The Web SSO Standard *OpenID Connect*: In-Depth Formal Security Analysis and Security Guidelines

Daniel Fett, Ralf Küsters, and Guido Schmitz
University of Stuttgart, Germany
Email: {daniel.fett,ralf.kuesters,guido.schmitz}@sec.uni-stuttgart.de

***Abstract*—Web-based single sign-on (SSO) services such as *Google Sign-In* and *Log In with Paypal* are based on the *OpenID Connect* protocol. This protocol enables so-called relying parties to delegate user authentication to so-called identity providers. OpenID Connect is one of the newest and most widely deployed single sign-on protocols on the web. Despite its importance, it has not received much attention from security researchers so far, and in particular, has not undergone any rigorous security analysis.**

In this paper, we carry out the first in-depth security analysis of OpenID Connect. To this end, we use a comprehensive generic model of the web to develop a detailed formal model of OpenID Connect. Based on this model, we then precisely formalize and prove central security properties for OpenID Connect, including authentication, authorization, and session integrity properties.

In our modeling of OpenID Connect, we employ security measures in order to avoid attacks on OpenID Connect that have been discovered previously and new attack variants that we document for the first time in this paper. Based on these security measures, we propose security guidelines for implementors of OpenID Connect. Our formal analysis demonstrates that these guidelines are in fact effective and sufficient.

I. Introduction

OpenID Connect is a protocol for delegated authentication in the web: A user can log into a relying party (RP) by authenticating herself at a so-called identity provider (IdP). For example, a user may sign into the website tripadvisor.com using her Google account.

Although the names might suggest otherwise, OpenID Connect (or *OIDC* for short) is not based on the older OpenID protocol. Instead, it builds upon the OAuth 2.0 framework, which defines a protocol for delegated *authorization* (e.g., a user may grant a third party website access to her resources at Facebook). While OAuth 2.0 was not designed to provide *authentication*, it has often been used for this purpose as well, leading to several severe security flaws in the past [12], [43].

OIDC was created not only to retrofit authentication into OAuth 2.0 by using cryptographically secured tokens and a precisely defined method for user authentication, but also to enable additional important features. For example, using the *Discovery* extension, RPs can automatically identify the IdP that is responsible for a given identity. With the *Dynamic Client Registration* extension, RPs do not need a manual set-up process to work with a specific IdP, but can instead register themselves at the IdP on the fly.

Created by the OpenID Foundation and standardized only in November 2014, OIDC is already very widely used. Among others, it is used and supported by Google, Amazon, Paypal, Salesforce, Oracle, Microsoft, Symantec, Verizon, Deutsche Telekom, PingIdentity, RSA Security, VMWare, and IBM. Many corporate and end-user single sign-on solutions are based on OIDC, for example, well-known services such as *Google Sign-In* and *Log In with Paypal*.

Despite its wide use, OpenID Connect has not received much attention from security researchers so far (in contrast to OpenID and OAuth 2.0). In particular, there have been no formal analysis efforts for OpenID Connect until now. In fact, the only previous works on the security of OpenID Connect are a large-scale study of deployments of Google's implementation of OIDC performed by Li and Mitchell [34] and an informal evaluation by Mainka et al. [36].

In this work, we aim to fill the gap and formally verify the security of OpenID Connect.

Contributions of this Paper. We provide the first in-depth formal security analysis of OpenID Connect. Based on a comprehensive formal web model and strong attacker models, we analyze the security of all flows available in the OIDC standard, including many of the optional features of OIDC and the important Discovery and Dynamic Client Registration extensions. More specifically, our contributions are as follows.

a) Attacks on OIDC and Security Guidelines: We first compile an overview of attacks on OIDC, common pitfalls, and their respective mitigations. Most of these attacks were documented before, but we point out new attack variants and aspects.

Starting from these attacks and pitfalls, we then derive security guidelines for implementors of OIDC. Our guidelines are backed-up by our formal security analysis, showing that the mitigations that we propose are in fact effective and sufficient.

b) Formal model of OIDC: Our formal analysis of OIDC is based on the expressive Dolev-Yao style model of the web infrastructure (FKS model) proposed by Fett, Küsters, and Schmitz [19]. This web model is designed independently of a specific web application and closely mimics published (de-facto) standards and specifications for the web, for instance, the HTTP/1.1 and HTML5 standards and associated (proposed) standards. It is the most comprehensive web model to date. Among others, HTTP(S) requests and responses, including several headers, such as cookie, location, referer, authorization, 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 modern technologies, such as web storage, web messaging (via postMessage), and referrer policies. JavaScript is modeled in an abstract way by so-called *scripts* which can be sent around and, among others, can create iframes, access other windows, and initiate XMLHttpRequests. Browsers may be corrupted dynamically by the adversary.

The FKS model has already been used to analyze the security of the BrowserID single sign-on system [19], [20], the security and privacy of the SPRESSO SSO system [21], and the security of OAuth 2.0 [22], each time uncovering new and severe attacks that have been missed by previous analysis attempts.

Using the generic FKS model, we build a formal model of OIDC, closely following the standard. We employ the defenses and mitigations discussed earlier in order to create a model with state-of-the-art security features in place. Our model includes RPs and IdPs that (simultaneously) support all modes of OIDC and can be dynamically corrupted by the adversary.

c) Formalization of security properties: Based on this model of OIDC, we formalize four main security properties of OIDC: authentication, authorization, session integrity for authentication, and session integrity for authorization. We also formalize further OIDC specific properties.

d) Proof of Security for OpenID Connect: Using the model and the formalized security properties, we then show, by a manual yet detailed proof, that OIDC in fact satisfies the security properties. This is the first proof of security of OIDC. Being based on an expressive and comprehensive formal model of the web, including a strong attacker model, as well as on a modeling of OpenID Connect which closely follows the standard, our security analysis covers a wide range of attacks.

Structure of this Paper. We provide an informal description of OIDC in Section II. Attacks and security guidelines are discussed in Section III. In Section IV, we briefly recall the FKS model. The model and analysis of OIDC are then presented in Section V. Related work is discussed in Section VI. We conclude in Section VII. All details of our work, including the proofs, are provided in our technical report [23].

II. OpenID Connect

The OpenID Connect protocol allows users to authenticate to RPs using their existing account at an IdP.[1] (Typically, this is an email account at the IdP.) OIDC was defined by the OpenID Foundation in a *Core* document [40] and in extension documents (e.g., [39], [41]). Supporting technologies were standardized at the IETF, e.g., [30], [31]. (Recall that OpenID Connect is not to be confused with the older OpenID standards, which are very different to OpenID Connect.)

Central to OIDC is a cryptographically signed document, the *id token*. It is created by the user's IdP and serves as a one-time proof of the user's identity to the RP.

A high-level overview of OIDC is given in Figure 1. First, the user requests to be logged in at some RP and provides her email address $\boxed{\text{A}}$. RP now retrieves operational information (e.g., some URLs) for the remaining protocol flow (*discovery*, $\boxed{\text{B}}$) and registers itself at the IdP $\boxed{\text{C}}$. The user is then redirected to the IdP, where she authenticates herself $\boxed{\text{D}}$ (e.g., using a password). The IdP issues an id token to RP $\boxed{\text{E}}$, which RP can then verify to ensure itself of the user's identity. (The way of how the IdP sends the id token to the RP is subject to the different modes of OIDC, which are described in detail later in this section. In short, the id token is either relayed via the user's browser or it is fetched by the RP from the IdP directly.) The id token includes an identifier for the IdP (the *issuer*),[2] a user identifier (unique at the respective IdP), and is signed by the IdP. The RP uses the issuer identifier and the user identifier to determine the user's identity. Finally, the RP may set a session cookie in the user's browser which allows the user to access the services of RP $\boxed{\text{F}}$.

