocument.subwindows}\,.$$

We write $\xrightarrow{\text{childof}^+}$ for the transitive closure.

In the following description of the web browser relation R^p we use the helper functions Subwindows, Docs, Clean, CookieMerge and AddCookie.

Given a browser state s, Subwindows(s) denotes the set of all pointers[20] to windows in the window list s.`windows`, their active documents, and (recursively) the subwindows of these documents. We exclude subwindows of inactive documents and their subwindows. With Docs(s) we denote the set of pointers to all active documents in the set of windows referenced by Subwindows(s).

[20] Recall the definition of a pointer in Definition 22.

Definition 35. For a browser state s we denote by $\mathsf{Subwindows}(s)$ the minimal set of pointers that satisfies the following conditions: (1) For all windows $w \in^{\langle\rangle} s.\texttt{windows}$ there is a $\overline{p} \in \mathsf{Subwindows}(s)$ such that $s.\overline{p} = w$. (2) For all $\overline{p} \in \mathsf{Subwindows}(s)$, the active document d of the window $s.\overline{p}$ and every subwindow w of d there is a pointer $\overline{p'} \in \mathsf{Subwindows}(s)$ such that $s.\overline{p'} = w$.

Given a browser state s, the set $\mathsf{Docs}(s)$ of pointers to active documents is the minimal set such that for every $\overline{p} \in \mathsf{Subwindows}(s)$, there is a pointer $\overline{p'} \in \mathsf{Docs}(s)$ with $s.\overline{p'} = s.\overline{p}.\texttt{activedocument}$.

By $\mathsf{Subwindows}^+(s)$ and $\mathsf{Docs}^+(s)$ we denote the respective sets that also include the inactive documents and their subwindows.

The function Clean will be used to determine which information about windows and documents the script running in the document d has access to.

Definition 36. Let s be a browser state and d a document. By $\mathsf{Clean}(s,d)$ we denote the term that equals $s.\texttt{windows}$ but with all inactive documents removed (including their subwindows etc.) and all subterms that represent non-same-origin documents w.r.t. d replaced by a limited document d' with the same nonce and the same subwindow list. Note that non-same-origin documents on all levels are replaced by their corresponding limited document.

The function CookieMerge merges two sequences of cookies together: When used in the browser, *oldcookies* is the sequence of existing cookies for some origin, *newcookies* is a sequence of new cookies that was output by some script. The sequences are merged into a set of cookies using an algorithm that is based on the *Storage Mechanism* algorithm described in RFC6265.

Definition 37. For a sequence of cookies (with pairwise different names) *oldcookies* and a sequence of cookies *newcookies*, the set $\mathsf{CookieMerge}(\mathit{oldcookies}, \mathit{newcookies})$ is defined by the following algorithm: From *newcookies* remove all cookies c that have $c.\texttt{content.httpOnly} \equiv \top$. For any c, $c' \in^{\langle\rangle}$ *newcookies*, $c.\texttt{name} \equiv c'.\texttt{name}$, remove the cookie that appears left of the other in *newcookies*. Let m be the set of cookies that have a name that either appears in *oldcookies* or in *newcookies*, but not in both. For all pairs of cookies $(c_{\text{old}}, c_{\text{new}})$ with $c_{\text{old}} \in^{\langle\rangle}$ *oldcookies*, $c_{\text{new}} \in^{\langle\rangle}$ *newcookies*, $c_{\text{old}}.\texttt{name} \equiv c_{\text{new}}.\texttt{name}$, add c_{new} to m if $c_{\text{old}}.\texttt{content.httpOnly} \equiv \bot$ and add c_{old} to m otherwise. The result of $\mathsf{CookieMerge}(\mathit{oldcookies}, \mathit{newcookies})$ is m.

The function AddCookie adds a cookie c received in an HTTP response to the sequence of cookies contained in the sequence *oldcookies*. It is again based on the algorithm described in RFC6265 but simplified for the use in the browser model.

Definition 38. For a sequence of cookies (with pairwise different names) *oldcookies* and a cookie c, the sequence $\mathsf{AddCookie}(\mathit{oldcookies}, c)$ is defined by the following algorithm: Let $m := \mathit{oldcookies}$. Remove any c' from m that has $c.\texttt{name} \equiv c'.\texttt{name}$. Append c to m and return m.

The function NavigableWindows returns a set of windows that a document is allowed to navigate. We closely follow [14], Section 5.1.4 for this definition.

Definition 39. The set $\mathsf{NavigableWindows}(\overline{w}, s')$ is the set $\overline{W} \subseteq \mathsf{Subwindows}(s')$ of pointers to windows that the active document in $\overline{w}$ is allowed to navigate. The set $\overline{W}$ is defined to be the minimal set such that for every $\overline{w'} \in \mathsf{Subwindows}(s')$ the following is true:

- If $s'.\overline{w}'.\texttt{activedocument.origin} \equiv s'.\overline{w}.\texttt{activedocument.origin}$ (i.e., the active documents in $\overline{w}$ and $\overline{w'}$ are same-origin), then $\overline{w'} \in \overline{W}$, and
- If $s'.\overline{w} \xrightarrow{\text{childof}^*} s'.\overline{w'} \wedge \nexists \overline{w}'' \in \mathsf{Subwindows}(s')$ with $s'.\overline{w}' \xrightarrow{\text{childof}^*} s'.\overline{w}''$ ($\overline{w'}$ is a top-level window and $\overline{w}$ is an ancestor window of $\overline{w'}$), then $\overline{w'} \in \overline{W}$, and

- If $\exists \overline{p} \in \mathsf{Subwindows}(s')$ such that $s'.\overline{w}' \xrightarrow{\text{childof}^+} s'.\overline{p}$ $\wedge\, s'.\overline{p}.\texttt{activedocument.origin} = s'.\overline{w}.\texttt{activedocument.origin}$ ($\overline{w'}$ is not a top-level window but there is an ancestor window $\overline{p}$ of $\overline{w'}$ with an active document that has the same origin as the active document in $\overline{w}$), then $\overline{w'} \in \overline{W}$, and
- If $\exists \overline{p} \in \mathsf{Subwindows}(s')$ such that $s'.\overline{w'}.\texttt{opener} = s'.\overline{p}.\texttt{nonce} \wedge \overline{p} \in \overline{W}$ ($\overline{w'}$ is a top-level window—it has an opener—and $\overline{w}$ is allowed to navigate the opener window of $\overline{w'}$, $\overline{p}$), then $\overline{w'} \in \overline{W}$.

D.2. Description of the Web Browser Atomic Process

We will now describe the relation R^p of a standard HTTP browser p. We define $((\langle a, f, m\rangle, s), (M, s'))$ to belong to R^p iff the non-deterministic algorithm presented below, when given $(\langle a, f, m\rangle, s)$ as input, terminates with **stop** M, s', i.e., with output M and s'. Recall that $\langle a, f, m\rangle$ is an (input) event and s is a (browser) state, M is a sequence of (output) protoevents, and s' is a new (browser) state (potentially with placeholders for nonces).

Notations.. The notation **let** $n \leftarrow N$ is used to describe that n is chosen non-deterministically from the set N. We write **for each** $s \in M$ **do** to denote that the following commands (until **end for**) are repeated for every element in M, where the variable s is the current element. The order in which the elements are processed is chosen non-deterministically. We will write, for example,

let x, y **such that** $\langle \texttt{Constant}, x, y\rangle \equiv t$ **if possible; otherwise** doSomethingElse

for some variables x, y, a string `Constant`, and some term t to express that $x := \pi_2(t)$, and $y := \pi_3(t)$ if $\texttt{Constant} \equiv \pi_1(t)$ and if $|\langle \texttt{Constant}, x, y\rangle| = |t|$, and that otherwise x and y are not set and doSomethingElse is executed.

Placeholders.. In several places throughout the algorithms presented next we use placeholders to generate "fresh" nonces as described in our communication model (see Definition 1). Figure 12 shows a list of all placeholders used.

Placeholder	Usage
ν_1	Algorithm 7, new window nonces
ν_2	Algorithm 7, new HTTP request nonce
ν_3	Algorithm 7, lookup key for pending HTTP requests entry
ν_4	Algorithm 5, new HTTP request nonce (multiple lines)
ν_5	Algorithm 5, new subwindow nonce
ν_6	Algorithm 6, new HTTP request nonce
ν_7	Algorithm 6, new document nonce
ν_8	Algorithm 4, lookup key for pending DNS entry
ν_9	Algorithm 1, new window nonce
$\nu_{10}, \ldots$	Algorithm 5, replacement for placeholders in scripting process output

Figure 12. List of placeholders used in browser algorithms.

Before we describe the main browser algorithm, we first define some functions.

Functions. In the description of the following functions we use a, f, m, and s as read-only global input variables. All other variables are local variables or arguments.

The following function, GETNAVIGABLEWINDOW, is called by the browser to determine the window that is *actually* navigated when a script in the window $s'.\overline{w}$ provides a window reference for navigation (e.g., for opening a link). When it is given a window reference (nonce) *window*, this function returns a pointer to a selected window term in s':

- If *window* is the string _BLANK, a new window is created and a pointer to that window is returned.
- If *window* is a nonce (reference) and there is a window term with a reference of that value in the windows in s', a pointer $\overline{w'}$ to that window term is returned, as long as the window is navigable by the current window's document (as defined by NavigableWindows above).

In all other cases, $\overline{w}$ is returned instead (the script navigates its own window).

Algorithm 1 Determine window for navigation.

```
1:  function GETNAVIGABLEWINDOW(w̄, window, noreferrer, s′)
2:      if window ≡ _BLANK then                    ▷ Open a new window when _BLANK is used
3:          if noreferrer ≡ ⊥ then
4:              let w′ := ⟨ν9, ⟨⟩, s′.w̄.nonce⟩
5:          else
6:              let w′ := ⟨ν9, ⟨⟩, ⊥⟩
7:          end if
8:          let s′.windows := s′.windows +^⟨⟩ w′
              ↪ and let w̄′ be a pointer to this new element in s′
9:          return w̄′
10:     end if
11:     let w̄′ ← NavigableWindows(w̄, s′) such that s′.w̄′.nonce ≡ window
          ↪ if possible; otherwise return w̄
12:     return w̄′
13: end function
```

The following function takes a window reference as input and returns a pointer to a window as above, but it checks only that the active documents in both windows are same-origin. It creates no new windows.

Algorithm 2 Determine same-origin window.

```
1:  function GETWINDOW(w̄, window, s′)
2:      let w̄′ ← Subwindows(s′) such that s′.w̄′.nonce ≡ window
          ↪ if possible; otherwise return w̄
3:      if s′.w̄′.activedocument.origin ≡ s′.w̄.activedocument.origin then
4:          return w̄′
5:      end if
6:      return w̄
7:  end function
```

The next function is used to stop any pending requests for a specific window. From the pending requests and pending DNS requests it removes any requests with the given window reference *n*.

Algorithm 3 Cancel pending requests for given window.

```
1:  function CANCELNAV(n, s′)
2:      remove all ⟨n, req, key, f⟩ from s′.pendingRequests for any req, key, f
3:      remove all ⟨x, ⟨n, message, url⟩⟩ from s′.pendingDNS
          ↪ for any x, message, url
4:      return s′
5:  end function
```

The following function takes an HTTP request *message* as input, adds cookie and origin headers to the message, creates a DNS request for the hostname given in the request and stores the request in s'.pendingDNS until the DNS resolution finishes. For normal HTTP requests, *reference* is a window reference. For XHRs, *reference* is a value of the form $\langle document, nonce \rangle$ where *document* is a document reference and *nonce* is some nonce that was chosen by the script that initiated the request. *url* contains

the full URL of the request (this is mainly used to retrieve the protocol that should be used for this message, and to store the fragment identifier for use after the document was loaded). *origin* is the origin header value that is to be added to the HTTP request.

Algorithm 4 Prepare headers, do DNS resolution, save message.

function SEND(*reference*, *message*, *url*, *origin*, *referrer*, s')
  if *message*.host $\in^{\langle\rangle}$ s'.sts **then**
    let *url*.protocol := S
  end if
  let *cookies* := $\langle\{\langle c.\texttt{name}, c.\texttt{content.value}\rangle | c \in^{\langle\rangle} s'.\texttt{cookies}[message.\texttt{host}]$
  ↪ $\wedge(c.\texttt{content.secure} \implies (url.\texttt{protocol} = \texttt{S}))\}\rangle$
  let *message*.headers[Cookie] := *cookies*
  if *origin* $\not\equiv \perp$ **then**
    let *message*.headers[Origin] := *origin*
  end if
  if *referrer* $\not\equiv \perp$ **then**
    let *message*.headers[Referer] := *referrer*
  end if
  let s'.pendingDNS[ν_8] := $\langle$*reference*, *message*, *url*$\rangle$
  stop $\langle\langle s'.\texttt{DNSaddress}, a, \langle\texttt{DNSResolve}, host, \nu_8\rangle\rangle\rangle$, s'
end function

The function RUNSCRIPT performs a script execution step of the script in the document $s'.\overline{d}$ (which is part of the window $s'.\overline{w}$). A new script and document state is chosen according to the relation defined by the script and the new script and document state is saved. Afterwards, the *command* that the script issued is interpreted.

Algorithm 5 Execute a script.

function RUNSCRIPT($\overline{w}$, $\overline{d}$, s')
  let *tree* := Clean(s', $s'.\overline{d}$)
  let *cookies* := $\langle\{\langle c.\texttt{name}, c.\texttt{content.value}\rangle | c \in^{\langle\rangle} s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right]$
  ↪ $\wedge c.\texttt{content.httpOnly} = \perp$
  ↪ $\wedge\left(c.\texttt{content.secure} \implies \left(s'.\overline{d}.\texttt{origin.protocol} \equiv \texttt{S}\right)\right)\}\rangle$
  let *tlw* ← s'.windows **such that** *tlw* is the top-level window containing $\overline{d}$
  let *sessionStorage* := $s'.\texttt{sessionStorage}\left[\langle s'.\overline{d}.\texttt{origin}, tlw.\texttt{nonce}\rangle\right]$
  let *localStorage* := $s'.\texttt{localStorage}\left[s'.\overline{d}.\texttt{origin}\right]$
  let *secrets* := $s'.\texttt{secrets}\left[s'.\overline{d}.\texttt{origin}\right]$
  let $R \leftarrow \text{script}^{-1}(s'.\overline{d}.\texttt{script})$
  let *in* := $\langle$*tree*, $s'.\overline{d}$.nonce, $s'.\overline{d}$.scriptstate, $s'.\overline{d}$.scriptinputs, *cookies*,
  ↪ *localStorage*, *sessionStorage*, s'.ids, *secrets*$\rangle$
  let *state*$'$ ← $\mathcal{T}_{\mathcal{N}}(V)$,
  ↪ *cookies*$'$ ← Cookies$^{\nu}$,
  ↪ *localStorage*$'$ ← $\mathcal{T}_{\mathcal{N}}(V)$,
  ↪ *sessionStorage*$'$ ← $\mathcal{T}_{\mathcal{N}}(V)$,
  ↪ *command* ← $\mathcal{T}_{\mathcal{N}}(V)$,
  ↪ *out*$^{\lambda}$:= $\langle$*state*$'$, *cookies*$'$, *localStorage*$'$, *sessionStorage*$'$, *command*$\rangle$
  ↪ **such that** $(in, out^{\lambda}) \in R$
  let *out* := $out^{\lambda}[\nu_{10}/\lambda_1, \nu_{11}/\lambda_2, \ldots]$
  let $s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right]$
  ↪ := $\langle$CookieMerge($s'.\texttt{cookies}\left[s'.\overline{d}.\texttt{origin.host}\right]$, *cookies*$'$)$\rangle$
  let $s'.\texttt{localStorage}\left[s'.\overline{d}.\texttt{origin}\right]$:= *localStorage*$'$
  let $s'.\texttt{sessionStorage}\left[\langle s'.\overline{d}.\texttt{origin}, tlw.\texttt{nonce}\rangle\right]$:= *sessionStorage*$'$

```
15:     let s'.d̄.scriptstate := state'
16:     switch command do
17:         case ⟨HREF, url, hrefwindow, noreferrer⟩
18:             let w̄' := GETNAVIGABLEWINDOW(w̄, hrefwindow, noreferrer, s')
19:             let req := ⟨HTTPReq, ν4, GET, url.host, url.path, ⟨⟩, url.parameters, ⟨⟩⟩
20:             if noreferrer ≡ ⊥ then
21:                 let referrer := s'.d̄.location
22:             else
23:                 let referrer := ⊥
24:             end if
25:             let s' := CANCELNAV(s'.w̄'.nonce, s')
26:             SEND(s'.w̄'.nonce, req, url, ⊥, referrer, s')
27:         case ⟨IFRAME, url, window⟩
28:             let w̄' := GETWINDOW(w̄, window, s')
29:             let req := ⟨HTTPReq, ν4, GET, url.host, url.path, ⟨⟩, url.parameters, ⟨⟩⟩
30:             let referrer := s'.w̄'.activedocument.location
31:             let w' := ⟨ν5, ⟨⟩, ⊥⟩
32:             let s'.w̄'.activedocument.subwindows
                ↪ := s'.w̄'.activedocument.subwindows +⟨⟩ w'
33:             SEND(ν5, req, url, ⊥, referrer, s')
34:         case ⟨FORM, url, method, data, hrefwindow⟩
35:             if method ∉ {GET, POST} then [21]
36:                 stop ⟨⟩, s'
37:             end if
38:             let w̄' := GETNAVIGABLEWINDOW(w̄, hrefwindow, ⊥, s')
39:             if method = GET then
40:                 let body := ⟨⟩
41:                 let parameters := data
42:                 let origin := ⊥
43:             else
44:                 let body := data
45:                 let parameters := url.parameters
46:                 let origin := s'.d̄.origin
47:             end if
48:             let req := ⟨HTTPReq, ν4, method, url.host, url.path, ⟨⟩, parameters, body⟩
49:             let referrer := s'.d̄.location
50:             let s' := CANCELNAV(s'.w̄'.nonce, s')
51:             SEND(s'.w̄'.nonce, req, url, origin, referrer, s')
52:         case ⟨SETSCRIPT, window, script⟩
53:             let w̄' := GETWINDOW(w̄, window, s')
54:             let s'.w̄'.activedocument.script := script
55:             stop ⟨⟩, s'
56:         case ⟨SETSCRIPTSTATE, window, scriptstate⟩
57:             let w̄' := GETWINDOW(w̄, window, s')
58:             let s'.w̄'.activedocument.scriptstate := scriptstate
59:             stop ⟨⟩, s'
60:         case ⟨XMLHTTPREQUEST, url, method, data, xhrreference⟩
61:             if method ∈ {CONNECT, TRACE, TRACK} ∧ xhrreference ∉ {𝒩, ⊥} then
62:                 stop ⟨⟩, s'
63:             end if
64:             if url.host ≢ s'.d̄.origin.host
```

[21] The working draft for HTML5 allowed for DELETE and PUT methods in HTML5 forms. However, these have since been removed. See http://www.w3.org/TR/2010/WD-html5-diff-20101019/#changes-2010-06-24.

```
            ↪ ∨ url ≢ s′.d̄.origin.protocol then
65:             stop ⟨⟩, s′
66:         end if
67:         if method ∈ {GET, HEAD} then
68:             let data := ⟨⟩
69:             let origin := ⊥
70:         else
71:             let origin := s′.d̄.origin
72:         end if
73:         let req := ⟨HTTPReq, ν4, method, url.host, url.path, , url.parameters, data⟩
74:         let referrer := s′.d̄.location
75:         SEND(⟨s′.d̄.nonce, xhrreference⟩, req, url, origin, referrer, s′)
76:     case ⟨BACK, window⟩ [22]
77:         let w̄′ := GETNAVIGABLEWINDOW(w̄, window, ⊥, s′)
78:         if ∃j̄ ∈ ℕ, j̄ > 1 such that s′.w̄′.documents.j̄.active ≡ ⊤ then
79:             let s′.w̄′.documents.j̄.active := ⊥
80:             let s′.w̄′.documents.(j̄ − 1).active := ⊤
81:             let s′ := CANCELNAV(s′.w̄′.nonce, s′)
82:         end if
83:         stop ⟨⟩, s′
84:     case ⟨FORWARD, window⟩
85:         let w̄′ := GETNAVIGABLEWINDOW(w̄, window, ⊥, s′)
86:         if ∃j̄ ∈ ℕ such that s′.w̄′.documents.j̄.active ≡ ⊤
            ↪ ∧ s′.w̄′.documents.(j̄ + 1) ∈ Documents then
87:             let s′.w̄′.documents.j̄.active := ⊥
88:             let s′.w̄′.documents.(j̄ + 1).active := ⊤
89:             let s′ := CANCELNAV(s′.w̄′.nonce, s′)
90:         end if
91:         stop ⟨⟩, s′
92:     case ⟨CLOSE, window⟩
93:         let w̄′ := GETNAVIGABLEWINDOW(w̄, window, ⊥, s′)
94:         remove s′.w̄′ from the sequence containing it
95:         stop ⟨⟩, s′
96:     case ⟨POSTMESSAGE, window, message, origin⟩
97:         let w̄′ ← Subwindows(s′) such that s′.w̄′.nonce ≡ window
98:         if ∃j̄ ∈ ℕ such that s′.w̄′.documents.j̄.active ≡ ⊤
            ↪ ∧(origin ≢ ⊥ ⟹ s′.w̄′.documents.j̄.origin ≡ origin) then
99:             let s′.w̄′.documents.j̄.scriptinputs
                ↪ := s′.w̄′.documents.j̄.scriptinputs
                ↪ +⟨⟩ ⟨POSTMESSAGE, s′.w̄.nonce, s′.d̄.origin, message⟩
100:        end if
101:        stop ⟨⟩, s′
102:    case else
103:        stop ⟨⟩, s′
104: end function
```

The function PROCESSRESPONSE is responsible for processing an HTTP response (*response*) that was received as the response to a request (*request*) that was sent earlier. In *reference*, either a window or a document reference is given (see explanation for Algorithm 4 above). *requestUrl* contains the URL used when retrieving the document.

[22] Note that navigating a window using the back/forward buttons does not trigger a reload of the affected documents. While real world browser may chose to refresh a document in this case, we assume that the complete state of a previously viewed document is restored. A reload can be triggered non-deterministically at any point (in the main algorithm).

The function first saves any cookies that were contained in the response to the browser state, then checks whether a redirection is requested (Location header). If that is not the case, the function creates a new document (for normal requests) or delivers the contents of the response to the respective receiver (for XHR responses).

Algorithm 6 Process an HTTP response.

1: **function** PROCESSRESPONSE(*response*, *reference*, *request*, *requestUrl*, s')
2: **if** `Set-Cookie` $\in$ *response*.`headers` **then**
3: **for each** $c \in^{\langle\rangle}$ *response*.`headers`[`Set-Cookie`], $c \in$ Cookies **do**
4: **let** s'.`cookies`[*request*.`host`] $\hookrightarrow$:= AddCookie(s'.`cookies`[*request*.`host`], c)
5: **end for**
6: **end if**
7: **if** `Strict-Transport-Security` $\in$ *response*.`headers` $\wedge$ *requestUrl*.`protocol` $\equiv$ `S` **then**
8: **let** s'.`sts` := s'.`sts` $+^{\langle\rangle}$ *request*.`host`
9: **end if**
10: **if** `Referer` $\in$ *request*.`headers` **then**
11: **let** *referrer* := *request*.`headers`[`Referer`]
12: **else**
13: **let** *referrer* := $\bot$
14: **end if**
15: **if** `Location` $\in$ *response*.`headers` $\wedge$ *response*.`status` $\in \{303, 307\}$ **then**
16: **let** *url* := *response*.`headers`[`Location`]
17: **if** *url*.`fragment` $\equiv \bot$ **then**
18: **let** *url*.`fragment` := *requestUrl*.`fragment`
19: **end if**
20: **let** *method*$'$:= *request*.`method`
21: **let** *body*$'$:= *request*.`body`
22: **if** `Origin` $\in$ *request*.`headers` **then**
23: **let** *origin* := $\langle$*request*.`headers`[`Origin`], $\langle$*request*.`host`, *url*.`protocol`$\rangle\rangle$
24: **else**
25: **let** *origin* := $\bot$
26: **end if**
27: **if** *response*.`status` $\equiv 303 \wedge$ *request*.`method` $\notin \{$`GET`, `HEAD`$\}$ **then**
28: **let** *method*$'$:= `GET`
29: **let** *body*$'$:= $\langle\rangle$
30: **end if**
31: **if** $\nexists \overline{w} \in$ Subwindows(s') **such that** $s'.\overline{w}$.`nonce` $\equiv$ *reference* **then**
▷ Do not redirect XHRs.
32: **stop** $\langle\rangle$, s
33: **end if**
34: **let** *req* := $\langle$`HTTPReq`, ν_6, *method*$'$, *url*.`host`, *url*.`path`, $\langle\rangle$, *url*.`parameters`, *body*$'\rangle$
35: SEND(*reference*, *req*, *url*, *origin*, *referrer*, s')
36: **end if**
37: **if** $\exists \overline{w} \in$ Subwindows(s') **such that** $s'.\overline{w}$.`nonce` $\equiv$ *reference* **then**
▷ normal response
38: **if** *response*.`body` $\not\sim \langle *, * \rangle$ **then**
39: **stop** $\{\}$, s'
40: **end if**
41: **let** *script* := π_1(*response*.`body`)
42: **let** *scriptstate* := π_2(*response*.`body`)
43: **let** $d := \langle \nu_7, requestUrl, referrer, script, scriptstate, \langle\rangle, \langle\rangle, \top \rangle$
44: **if** $s'.\overline{w}$.`documents` $\equiv \langle\rangle$ **then**

45: **let** $s'.\overline{w}.\texttt{documents} := \langle d \rangle$
46: **else**
47: **let** $\overline{i} \leftarrow \mathbb{N}$ **such that** $s'.\overline{w}.\texttt{documents}.\overline{i}.\texttt{active} \equiv \top$
48: **let** $s'.\overline{w}.\texttt{documents}.\overline{i}.\texttt{active} := \bot$
49: **remove** $s'.\overline{w}.\texttt{documents}.(\overline{i}+1)$ and all following documents
↪ from $s'.\overline{w}.\texttt{documents}$
50: **let** $s'.\overline{w}.\texttt{documents} := s'.\overline{w}.\texttt{documents} +^{\langle\rangle} d$
51: **end if**
52: **stop** $\{\}, s'$
53: **else if** $\exists \overline{w} \in \mathsf{Subwindows}(s'), \overline{d}$ **such that** $s'.\overline{d}.\texttt{nonce} \equiv \pi_1(\mathit{reference})$
↪ $\wedge\, s'.\overline{d} = s'.\overline{w}.\texttt{activedocument}$ **then** ▷ process XHR response
54: **let** $s'.\overline{d}.\texttt{scriptinputs} := s'.\overline{d}.\texttt{scriptinputs} +^{\langle\rangle}$
$\langle \texttt{XMLHTTPREQUEST}, \mathit{response}.\texttt{body}, \pi_2(\mathit{reference}) \rangle$
55: **end if**
56: **end function**

Main Algorithm.. This is the main algorithm of the browser relation. It receives the message m as input, as well as a, f and s as above.

Algorithm 7 Main Algorithm

Input: $\langle a, f, m \rangle, s$
1: **let** $s' := s$
2: **if** $s.\texttt{isCorrupted} \not\equiv \bot$ **then**
3: **let** $s'.\texttt{pendingRequests} := \langle m, s.\texttt{pendingRequests} \rangle$ ▷ Collect incoming messages
4: **let** $m' \leftarrow d_V(s')$
5: **let** $a' \leftarrow \mathsf{IPs}$
6: **stop** $\langle \langle a', a, m' \rangle \rangle, s'$
7: **end if**
8: **if** $m \equiv \texttt{TRIGGER}$ **then** ▷ A special trigger message.
9: **let** $\mathit{switch} \leftarrow \{1,2,3\}$
10: **if** $\mathit{switch} \equiv 1$ **then** ▷ Run some script.
11: **let** $\overline{w} \leftarrow \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w}.\texttt{documents} \neq \langle\rangle$
↪ **if possible; otherwise stop** $\langle\rangle, s'$
12: **let** $\overline{d} := \overline{w} +^{\langle\rangle} \texttt{activedocument}$
13: $\mathsf{RUNSCRIPT}(\overline{w}, \overline{d}, s')$
14: **else if** $\mathit{switch} \equiv 2$ **then** ▷ Create some new request.
15: **let** $w' := \langle \nu_1, \langle\rangle, \bot \rangle$
16: **let** $s'.\texttt{windows} := s'.\texttt{windows} +^{\langle\rangle} w'$
17: **let** $\mathit{protocol} \leftarrow \{\texttt{P}, \texttt{S}\}$
18: **let** $\mathit{host} \leftarrow \mathsf{Doms}$
19: **let** $\mathit{path} \leftarrow \mathbb{S}$
20: **let** $\mathit{fragment} \leftarrow \mathbb{S}$
21: **let** $\mathit{parameters} \leftarrow [\mathbb{S} \times \mathbb{S}]$
22: **let** $\mathit{url} := \langle \texttt{URL}, \mathit{protocol}, \mathit{host}, \mathit{path}, \mathit{parameters}, \mathit{fragment} \rangle$
23: **let** $\mathit{req} := \langle \texttt{HTTPReq}, \nu_2, \texttt{GET}, \mathit{host}, \mathit{path}, \langle\rangle, \mathit{parameters}, \langle\rangle \rangle$
24: $\mathsf{SEND}(\nu_1, \mathit{req}, \mathit{url}, \bot, s')$
25: **else if** $\mathit{switch} \equiv 3$ **then** ▷ Reload some document.
26: **let** $\overline{w} \leftarrow \mathsf{Subwindows}(s')$ **such that** $s'.\overline{w}.\texttt{documents} \neq \langle\rangle$
↪ **if possible; otherwise stop** $\langle\rangle, s'$
27: **let** $\mathit{url} := s'.\overline{w}.\texttt{activedocument.location}$
28: **let** $\mathit{req} := \langle \texttt{HTTPReq}, \nu_2, \texttt{GET}, \mathit{url}.\texttt{host}, \mathit{url}.\texttt{path}, \langle\rangle, \mathit{url}.\texttt{parameters}, \langle\rangle \rangle$
29: **let** $\mathit{referrer} := s'.\overline{w}.\texttt{activedocument.referrer}$
30: **let** $s' := \mathsf{CANCELNAV}(s'.\overline{w}.\texttt{nonce}, s')$
31: $\mathsf{SEND}(s'.\overline{w}.\texttt{nonce}, \mathit{req}, \mathit{url}, \bot, \mathit{referrer}, s')$

```
    end if
else if m ≡ FULLCORRUPT then                                   ▷ Request to corrupt browser
    let s′.isCorrupted := FULLCORRUPT
    stop ⟨⟩, s′
else if m ≡ CLOSECORRUPT then                                  ▷ Close the browser
    let s′.secrets := ⟨⟩
    let s′.windows := ⟨⟩
    let s′.pendingDNS := ⟨⟩
    let s′.pendingRequests := ⟨⟩
    let s′.sessionStorage := ⟨⟩
    let s′.cookies ⊂^⟨⟩ Cookies such that
      ↪ (c ∈^⟨⟩ s′.cookies) ⟺ (c ∈^⟨⟩ s.cookies ∧ c.content.session ≡ ⊥)
    let s′.isCorrupted := CLOSECORRUPT
    stop ⟨⟩, s′
else if ∃⟨reference, request, url, key, f⟩ ∈^⟨⟩ s′.pendingRequests
  ↪ such that π1(dec_s(m, key)) ≡ HTTPResp then                ▷ Encrypted HTTP response
    let m′ := dec_s(m, key)
    if m′.nonce ≢ request.nonce then
        stop ⟨⟩, s
    end if
    remove ⟨reference, request, url, key, f⟩ from s′.pendingRequests
    PROCESSRESPONSE(m′, reference, request, url, s′)
else if π1(m) ≡ HTTPResp ∧ ∃⟨reference, request, url, ⊥, f⟩ ∈^⟨⟩ s′.pendingRequests
  ↪ such that m′.nonce ≡ request.key then
    remove ⟨reference, request, url, ⊥, f⟩ from s′.pendingRequests
    PROCESSRESPONSE(m, reference, request, url, s′)
else if m ∈ DNSResponses then                                  ▷ Successful DNS response
    if m.nonce ∉ s.pendingDNS ∨ m.result ∉ IPs ∨ m.domain ≢ π2(s.pendingDNS).host then
        stop ⟨⟩, s
    end if
    let ⟨reference, message, url⟩ := s.pendingDNS[m.nonce]
    if url.protocol ≡ S then
        let s′.pendingRequests := s′.pendingRequests
          ↪ +^⟨⟩ ⟨reference, message, url, ν3, m.result⟩
        let message := enc_a(⟨message, ν3⟩, s′.keyMapping[message.host])
    else
        let s′.pendingRequests := s′.pendingRequests
          ↪ +^⟨⟩ ⟨reference, message, url, ⊥, m.result⟩
    end if
    let s′.pendingDNS := s′.pendingDNS − m.nonce
    stop ⟨⟨m.result, a, message⟩⟩, s′
end if
stop ⟨⟩, s
```