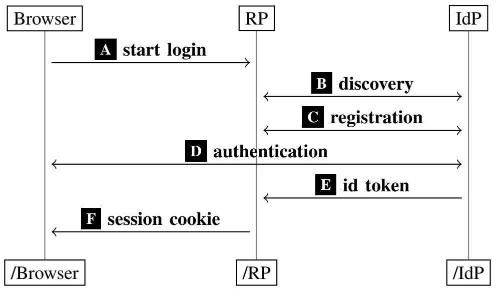

Figure 1. OpenID Connect — high level overview.

Before we explain the modes of operation of OIDC, we first present some basic concepts used in OIDC. At the end of this section, we discuss the relationship of OIDC to OAuth 2.0.

A. Basic Concepts

We have seen above that id tokens are essential to OIDC. Also, to allow users to use any IdP to authenticate to any RP, the RP needs to *discover* some information about the IdP. Additionally, the IdP and the RP need to establish some sort of relationship between each other. The process to establish such a relationship is called *registration*. Both, discovery and registration, can be either a manual task or a fully automatic process. Further, OIDC allows users to *authorize* an RP to access user's data at IdP on the user's behalf. All of these concepts are described in the following.

1) Authentication and ID Tokens: The goal of OIDC is to *authenticate* a user to an RP, i.e., the RP gets assured of the identity of the user interacting with the RP. This assurance is based on id tokens. As briefly mentioned before, an id token is a document signed by the IdP. It contains several *claims*, i.e., information about the user and further meta data. More precisely, an id token contains a user identifier (unique at the respective IdP) and the issuer identifier of the IdP. Both identifiers in combination serve as a global user identifier for authentication. Also, every id token contains an identifier for the RP at the IdP, which is assigned during registration (see below). The id token may also contain a nonce chosen by

[1]Note that the OIDC standard also uses the terms *client* for RP and *OpenID provider (OP)* for the IdP. We here use the more common terms RP and IdP.

[2]The issuer identifier of an IdP is an HTTPS URL without any query or fragment components.

the RP during the authentication flow as well as an expiration timestamp and a timestamp of the user's authentication at the IdP to prevent replay attacks. Further, an id token may contain information about the particular method of authentication and other claims, such as data about the user and a hash of some data sent outside of the id token.

When an RP validates an id token, it checks in particular whether the signature of the token is correct (we will explain below how RP obtains the public key of the IdP), the issuer identifier is the one of the currently used IdP, the id token is issued for this RP, the nonce is the one RP has chosen during this login flow, and the token has not expired yet. If the id token is valid, the RP trusts the claims contained in the id token and is confident in the user's identity.

2) Discovery and Registration: The OIDC protocol is heavily based on redirection of the user's browser: An RP redirects the user's browser to some IdP and vice-versa. Hence, both parties, the RP and the IdP, need some information about the respective URLs (so-called *endpoints*) pointing to each other. Also, the RP needs a public key of the IdP to verify the signature of id tokens. Further, an RP can contact the IdP directly to exchange protocol information. This exchange may include authentication of the RP at the IdP.

More specifically, an RP and an IdP need to exchange the following information: (1) a URL where the user can authenticate to the IdP (*authorization endpoint*), (2) one or more URLs at RP where the user's browser can be redirected to by the IdP after authentication (*redirection endpoint*), (3) a URL where the RP can contact the IdP in order to retrieve an id token (*token endpoint*), (4) the issuer identifier of the IdP, (5) the public key of the IdP to verify the id token's signature, (6) an identifier of the RP at IdP (*client id*), and optionally (7) a secret used by RP to authenticate itself to the token endpoint (*client secret*). (Recall that *client* is another term for RP, and in particular does not refer to the browser.)

This information can be exchanged manually by the administrator of the RP and the administrator of the IdP, but OIDC also allows one to completely automate the discovery of IdPs [41] and dynamically register RPs at an IdP [39].

During the automated discovery, the RP first determines which IdP is responsible for the email address provided by the user who wants to log in using the WebFinger protocol [31]. As a result, the RP learns the issuer identifier of the IdP and can retrieve the URLs of the authorization endpoint and the token endpoint from the IdP. Furthermore, the RP receives a URL where it can retrieve the public key to verify the signature of the id token (*JWKS URI*), and a URL where the RP can register itself at the IdP (*client registration endpoint*).

If the RP has not registered itself at this IdP before, it starts the registration ad-hoc at the client registration endpoint: The RP sends its redirection endpoint URLs to the IdP and receives a new client id and (optionally) a client secret in return.

3) Authorization and Access Tokens: OIDC allows users to authorize RPs to access the user's data stored at IdPs or act on the user's behalf at IdPs. For example, a photo printing service (the RP) might access or manage the user's photos on Google Drive (the IdP). For authorization, the RP receives a so-called *access token* (besides the id token). Access tokens follow the concept of so-called *bearer tokens*, i.e., they are used as the only authentication component in requests from an RP to an IdP. In our example, the photo printing service would have to add the access token to each HTTP request to Google Drive.

B. Modes

OIDC defines three modes: the *authorization code mode*, the *implicit mode*, and the *hybrid mode*. While in the authorization code mode, the id token is retrieved by an RP from an IdP in direct server-to-server communication (back channel), in the implicit mode, the id token is relayed from an IdP to an RP via the user's browser (front channel). The hybrid mode is a combination of both modes and allows id tokens to be exchanged via the front and the back channel at the same time.

We now provide a detailed description of all three modes.

1) Authorization Code Mode: In this mode, an RP redirects the user's browser to an IdP. At the IdP, the user authenticates and then the IdP issues a so-called *authorization code* to the RP. The RP now uses this code to obtain an id token from the IdP.

a) Step-by-Step Protocol Flow: The protocol flow is depicted in Figure 2. First, the user starts the login process by entering her email address[3] in her browser (at some web page of an RP), which sends the email address to the RP in $\boxed{1}$.

Now, the RP uses the OIDC discovery extension [41] to gather information about the IdP: As the first step (in this extension), the RP uses the WebFinger mechanism [31] to discover information about which IdP is responsible for this email address. For this discovery, the RP contacts the server of the email domain in $\boxed{2}$ (in the figure, the server of the user's email domain is depicted as the same party as the IdP). The result of the WebFinger request in $\boxed{3}$ contains the issuer identifier of the IdP (which is also a URL). With this information, the RP can continue the discovery by requesting the OIDC configuration from the IdP in $\boxed{4}$ and $\boxed{5}$. This configuration contains meta data about the IdP, including all endpoints at the IdP and a URL where the RP can retrieve the public key of the IdP (used to later verify the id token's signature). If the RP does not know this public key yet, the RP retrieves the key (Steps $\boxed{6}$ and $\boxed{7}$). This concludes the OIDC discovery in this login flow.

Next, if the RP is not registered at the IdP yet, the RP starts the OIDC dynamic client registration extension [39]: In Step $\boxed{8}$ the RP contacts the IdP and provides its redirect URIs. In return, the IdP issues a client id and (optionally) a client secret to the RP in Step $\boxed{9}$. This concludes the registration.

Now, the core part of the OIDC protocol starts: the RP redirects the user's browser to the IdP in $\boxed{10}$. This redirect contains the information that the authorization code mode is used. Also, this redirect contains the client id of the RP, a redirect URI, and a *state* value, which serves as a Cross-Site Request Forgery (CSRF) token when the browser is later

[3]Note that OIDC also allows other types of user ids, such as personal URLs.