E. Formal Model of OAuth

We here present the full details of our formal model of OAuth which we use to analyze the authentication properties.

We model OAuth as a web system (in the sense of Appendix B.3). We call a web system $\mathcal{OWS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ an *OAuth web system* if it is of the form described in what follows.

E.1. Outline

The system $\mathcal{W} = \mathsf{Hon} \cup \mathsf{Net}$ consists of network attacker processes (in Net), a finite set B of web browsers, a finite set RP of web servers for the relying parties, a finite set IDP of web servers for the identity providers, with $\mathsf{Hon} := \mathsf{B} \cup \mathsf{RP} \cup \mathsf{IDP}$. More details on the processes in $\mathcal{W}$ are provided below. We do not model DNS servers, as they are subsumed by the network attacker. Figure 13 shows the set of scripts $\mathcal{S}$ and their respective string representations that are defined by the mapping script. The set E^0 contains only the trigger events as specified in Appendix B.3.

$s \in \mathcal{S}$	$\mathsf{script}(s)$
R^{att}	`att_script`
script_rp_index	`script_rp_index`
script_rp_implicit	`script_rp_implicit`
script_idp_form	`script_idp_form`

Figure 13. List of scripts in $\mathcal{S}$ and their respective string representations.

This outlines $\mathcal{OWS}$. We will now define the DY processes in $\mathcal{OWS}$ and their addresses, domain names, and secrets in more detail.

E.2. Addresses and Domain Names

The set IPs contains for every network attacker in Net, every relying party in RP, every identity provider in IDP, and every browser in B a finite set of addresses each. By addr we denote the corresponding assignment from a process to its address. The set Doms contains a finite set of domains for every relying party in RP, every identity provider in IDP, and every network attacker in Net. Browsers (in B) do not have a domain.

By addr and dom we denote the assignments from atomic processes to sets of IPs and Doms, respectively.

E.3. Keys and Secrets

The set $\mathcal{N}$ of nonces is partitioned into five sets, an infinite sequence N, an infinite set K_{SSL}, an infinite set K_{sign}, and finite sets Passwords, RPSecrets′ and ProtectedResources. We thus have

$$\mathcal{N} = \underbrace{N}_{\text{infinite sequence}} \dot\cup \underbrace{K_{\text{SSL}}}_{\text{finite}} \dot\cup \underbrace{\mathsf{Passwords}}_{\text{finite}} \dot\cup \underbrace{\mathsf{RPSecrets}'}_{\text{finite}} \dot\cup \underbrace{\mathsf{ProtectedResources}}_{\text{finite}} .$$

We then define $\mathsf{RPSecrets} := \mathsf{RPSecrets} \cup \{\bot\}$. These sets are used as follows:

- The set N contains the nonces that are available for each DY process in $\mathcal{W}$ (it can be used to create a run of $\mathcal{W}$).
- The set K_{SSL} contains the keys that will be used for SSL encryption. Let $\mathsf{sslkey} \colon \mathsf{Doms} \to K_{\text{SSL}}$ be an injective mapping that assigns a (different) private key to every domain. For an atomic DY process p we define $sslkeys^p = \langle \{ \langle d, \mathsf{sslkey}(d) \rangle \mid d \in \mathsf{dom}(p) \} \rangle$.
- The set Passwords is the set of passwords (secrets) the browsers share with the identity providers. These are the passwords the users use to log in at the IdPs.

- The set $\mathsf{RPSecrets}$ is the set of passwords (secrets) the relying parties share with the identity providers. These are the passwords the relying parties use to log in at the IdPs. The passwords can also be blank ($\bot$).
- The set $\mathsf{ProtectedResources}$ contains a secret for each combination of IdP, client, and user. These are thought of as protected resources that only the owner of the resource (i.e., the user) should be able to read.

E.4. Identities, Passwords, and Protected Resources

Identites consist, similar to email addresses, of a user name and a domain part. For our model, this is defined as follows:

Definition 40. An *identity* (email address) i is a term of the form $\langle name, domain\rangle$ with $name \in \mathbb{S}$ and $domain \in \mathsf{Doms}$.

Let ID be the finite set of identities. By ID^y we denote the set $\{\langle name, domain\rangle \in \mathsf{ID} \,|\, domain \in \mathsf{dom}(y)\}$.

We say that an ID is *governed* by the DY process to which the domain of the ID belongs. Formally, we define the mapping $\mathsf{governor} : \mathsf{ID} \to \mathcal{W}$, $\langle name, domain\rangle \mapsto \mathsf{dom}^{-1}(domain)$.

The governor of an ID will usually be an IdP, but could also be the attacker. Besides $\mathsf{governor}$, we define the following mappings:

- By $\mathsf{secretOfID} : \mathsf{ID} \to \mathsf{Passwords}$ we denote the bijective mapping that assigns secrets to all identities.
- Let $\mathsf{ownerOfSecret} : \mathsf{Passwords} \to \mathsf{B}$ denote the mapping that assigns to each secret a browser that *owns* this secret. Now, we define the mapping $\mathsf{ownerOfID} : \mathsf{ID} \to \mathsf{B}$, $i \mapsto \mathsf{ownerOfSecret}(\mathsf{secretOfID}(i))$, which assigns to each identity the browser that owns this identity (we say that the identity belongs to the browser).
- Let $\mathsf{trustedRPs} : \mathsf{Passwords} \to 2^{\mathsf{RP}}$ denote a mapping that assigns a set of *trusted relying parties* to each password. Intuitively a trusted relying party is a relying party the user entrusts with her password (in the resource owner password credentials grant mode of OAuth).
- Let $\mathsf{clientIDOfRP} : \mathsf{RP} \times \mathsf{IDP} \to \mathbb{S}$ denote a mapping that assigns an OAuth client id for an relying party to each combination of a relying party and an identity provider. We require that $\mathsf{clientIDOfRP}(\cdot, i)$ is bijective for any $i \in \mathsf{IDP}$.
- Let $\mathsf{secretOfRP} : \mathsf{RP} \times \mathsf{IDP} \to \mathsf{RPSecrets}$ denote a bijective mapping that assigns a relying party password (or the empty password $\bot$) to each combination of a relying party and an identity provider.
- As a shortcut, we define the mapping $\mathsf{secretOfClientID} : \mathbb{S} \times \mathsf{IDP} \to \mathsf{RPSecrets}$ to return the relying party password to a relying party identified by an OAuth client id (at some specific identity provider), i.e., $\mathsf{secretOfClientID}(s, i)$ maps to $\mathsf{secretOfRP}(r, i)$ with r such that $s = \mathsf{clientIDOfRP}(r, i)$.
- By $\mathsf{resourceOf} : \mathsf{IDP} \times \mathsf{RP} \times (\mathsf{ID} \cup \{\bot\}) \to \mathsf{ProtectedResources}$ we denote the injective mapping that assigns a protected resource to each combination of user identity, IdP and client (RP). We also include protected resources that are not assigned to a specific user (in this case, the user is $\bot$). (Note that a protected resource depends not only on the IdP and user ID but also the RP. This is

motivated by the fact that different RPs may get access to different protected resources at one IdP, even if they access the resources of the same user.)

E.5. Corruption

RPs and IdPs can become corrupted: If they receive the message `CORRUPT`, they start collecting all incoming messages in their state and (upon triggering) send out all messages that are derivable from their state and collected input messages, just like the attacker process. We say that an RP or an IdP is *honest* if the according part of their state ($s.\texttt{corrupt}$) is $\bot$, and that they are corrupted otherwise.

We are now ready to define the processes in $\mathcal{W}$ as well as the scripts in $\mathcal{S}$ in more detail.

E.6. Processes in $\mathcal{W}$ (Overview)

We first provide an overview of the processes in $\mathcal{W}$. All processes in $\mathcal{W}$ contain in their initial states all public keys and the private keys of their respective domains (if any). We define $I^p = \mathsf{addr}(p)$ for all $p \in \mathsf{Hon}$.

Network Attacker. There is one atomic DY process $na \in \mathsf{Net}$ which is a network attacker (see Appendix B.3), who uses all addresses for sending and listening.

Browsers. Each $b \in \mathsf{B}$ is a web browser as defined in Appendix D. The initial state contains all secrets owned by b, stored under the origins of the respective IdP and of all trusted RPs for the respective secret. See Appendix E.8 for details.

Relying Parties. Each relying party is a web server modeled as an atomic DY process following the description in Section 2 and the fixes discussed in Section 3. 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 only used during the login phase of a user, and the *service sessions* (modeled by a *service token* as described above).

When receiving a special message (`CORRUPT`) RPs can become corrupted. Similar to the definition of corruption for the browser, RPs then start sending out all messages that are derivable from their state.

Identity Providers. Each IdP is a web server modeled as an atomic DY process following the description in Section 2 and the fixes discussed in Section 3. In particular, users can authenticate to the IdP with their credentials. Authenticated users can interact with the authorization endpoint of the IdP (e.g., to acquire an authorization code). Just as RPs, IdPs can become corrupted.

E.7. Network Attackers

As mentioned, the network attacker *na* is modeled to be a network attacker as specified in Appendix B.3. We allow it to listen to/spoof all available IP addresses, and hence, define $I^{na} = \mathsf{IPs}$. The initial state is $s_0^{na} = \langle attdoms, sslkeys, signkeys \rangle$, where *attdoms* is a sequence of all domains along with the corresponding private keys owned by the attacker *na*, *sslkeys* is a sequence of all domains and the corresponding public keys, and *signkeys* is a sequence containing all public signing keys for all IdPs.

E.8. Browsers

Each $b \in \mathsf{B}$ is a web browser as defined in Appendix D, with $I^b := \mathsf{addr}(b)$ being its addresses.

To define the inital state, first let $\mathsf{ID}^b := \mathsf{ownerOfID}^{-1}(b)$ be the set of all IDs of b. We then define the set of passwords that a browser b gives to an origin o to consist of two parts: (1) If the origin belongs to an IdP, then the user's passwords of this IdP are contained in the set. (2) If the origin belongs to an

RP, then those passwords with which the user entrusts this RP are contained in the set. To define this mapping in the initial state, we first define for some process p

$$\mathsf{Secrets}^{b,p} = \Big\{ s \;\Big|\; b = \mathsf{ownerOfSecret}(s) \wedge \big((\exists i : s = \mathsf{secretOfID}(i) \wedge i \in \mathsf{governor}^{-1}(p)) \\ \vee (\exists R : p \in R \wedge s \in \mathsf{trustedRPs}^{-1}(R))\big) \Big\}.$$

Then, the initial state s_0^b is defined as follows: the key mapping maps every domain to its public (ssl) key, according to the mapping sslkey; the DNS address is an address of the network attacker; the list of secrets contains an entry $\langle\langle d, \mathsf{S}\rangle, \langle \mathsf{Secrets}^{b,p}\rangle\rangle$ for each $p \in \mathsf{RP} \cup \mathsf{IDP}$ and $d \in \mathsf{dom}(p)$; *ids* is $\langle \mathsf{ID}^b \rangle$; *sts* is empty.

E.9. Relying Parties

A relying party $r \in \mathsf{RP}$ is a web server modeled as an atomic DY process (I^r, Z^r, R^r, s_0^r) with the addresses $I^r := \mathsf{addr}(r)$. Its initial state s_0^r contains its domains, the private keys associated with its domains, the DNS server address, and information about IdPs RP is registered at. The full state additionally contains the sets of service tokens and login session identifiers the RP has issued as well as information about pending DNS and pending HTTPS requests (similar to browsers). RP only accepts HTTPS requests.

RP manages two kinds of sessions: The *login sessions*, which are only used during the login phase of a user, and the *service sessions* (we call the session identifier of a service session a *service token*). Service sessions allow a user to use RP's services. The ultimate goal of a login flow is to establish such a service session.

We now first describe how r can become corrupted, then we describe the handling of DNS and HTTPS requests and responses, before we describe the behaviour of r during a login flow.

Corruption. When r receives a corrupt message, it becomes corrupt and acts like the attacker from then on (i.e., it collects all incoming messages and non-deterministically sends out all messages derivable from its state).

Pending DNS Requests and Pending HTTPS Requests. Since the RP r also acts as an HTTPS client, it manages two kinds of records for messages that have been sent out into the network and are waiting for corresponding responses. When an HTTPS message is to be sent, the RP first needs to resolve the hostname into an IP address. To this end, the RP first stores the HTTPS request (together with some state information) in a subterm of its state called *pendingDNS* and (instead of sending the HTTPS request immediately) sends out a DNS request to the DNS server. When a DNS response arrives that matches one of the entries in this subterm, the HTTPS request is sent out over the network (to the resolved IP address) and stored in the subterm *pendingRequests* of the RP's state. Note that this mechanism is very similar to (generic) browsers (see Appendix D).

Initial Request. In a typical flow, r will first receive an HTTP GET request from a browser for the path `/`. In this case, r returns the script `script_rp_index`. In the browser, this script non-deterministically selects an identity of the user, i.e., a combination of a username and a domain of an IdP. Further this script non-deterministically decides whether an interactive login (i.e., authorization code mode or implicit mode) or a non-interactive login (i.e., resource owner password credentials mode) is used. If an interactive login is chosen, the script instructs the browser to send an HTTPS POST request to r for the path `/startInteractiveLogin`. This POST request contains in its body the domain of the IdP.[23] If

[23] Note that while the script has selected an identity of the user, only the domain of the IdP is used in this case and during the authentication to the IdP, a different username may be chosen.

the script chooses a non-interactive login, the domain of the IdP, the username, and the user's password are sent to *r* in an HTTPS POST request for the path `/passwordLogin`.

As the flow now forks into different branches, we will explain (the first part of) each of these branches separately: If the script has chosen to run an interactive login, we continue our description in the paragraph *Interactive Login* below. Else, if the script has chosen to run a non-interactive login, we continue our description of this in the paragraph *Non-Interactive Login*.

Interactive Login. In this case, `script_rp_index` has sent an HTTPS POST request for the path `/startInteractiveLogin` to *r* containing the name of an IdP in its body. When *r* receives such a request, *r* non-deterministically decides whether the OAuth authorization code mode or the OAuth implicit mode is used. Also, *r* non-deterministically selects a redirect URI *redirect_uri* of its redirection endpoints (and appends the domain of the IdP to this redirect URI) or selects no redirect URI. Further, *r* non-deterministically selects a (fresh) nonce *state* and a (fresh) nonce as login session id. Then, *r* saves all the chosen information in its state. Now, *r* constructs and sends an HTTPS response containing an HTTP 303 location redirect or an HTTP 307 location redirect[24] (chosen non-deterministically) which points to the corresponding authorization endpoint at the IdP along with *r*'s OAuth client id for this IdP, *state* and information which OAuth mode *r* has chosen. Additionally, this response also contains a Set-Cookie header, which sets a cookie containing the login session id. *r* also stores a record in the subterm *loginSessions* of its state. This record contains the login session id, the chosen OAuth mode, and the domain of the IdP.

Later, when IdP redirects the user's browser to *r*'s redirection endpoint, *r* will receive an HTTPS GET request for the path `/redirectionEndpoint`. This request must contain a login session id cookie, which refers to the information stored in the subterm *loginSessions* in *r*'s state. The request must also contain a parameter with the domain of the IdP and this domain must match the domain stored for this login session.

If *r* has stored that for this login session the OAuth authorization code mode is used, *r* checks if the *state* value contained in a parameter is correct (i.e., the value of this parameter is congruent to the value recorded in *r*'s state). Then, *r* extracts the authorization code *code* from the parameters of the incoming request and prepares an HTTPS POST request to the IdP's token endpoint to obtain an access token as follows: *r* adds the authorization code to the request's body. If a redirect URI has been set by *r* before (according to *r*'s state for this login session), the redirect URI is included in the request's body. If *r* knows an OAuth client secret for the IdP, *r* adds its OAuth client id and its OAuth client secret for the IdP to the header of the request, else *r* adds its OAuth client id for the IdP to the request's body. Now, *r* sends a DNS request for the domain of the IdP's token endpoint to the DNS server (according to *r*'s state), saves this (prepared) request and all information belonging to the (incoming) HTTPS request *r* received from the browser (such as IP addresses, temporary HTTPS keys) in *pendingDNS* in its state. We will continue our description of which requests *r* will process next in the OAuth authorization code mode in the paragraph *Token Response* below.

If the (incoming) HTTPS request's login session at *r* states that implicit mode is used, *r* instead sends an HTTPS response to the sender of the incoming message. This HTTPS response contains the script *script_rp_implicit* and the initial state for this script in this response contains the domain of the IdP.

In a browser, this script extracts *access_token* and *state* from the fragment part of its URL and extracts the domain of the IdP from its initial state. The script then sends this information in the body of an HTTPS POST request for the path `/receiveTokenFromImplicitGrant` to *r*.

When *r* receives such an HTTPS POST request (for the path `/receiveTokenFromImplicitGrant`), *r* checks if this request contains a login session id cookie, which refers to the information stored in its

[24] Note that while in this paper we present an attack against OAuth based on an HTTP 307 location redirect, our analysis shows that an HTTP 307 location redirect is safe at this point in the protocol flow.

state and if the values of *state* and *idp* (contained in the request) match the information there. Next, r prepares an HTTPS request to IdP's inspection endpoint containing the access token just received. r saves all information belonging to this new request and the (incoming) request it had just received in *pendingDNS* in its state and sends out a DNS request for the domain of the IdP's inspection endpoint to the DNS server.

We describe what happens when r later receives the response from IdP in the paragraph *Inspection Response* below.

Non-Interactive Login. In this case, `script_rp_index` has sent an HTTPS POST request for the path `/passwordLogin` to r containing a domain of an IdP, a username and a user's password in its body. Next, r constructs an HTTPS POST request to the token endpoint of the IdP. This request contains the username and the user's password in its body and if r knows an OAuth client secret for the IdP, the request contains an HTTP header with r's OAuth client id and OAuth client secret. r saves all information belonging to this new request and the (incoming) request r has just received in the subterm *pendingDNS* in r's state and sends out a DNS request for the domain of the IdP's token endpoint to the DNS server.

We describe what happens when r later receives the response from the IdP in the paragraph *Token Response* below.

Client Credentials Mode. When r receives a `TRIGGER` message (which models that r non-deterministically starts an OAuth flow in the client credentials mode), r first non-deterministically selects a domain of an IdP. Then, r constructs an HTTPS POST request to the token endpoint of the IdP. This request contains an HTTP header with r's OAuth client id and OAuth client secret.[25] r saves all information belonging to this (prepared) request in *pendingDNS* and sends out a DNS request for the domain of the IdP's token endpoint to the DNS server.

We describe what happens when r later receives the response from IdP in the paragraph *Token Response* below.

Token Response. When r receives an encrypted HTTP response that matches a record in the subterm *pendingRequests* of its state and belongs to a request for an access token from an IdP (according to the information recorded in *pendingRequests*), then r extracts the access token and prepares an HTTPS request to the IdP's inspection endpoint containing the access token. r saves all information belonging to this new request in *pendingDNS*. Further, r also stores selected information, which is passed along in r's state in the corresponding record of the incoming request, such as the IP address of the sender and the HTTPS response key of the request which initiated r's request for the access token before. Then, r sends out a DNS request for the domain of the IdP's inspection endpoint to the DNS server.

Inspection Response. When r receives an encrypted HTTP response that matches a record in the subterm *pendingRequests* in its state and this record belongs to a request to an IdP's inspection endpoint, r behaves as follows: If the response belongs to a flow in client credentials mode (according to the record), r stops. Else, if the response contains r's OAuth client id, r retrieves the user id from the response and non-deterministically chooses a fresh nonce as a service token. r records in its state that the service token belongs to this user at the IdP. Now, r sends out an HTTPS response to the IP address recorded in the record in *pendingRequests* (which contains the IP address of the browser, which initially sent either user credentials, an authorization code, or an access token). This response contains the service token.

This concludes the description of the behaviour of an RP.

Formal description. We now provide the formal definition of r as an atomic DY process (I^r, Z^r, R^r, s_0^r).

[25] Note that in our model, r may even construct such a request if r does not have an OAuth client secret for the IdP. In this case, the symbol $\perp$ is placed in this header instead of an OAuth client secret. The IdP, however, will drop such a request, as it is not authenticated.

As mentioned, we define $I^r = \mathsf{addr}(r)$. Next, we define the set Z^r of states of r and the initial state s_0^r of r.

Definition 41. An *IdP registration record* is a term of the form

$$\langle \mathit{tokenEndpoint}, \mathit{authorizationEndpoint}, \mathit{inspectionEndpoint}, \mathit{clientId}, \mathit{clientPassword} \rangle$$

with $\mathit{tokenEndpoint}$, $\mathit{authorizationEndpoint}$, $\mathit{inspectionEndpoint} \in \mathsf{URLs}$, $\mathit{clientId} \in \mathbb{S}$, and $\mathit{clientPassword} \in \mathcal{N}$.

An *IdP registration record for an identity provider i at a relying party r* is an IdP registration record with $\mathit{tokenEndpoint}.\mathtt{host}$, $\mathit{authorizationEndpoint}.\mathtt{host}$, $\mathit{inspectionEndpoint}.\mathtt{host} \in \mathsf{dom}(i)$, $\mathit{clientId} = \mathsf{clientIDOfRP}(r,i)$, and $\mathit{clientPassword} = \mathsf{secretOfRP}(r,i)$.

Definition 42. A *state $s \in Z^r$ of an RP r* is a term of the form $\langle \mathit{DNSAddress}, \mathit{idps}, \mathit{serviceTokens}, \mathit{loginSessions}, \mathit{keyMapping}, \mathit{sslkeys}, \mathit{pendingDNS}, \mathit{pendingRequests}, \mathit{corrupt} \rangle$ where $\mathit{DNSAddress} \in \mathsf{IPs}$, $\mathit{idps} \in \left[\mathsf{Doms} \times \mathcal{T}_{\mathcal{N}}\right]$ is a dictionary of IdP registration records, $\mathit{serviceTokens} \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{loginSessions} \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$ is a dictionary of login session records, $\mathit{keyMapping} \in [\mathbb{S} \times \mathcal{N}]$, $\mathit{sslkeys} = \mathit{sslkeys}^r$, $\mathit{pendingDNS} \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{pendingRequests} \in \left[\mathcal{N} \times \mathcal{T}_{\mathcal{N}}\right]$, $\mathit{corrupt} \in \mathcal{T}_{\mathcal{N}}$.

An *initial state s_0^r of r* is a state of r with $s_0^r.\mathtt{idps}$ being a dictionary that maps each domain of all identity providers i to an IdP registration record for i at r, $s_0^r.\mathtt{serviceTokens} = s_0^r.\mathtt{loginSessions} = \langle\rangle$, $s_0^r.\mathtt{corrupt} = \bot$, and $s_0^r.\mathtt{keyMapping}$ is the same as the keymapping for browsers above.

We now specify the relation R^r. Just like in Appendix D, we describe this relation by a non-deterministic algorithm. In several places throughout this algorithm we use placeholders to generate "fresh" nonces as described in our communication model (see Definition 1). Figure 14 shows a list of all placeholders used.

Placeholder	Usage
ν_1	new HTTP request nonce
ν_2	lookup key for pending DNS entry
ν_3	new service token
ν_4	fresh HTTPS response key
ν_5	new HTTP request nonce
ν_6	lookup key for pending DNS entry
ν_7	new CSRF token
ν_8	new login session cookie
ν_9	new HTTP request nonce
ν_{10}	lookup key for pending DNS entry
ν_{11}	new HTTP request nonce
ν_{12}	lookup key for pending DNS entry
ν_{13}	new HTTP request nonce
ν_{14}	lookup key for pending DNS entry

Figure 14. List of placeholders used in the relying party algorithm.

Algorithm 8 Relation of a Relying Party R^r

Input: $\langle a, f, m \rangle, s$

1: **if** $s'.\mathtt{corrupt} \not\equiv \bot \vee m \equiv \mathtt{CORRUPT}$ **then**

```
2:      let s′.corrupt := ⟨⟨a, f, m⟩, s′.corrupt⟩
3:      let m′ ← d_V(s′)
4:      let a′ ← IPs
5:      stop ⟨⟨a′, a, m′⟩⟩, s′
6:  end if
7:  if ∃⟨reference, request, key, f⟩ ∈^⟨⟩ s′.pendingRequests
        ↪ such that π1(dec_s(m, key)) ≡ HTTPResp then                ▷ Encrypted HTTP response
8:      let m′ := dec_s(m, key)
9:      if m′.nonce ≢ request.nonce then
10:         stop ⟨⟩, s
11:     end if
12:     remove ⟨reference, request, key, f⟩ from s′.pendingRequests
13:     let mode := π1(reference)
14:     if mode ≡ code ∨ mode ≡ password ∨ mode ≡ client_credentials then
15:         let idp, a′, f′, n′, k′ such that ⟨mode, idp, a′, f′, n′, k′⟩ ≡ reference if possible; otherwise stop ⟨⟩, s
16:         let token := m′.body[access_token]
17:         let inspectionEndpoint := s′.idps[idp]
18:         let parameters := inspectionEndpoint.parameters
19:         let parameters := parameters +^⟨⟩ ⟨token, token⟩
20:         let host := inspectionEndpoint.domain
21:         let path := inspectionEndpoint.path
22:         let message := ⟨HTTPReq, ν1, GET, host, path, parameters, ⟨⟩, ⟨⟩⟩
23:         let s′.pendingDNS[ν2] := ⟨⟨inspect, mode, idp, a′, f′, n′, k′⟩, message⟩
24:         stop ⟨⟨s′.DNSaddress, a, ⟨DNSResolve, inspectionEndpoint.domain, ν2⟩⟩⟩, s′
25:     else if mode ≡ inspect then
26:         let mode, idp, a′, f′, n′, k′ such that ⟨inspect, mode, idp, a′, f′, n′, k′⟩ ≡ reference
              ↪ if possible; otherwise stop ⟨⟩, s
27:         if mode ≡ client_credentials then
28:             stop ⟨⟩, s                ▷ In client credential grant mode, no service token is issued.
29:         end if
30:         let clientId := body[client_id]
31:         if clientId ≡ s′.idps[idp].clientId ∨ (clientId ≡ ⟨⟩ ∧ mode ≡ password ∧
              ↪ s′.idps[idp].clientPassword ≡ ⊥) then
32:             let user := body[user]
33:         else
34:             stop ⟨⟩, s
35:         end if
36:         let serviceToken := ν3
37:         let s′.serviceTokens[serviceToken] := ⟨user, idp⟩
38:         let m′ := enc_s(⟨HTTPResp, n′, 200, ⟨⟩, ⟨serviceToken⟩⟩, k′)
39:         stop ⟨⟨f′, a′, m′⟩⟩, s′
40:     end if
41:     stop ⟨⟩, s
42: else if m ∈ DNSResponses then                                    ▷ Successful DNS response
43:     if m.nonce ∉ s.pendingDNS ∨ m.result ∉ IPs ∨ m.domain ≢ π2(s.pendingDNS).host then
44:         stop ⟨⟩, s
45:     end if
46:     let ⟨reference, message⟩ := s.pendingDNS[m.nonce]
47:     let s′.pendingRequests := s′.pendingRequests
            ↪ +^⟨⟩ ⟨reference, message, ν4, m.result⟩
48:     let message := enc_a(⟨message, ν4⟩, s′.keyMapping[message.host])
49:     let s′.pendingDNS := s′.pendingDNS − m.nonce
50:     stop ⟨⟨m.result, a, message⟩⟩, s′
51: else if m ≡ TRIGGER then                                         ▷ Start Client Credentials Grant
```