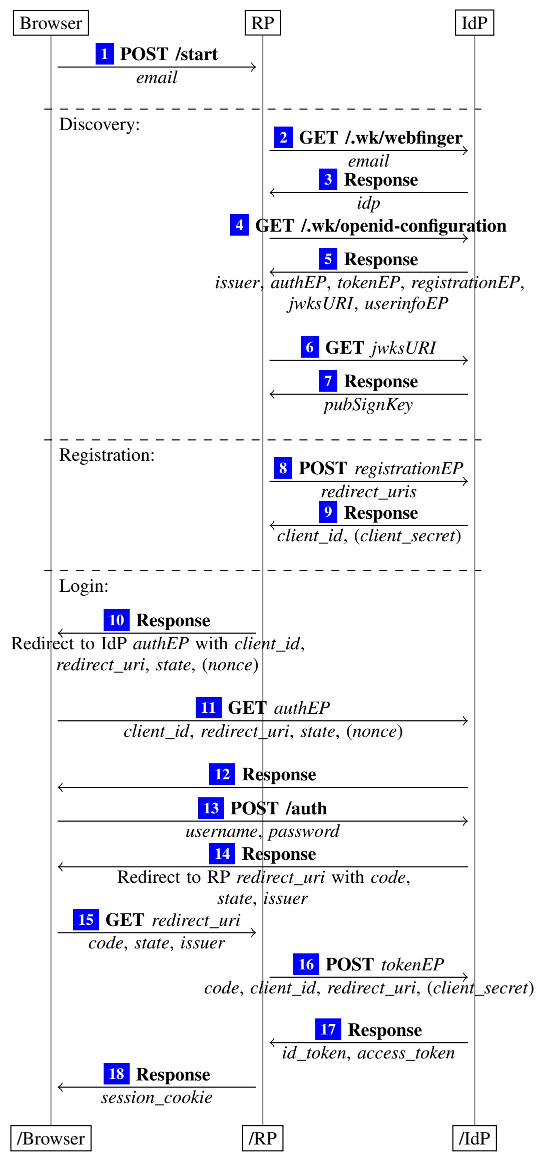

Figure 2. OpenID Connect authorization code mode. Note that data depicted below the arrows is either transferred in URI parameters, HTTP headers, or POST bodies.

redirected back to the RP. The redirect may also optionally include a nonce, which will be included in the id token issued later in this flow. This data is sent to the IdP by the browser $\boxed{11}$. The user authenticates to the IdP $\boxed{12}$, $\boxed{13}$, and the IdP redirects the user's browser back to the RP in $\boxed{14}$ and $\boxed{15}$ (using the redirect URI from the request in $\boxed{11}$). This redirect contains an authorization code, the *state* value as received in $\boxed{10}$, and the issuer identifier.[4] If the *state* value and the issuer identifier are correct, the RP contacts the IdP in $\boxed{16}$ at the token endpoint with the received authorization code, its client id, its client secret (if any), and the redirect URI used to obtain the authorization code. If these values are correct, the IdP responds with a fresh access token and an id token to the RP in $\boxed{17}$. If the id token is valid, then the RP considers the user to be logged in (under the identifier composed from the user id in the id token and the issuer identifier). Hence, the RP may set a session cookie at the user's browser in $\boxed{18}$.

2) Implicit Mode: The implicit mode (depicted in Figure 3 in Appendix A of our technical report [23]) is similar to the authorization code mode, but instead of providing an authorization code, the IdP issues an id token right away to the RP (via the user's browser) when the user authenticates to the IdP. Hence, the Steps $\boxed{1}$–$\boxed{13}$ of the authorization code mode (Figure 2) are the same. After these steps, the IdP redirects the user's browser to the redirection endpoint at the RP, providing an id token, (optionally) an access token, the *state* value, and the issuer identifier. These values are not provided as a URL parameter but in the URL fragment instead. Hence, the browser does not send them to the RP at first. Instead, the RP has to provide a JavaScript that retrieves these values from the fragment and sends them to the RP. If the id token is valid, the issuer is correct, and the *state* matches the one previously chosen by the RP, the RP considers the user to be logged in and issues a session cookie.

3) Hybrid Mode: The hybrid mode (depicted in Figure 4 in Appendix A of our technical report [23]) is a combination of the authorization code mode and the implicit mode: First, this mode works like the implicit mode, but when IdP redirects the browser back to RP, the IdP issues an authorization code, and either an id token or an access token or both.[5] The RP then retrieves these values as in the implicit mode (as they are sent in the fragment like in the implicit mode) and uses the authorization code to obtain a (potentially second) id token and a (potentially second) access token from IdP.

C. Relationship to OAuth 2.0

Technically, OIDC is derived from OAuth 2.0. It goes, however, far beyond what was specified in OAuth 2.0 and introduces many new concepts: OIDC defines a method for authentication (while retaining the option for authorization) using a new type of tokens, the id token. Some messages and tokens in OIDC can be cryptographically signed or encrypted while OAuth 2.0 does neither use signing nor encryption. The new hybrid flow combines features of the implicit mode and the authorization code mode. Importantly, with ad-hoc discovery and dynamic registration, OIDC standardizes and automates a process that is completely out of the scope of OAuth 2.0.

These new features and their interplay potentially introduce new security flaws. It is therefore not sufficient to analyze the security of OAuth 2.0 to derive any guarantees for OIDC. OIDC rather requires a new security analysis. (See Section V for a more detailed discussion. In Section III we describe attacks that cannot be applied to OAuth 2.0.)

[4]The issuer identifier will be included in this message in an upcoming revision of OIDC to mitigate the IdP Mix-Up attack, see Section III-A1.

[5]The choice of the IdP to issue either an id token or an access token or both depends on the IdP's configuration and the first request to *authEP* received by the IdP.

III. Attacks and Security Guidelines

In this section, we present a concise overview of known attacks on OIDC and present additions that have not been documented so far. We also summarize mitigations and implementation guidelines that have to be implemented to avoid these attacks.

The main focus of this work is to prove central security properties of OIDC, by which these mitigations and implementation guidelines are backed up. Moreover, further (potentially unknown types of) attacks on OIDC that can be captured by our security analysis are ruled out as well.

The rest of the section is structured as follows: we first present the attacks, mitigations and guidelines, then point out differences to OAuth 2.0, and finally conclude with a brief discussion.

A. Attacks, Mitigations, and Guidelines

(Mitigations and guidelines are presented along with every class of attack.)

1) IdP Mix-Up Attacks: In two previously reported attacks [22], [36], the aim was to confuse the RP about the identity of the IdP. In both attacks, the user was tricked into using an *honest* IdP to authenticate to an *honest* RP, while the RP is made to believe that the user authenticated to the attacker. The RP therefore, after successful user authentication, tries to use the authorization code or access token at the attacker, which then can impersonate the user or access the user's data at the IdP. We present a detailed description of an application of the IdP Mix-Up attack to OpenID Connect in Appendix A of our technical report [23].

The IETF OAuth Working Group drafted a proposal for a mitigation technique [29] that is based on a proposal in [22] and that also applies to OpenID Connect. The proposal is that the IdP puts its identity into the response from the authorization endpoint. (This is already included in our description of OIDC above, see the issuer in Step $\boxed{14}$ in Figure 2.) The RP can then check that the user authenticated to the expected IdP.

2) Attacks on the State Parameter: The *state* parameter is used in OIDC to protect against attacks on session integrity, i.e., attacks in which an attacker forces a user to be logged in at some RP (under the attacker's account). Such attacks can arise from session swapping or CSRF vulnerabilities.

OIDC recommends the use of the *state* parameter. It should contain a nonce that is bound to the user's session. Attacks that can result from omitting or incorrectly using *state* were described in the context of OAuth 2.0 in [8], [33], [35], [42].

The nonce for the *state* value should be chosen freshly for *each login attempt* to prevent an attack described in [22] (Section 5.1) where the same state value is used first in a user-initiated login flow with a malicious IdP and then in a login flow with an honest IdP (forcefully initiated by the attacker with the attacker's account and the user's browser).

3) Code/Token/State Leakage: Care should be taken that a value of *state* or an authorization code is not inadvertently sent to an untrusted third party through the *Referer* header. The *state* and the authorization code parameters are part of the redirection endpoint URI (at the RP), the *state* parameter is also part of the authorization endpoint URI (at the IdP). If, on either of these two pages, a user clicks on a link to an external page, or if one of these pages embeds external resources (images, scripts, etc.), then the third party will receive the full URI of the endpoint, including these parameters, in the Referer header that is automatically sent by the browser.

Documents delivered at the respective endpoints should therefore be vetted carefully for links to external pages and resources. In modern browsers, *referrer policies* [18] can be used to suppress the Referer header. As a second line of defense, both parameters should be made single-use, i.e., *state* should expire after it has been used at the redirection endpoint and authorization code after it has been redeemed at IdP.

In a related attack, an attacker that has access to log files or browsing histories (e.g., through malicious browser extensions) can steal authentication codes, access tokens, id tokens, or *state* values and re-use these to impersonate a user or to break session integrity. A subset of these attacks was dubbed *Cut-and-Paste Attacks* by the IETF OAuth working group [29].

There are drafts for RFCs that tackle specific aspects of these leakage attacks, e.g., [13] which discusses binding the *state* parameter to the browser instance, and [28] which discusses binding the access token to a TLS session. Since these mitigations are still very early IETF drafts, subject to change, and not easy to implement in the majority of the existing OIDC implementations, we did not model them.

In our analysis, we assume that implementations keep log files and browsing histories (of honest browsers) secret and employ referrer policies as described above.

4) Naïve RP Session Integrity Attack: So far, we have assumed that after Step $\boxed{10}$ (Figure 2), the RP remembers the user's choice (which IdP is used) in a session; more precisely, the user's choice is stored in RP's session data. This way, in Step $\boxed{15}$, the RP looks up the user's selected IdP in the session data. In [22], this is called *explicit user intention tracking*.

There is, however, an alternative to storing the IdP in the session. As pointed out by [22], some implementations put the identity of the IdP into the *redirect_uri* (cf. Step $\boxed{10}$), e.g., by appending it as the last part of the path or in a parameter. Then, in Step $\boxed{15}$, the RP can retrieve this information from the URI. This is called *naïve user intention tracking*.

RPs that use naïve user intention tracking are susceptible to the naïve RP session integrity attack described in [22]: An attacker obtains an authorization code, id token, or access token for his own account at an honest IdP (HIdP). He then waits for a user that wants to log in at some RP using the attacker's IdP (AIdP) such that AIdP obtains a valid *state* for this RP. AIdP then redirects the user to the redirection endpoint URI of RP using the identity of HIdP plus the obtained *state* value and code or (id) token. Since the RP cannot see that the user originally wanted to log in using AIdP instead of HIdP, the user will now be logged in under the attacker's identity.

Therefore, an RP should always use sessions to store the user's chosen IdP (explicit user intention tracking), which, as mentioned, is also what we do in our formal OIDC model.

5) 307 Redirect Attack: Although OIDC explicitly allows for any redirection method to be used for the redirection in Step $\boxed{14}$ of Figure 2, IdPs should not use an HTTP 307 status code for redirection. Otherwise, credentials entered by the user at an IdP will be repeated by the browser in the request to RP (Step $\boxed{15}$ of Figure 2), and hence, malicious RPs would learn these credentials and could impersonate the user at the IdP. This attack was presented in [22]. In our model, we exclusively use the 303 status code, which prevents re-sending of form data.

6) Injection Attacks: It is well known that Cross-Site Scripting (XSS) and SQL Injection attacks on RPs or IdPs can lead to theft of access tokens, id tokens, and authorization codes (see, for example, [8], [26], [35], [36], [42]). XSS attacks can, for example, give an attacker access to session ids. Besides using proper escaping (and Content Security Policies [44] as a second line of defense), OIDC endpoints should therefore be put on domains separate from other, potentially more vulnerable, web pages on IdPs and RPs.[6] (See *Third-Party Resources* below for another motivation for this separation.)

In OIDC implementations, data that can come from untrusted sources (e.g., client ids, user attributes, *state* and *nonce* values, redirection URIs) must be treated as such: For example, a malicious IdP might try to inject user attributes containing malicious JavaScript to the RP. If the RP displays this data without applying proper escaping, the JavaScript is executed.

We emphasize that in a similar manner, attackers can try to inject additional parameters into URIs by appending them to existing parameter values, e.g., the *state*. Since data is often passed around in OIDC, proper escaping of such parameters can be overlooked easily.

As a result of such parameter injection attacks or independently, parameter pollution attacks can be a threat for OIDC implementations. In these attacks, an attacker introduces duplicate parameters into URLs (see, e.g., [6]). For example, a simple parameter pollution attack could be launched as follows: A malicious RP could redirect a user to an honest IdP, using a client id of some honest RP but appending two redirection URI parameters, one pointing to the honest RP and one pointing to the attacker's RP. Now, if the IdP checks the first redirection URI parameter, but afterwards redirects to the URI in the second parameter, the attacker learns authentication data that belongs to the honest RP and can impersonate the user.

Mitigations against all these kinds of injection attacks are well known: implementations have to vet incoming data carefully, and properly escape any output data. In our model, we assume that these mitigations are implemented.

7) CSRF Attacks and Third-Party Login Initiation: Some endpoints need protection against CSRF in addition to the protection that the *state* parameter provides, e.g., by checking the origin header. Our analysis shows that the RP only needs to protect the URI on which the login flow is started (otherwise, an attacker could start a login flow using his own identity in a user's browser) and for the IdP to protect the URI where the user submits her credentials (otherwise, an attacker could submit his credentials instead).

In the OIDC Core standard [40], a so-called *login initiation endpoint* is described which allows a third party to start a login flow by redirecting a user to this endpoint, passing the identity of an IdP in the request. The RP will then start a login flow at the given IdP. Members of the OIDC foundation confirmed to us that this endpoint is essentially an intentional CSRF protection bypass. We therefore recommend login initiation endpoints not to be implemented (they are not a mandatory feature), or to require explicit confirmation by the user.

8) Server-Side Request Forgery (SSRF): SSRF attacks can arise when an attacker can instruct a server to send HTTP(S) requests to other hosts, causing unwanted side-effects or revealing information [38]. For example, if an attacker can instruct a server behind a firewall to send requests to other hosts behind this firewall, the attacker might be able to call services or to scan the internal network (using timing attacks). He might also instruct the server to retrieve very large documents from other sources, thereby creating Denial of Service attacks.

SSRF attacks on OIDC were described for the first time in [36], in the context of the OIDC Discovery extension: An attacker could set up a malicious discovery service that, when queried by an RP, answers with links to arbitrary, network-internal or external servers (in Step $\boxed{5}$ of Figure 2).

We here, for the first time, point out that not only RPs can be vulnerable to SSRF, but also IdPs. OIDC defines a way to indirectly pass parameters for the authorization request (cf. Step $\boxed{11}$ in Figure 2). To this end, the IdP accepts a new parameter, *request_uri* in the authorization request. This parameter contains a URI from which the IdP retrieves the additional parameters (e.g., *redirect_uri*). The attacker can use this feature to easily mount an SSRF attack against the IdP even without any OIDC extensions: He can put an arbitrary URI in an authorization request causing the IdP to contact this URI.

This new attack vector shows that not only RPs but also IdPs have to protect themselves against SSRF by using appropriate filtering and limiting mechanisms to restrict unwanted requests that originate from a web server (cf. [38]).

SSRF attacks typically depend on an application specific context, such as the structure of and (vulnerable) services in an internal network. In our model, attackers can trigger SSRF requests, but the model does not contain vulnerable applications/services aside from OIDC. (Our analysis focuses on the security of the OIDC standard itself, rather than on specific applications and services.) Timing and performance properties, while sometimes relevant for SSRF attacks, are also outside of our analysis.