52: **let** *idpEntry* $\leftarrow s'.\texttt{idps}$
53: **let** *idp* $:= \pi_1(idpEntry)$
54: **let** *tokenEndpoint* $:= s'.\texttt{idps}[idp].\texttt{tokenEndpoint}$ ▷ *tokenEndpoint* is a URL
55: **let** *host* $:=$ *tokenEndpoint*.domain
56: **let** *path* $:=$ *tokenEndpoint*.path
57: **let** *parameters* $:=$ *tokenEndpoint*.parameters
58: **let** *headers* $:= \langle\langle \texttt{Authorization}, \langle s'.\texttt{idps}[idp].\texttt{clientId}, s'.\texttt{idps}[idp].\texttt{clientPassword}\rangle\rangle\rangle$
59: **let** *message* :=
↪ $\langle \texttt{HTTPReq}, \nu_5, \texttt{POST}, host, path, parameters, headers, \langle\langle \texttt{grant\_type}, \texttt{client\_credentials}\rangle\rangle\rangle$
60: **let** $s'.\texttt{pendingDNS}[\nu_6] := \langle\langle \texttt{client\_credentials}, idp, \bot, \bot, \bot, \bot\rangle, message\rangle$
61: **stop** $\langle\langle s'.\texttt{DNSaddress}, a, \langle \texttt{DNSResolve}, idp.\texttt{tokenEndpoint.domain}, \nu_6\rangle\rangle\rangle, s'$
62: **else** ▷ Handle HTTP requests
63: **let** m_{dec}, *k*, *k'*, *inDomain* **such that**
↪ $\langle m_{\text{dec}}, k\rangle \equiv \text{dec}_{\text{a}}(m, k') \wedge \langle inDomain, k'\rangle \in s.\texttt{sslkeys}$
↪ **if possible; otherwise stop** $\langle\rangle, s$
64: **let** *n*, *method*, *path*, *parameters*, *headers*, *body* **such that**
↪ $\langle \texttt{HTTPReq}, n, method, inDomain, path, parameters, headers, body\rangle \equiv m_{\text{dec}}$
↪ **if possible; otherwise stop** $\langle\rangle, s$
65: **if** $path \equiv /$ **then** ▷ Serve index page.
66: **let** $m' := \text{enc}_{\text{s}}(\langle \texttt{HTTPResp}, n, 200, \langle\rangle, \langle \texttt{script\_rp\_index}, \langle\rangle\rangle\rangle, k)$
67: **stop** $\langle\langle f, a, m'\rangle\rangle, s'$
68: **else if** $path \equiv /\texttt{startInteractiveLogin} \wedge method \equiv \texttt{POST}$ **then** ▷ Serve start interactive login request.
69: **let** *idp* $:=$ *body*
70: **if** $idp \notin s'.\texttt{idps}$ **then**
71: **stop** $\langle\rangle, s$
72: **end if**
73: **let** *state* $:= \nu_7$
74: **let** *mode* $\leftarrow \{\texttt{code}, \texttt{token}\}$
75: **let** *responseStatus* $\leftarrow \{303, 307\}$
76: **let** *authEndpoint* $:= s'.\texttt{idps}[idp].\texttt{authorizationEndpoint}$ ▷ *authEndpoint* is a URL
77: **let** *authEndpoint*.parameters $:=$ *authEndpoint*.parameters $+^{\langle\rangle} \langle \texttt{response\_type}, mode\rangle$
78: **let** *authEndpoint*.parameters $:=$ *authEndpoint*.parameters $+^{\langle\rangle}$
↪ $\langle \texttt{client\_id}, s'.\texttt{idps}[idp].\texttt{clientId}\rangle$
79: **let** *authEndpoint*.parameters $:=$ *authEndpoint*.parameters $+^{\langle\rangle} \langle \texttt{state}, state\rangle$
80: **let** *redirectUri* $\leftarrow \{\bot, \top\}$
81: **if** $redirectUri \equiv \top$ **then**
82: **let** *sslkey'* $\leftarrow s'.\texttt{sslkeys}$ ▷ Choose one of RP's domains non-deterministically
83: **let** *host'* $:= \pi_1(sslkey')$
84: **let** *redirectUri* $:= \langle \texttt{URL}, \texttt{S}, host', /\texttt{redirectionEndpoint}, \langle\langle \texttt{idp}, idp\rangle\rangle, \langle\rangle\rangle$
85: **end if**
86: **let** *loginSessionId* $:= \nu_8$
87: **let** $s'.\texttt{loginSessions} := s'.\texttt{loginSessions} +^{\langle\rangle} \langle loginSessionId, \langle idp, state, mode, redirectUri\rangle\rangle$
88: **let** *setCookies* $:= \langle\langle \texttt{loginSessionId}, loginSessionId, \top, \top, \top\rangle\rangle$
89: **let** $m' := \text{enc}_{\text{s}}(\langle \texttt{HTTPResp}, n, responseStatus, \langle\langle \texttt{Location}, authEndpoint\rangle,$
↪ $\langle \texttt{Set-Cookie}, setCookies\rangle\rangle, \bot\rangle, k)$
90: **stop** $\langle\langle f, a, m'\rangle\rangle, s'$
91: **else if** $path \equiv /\texttt{redirectionEndpoint}$ **then**
92: **let** *loginSessionId* $:= headers[\texttt{Cookie}][\texttt{loginSessionId}]$
93: **let** *idp*, *state*, *mode*, *redirectUri* **such that** $\langle idp, state, mode, redirectUri\rangle \equiv$
↪ $s'.\texttt{loginSessions}[loginSessionId]$ **if possible; otherwise stop** $\langle\rangle, s$
94: **if** $idp \not\equiv parameters[\texttt{idp}]$ **then**
95: **stop** $\langle\rangle, s$
96: **end if**
97: **if** $mode \equiv \texttt{code}$ **then** ▷ Continue Authorization Code Grant

98: **if** *parameters*[`state`] $\not\equiv$ *state* **then**
99: **stop** $\langle\rangle$, *s*
100: **end if**
101: **let** *code* := *parameters*[`code`]
102: **let** *tokenRequestHeaders* := $\langle\rangle$
103: **let** *tokenRequestBody* := $\langle\langle$`grant_type`, `authorization_code`$\rangle$, $\langle$`code`, *code*$\rangle\rangle$
104: **if** *redirectUri* $\not\equiv \bot$ **then**
105: **let** *tokenRequestBody* := *tokenRequestBody* $+^{\langle\rangle}$ $\langle$`redirect_uri`, *redirectUri*$\rangle$
106: **end if**
107: **let** *clientId* := *s*′.`idps`[*idp*].`clientId`
108: **let** *clientPassword* := *s*′.`idps`[*idp*].`clientPassword`
109: **if** *clientPassword* $\equiv \bot$ **then**
110: **let** *tokenRequestBody* := *tokenRequestBody* $+^{\langle\rangle}$ $\langle$`client_id`, *clientId*$\rangle$
111: **else**
112: **let** *tokenRequestHeaders* := *tokenRequestHeaders* $+^{\langle\rangle}$ $\hookrightarrow$ $\langle$`Authorization`, $\langle$*clientId*, *clientPassword*$\rangle\rangle$
113: **end if**
114: **let** *tokenEndpoint* := *s*′.`idps`[*idp*].`tokenEndpoint`
115: **let** *message* := $\langle$`HTTPReq`, ν_9, `POST`, *tokenEndpoint*.`domain`, *tokenEndpoint*.`path`, *tokenEndpoint*.`parameters`, *tokenRequestHeaders*, *tokenRequestBody*$\rangle$
116: **let** *s*′.`pendingDNS`[ν_{10}] := $\langle\langle$`code`, *idp*, *a*, *f*, *n*, *k*$\rangle$, *message*$\rangle$
117: **stop** $\langle\langle s'$.`DNSaddress`, *a*, $\langle$`DNSResolve`, *tokenEndpoint*.`domain`, $\nu_{10}\rangle\rangle\rangle$, *s*′
118: **else if** *mode* $\equiv$ `token` **then** ▷ Continue Implicit Grant
119: **let** *m*′ := $\mathsf{enc_s}(\langle$`HTTPResp`, *n*, 200, $\langle\rangle$, $\langle$`script_rp_implicit`, *idp*$\rangle\rangle$, *k*$)$
120: **stop** $\langle\langle f, a, m'\rangle\rangle$, *s*′
121: **end if**
122: **stop** $\langle\rangle$, *s*
123: **else if** *path* $\equiv$ /`passwordLogin` $\wedge$ *method* $\equiv$ `POST` **then**
124: **let** *idp, username, password* **such that** $\langle\langle$*username*, *idp*$\rangle$, *password*$\rangle \equiv$ *body* **if possible; otherwise** $\hookrightarrow$ **stop** $\langle\rangle$, *s*
125: **let** *tokenRequestHeaders* := $\langle\rangle$
126: **let** *tokenRequestBody* := $\langle\langle$`grant_type`, `password`$\rangle$, $\langle$`username`, $\langle$*username*, *idp*$\rangle\rangle$, $\hookrightarrow$ $\langle$`password`, *password*$\rangle\rangle$
127: **let** *clientId* := *s*′.`idps`[*idp*].`clientId`
128: **let** *clientPassword* := *s*′.`idps`[*idp*].`clientPassword`
129: **if** *clientPassword* $\not\equiv \bot$ **then**
130: **let** *tokenRequestHeaders* := *tokenRequestHeaders* $+^{\langle\rangle}$ $\hookrightarrow$ $\langle$`Authorization`, $\langle$*clientId*, *clientPassword*$\rangle\rangle$
131: **end if**
132: **let** *tokenEndpoint* := *s*′.`idps`[*idp*].`tokenEndpoint`
133: **let** *message* := $\langle$`HTTPReq`, ν_{11}, `POST`, *tokenEndpoint*.`domain`, *tokenEndpoint*.`path`, *tokenEndpoint*.`parameters`, *tokenRequestHeaders*, *tokenRequestBody*$\rangle$
134: **let** *s*′.`pendingDNS`[ν_{12}] := $\langle\langle$`password`, *idp*, *a*, *f*, *n*, *k*$\rangle$, *message*$\rangle$
135: **stop** $\langle\langle s'$.`DNSaddress`, *a*, $\langle$`DNSResolve`, *tokenEndpoint*.`domain`, $\nu_{12}\rangle\rangle\rangle$, *s*′
136: **else if** *path* $\equiv$ /`receiveTokenFromImplicitGrant` $\wedge$ *method* $\equiv$ `POST` **then**
137: **let** *loginSessionId* := *headers*[`Cookie`][`loginSessionId`]
138: **let** *idp, state, mode, redirectUri* **such that** $\langle$*idp*, *state*, *mode*, *redirectUri*$\rangle \equiv$ $\hookrightarrow$ *s*′.`loginSessions`[*loginSessionId*] **if possible; otherwise stop** $\langle\rangle$, *s*
139: **let** *token* **such that** $\langle$*token*, *state*, *idp*$\rangle \equiv$ *body* **if possible; otherwise stop** $\langle\rangle$, *s*
140: **let** *inspectionEndpoint* := *s*′.`idps`[*idp*].`inspectionEndpoint` ▷ *inspectionEndpoint* is a URL
141: **let** *parameters*′ := *inspectionEndpoint*.`parameters`
142: **let** *parameters*′ := *parameters*′ $+^{\langle\rangle}$ $\langle$`token`, *token*$\rangle$
143: **let** *host* := *inspectionEndpoint*.`domain`
144: **let** *path*′ := *inspectionEndpoint*.`path`

145: **let** *message* := $\langle \texttt{HTTPReq}, \nu_{13}, \texttt{GET}, host, path', parameters', \langle\rangle, \langle\rangle \rangle$
146: **let** $s'.\texttt{pendingDNS}[\nu_{14}] := \langle \langle \texttt{inspect}, \texttt{implicit}, idp, a, f, n, k \rangle, message \rangle$
147: **stop** $\langle \langle s'.\texttt{DNSaddress}, a, \langle \texttt{DNSResolve}, inspectionEndpoint.\texttt{domain}, \nu_{14} \rangle \rangle, s'$
148: **end if**
149: **end if**
150: **stop** $\langle\rangle, s$

In the following scripts, to extract the current URL of a document, the function $\mathsf{GETURL}(tree, docnonce)$ is used. We define this function as follows: It searches for the document with the identifier *docnonce* in the (cleaned) tree *tree* of the browser's windows and documents. It then returns the URL u of that document. If no document with nonce *docnonce* is found in the tree *tree*, $\diamond$ is returned.

Algorithm 9 Relation of *script_rp_index*

Input: $\langle$*tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets*$\rangle$
1: **let** *url* := $\mathsf{GETURL}(tree, docnonce)$
2: **let** *id* $\leftarrow$ *ids*
3: **let** *interactive* $\leftarrow \{\bot, \top\}$
4: **if** *interactive* $\equiv \top$ **then**
5: **let** $url' := \langle \texttt{URL}, \texttt{S}, url.\texttt{host}, /\texttt{startInteractiveLogin}, \langle\rangle, \langle\rangle \rangle$
6: **let** *command* := $\langle \texttt{FORM}, url', \texttt{POST}, \pi_2(id), \bot \rangle$
7: **else**
8: **let** $url' := \langle \texttt{URL}, \texttt{S}, url.\texttt{host}, /\texttt{passwordLogin}, \langle\rangle, \langle\rangle \rangle$
9: **let** *secret* **such that** $secret = \mathsf{secretOfID}(id) \wedge secret \in secrets$ **if possible; otherwise** ↪ **stop** $\langle s, cookies, localStorage, sessionStorage, \langle\rangle \rangle$
10: **let** *command* := $\langle \texttt{FORM}, url', \texttt{POST}, \langle id, secret \rangle, \bot \rangle$
11: **end if**
12: **stop** $\langle s, cookies, localStorage, sessionStorage, command \rangle$

Algorithm 10 Relation of *script_rp_implicit*

Input: $\langle$*tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets*$\rangle$
1: **let** *url* := $\mathsf{GETURL}(tree, docnonce)$
2: **let** $url' := \langle \texttt{URL}, \texttt{S}, url.\texttt{host}, /\texttt{receiveTokenFromImplicitGrant}, \langle\rangle, \langle\rangle \rangle$
3: **let** *body* := $\langle url.\texttt{fragment}[\texttt{access\_token}], url.\texttt{fragment}[\texttt{state}], scriptstate \rangle$
4: **let** *command* := $\langle \texttt{FORM}, url', \texttt{POST}, body, \bot \rangle$
5: **stop** $\langle s, cookies, localStorage, sessionStorage, command \rangle$

E.10. Identity Providers

An identity provider $i \in \mathsf{IdPs}$ is a web server modeled as an atomic process (I^i, Z^i, R^i, s_0^i) with the addresses $I^i := \mathsf{addr}(i)$. Its initial state s_0^i contains a list of its domains and (private) SSL keys, the paths for the endpoints (authorization and token), a list of users, a list of clients, and information about the corruption status (initially, the IdP is not corrupted). Besides this, the full state of i further contains a list of issued authorization codes and access tokens.

Once the IdP becomes corrupted (when it receives the message `corrupt`), it starts collecting all input messages and non-deterministically sending out whatever messages are derivable from its state.

Otherwise, IdPs react to three types of requests:

Requests to the authorization endpoint path: In this case, the IdP expects a POST request containing valid user credentials. If the user credentials are not supplied, or the request is not a POST request, the answer contains a script which shows a form to the user to enter her user credentials. In our model, the script just extracts the user credentials from the browser and sends a request to the IdP containing the

user credentials and any OAuth parameters contained in the original request (e.g., the intended redirect URI).

If the IdP received a POST request with valid user credentials, it checks the contained client identifier against its own list of clients. If the client identifier is unknown, the IdP aborts. Otherwise, it ensures that the redirect URI, if contained in the request, is valid. For this, it checks the list of redirect URIs stored along with the client identifier. If none of the redirect URIs match the redirect URI presented in the request (see "Matching Redirect URIs" below), the IdP aborts. If no redirect URI is provided in the request, the first URI in the list of redirect URIs is chosen as the redirect URI.

Now the IdP creates a new authorization code and saves this code together with the client identifier and the redirect URI (if provided in the request) to the list of authorization codes.

Now, if the response type parameter in the request is "code", the IdP issues a Location redirect header to the redirect URI, appending (as parameters) the newly created authorization code and the state (if provided in the request).

If the reponse type is "token", the IdP redirects the browser to the redirect URI, but appends the authorization code, the state (if provided) and a fixed string (containing the token type, which is "bearer") to the hash of the redirect URI.

Requests to the token endpoint path: Requests to the token endpoint path are only accepted by the IdP if they are POST requests. The IdP then checks that the request either contains a valid client ID, provided as a parameter, or a pair of client ID and client password in a basic authentication header.