9) Third-Party Resources: RPs and IdPs that include third-party resources, e.g., tracking or advertisement scripts, subject their users to token theft and other attacks by these third parties. If possible, RPs and IdPs should therefore avoid including third-party resources on any web resources delivered from the same origins[6] as the OIDC endpoints (see also Section V-F). For newer browsers, *subresource integrity* [2] can help to reduce the

[6]Since scripts on one origin can often access documents on the same origin, origins of the OIDC endpoints should be free from untrusted scripts.

risks associated with embedding third-party resources. With subresource integrity, websites can instruct supporting web browsers to reject third-party content if this content does not match a specific hash. In our model, we assume that websites do not include untrusted third-party resources.

10) Transport Layer Security: The security of OIDC depends on the confidentiality and integrity of the transport layer. In other words, RPs and IdPs should use HTTPS. Endpoint URIs that are provided for the end user and that are communicated, e.g., in the discovery phase of the protocol, should only use the `https://` scheme. HTTPS Strict Transport Security and Public Key Pinning can be used to further strengthen the security of the OIDC endpoints. (In our model, we assume that users enter their passwords only over HTTPS web sites because otherwise, any authentication could be broken trivially.)

11) Session Handling: Sessions are typically identified by a nonce that is stored in the user's browser as a cookie. It is a well known best practice that cookies should make use of the *secure* attribute (i.e., the cookie is only ever used over HTTPS connections) and the *HttpOnly* flag (i.e., the cookie is not accessible by JavaScript). Additionally, after the login, the RP should replace the session id of the user by a freshly chosen nonce in order to prevent session fixation attacks: Otherwise, a network attacker could set a login session cookie that is bound to a known *state* value into the user's browser (see [46]), lure the user into logging in at the corresponding RP, and then use the session cookie to access the user's data at the RP (*session fixation*, see [37]). In our model, RPs use two kinds of sessions: Login sessions (which are valid until just before a user is authenticated at the RP) and service sessions (which signify that a user is already signed in to the RP). For both sessions, the secure and HttpOnly flags are used.

B. Relationship to OAuth 2.0

Many, but not all of the attacks described above can also be applied to OAuth 2.0. The following attacks in particular are only applicable to OIDC: (1) Server-side request forgery attacks are facilitated by the ad-hoc discovery and dynamic registration features. (2) The same features enable new ways to carry out injection attacks. (3) The new OIDC feature third-party login initiation enables new CSRF attacks. (4) Attacks on the id token only apply to OIDC, since there is no such token in OAuth 2.0.

It is interesting to note that on the other hand, some attacks on OAuth 2.0 cannot be applied to OIDC (see [22], [35] for further discussions on these attacks): (1) OIDC setups are less prone to open redirector attacks since placeholders are not allowed in redirection URIs. (2) TLS is mandatory for some messages in OIDC, while it is optional in OAuth 2.0. (3) The nonce value can prevent some replay attacks when the state value is not used or leaks to an attacker.

C. Discussion

In this section, our focus was to provide a concise overview of known attacks on OIDC and present some additions, namely SSRF at IdPs and third-party login initiation, along with mitigations and implementation guidelines. Our formal analysis of OIDC, which is the main focus of our work and is presented in the next sections, shows that the mitigations and implementation guidelines presented above are effective and that we can exclude other, potentially unknown types of attacks.

IV. The FKS Web Model

Our formal security analysis of OIDC is based on the FKS model, a generic Dolev-Yao style web model proposed by Fett et al. in [19]. Here, we only briefly recall this model following the description in [22] (see Appendices B ff. of our technical report [23] for a full description, and [19]–[21] for comparison with other models and discussion of its scope and limitations).

The FKS model is designed independently of a specific web application and closely mimics published (de-facto) standards and specifications for the web, for example, the HTTP/1.1 and HTML5 standards and associated (proposed) standards. The FKS model defines a general communication model, and, based on it, web systems consisting of web browsers, DNS servers, and web servers as well as web and network attackers.

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

As usual in Dolev-Yao models (see, e.g., [1]), messages are expressed as formal terms over a signature Σ. The signature contains constants (for (IP) addresses, strings, nonces) as well as sequence, projection, and function symbols (e.g., for encryption/decryption and signatures). For example, in the web model, an HTTP request is represented as a term r containing a nonce, an HTTP method, a domain name, a path, URI parameters, request headers, and a message body. For instance, an HTTP request for the URI http://ex.com/show?p=1 is represented as $r := \langle \mathtt{HTTPReq}, n_1, \mathtt{GET}, \mathtt{ex.com}, \mathtt{/show}, \langle\langle \mathtt{p}, 1\rangle\rangle, \langle\rangle, \langle\rangle\rangle$ where the body and the list of request headers is 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.

The *equational theory* associated with Σ is defined as usual in Dolev-Yao models. The theory induces a congruence relation $\equiv$ on terms, capturing the meaning of the function symbols in Σ. For instance, the equation in the equational theory which captures asymmetric decryption is $\mathsf{dec_a}(\mathsf{enc_a}(x, \mathsf{pub}(y)), y) = x$. With this, we have that, for example, $\mathsf{dec_a}(\mathsf{enc_a}(\langle r, k'\rangle, \mathsf{pub}(k_{\text{ex.com}})), k_{\text{ex.com}}) \equiv \langle r, k'\rangle$, i.e., these two terms are equivalent w.r.t. the equational theory.

A *(Dolev-Yao) process* consists of a set of addresses the process listens to, a set of states (terms), an initial state, and a relation that takes an event and a state as input and (non-deterministically) returns a new state and a sequence of events. The relation models a computation step of the process. It is required that the output can be computed (formally, derived in the usual Dolev-Yao style) from the input event and the state.

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

A *script* models JavaScript running in a browser. Scripts are defined similarly to Dolev-Yao processes. When triggered by a browser, a script is provided with state information. The script then outputs a term representing a new internal state and a command to be interpreted by the browser (see also the specification of browsers below). We give an annotated example for a script in Algorithm 22 in the appendix of our technical report [23]. Similarly to an attacker process, the so-called *attacker script* outputs everything that is derivable from the input.

A *system* is a set of processes. A *configuration* of this system consists of the states of all processes in the system, the pool of waiting events, and a sequence of unused nonces. Systems induce *runs*, i.e., sequences of configurations, where each configuration is obtained by delivering one of the waiting events of the preceding configuration to a process, which then performs a computation step. The transition from one configuration to the next configuration in a run is called a *processing step*. We write, for example, $Q = (S,E,N) \to (S',E',N')$ to denote the transition from the configuration (S,E,N) to the configuration (S',E',N'), where S and S' are the states of the processes in the system, E and E' are pools of waiting events, and N and N' are sequences of unused nonces.

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

In our analysis of OIDC, we consider either one network attacker or a set of web attackers (see Section V). In our OIDC model, we need to specify only the behavior of servers and scripts. These are not defined by the FKS model since they depend on the specific application, unless they become corrupted, in which case they behave like attacker processes and attacker scripts; browsers are specified by the FKS model (see below). The modeling of OIDC servers and scripts is outlined in Section V and with full details provided in Appendices F and G of our technical report [23].

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

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

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

V. Analysis

We now present our security analysis of OIDC, including a formal model of OIDC, the specifications of central security properties, and our theorem which establishes the security of OIDC in our model.

More precisely, our formal model of OIDC uses the FKS model as a foundation and is derived by closely following the OIDC standards Core, Discovery, and Dynamic Client Registration [39]–[41]. (As mentioned above, the goal in this work is to analyze OIDC itself instead of concrete implementations.) We then formalize the main security properties for OIDC, namely authentication, authorization, session integrity for authentication, and session integrity for authorization. We also formalize secondary security properties that capture important aspects of the security of OIDC, for example, regarding the outcome of the dynamic client registration. We then state and prove our main theorem. Finally, we discuss the relationship of our work to the analysis of OAuth 2.0 presented in [22] and conclude with a discussion of the results.

We refer the reader to Appendices F–I of our technical report [23] for full details, including definitions, specifications, and proofs. To provide an intuition of the abstraction level, syntax, and concepts that we use for the modeling without reading all details, we extensively annotated Algorithms 17, 20, and 22 in Appendix F of our technical report [23].

A. Model

Our model of OIDC includes all features that are commonly found in real-world implementations, for example, all three modes, a detailed model of the Discovery mechanism [41] (including the WebFinger protocol [31]), and Dynamic Client Registration [39] (including dynamic exchange of signing keys). RPs, IdPs, and, as usual in the FKS model, browsers can be corrupted by the adversary dynamically.

We do not model less used features, in particular OIDC logout, self-issued OIDC providers ("personal, self-hosted OPs that issue self-signed ID Tokens", [40]), and ACR/AMR (Authentication Class/Methods Reference) values that can be used to indicate the level of trust in the authentication of the user to the IdP.

Since the FKS model has no notion of time, we overapproximate by never letting tokens, e.g., id tokens, expire. Moreover, we subsume user claims (information about the user that can be retrieved from IdPs) by user identifiers, and hence, in our model users have identities, but no other properties.

We have two versions of our OIDC model, one with a network attacker and one with an unbounded number of web attackers, as explained next. The reason for having two versions is that while the authentication and authorization properties can be proven assuming a network attacker, such an attacker could easily break session integrity. Hence, for session integrity we need to assume web attackers (see the explanations for session integrity in Section V-B).

1) OIDC Web System with a Network Attacker: We model OIDC as a class of web systems (in the sense of Section IV) which can contain an unbounded finite number of RPs, IdPs, browsers, and one network attacker.

More formally, an *OIDC web system with a network attacker* ($\mathcal{OIDC}^n$) consists of a network attacker, a finite set of web browsers, a finite set of web servers for the RPs, and a finite set of web servers for the IdPs. Recall that in $\mathcal{OIDC}^n$, since we have a network attacker, we do not need to consider web attackers (as the network attacker subsumes all web attackers). All non-attacker parties are initially honest, but can become corrupted dynamically upon receiving a special message and then behave just like a web attacker process.

As already mentioned in Section IV, to model OIDC based on the FKS model, we have to specify the protocol specific behavior only, i.e., the servers for RPs and IdPs as well as the scripts that they use. We start with a description of the servers.

a) Web Servers: Since RPs and IdPs both are web servers, we developed a generic model for HTTPS server processes for the FKS model. We call these processes *HTTPS server base processes*. Their definition covers decrypting received HTTPS messages and handling HTTP(S) requests to external webservers, including DNS resolution.

RPs and IdPs are derived from this HTTPS server base process. Their models follow the OIDC standard closely and include the mitigations discussed in Section III.

An RP waits for users to start a login flow and then non-deterministically decides which mode to use. If needed, it starts the discovery and dynamic registration phase of the protocol, and finally redirects the user to the IdP for user authentication. Afterwards, it processes the received tokens and uses them according to their type (e.g., with an access token, the RP would retrieve an id token from the IdP). If an id token is received that passes all checks, the user will be logged in. As mentioned briefly in Section III, RPs manage two kinds of sessions: The *login sessions*, which are used only during the user login phase, and *service sessions*.

The IdP provides several endpoints according to its role in the login process, including its OIDC configuration endpoint and endpoints for receiving authentication and token requests.

b) Scripts: Three scripts (altogether 30 lines of code) can be sent from honest IdPs and RPs to web browsers. The script *script_rp_index* is sent by an RP when the user visits the RP's web site. It starts the login process. The script *script_rp_get_fragment* is sent by an RP during an implicit or hybrid mode flow to retrieve the data from the URI fragment. It extracts the access token, authorization code, and *state* from the fragment part of its own URI and sends this information in the body of a POST request back to the RP. IdP sends the script *script_idp_form* for user authentication at the IdP.

2) OIDC Web System with Web Attackers: We also consider a class of web systems where the network attacker is replaced by an unbounded finite set of web attackers and a DNS server is introduced. We denote such a system by $\mathcal{OIDC}^w$ and call it an *OIDC web system with web attackers*. Such web systems are used to analyze session integrity, see below.

B. Main Security Properties

Our primary security properties capture authentication, authorization and session integrity for authentication and authorization. We will present these security properties in the following, with full details in Appendix H of our technical report [23].

a) Authentication Property: The most important property for OIDC is the authentication property. In short, it captures that a network attacker (and therefore also web attackers) should be unable to log in as an honest user at an honest RP using an honest IdP.

Before we define this property in more detail, recall that in our modeling, an RP uses two kinds of sessions: login sessions, which are only used for the login flow, and service sessions, which are used after a user/browser was logged in (see Section III-A11 for details). When a login session has finished successfully (i.e., the RP received a valid id token), the RP uses a fresh nonce as the service session id, stores this id in the session data of the login session, and sends the service session id as a cookie to the browser. In the same step, the RP also stores the issuer, say d, that was used in the login flow and the identity (email address) of the user, say id, as a pair $\langle d, id \rangle$, referred to as a global user identifier in Section II-A.

Now, our authentication property defines that a network attacker should be unable to get hold of a service session id by which the attacker would be considered to be logged in at an honest RP under an identity governed by an honest IdP for an honest user/browser.

In order to define the authentication property formally, we first need to define the precise notion of a service session. In the following, as introduced in Section IV, (S,E,N) denotes a configuration in the run ρ with its components S, a mapping from processes to states of these processes, E, a set of events in the network that are waiting to be delivered to some party, and N, a set of nonces that have not been used yet. By $\mathsf{governor}(id)$ we denote the IdP that is responsible for a given user identity (email address) id, and by $\mathsf{dom}(\mathsf{governor}(id))$, we denote the set of domains that are owned by this IdP. By $S(r).\mathtt{sessions}[lsid]$ we denote a data structure in the state of r that contains information about the login session identified by $lsid$. This data structure contains, for example, the identity for which the login session with the id $lsid$ was started and the service session id that was issued after the login session.

We can now define that there is a service session identified by a nonce n for an identity id at some RP r iff there exists a login session (identified by some nonce $lsid$) such that n is the service session associated with this login session, and r has stored that the service session is logged in for the id id using an issuer d (which is some domain of the governor of id).

Definition 1 (Service Sessions). We say that there is a *service session identified by a nonce n for an identity id at some RP r* in a configuration (S,E,N) of a run ρ of an OIDC web system iff there exists some login session id $lsid$ and a domain $d \in \mathsf{dom}(\mathsf{governor}(id))$ such that $S(r).\mathtt{sessions}[lsid][\mathtt{loggedInAs}] \equiv \langle d, id\rangle$ and $S(r).\mathtt{sessions}[lsid][\mathtt{serviceSessionId}] \equiv n$.

By $d_\emptyset(S(\mathsf{attacker}))$ we denote all terms that can be computed (derived in the usual Dolev-Yao style, see Section IV) from the attacker's knowledge in the state S. We can now define that an OIDC web system with a network attacker is secure w.r.t. authentication iff the attacker can never get hold of a service session id (n) that was issued by an honest RP r for an identity id of an honest user (browser) at some honest IdP (governor of id).

Definition 2 (Authentication Property). Let $\mathcal{OIDC}^n$ be an OIDC web system with a network attacker. We say that *$\mathcal{OIDC}^n$ is secure w.r.t. authentication* iff for every run ρ of $\mathcal{OIDC}^n$, every configuration (S,E,N) in ρ, every $r \in \mathsf{RP}$ that is honest in S, every browser b that is honest in S, every identity $id \in \mathsf{ID}$ with $\mathsf{governor}(id)$ being an honest IdP, every service session identified by some nonce n for id at r, we have that n is not derivable from the attackers knowledge in S (i.e., $n \not\in d_\emptyset(S(\mathsf{attacker}))$).

b) Authorization Property: Intuitively, authorization for OIDC means that a network attacker should not be able to obtain or use a protected resource available to some honest RP at an IdP for some user unless certain parties involved in the authorization process are corrupted. As the access control for such protected resources relies only on access tokens, we require that an attacker does not learn access tokens that would allow him to gain unauthorized access to these resources.