If the grant type parameter is *authorization code*, then the IdP checks that the authorization code delivered to it is contained in the list of codes. It checks that the client ID and redirect URI are the same as those stored in the list of codes. It then creates an access token and returns it in the HTTPS response (with token type "bearer").

If the grant type is *password*, the IdP checks the provided username and password and creates an access token as above.

If the grant type is *client credentials*, the IdP checks that the client was authorized with client ID and client password above. If so, it creates an access token as above.

Requests to the inspection endpoint path: In this case, the IdP expects an access token in the parameters of the request. If the access token is valid, the IdP returns the client and user id for which the access token was issued along with the protected resource for this client, user, and IdP.

Formal description. In the following, we will first define the (initial) state of i formally and afterwards present the definition of the relation R^i.

To define the initial state, we will need to add a list of all protected resources that this IdP manages. We therefore define $srlist^i := \langle \{\mathsf{resourceOf}(i,c,u) \mid c \in \mathsf{RP}, u \in \mathsf{ID}\} \rangle$ for some IdP i. (Note that we do not use this term for term manipulations in the algorithm. Instead, this term ensures that the output of the atomic process is derivable from the input.)

Definition 43. A *state* $s \in Z^i$ *of an IdP* i is a term of the form $\langle sslkeys, srlist, authEndpoint, tokenEndpoint, inspectEndpoint, clients, codes, corrupt \rangle$ where $sslkeys = sslkeys^i$, $srlist = srlist^i$, $authEndpoint, tokenEndpoint, inspectEndpoint \in \mathbb{S}$, $clients \in [\mathbb{S} \times \mathcal{T}_{\mathcal{N}}]$, $codes \in \mathcal{T}_{\mathcal{N}}$, $atokens \in [\mathcal{N} \times \mathbb{S}]$.

An *initial state* s_0^i *of* i is a state of the form $\langle sslkeys^i, srlist^i, w, x, y, clients^i, \langle\rangle, \langle\rangle, \bot \rangle$ for some strings w, x and y and a dictionary $clients^i$ that for each relying party r contains an entry of the form $\langle \mathsf{clientIDOfRP}(r,i), z \rangle$ where z is a sequence of URL terms that may contain the wildcard $*$ (see Definition 4) where for every $u \in^{\langle\rangle} z$ we have that $u.\mathtt{protocol} \equiv \mathtt{S}$, $u.\mathtt{host} \in \mathsf{dom}(r)$ and $u.\mathtt{parameters}[\mathtt{idp}] \equiv d$ for some $d \in \mathsf{dom}(i)$.

The relation R^i that defines the behavior of the IdP i is defined as follows:

Algorithm 11 Relation of IdP R^i

Input: $\langle a, f, m\rangle, s$

if $s'.\texttt{corrupt} \not\equiv \bot \vee m \equiv \texttt{CORRUPT}$ **then**
  let $s'.\texttt{corrupt} := \langle\langle a, f, m\rangle, s'.\texttt{corrupt}\rangle$
  let $m' \leftarrow d_V(s')$
  let $a' \leftarrow \mathsf{IPs}$
  stop $\langle\langle a', a, m'\rangle\rangle$, s'
end if
let $s' := s$
let m_{dec}, k, k', *inDomain* **such that**
  $\hookrightarrow$ $\langle m_{\text{dec}}, k\rangle \equiv \mathsf{dec_a}(m, k') \wedge \langle \mathit{inDomain}, k'\rangle \in s.\texttt{sslkeys}$
  $\hookrightarrow$ **if possible; otherwise stop** $\langle\rangle$, s
let *n, method, path, parameters, headers, body* **such that**
  $\hookrightarrow$ $\langle \texttt{HTTPReq}, n, \mathit{method}, \mathit{inDomain}, \mathit{path}, \mathit{parameters}, \mathit{headers}, \mathit{body}\rangle \equiv m_{\text{dec}}$
  $\hookrightarrow$ **if possible; otherwise stop** $\langle\rangle$, s
if $\mathit{path} \equiv s.\texttt{authEndpoint}$ **then** ▷ Authorization Endpoint.
  if $\mathit{method} \equiv \texttt{GET} \vee (\mathit{method} \equiv \texttt{POST} \wedge (\mathit{body}[\texttt{username}] \equiv \langle\rangle \vee \mathit{body}[\texttt{password}] \equiv \langle\rangle))$ **then**
    let $\mathit{data} := \mathit{parameters}$
    let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle\rangle, \langle \texttt{script\_idp\_form}, \mathit{data}\rangle\rangle, k)$
    stop $\langle\langle f, a, m'\rangle\rangle$, s'
  else if $\mathit{method} \equiv \texttt{POST}$ **then**
    if $\mathit{headers}[\texttt{Origin}] \not\equiv \langle \mathit{inDomain}, \mathtt{S}\rangle$ **then** ▷ CSRF protection.
      stop $\langle\rangle$, s
    end if
    let $\mathit{username} := \mathit{body}[\texttt{username}]$
    let $\mathit{password} := \mathit{body}[\texttt{password}]$
    let $\mathit{clientid} := \mathit{body}[\texttt{client\_id}]$
    let $\mathit{allowedredirects} := s.\texttt{clients}[\mathit{clientid}]$
    if $\mathit{password} \not\equiv \mathsf{secretOfID}(\mathit{username})$ **then**
      stop $\langle\rangle$, s
    end if
    if $\mathit{allowedredirects} \equiv \langle\rangle$ **then**
      stop $\langle\rangle$, s
    end if
    let $\mathit{redirecturi} := \mathit{body}[\texttt{redirect\_uri}]$
    if $\mathit{redirecturi} \not\equiv \langle\rangle$ **then**
      if not $\mathit{redirecturi} \dot{\sim} \mathit{allowedredirects}$ **then**
        stop $\langle\rangle$, s
      end if
    else
      let $\mathit{redirecturi} \leftarrow \mathit{allowedredirects}$ ▷ Take one from list of redir URIs.
    end if
    if $\mathit{body}[\texttt{response\_type}] \equiv \texttt{code}$ **then**
      let $s'.\texttt{codes} := s'.\texttt{codes} +^{\langle\rangle} \langle \nu_1, \langle \mathit{clientid}, \mathit{body}[\texttt{redirect\_uri}], \mathit{username}\rangle\rangle$ ▷ Create authorization code.
      let $\mathit{redirecturi}.\texttt{parameters} := \mathit{redirecturi}.\texttt{parameters} +^{\langle\rangle} \langle \texttt{code}, \nu_1\rangle$
      let $\mathit{redirecturi}.\texttt{parameters} := \mathit{redirecturi}.\texttt{parameters} +^{\langle\rangle} \langle \texttt{state}, \mathit{body}[\texttt{state}]\rangle$
      let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 303, \langle\langle \texttt{Location}, \mathit{redirecturi}\rangle\rangle, \langle\rangle\rangle, k)$
      stop $\langle\langle f, a, m'\rangle\rangle$, s'
    else ▷ Assume response type token.
      let $s'.\texttt{atokens} := s'.\texttt{atokens} +^{\langle\rangle} \langle \nu_1, \mathit{clientid}, \mathit{username}\rangle$
      let $\mathit{redirecturi}.\texttt{fragment} := \mathit{redirecturi}.\texttt{fragment} +^{\langle\rangle} \langle \texttt{access\_token}, \nu_1\rangle$
      let $\mathit{redirecturi}.\texttt{fragment} := \mathit{redirecturi}.\texttt{fragment} +^{\langle\rangle} \langle \texttt{token\_type}, \texttt{bearer}\rangle$
      let $\mathit{redirecturi}.\texttt{fragment} := \mathit{redirecturi}.\texttt{fragment} +^{\langle\rangle} \langle \texttt{state}, \mathit{body}[\texttt{state}]\rangle$
      let $m' := \mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 303, \langle\langle \texttt{Location}, \mathit{redirecturi}\rangle\rangle, \langle\rangle\rangle, k)$

```
49:              stop ⟨⟨f, a, m′⟩⟩, s′
50:          end if
51:      end if
52:  else if path ≡ s.tokenEndpoint then                                  ▷ Token Endpoint.
53:      if method ≢ POST then
54:          stop ⟨⟩, s
55:      end if
56:      let auth := ⊥
57:      let clientid := ⊥
58:      if body[client_id] ≢ ⟨⟩ then                  ▷ Only client ID is provided, no password.
59:          let clientid := body[client_id]
60:          let clientinfo := s.clients[clientid]
61:          if clientinfo ≡ ⟨⟩ ∨ secretOfClientID(clientid, i) ≢ ⊥ then      ▷ Empty client secret allowed?
62:              stop ⟨⟩, s
63:          end if
64:      else if headers[Authorization].1 ≢ ⟨⟩ then
65:          let clientid := headers[Authorization].1
66:          let clientpw := headers[Authorization].2
67:          if secretOfClientID(clientid, i) ≢ clientpw ∨ clientpw ≡ ⊥ then
68:              stop ⟨⟩, s
69:          end if
70:          let auth := clientid                  ▷ Authentication with client credentials.
71:      end if
72:      if body[grant_type] ≡ authorization_code then
73:          if clientid ≡ ⊥ then
74:              stop ⟨⟩, s
75:          end if
76:          let codeinfo := s.codes[body[code]]
77:          if codeinfo ≡ ⟨⟩ ∨ codeinfo.1 ≢ clientid ∨ codeinfo.2 ≢ body[redirect_uri] then
78:              stop ⟨⟩, s
79:          end if
80:          let s′.atokens := s′.atokens +^⟨⟩ ⟨ν1, clientid, codeinfo.3⟩ ▷ Add nonce, client ID and user ID to list
     of tokens.
81:          let m′ := enc_s(⟨HTTPResp, n, 200, ⟨⟩, ⟨⟨access_token, ν1⟩, ⟨token_type, bearer⟩⟩⟩, k)
82:      else if body[grant_type] ≡ password then
83:          let username := body[username]
84:          let password := body[password]
85:          if password ≢ secretOfID(username) then
86:              stop ⟨⟩, s
87:          end if
88:          let s′.atokens := s′.atokens +^⟨⟩ ⟨ν1, clientid, username⟩
89:          let m′ := enc_s(⟨HTTPResp, n, 200, ⟨⟩, ⟨⟨access_token, ν1⟩, ⟨token_type, bearer⟩⟩⟩, k)
90:      else if body[grant_type] ≡ client_credentials then
91:          if auth ≡ ⊥ then
92:              stop ⟨⟩, s
93:          end if
94:          let s′.atokens := s′.atokens +^⟨⟩ ⟨ν1, clientid, ⊥⟩
95:          let m′ := enc_s(⟨HTTPResp, n, 200, ⟨⟩, ⟨⟨access_token, ν1⟩, ⟨token_type, bearer⟩⟩⟩, k)
96:      end if
97:  else if path ≡ s.inspectEndpoint then                                  ▷ Inspection Endpoint.
98:      if method ≢ GET then
99:          stop ⟨⟩, s
100:     end if
101:     let atoken := parameters[token]
```

102: **let** *clientid*, *userid* **such that** $\langle atoken, clientid, userid\rangle \in^{\langle\rangle} s'.\texttt{atokens}$ **if possible; otherwise stop** $\langle\rangle, s$
103: **let** *secret* := $\mathsf{resourceOf}(i, clientid, \bot)$
104: **let** *body*′ := $\langle\langle \texttt{secret\_resource}, secret\rangle, \langle \texttt{client\_id}, clientid\rangle, \langle \texttt{user}, userid\rangle\rangle$
105: **let** m' := $\mathsf{enc_s}(\langle \texttt{HTTPResp}, n, 200, \langle\rangle, body'\rangle, k)$
106: **end if**
107: **stop** $\langle\rangle, s$

Algorithm 12 Relation of *script_idp_form*

Input: $\langle$*tree, docnonce, scriptstate, scriptinputs, cookies, localStorage, sessionStorage, ids, secrets*$\rangle$
1: **let** *url* := $\mathsf{GETURL}(tree, docnonce)$
2: **let** *url*.path $\leftarrow \mathbb{S}$
3: **let** *formdata* := *scriptstate*
4: **let** *id* $\leftarrow$ *ids*
5: **let** *secret* $\leftarrow$ *secrets*
6: **let** *formdata* := *formdata* $+^{\langle\rangle}$ $\langle \texttt{username}, id\rangle$
7: **let** *formdata* := *formdata* $+^{\langle\rangle}$ $\langle \texttt{password}, secret\rangle$
8: **let** *command* := $\langle \texttt{FORM}, url, \texttt{POST}, formdata, \bot\rangle$
9: **stop** $\langle s, cookies, localStorage, sessionStorage, command\rangle$

F. Formal Security Properties

The security properties for OAuth are formally defined as follows.

Definition 44 (Authorization Property). Let $\mathcal{OWS}$ be an OAuth web system. We say that $\mathcal{OWS}$ *is secure w.r.t. authorization* if for every run ρ of $\mathcal{OWS}$, every state (S^j, E^j, N^j) in ρ, every IdP $i \in \mathsf{IDP}$, every RP $r \in \mathsf{RP}$ that is honest in S^j, every $u \in \mathsf{ID} \cup \{\bot\}$, for $n = \mathsf{resourceOf}(i, r, u)$, n is derivable from the attackers knowledge in S^j (i.e., $n \in d_\emptyset(S^j(\mathsf{attacker}))$), it follows that

1. i is corrupted in S^j, or
2. $u \neq \bot$ and (i) the browser b owning u is fully corrupted in S^j or (ii) some $r' \in \mathsf{trustedRPs}(\mathsf{secretOfID}(u))$ is corrupted in S^j.

Definition 45 (Authentication Property). Let $\mathcal{OWS}$ be an OAuth web system. We say that $\mathcal{OWS}$ *is secure w.r.t. authentication* if for every run ρ of $\mathcal{OWS}$, every state (S^j, E^j, N^j) in ρ, every $r \in \mathsf{RP}$ that is honest in S^j, every $i \in \mathsf{IDP}$, every $g \in \mathsf{dom}(i)$, every $u \in \mathsf{governor}^{-1}(i)$, every RP service token of the form $\langle n, \langle u, g\rangle\rangle$ recorded in $S^j(r).\texttt{serviceTokens}$, and n being derivable from the attackers knowledge in S^j (i.e., $n \in d_\emptyset(S^j(\mathsf{attacker}))$), then the browser b owning u is fully corrupted in S^j (i.e., the value of *isCorrupted* is FULLCORRUPT), some $r' \in \mathsf{trustedRPs}(\mathsf{secretOfID}(u))$ is corrupted in S^j, or i is corrupted in S^j.

G. Proof of Theorem 1

Before we prove Theorem 1, we show some general properties of the $\mathcal{OWS}$. We then first prove the authentication property and then the authorization property.

G.1. Properties of $\mathcal{OWS}$

Let $\mathcal{OWS} = (\mathcal{W}, \mathcal{S}, \mathsf{script}, E^0)$ be a web system. Let ρ be a run of $\mathcal{OWS}$. We write $s_x = (S^x, E^x, N^x)$ for the states in ρ.

Definition 46. In what follows, given an atomic process p and a message m, we say that *p emits m* in a run $\rho = (s_0, s_1, \ldots)$ if there is a processing step of the form

$$s_{u-1} \xrightarrow[p \to E]{} s_u$$

for some $u \in \mathbb{N}$, a set of events E and some addresses x, y with $\langle x, y, m \rangle \in E$.