To define the authorization property formally, we need to reason about the state of an honest IdP, say i. In this state, i creates *records* containing data about successful authentications of users at i. Such records are stored in $S(i).\mathtt{records}$. One such record, say x, contains the authenticated user's identity in $x[\mathtt{subject}]$, two[7] access tokens in $x[\mathtt{access\_tokens}]$, and the client id of the RP in $x[\mathtt{client\_id}]$.

We can now define the authorization property. It defines that an OIDC web system with a network attacker is secure w.r.t. authorization iff the attacker cannot get hold of an access token that is stored in one of i's records for an identity of an honest user/browser b and an honest RP r.

Definition 3 (Authorization Property). Let $\mathcal{OIDC}^n$ be an OIDC web system with a network attacker. We say that *$\mathcal{OIDC}^n$ is secure w.r.t. authorization* iff for every run ρ of $\mathcal{OIDC}^n$, every configuration (S,E,N) in ρ, every $r \in \mathsf{RP}$ that is honest in S, every $i \in \mathsf{IdP}$ that is honest in S, every browser b that is honest in S, every identity $id \in \mathsf{ID}$ owned by b and $\mathsf{governor}(id) = i$, every nonce n, every term $x \in S(i).\mathtt{records}$ with $x[\mathtt{subject}] \equiv id$, $n \in x[\mathtt{access\_tokens}]$, and the client id $x[\mathtt{client\_id}]$ having been issued by i to r,[8] we have that n is not derivable from the attackers knowledge in S (i.e., $n \not\in d_\emptyset(S(\mathsf{attacker}))$).

c) Session Integrity for Authentication: The two session integrity properties capture that an attacker should be unable to forcefully log a user/browser in at some RP. This includes attacks such as CSRF and session swapping. Note that we define these properties over $\mathcal{OIDC}^w$, i.e., we consider web attackers instead of a network attacker. The reason is that OIDC deployments typically use cookies to track the login sessions of users. Since a network attacker can put cookies into browsers over unencrypted connections and these cookies are then also used for encrypted connections, cookies have no integrity in the presence of a network attacker (see also [46]). In particular, a network attacker could easily break the session integrity of typical OIDC deployments.

For session integrity for authentication we say that a user/browser that is logged in at some RP must have expressed her wish to be logged in to that RP in the beginning of the login flow. Note that not even a malicious IdP should be able to forcefully log in its users (more precisely, its user's browsers) at an honest RP. If the IdP is honest, then the user must additionally have authenticated herself at the IdP with the same user account that RP uses for her identification. This excludes, for example, cases where (1) the user is forcefully logged in to an RP by an attacker that plays the role of an IdP, and (2) where an attacker can force an honest user to be logged in at some RP under a false identity issued by an honest IdP.

In our formal definition of session integrity for authentication (below), $\mathsf{loggedIn}^{Q}_{\rho}(b,r,u,i,lsid)$ denotes that in the

[7]In the hybrid mode, IdPs can issue two access tokens, cf. Section II-B3.
[8]See Definition 54 in Appendix H-B of our technical report [23].

processing step Q (see below), the browser b was authenticated (logged in) to an RP r using the IdP i and the identity u in an RP login session with the session id *lsid*. (Here, the processing step Q corresponds to Step $\boxed{18}$ in Figure 2.) The user authentication in the processing step Q is characterized by the browser b receiving the service session id cookie that results from the login session *lsid*.

By $\mathsf{started}^{Q'}_{\rho}(b,r,lsid)$ we denote that the browser b, in the processing step Q' triggered the script *script_rp_index* to start a login session which has the session id *lsid* at the RP r. (Compare Section IV on how browsers handle scripts.) Here, Q' corresponds to Step $\boxed{1}$ in Figure 2.

By $\mathsf{authenticated}^{Q''}_{\rho}(b,r,u,i,lsid)$ we denote that in the processing step Q'', the user/browser b authenticated to the IdP i. In this case, authentication means that the user filled out the login form (in *script_idp_form*) at the IdP i and, by this, consented to be logged in at r (as in Step $\boxed{13}$ in Figure 2).

Using these notations, we can now define security w.r.t. session integrity for authentication of an OIDC web system with web attackers in a straightforward way:

Definition 4 (Session Integr. for Authentication). Let $\mathcal{OIDC}^w$ be an OIDC web system with web attackers. We say that *$\mathcal{OIDC}^w$ is secure w.r.t. session integrity for authentication* iff for every run ρ of $\mathcal{OIDC}^w$, every processing step Q in ρ with $Q = (S,E,N) \to (S',E',N')$ (for some S, S', E, E', N, N'), every browser b that is honest in S, every $i \in \mathsf{IdP}$, every identity u that is owned by b, every $r \in \mathsf{RP}$ that is honest in S, every nonce *lsid*, with $\mathsf{loggedIn}^{Q}_{\rho}(b,r,u,i,lsid)$, we have that (1) there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}^{Q'}_{\rho}(b,r,lsid)$, and (2) if i is honest in S, then there exists a processing step Q'' in ρ (before Q) such that $\mathsf{authenticated}^{Q''}_{\rho}(b,r,u,i,lsid)$.

d) Session Integrity for Authorization: For session integrity for authorization we say that if an RP uses some access token at some IdP in a session with a user, then that user expressed her wish to authorize the RP to interact with *some* IdP. Note that one cannot guarantee that the IdP with which RP interacts is the one the user authorized the RP to interact with. This is because the IdP might be malicious. In this case, for example in the discovery phase, the malicious IdP might just claim (in Step $\boxed{3}$ in Figure 2) that some other IdP is responsible for the authentication of the user. If, however, the IdP the user is logged in with is honest, then it should be guaranteed that the user authenticated to that IdP and that the IdP the RP interacts with on behalf of the user is the one intended by the user.

For the formal definition, we use two additional predicates: $\mathsf{usedAuthorization}^{Q}_{\rho}(b,r,i,lsid)$ means that the RP r, in a login session (session id *lsid*) with the browser b used some access token to access services at the IdP i. By $\mathsf{actsOnUsersBehalf}^{Q}_{\rho}(b,r,u,i,lsid)$ we denote that the RP r not only used *some* access token, but used one that is bound to the user's identity at the IdP i.

Again, starting from our informal definition above, we define security w.r.t. session integrity for authorization of an OIDC web system with web attackers in a straightforward way (and similarly to session integrity for authentication):

Definition 5 (Session Integr. for Authorization). Let $\mathcal{OIDC}^w$ be an OIDC web system with web attackers. We say that *$\mathcal{OIDC}^w$ is secure w.r.t. session integrity for authentication* iff for every run ρ of $\mathcal{OIDC}^w$, every processing step Q in ρ with $Q = (S,E,N) \to (S',E',N')$ (for some S, S', E, E', N, N'), every browser b that is honest in S, every $i \in \mathsf{IdP}$, every identity u that is owned by b, every $r \in \mathsf{RP}$ that is honest in S, every nonce *lsid*, we have that (1) if $\mathsf{usedAuthorization}^{Q}_{\rho}(b,r,i,lsid)$, then there exists a processing step Q' in ρ (before Q) such that $\mathsf{started}^{Q'}_{\rho}(b,r,lsid)$, and (2) if i is honest in S and $\mathsf{actsOnUsersBehalf}^{Q}_{\rho}(b,r,u,i,lsid)$, then there exists a processing step Q'' in ρ (before Q) such that $\mathsf{authenticated}^{Q''}_{\rho}(b,r,u,i,lsid)$.