Definition 47. We say that a term t *is derivably contained in (a term) t' for (a set of DY processes) P (in a processing step $s_i \to s_{i+1}$ of a run $\rho = (s_0, s_1, \ldots)$)* if t is derivable from t' with the knowledge available to P, i.e.,

$$t \in d_\emptyset(\{t'\} \cup \bigcup_{p \in P} S^{i+1}(p))$$

Definition 48. We say that *a set of processes P leaks a term t (in a processing step $s_i \to s_{i+1}$) to a set of processes P'* if there exists a message m that is emitted (in $s_i \to s_{i+1}$) by some $p \in P$ and t is derivably contained in m for P' in the processing step $s_i \to s_{i+1}$. If we omit P', we define $P' := \mathcal{W} \setminus P$. If P is a set with a single element, we omit the set notation.

Definition 49. We say that an DY process p *created* a message m (at some point) in a run if m is derivably contained in a message emitted by p in some processing step and if there is no earlier processing step where m is derivably contained in a message emitted by some DY process p'.

Definition 50. We say that *a browser b accepted* a message (as a response to some request) if the browser decrypted the message (if it was an HTTPS message) and called the function PROCESSRESPONSE, passing the message and the request (see Algorithm 6).

Definition 51. We say that an atomic DY process p *knows a term t* in some state $s = (S, E, N)$ of a run if it can derive the term from its knowledge, i.e., $t \in d_\emptyset(S(p))$.

Definition 52. We say that a *script initiated a request r* if a browser triggered the script (in Line 10 of Algorithm 5) and the first component of the *command* output of the script relation is either `HREF`, `IFRAME`, `FORM`, or `XMLHTTPREQUEST` such that the browser issues the request r in the same step as a result.

The following lemma captures properties of RP when it uses HTTPS. For example, the lemma says that other parties cannot decrypt messages encrypted by RP.

Lemma 1 (RP messages are protected by HTTPS). If in the processing step $s_i \to s_{i+1}$ of a run ρ of $\mathcal{OWS}$ an honest relying party r (I) emits an HTTPS request of the form

$$m = \mathsf{enc_a}(\langle req, k \rangle, \mathsf{pub}(k'))$$

(where *req* is an HTTP request, k is a nonce (symmetric key), and k' is the private key of some other DY process u), and (II) in the initial state s_0 the private key k' is only known to u, and (III) u never leaks k', then all of the following statements are true:

1. There is no state of $\mathcal{OWS}$ where any party except for u knows k', thus no one except for u can decrypt *req*.
2. If there is a processing step $s_j \to s_{j+1}$ where the RP r leaks k to $\mathcal{W} \setminus \{u, r\}$ there is a processing step $s_h \to s_{h+1}$ with $h < j$ where u leaks the symmetric key k to $\mathcal{W} \setminus \{u, r\}$ or r is corrupted in s_j.

3. The value of the host header in *req* is the domain that is assigned the public key $\mathtt{pub}(k')$ in RP's keymapping $s_0.\mathtt{keyMapping}$ (in its initial state).

4. If r accepts a response (say, m') to m in a processing step $s_j \to s_{j+1}$ and r is honest in s_j and u did not leak the symmetric key k to $\mathcal{W} \setminus \{u,r\}$ prior to s_j, then either u or r created the HTTPS response m' to the HTTPS request m, in particular, the nonce of the HTTP request *req* is not known to any atomic process p, except for the atomic DY processes r and u.

PROOF. **(1)** follows immediately from the condition. If k' is initially only known to u and u never leaks k', i.e., even with the knowledge of all nonces (except for those of u), k' can never be derived from any network output of u, k' cannot be known to any other party. Thus, nobody except for u can derive *req* from m.

(2) We assume that r leaks k to $\mathcal{W} \setminus \{u,r\}$ in the processing step $s_j \to s_{j+1}$ without u prior leaking the key k to anyone except for u and r and that the RP is not fully corrupted in s_j, and lead this to a contradiction.

The RP is honest in s_i. From the definition of the RP, we see that the key k is always a fresh nonce that is not used anywhere else. Further, the key is stored in *pendingRequests* (ν_4 in Lines 47f. of Algorithm 8). The information from *pendingRequests* is not extracted or used anywhere else, except when handling the received messages, where the key is only checked against and used to decrypt the message (Lines 7ff. of Algorithm 8). Hence, r does not leak k to any other party in s_j (except for u and r). This proves (2).

(3) Per the definition of RPs (Algorithm 8), a host header is always contained in HTTP requests by RPs. From Line 48 of Algorithm 8 we can see that the encryption key for the request *req* was chosen using the host header of the message. It is chosen from the *keyMapping* in RP's state, which is never changed during ρ. This proves (3).

(4) An HTTPS response m' that is accepted by r as a response to m has to be encrypted with k. The nonce k is stored by the RP in the *pendingRequests* state information (see Line 47 of Algorithm 8). The RP only stores freshly chosen nonces there (i.e., the nonces are not used twice, or for other purposes than sending one specific request). The information cannot be altered afterwards (only deleted) and cannot be read except when the RP checks incoming messages. The nonce k is only known to u (which did not leak it to any other party prior to s_j) and r (which did not leak it either, as u did not leak it and r is honest, see (2)). This proves (4). ∎

On a high level, the following lemma shows that the contents in the list of pending HTTP requests are immutable.

Lemma 2 (Pending DNS messages become pending requests). Let r be some honest relying party in $\mathcal{OWS}$, $\nu \in \mathcal{N}$, $l > 0$ such that (S^l, E^l, N^l) is a state in ρ, and let $ref \in \mathcal{T}_{\mathcal{N}}$, $req \in \mathsf{HTTPRequests}$ such that $S^l(r).\mathtt{pendingDNS} \equiv S^{l-1}(r).\mathtt{pendingDNS} +^{\langle\rangle} \langle \nu, \langle ref, req \rangle \rangle$. Then we have that $\forall l'$: if there exist $ref', req', x, y \in \mathcal{T}_{\mathcal{N}}$ with $req.\mathtt{nonce} \equiv req'.\mathtt{nonce}$ and $\langle ref', req', x, y \rangle \in^{\langle\rangle} S^{l'}(r).\mathtt{pendingRequests}$ then $req \equiv req' \wedge ref \equiv ref'$.

PROOF. We first note that Algorithm 8 (of relying parties) modifies the subterm *pendingDNS* of the RP's state only in such a way that entries are appended to or removed from this subterm, but never modified. Entries are appended in Lines 23, 60, 116, 134, and 146. At all these places in the algorithm, an HTTP message term, say *req*, having a fresh (HTTP) nonce, is appended (together with some term *ref*) to the subterm *pendingDNS*. (A processing step executing one of these parts of the algorithm results in the state (S^l, E^l, N^l) of ρ.) Entries are only removed in Line 49. In this part of the algorithm, a sequence $\langle ref'', req'', x, y \rangle$ with $x, y \in \mathcal{T}_{\mathcal{N}}$ and $req'' \equiv req$ and $ref'' \equiv ref$ (which could not have been altered in any processing step) are appended to the subterm *pendingRequests* of RP's state (in Line 47). Besides

Line 12, where some entry is removed from this subterm, there is no other part of the algorithm that alters *pendingRequests* in any way. Hence, there we cannot have any state $(S^{l'}, E^{l'}, N^{l'})$ of ρ where we have an request in *pendingRequests* with the same (HTTP) nonce but a different req' or a different ref'. ■

Lemma 3 (RPs never send requests to themselves). An honest RP never sends an HTTP request to any RP (including itself), and only sends HTTPS requests to RPs that the receiving RP cannot decrypt.

PROOF. Honest RPs send HTTP requests only in Lines 22, 59, 115, 133, and 145. In all of these cases, they send the HTTPS request to an endpoint configured in the state (in `idps`). With Definition 42, it follows that the domains to which these requests are sent, are never a domain of an RP. All requests are sent over HTTPS, and the "correct" encryption keys (as stored in `keyMapping`) are used (i.e., even if the attacker changes the DNS response such that an HTTPS request is sent to an RP, it cannot be decrypted by the RP). ■

G.2. Proof of Authentication

We here want to show that every OAuth web system is secure w.r.t. authentication, and therefore assume that there exists an OAuth web system that is not secure w.r.t. authentication. We then lead this to a contradiction, thereby showing that all OAuth web systems are secure w.r.t. authentication. In detail, we assume:

Assumption 1. There exists an OAuth web system $\mathcal{OWS}$, a run ρ of $\mathcal{OWS}$, a state (S^j, E^j, N^j) in ρ, some $r \in \mathsf{RP}$ that is honest in S^j, some $i \in \mathsf{IDP}$ that is honest in S^j, some $g \in \mathsf{dom}(i)$, some $u \in \mathsf{governor}^{-1}(i)$ with the browser b owning u being not fully corrupted in S^j and all $r' \in \mathsf{trustedRPs}(\mathsf{secretOfID}(u))$ being honest, some RP service token of the form $\langle n, \langle u, g \rangle\rangle$ recorded in $S^j(r).\mathtt{serviceTokens}$ such that n is derivable from the attackers knowledge in S^j (i.e., $n \in d_\emptyset(S^j(\mathsf{attacker}))$).

To show that this is a contradiction, we first show some lemmas:

Lemma 4 (Attacker does not learn passwords). There exists no $l \leq j$, (S^l, E^l, N^l) being a state in ρ such that $\mathsf{secretOfID}(u) \in d_\emptyset(S^l(\mathsf{attacker}))$.

PROOF. Let $s := \mathsf{secretOfID}(u)$ and $R := \mathsf{trustedRPs}(s)$. Initially, in S^0, s is only contained in $S^0(b).\mathtt{secrets}[\langle d, \mathtt{S}\rangle]$ for any $d \in \bigcup_{r' \in R} \mathsf{dom}(r') \cup \mathsf{dom}(i)$ and in no other states (or waiting events). By the definition of the browser, we can see that only scripts loaded from the origins $\langle d, \mathtt{S}\rangle$ can access s. We know that i and all $r' \in R$ are honest (from the assumption). We therefore have that only the scripts *script_rp_index*, *script_rp_implicit*, and *script_idp_form* can access s (if loaded from their respective origins) and that the browser does not use or leak s in any other way. *script_rp_implicit* does not use any browser secrets. We therefore focus on the remaining two scripts:

script_rp_index. If this script was loaded and has access to s, it must have been loaded from origin $\langle d, \mathtt{S}\rangle$ for a domain d of some trusted relying party, say t ($\in R$). If *script_rp_index* selects the secret s in Line 9 of Algorithm 9, we know that it must have selected the id u in Line 2. We therefore know that in Line 10, the browser b is instructed to send (using HTTPS) $\langle u, s\rangle$ to the path `/passwordLogin` at d. If b sends such a request, t is the only party able to decrypt this request (see the general security properties in [9]). This message is then processed by t according to Lines 123ff. There, username and password are forwarded to some IdP, say i', using an HTTPS POST request. More precisely, this request is sent to the domain of the token endpoint URL contained in the IdP registration record for the domain contained in u. From Definitions 41 and 42 and the fact that this part of the state (of relying parties) is never changed, we can see that the

request is sent to a domain of i, and therefore $i' = i$. (The attacker can also not modify or read this request, see Lemma 1.) The body of the HTTPS POST request sent to i is of the following form:

$$\langle\langle \texttt{grant\_type}, \texttt{password}\rangle, \langle \texttt{username}, u\rangle, \langle \texttt{password}, s\rangle\rangle\,.$$

Such a request can processed by IdP only in Lines 82ff. of Algorithm 11. There, IdP checks s and discards it. Therefore, s does not leak from i, t, or b to the attacker (or any other party).

script_idp_form. If this script was loaded and has access to s, it must have been loaded from origin $\langle d, \mathtt{S}\rangle$ for a domain d of i. This script sends s to d in an HTTPS POST request. If b sends such a request, i is the only party able to decrypt this request (see the general security properties in [9]). This message is then processed by i according to Lines 15ff. of Algorithm 11. There, the IdP i checks s and discards it. Therefore, s does not leak from i or b to the attacker (or any other party).

This proves Lemma 4.

Lemma 5 (Attacker does not learn authorization codes). There exists no $l \leq j$, (S^l, E^l, N^l) being a state in ρ, $v \in \mathcal{N}$, $y \in \mathcal{T}_{\mathcal{N}}$ such that $v \in d_\emptyset(S^l(\mathsf{attacker}))$ and $\langle v, \langle \mathsf{clientIDOfRP}(r,i), y, u\rangle\rangle \in^{\langle\rangle} S^l(i).\texttt{codes}$.

PROOF. $S^l(i).\texttt{codes}$ is initially empty and appended to only in Line 38 of Algorithm 11 (where an authorization code is created). From Line 15ff. it is easy to see that the request which triggers the creation of the authorization code must carry a valid password for the specific identity in the request body. With Lemma 4, we can see that such a request can not come from the attacker, as the attacker does not know the password needed in the request. It can also not originate from an IdP, as IdPs do not send requests. Further, the request can not originate from any corrupted party or an attacker-controlled origin in the honest browser (as otherwise there would be a flow where the attacker would learn the password by sending it to himself, which can be ruled out by Lemma 4). It is also impossible that the request originated from any non-attacker controlled origin in the honest browser: Such a request could be caused by either a Location redirect or a script. (We will refer to the following as *.) A Location redirect must have been issued by an honest party (otherwise, the attacker would have learned the password by the time he issued the response, see Lemma 4). There are two occasions where honest parties issue Location redirect headers:

IdP in Lines 41/48 of Algorithm 11 In this case, an HTTP status code of 303 is sent. While this causes the browser to do a new request, the new request has an empty body in any case.[26]

RP in Line 89 of Algorithm 8 In this case, a 307 redirect could be issued, causing the browser to preserve the request body. We therefore have to check what could have caused the browser to issue a request that caused this Location redirect response, and what body could be contained in such a request. For clarity, we call the request causing the redirection m. It is clear that m cannot come from the attacker (as it contains the password). It must therefore come from an honest browser. If it was caused by a redirect in the honest browser, (*) applies recursively. Otherwise, there are three scripts that could send such a request to RP: *script_rp_index*, *script_rp_implicit*, and *script_idp_form*. Of these, only *script_rp_index* causes a request for the path `/startInteractiveLogin` (which triggers the redirection in Line 89 of Algorithm 8), which, however, does not contain any secret.

[26] Note that at this point it is important that a 303 redirect is performed, not a 307 redirect. See Line 29 of Algorithm 6 for details.

A Location redirect can therefore be ruled out as the cause of the request. There are three scripts that could send such a request: *script_rp_index*, *script_rp_implicit*, and *script_idp_form*. The first two, *script_rp_index*, *script_rp_implicit*, do not send requests to any IdP (instead, they only send requests to the RP that sent the scripts to the browser, IdP does not send these scripts to the browser). The latter script, *script_idp_form*, can send the request. In this (last remaining) case, the IdP responds with a Location redirect header in the response, which, among others, carries a URL containing the critical value v (in Line 41). In this case, the browser receives the response, and immediately triggers a new request to the redirection URL. This URL was composed by the IdP using the list of valid redirection URIs from $S^l(i).\texttt{clients}$, a part of the state of i that is not changed during any run. Definition 43 defines how $S^l(i).\texttt{clients}$ is initialized: For the client id $c := \mathsf{clientIDOfRP}(r,i)$, all redirection URLs carry hosts (domains) of r, have the protocol `S` (HTTPS), and contain a query parameter component identifying the IdP i. In the checks in Lines 22ff., it is ensured that in any case, this restriction on domain and protocol applies to the resulting redirection URI (called *redirecturi* in the algorithm) as well. Therefore, the browser's GET request which is triggered by the Location header and contains the value v is sent to r over HTTPS.

The RP r can process such a GET request only in Lines 65 and 91 of Algorithm 8. It is clear, that in Line 65, the value v does not leak to the attacker: An honest script is loaded into the browser, which does not use v in any form.

In Lines 91ff., v is forwarded to the IdP for checking its validity and retrieving the access token (there is also code for retrieving the access code from the implicit flow in this part of the code, which is not of interest here). When sending the authorization code, it is critical to ensure that v is forwarded to an honest IdP (in particular, i), and not to the attacker. This is ensured by checking the redirection URL parameters, which, as mentioned above, contain a hint for the IdP in use, in this case i. In Line 94 it is checked that the IdP, to which v is eventually sent, is i.

Therefore, we know that v is sent via POST to the honest IdP i. There, it can only be processed in Lines 52ff. Here, it is easy to see that the value v (called $body[\texttt{code}]$ in the algorithm) is checked. However, the value is never sent out to any other party and therefore does not leak.

We have shown that the value v cannot be known to the attacker, which proves Lemma 5. ∎

Lemma 6 (Attacker does not learn access tokens). There exists no $l \leq j$, (S^l, E^l, N^l) being a state in ρ, $v \in \mathcal{N}$, such that $v \in d_\emptyset(S^l(\mathsf{attacker}))$ and $\langle v, \mathsf{clientIDOfRP}(r,i), u\rangle \in^{\langle\rangle} S^l(i).\texttt{atokens}$.

PROOF. Initially, we have $S^0(i).\texttt{atokens} \equiv \langle\rangle$. $S^l(i).\texttt{atokens}$ is appended to only in Lines 44, 80, 88, and 94 (where in each an access token is issued) of Algorithm 11 and not altered in any other way.

In Line 94, a term of the form $\langle *, *, \bot\rangle$ is appended, which is not of the form $\langle v, \mathsf{clientIDOfRP}(r,i), u\rangle$. In what follows, we will distinguish between the lines of Algorithm 11 were $\langle v, \mathsf{clientIDOfRP}(r,i), u\rangle$ is created:

Line 44. It is easy to see, that i must have received an HTTPS POST request containing an Origin header with one of its HTTPS origins and containing (in its body) a dictionary with the entries $\langle \texttt{username}, u\rangle$, $\langle \texttt{password}, \mathsf{secretOfID}(u)\rangle$, and $\langle \texttt{client\_id}, \mathsf{clientIDOfRP}(r,i)\rangle$. From Lemma 4 it follows that such a request cannot be assembled by the attacker. Also, neither an IdP nor an RP sends such a request. Hence, this request must have be sent from a browser. In the browser, only the scripts `script_idp_form` and the attacker script R^{att} can instruct the browser to send such a request. From Lemma 4 we know that the attacker script cannot access $\mathsf{secretOfID}(u)$ (otherwise, there would be a run ρ' in which the attacker script would send $\mathsf{secretOfID}(u)$ to the attacker instead). Hence, this request must originate from a command returned by `script_idp_form` and it must be created by the browser b (which is $\mathsf{ownerOfID}(u)$). This script only sends such a request to its own origin, which must be an HTTPS origin (it would

not have access to $\mathsf{secretOfID}(u)$ otherwise). The IdP responds with a Location redirect header in the response, which among others, carries a URL containing the critical value v (in Line 48) in the fragment of the URL. In this case, the browser receives the response, and immediately triggers a new request to the redirection URL. This URL was composed by the IdP using the list of valid redirection URIs from $S^l(i).\texttt{clients}$, a part of the state of i that is not changed during any run. Definition 43 defines how $S^l(i).\texttt{clients}$ is initialized: For the client id $c := \mathsf{clientIDOfRP}(r,i)$, all redirection URLs carry hosts (domains) of r, have the protocol `S` (HTTPS), and contain a query parameter component identifying the IdP i. In the checks in Lines 22ff., it is ensured that in any case, this restriction on domain and protocol applies to the resulting redirection URI (called *redirecturi* in the algorithm) as well. Therefore, the browser's GET request which is triggered by the Location header and contains the value v in the fragment, is sent to r over HTTPS.

The RP r can process such a GET request only in Lines 65 and 91 of Algorithm 8. It is clear, that in Line 65, the value v does not leak to the attacker: The honest script *script_rp_index* is loaded into the browser, which does not use v in any form.

In Lines 91ff., RP's algorithm branches into two different flows: (1) RP takes some value from the URL parameters (which do not contain v) and sends it to some process. RP defers its response to the browser and will (later) only send out the response in Lines 36ff. This response, however, does not contain a script and hence, the browser will not be instructed to create any new messages from the resulting document. Hence, v does not leak in this case. (2) RP sends an HTTPS response containing the script *script_rp_implicit* (and, in the script's initial state, a domain of i derived from the redirection URL), which takes v from the URL parameters and instructs the browser to send an HTTPS POST request containing v and the domain of i to the script's (secure) origin at path `/receiveTokenFromImplicitGrant`. RP processes such a request in Lines 136ff. where it forwards v to the IdP for checking its validity. Here, it is critical to ensure that v is forwarded to an honest IdP (in particular, i), and not to the attacker. This is fulfilled since a domain of i is contained in the request's body, and, before forwarding, it is checked that v is only forwarded to this domain.

Therefore, we know that v is sent via GET to the honest IdP i. There, it can only be processed in Lines 97ff. Here, it is easy to see that the value v is never sent out to any other party and therefore does not leak.

Line 80. In this case, i must have received an HTTPS POST request carrying a dictionary in its body containing the entries $\langle \texttt{grant\_type}, \texttt{authorization\_code} \rangle$ and $\langle \texttt{code}, \mathit{code} \rangle$ with $\mathit{code} \in \mathcal{N}$ such that $\langle \mathit{code}, \langle \mathsf{clientIDOfRP}(r,i), y, u \rangle \rangle \in^{\langle\rangle} S^{l'}(i).\texttt{codes}$ for some $y \in \mathcal{T}_{\mathcal{N}}$ and $l' \leq l$. From Lemma 5 it follows that such a request can neither be constructed by the attacker nor by a browser instructed by the attacker script R^{att}. In a browser, the remaining honest scripts do not instruct the browser to send such a request. (Honest) IdPs do not send such requests. Hence, such a request must have been constructed by an (honest) RP. An RP prepares such a request only in Lines 103ff. (of Algorithm 8) and finally sends out this request in Line 50 (after a DNS response). With Lemma 2 and Lemma 1 we know that *reference* contains a term of the form $\langle \texttt{code}, \mathit{idp}, *, *, *, * \rangle$ with $\mathit{idp} \in \mathsf{dom}(i)$ (as the request was sent encrypted for and to i). When RP receives the response from i, RP processes this response in Lines 7ff. where RP distinguishes between two cases based on the first subterm in *reference*. As we know that this subterm is `code`, we have that the response is processed only in Lines 15ff. RP takes a subterm from the response's body which might contain[27] v in Line 16 and prepares an HTTPS POST request to an URL of i (which is taken from the subterm *idps* of RP's state and this subterm is never altered and initially configured such that

[27] The subterm actually is v.

the URLs under the dictionary key *idp* are actually belonging to i). This HTTPS POST request contains v in the parameter `token`. This request is finally sent out this request in Line 50 (after a DNS response) encrypted for and to i.

It is now easy to see that i only accepts the request only in Lines 97ff. (of Algorithm 11). There, the IdP only checks the parameter `token` against its state and discards it afterwards. Hence, v does not leak.

Line 88. In this case, i must have received an HTTPS POST request carrying a dictionary in its body containing the entries $\langle \texttt{grant\_type}, \texttt{password} \rangle$, $\langle \texttt{username}, u \rangle$, and $\langle \texttt{password}, \mathsf{secretOfID}(u) \rangle$. From Lemma 4 it follows that such a request cannot be constructed by the attacker, dishonest scripts in browsers, or any other dishonest party. (Honest) IdPs do not construct such a request. All honest scripts do not instruct a browser to send such a request. Hence, the request must have been constructed by an honest RP. An RP prepares such a request only in Lines 126ff. (of Algorithm 8) and finally sends out this request in Line 50 (after a DNS response). With Lemma 2 and Lemma 1 we know that *reference* contains a term of the form $\langle \texttt{password}, idp, *, *, *, * \rangle$ with $idp \in \mathsf{dom}(i)$ (as the request was sent encrypted for and to i). When RP receives the response from i, RP processes this response in Lines 7ff. where RP distinguishes between two cases based on the first subterm in *reference*. As we know that this subterm is `code`, we have that the response is processed only in Lines 15ff. RP takes a subterm from the response's body which might contain[28] v in Line 16 and prepares an HTTPS POST request to an URL of i (which is taken from the subterm *idps* of RP's state and this subterm is never altered and initially configured such that the URLs under the dictionary key *idp* are actually belonging to i). This HTTPS POST request contains v in the parameter `token`. This request is finally sent out this request in Line 50 (after a DNS response) encrypted for and to i. It is now easy to see that i only accepts the request only in Lines 97ff. (of Algorithm 11). There, the IdP only checks the parameter `token` against its state and discards it afterwards. Hence, v does not leak.

We have shown that the value v cannot be known to the attacker, which proves Lemma 6. ∎

We can now show that Assumption 1 is a contradiction.

Lemma 7. Assumption 1 is a contradiction.

PROOF. The service token $\langle n, \langle u, g \rangle \rangle$ can only be created and added to the state $S^j(r).\texttt{serviceTokens}$ in Line 36 of Algorithm 8. To get to this point in the algorithm, in Line 26, it is checked that *reference* is a tupel of the form $\langle \texttt{inspect}, mode, g, a', f', n', k' \rangle$. This is taken from the pending requests, where the value is transferred to from the pending DNS subterm (see Lemma 2). Such a term (starting with `inspect`) is added to the `pendingDNS` subterm only in Lines 23 and 146. We can now do a case distinction between these two possibilities to identify the request m' to which the response containing the service token will be sent.

Subterm was added in Line 23. In this case, in Line 15, an entry of the form $\langle mode, g, a', f', n', k' \rangle$ must have existed as a reference in the pending HTTP requests, where *mode* is either `code` or `password`.[29] Such entries are created in the following lines:

Line 116. Here, a request m' must have been received which contained a valid authorization code for the identity u at the IdP i.[30] The attacker cannot know such an authorization code (see

[28] The subterm actually is v.

[29] If *mode* was `client_credentials`, no service token is created.

[30] Otherwise, the IdP would not have returned an access token for the identity u. As $g = idp$ is the value stored in the reference, it is also clear that the authorization code was, in fact, sent to i for retrieving the access token, and not to the attacker or another identity provider. Also, the request to i was sent over HTTPS, and therefore, Lemma 1 applies.

Lemma 5). The RP r does not send requests to itself or to other RPs (see Lemma 3), and no IdPs send requests. Therefore, m' must have originated from an honest browser.

Line 134. In this case, a request m' was received which contained a valid username and password combination for u at i. (As above, we know that i was used to verify that information as g is a domain of i, and $idp = g$.) Only the honest browser b and some relying parties know this password (see Lemma 4), but the RPs would not send such a request. The request m' was therefore sent from the browser b.

Subterm was added in Line 146. If the subterm $\langle \texttt{inspect}, mode, g, a', f', n', k' \rangle$ was added in this line, the request causing this (m') must have carried a valid access token for the identity u at i. (As above, the access token was sent to i for validation.) The attacker does not know such an access token (see Lemma 6), and other RPs or IdPs cannot send m'. Therefore, an honest browser must have sent m'.

We therefore have that in all cases, m' was sent by an honest browser. Further, m' must have been an HTTPS request (by the definition of RPs). If the request was sent as the result of an XMLHTTPRequest command from a script, that script must have been loaded from the origin $\langle g_r, \mathtt{S} \rangle$ with $g_r \in \mathsf{dom}(r)$. This is a contradiction (there are no honest scripts that use XMLHTTPRequest). Otherwise, it was a "regular" request. In this case, the browser tries to load the service token as a document (which will fail). In particular, the service token $\langle n, \langle u, g \rangle \rangle$ never leaks to the attacker.

We therefore know that the attacker cannot know the service token, which is a contradiction to the assumption. ∎

G.3. Proof of Authorization

As above, we assume that there exists an OAuth web system that is not secure w.r.t. authorization and lead this to a contradiction. Note that in the following, some of the lemmas shown above are used.

Assumption 2. There exists a run ρ of $\mathcal{OWS}$, a state (S^j, E^j, N^j) in ρ, some IdP $i \in \mathsf{IDP}$ that is honest in S^j, some RP $r \in \mathsf{RP}$ that is honest in S^j, some $u \in \mathsf{ID} \cup \{\bot\}$, some $n = \mathsf{resourceOf}(i, r, u)$, n being derivable from the attackers knowledge in S^j (i.e., $n \in d_\emptyset(S^j(\mathsf{attacker}))$), and $u = \bot$ or ((i) the browser b owning u is not fully corrupted in S^j and (ii) all $r' \in \mathsf{trustedRPs}(\mathsf{secretOfID}(u))$ are honest S^j).

Lemma 8 (Attacker does not learn RP secrets.). There exists no $l \leq j$, (S^l, E^l, N^l) being a state in ρ such that $\mathsf{secretOfRP}(r, i) \in d_\emptyset(S^l(\mathsf{attacker}))$ unless $\mathsf{secretOfRP}(r, i) \equiv \bot$.

PROOF. Following the definition of the initial states of all atomic processes (in particular Definition 42), initially, $\mathsf{secretOfRP}(r, i)$ is only known to r.

The secret is being used and sent out in an HTTPS message in Lines 52ff. of Algorithm 8 The message is being sent to the token endpoint configured for i, which, according to Definition 41, bears a host name belonging to i. With the definition of *sslkeys* in Definition 42 and Lemma 1 it can be seen that this outgoing HTTP POST request can therefore only be read by the intended receiver, i.

In i, the message cannot be processed in the authentication endpoint, Lines 15 to 51 of Algorithm 11, since it does not carry an Origin header. It can be processed in Lines 52 to 96. It is easy to see that the secret in the message is not used in any outgoing message, neither stored in the IdP's data structures. The message not be processed in Line 97ff., since it is a POST request.

The same applies when the client sends the password in Line 107ff. or Line 127ff. of Algorithm 8.

Therefore, the secret $\mathsf{secretOfRP}(r, i)$ cannot be known to the attacker. ∎

Lemma 9. Assumption 2 is a contradiction.

PROOF. At the beginning of each run, the attacker cannot know n (as defined in the initial states). Only the IdP i can send out the protected resource n, in Line 105 of Algorithm 11. In a state $(S^{l'}, E^{l'}, N^{l'})$ in ρ for some $l' < j$, for i to send out n, an HTTPS request must be received by i which contains, among others, an access token a such that $\langle a, \mathsf{clientIDOfRP}(r,i), u\rangle \in^{\langle\rangle} S^{l'}(i).\texttt{atokens}$. We therefore note that for the attacker to learn n, it has to know a. We also note that if r requests n at the IdP i, the attacker cannot read n or a from such messages (see Lemma 1).

We now have to distinguish two cases:

Anonymous Resource, i.e., $u \equiv \bot$. In this case, the access token a was chosen by i in Line 94 of Algorithm 11. There, a is sent out in response to a request that must have contained the client credentials for r, where the client secret cannot be $\bot$ (see Line 64. With Lemma 8 we see that the attacker cannot send such a request, and therefore, cannot learn a. This implies that the attacker cannot send the request to learn n from i.

User Resource, i.e., $u \not\equiv \bot$. In this Case, Lemma 6 shows that it is not possible for the attacker to send a request to learn n.

With this, we have shown that the attacker cannot learn n, and therfore, Assumption 2 is a contradication. ■