C. Secondary Security Properties

We define the following secondary security properties that capture specific aspects of OIDC. We use these secondary security properties during our proof of the above main security properties. Nonetheless, these secondary security properties are important and interesting in their own right.

We define and prove the following properties (see the corresponding lemmas in Appendices I-C and I-E of our technical report [23] for details):

Integrity of Issuer Cache: If a relying party requests the issuer identifier from an identity provider (cf. Steps $\boxed{2}$–$\boxed{3}$ in Figure 2), then the RP will only receive an origin that belongs to this IdP in the response. In other words, honest IdPs do not use attacker-controlled domains as issuer identifiers, and the attacker is unable to alter this information on the way to the RP or in the *issuer cache* at the RP.

Integrity of OIDC Configuration Cache: (1) Honest IdPs only use endpoints under their control in their OIDC configuration document (cf. Steps $\boxed{4}$–$\boxed{5}$ in Figure 2) and (2) this information (which is stored at the RP in the so-called *OIDC configuration cache*) cannot be altered by an attacker.

Integrity of JWKS Cache: RPs receive only "correct" signing keys from honest IdPs, i.e., keys that belong to the respective IdP (cf. Steps $\boxed{6}$–$\boxed{7}$ in Figure 2).

Integrity of Client Registration: Honest RPs register only redirection URIs that point to themselves and that these URIs always use HTTPS. Recall that when an RP registers at an IdP, the IdP issues a freshly chosen client id to the RP and then stores RP's redirection URIs.

Third Parties Do Not Learn Passwords: Attackers cannot learn user passwords. More precisely, we define that $\mathsf{secretOfID}(id)$, which denotes the password for a given identity *id*, is not known to any party except for the browser b owning the id and the identity provider i governing the id (as long as b and i are honest).

Attacker Does Not Learn ID Tokens: Attackers cannot learn id tokens that were issued by honest IdPs for honest RPs and identities of honest browsers.

Third Parties Do Not Learn State: If an honest browser logs in at an honest RP using an honest IdP, then the attacker cannot learn the *state* value used in this login flow.

D. Theorem

The following theorem states that OIDC is secure w.r.t. authentication and authorization in presence of the network attacker, and that OIDC is secure w.r.t. session integrity for authentication and authorization in presence of web attackers. For the proof we refer the reader to Appendix I of our technical report [23].

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

E. Comparison to OAuth 2.0

As described in Section II-C, OIDC is based on OAuth 2.0. Since a formal proof for the security of OAuth 2.0 was conducted in [22], one might be tempted to think that a proof for the security of OIDC requires little more than an extension of the proof in [22]. The specific set of features of OIDC introduces, however, important differences that call for new formulations of security properties and require new proofs:

Dynamic Discovery and Registration: Due to the dynamic discovery and registration, RPs can directly influence and manipulate the configuration data that is stored in IdPs. In OAuth, this configuration data is fixed and assumed to be "correct", greatly limiting the options of the attacker. See, for example, the variant [36] of the IdP Mix-up attack that only works in OIDC (mentioned in Section III-A).

Different set of modes: Compared to OAuth, OIDC introduces the hybrid mode, but does not use the resource owner password credentials mode and the client credentials mode.

New endpoints, messages, and parameters: With additional endpoints (and associated HTTPS messages), the attack surface of OIDC is, also for this reason, larger than that of OAuth. The registration endpoints, for example, could be used in ways that break the security of the protocol, which is not possible in OAuth where these endpoints do not exist. In a similar vein, new parameters like nonce, request_uri, and the id token, are contained in several messages (some of which are also present in the original OAuth flow) and potentially change the security requirements for these messages.

Authentication mechanism: The authentication mechanisms employed by OIDC and OAuth are quite different. This shows, in particular, in the fact that OIDC uses the id token mechanism for authentication, while OAuth uses a different, non-standardized mechanism. Additionally, unlike in OAuth, authentication can happen multiple times during one OIDC flow (see the description of the hybrid mode in Section II-B). This greatly influences (the formulation of) security properties, and hence, also the security proofs.

In summary, taking all these differences into account, our security proofs had to be carried out from scratch. At the same time, our proof is more modular than the one in [22] due to the secondary security properties we identified. Moreover, our security properties are similar to the ones by Fett et al. in [22] only on a high level. The underlying definitions in many aspects differ from the ones used for OAuth.[9]

F. Discussion

Using our detailed formal model, we have shown that OIDC enjoys a high level of security regarding authentication, authorization, and session integrity. To achieve this security, it is essential that implementors follow the security guidelines that we stated in Section III. Clearly, in practice, this is not always feasible—for example, many RPs want to include third-party resources for advertisement or user tracking on their origins. As pointed out, however, not following the security guidelines we outline can lead to severe attacks.

We have shown the security of OIDC in the most comprehensive model of the web infrastructure to date. Being a model, however, some features of the web are not included in the FKS model, for example browser plugins. Such technologies can under certain circumstances also undermine the security of OIDC in a manner that is not reflected in our model. Also, user-centric attacks such as phishing or clickjacking attacks are also not covered in the model.

Nonetheless, our formal analysis and the guidelines (along with the attacks when these guidelines are not followed) provide a clear picture of the security provided by OIDC for a large class of adversaries.

VI. Related Work

As already mentioned in the introduction, the only previous works on the security of OIDC are [34], [36]. None of these works establish security guarantees for the OIDC standard: In [34], the authors find implementation errors in deployments of Google Sign-In (which, as mentioned before, is based on OIDC). In [36], the authors describe a variant of the IdP Mix-Up attack (see Section III), highlight the possibility of SSRF attacks at RPs, and show some implementation-specific flaws. In our work, however, we aim at establishing and proving security properties for OIDC.

In general, there have been only few formal analysis efforts for web applications, standards, and browsers so far. Most of the existing efforts are based on formal representations of (parts of) web browsers or very limited models of web mechanisms and applications [3]–[5], [9]–[11], [14]–[17], [24], [25], [27], [32], [45].

Only [7], [8] and [19]–[22] were based on a generic formal model of the web infrastructure. In [8], Bansal, Bhargavan, Delignat-Lavaud, and Maffeis analyze the security of

[9]As an example, in [22], the definitions rely on a notion of *OAuth sessions* which are defined by *connected HTTP(S) messages*, i.e., messages that are created by a browser or server in response to another message. In our model, the attacker is involved in each flow of the protocol (for providing the client id, without receiving any prior message), making it hard to apply the notion of OAuth sessions. We instead define the properties using the existing session identifiers. (See Definitions 54, 52, 56–60 in Appendix H of our technical report [23] for details.)

OAuth 2.0 with the tool ProVerif in the applied pi-calculus and the WebSpi library. They identify previously unknown attacks on the OAuth 2.0 implementations of Facebook, Yahoo, Twitter, and many other websites. They do not, however, establish security guarantees for OAuth 2.0 and their model is much less expressive than the FKS model.

The relationship of our work to [19]–[22] has been discussed in detail throughout the paper.

VII. Conclusion

Despite being the foundation for many popular and critical login services, OpenID Connect had not been subjected to a detailed security analysis, let alone a formal analysis, before. In this work, we filled this gap.

We developed a detailed and comprehensive formal model of OIDC based on the FKS model, a generic and expressive formal model of the web infrastructure. Using this model, we stated central security properties of OIDC regarding authentication, authorization, and session integrity, and were able to show that OIDC fulfills these properties in our model. By this, we could, for the first time, provide solid security guarantees for one of the most widely deployed single sign-on systems.

To avoid previously known and newly described attacks, we analyzed OIDC with a set of practical and reasonable security measures and best practices in place. We documented these security measures so that they can now serve as guidelines for secure implementations of OIDC.

Acknowledgements

This work was partially supported 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] Subresource Integrity – W3C Recommendation 23 June 2016. Jun. 23, 2016.

[3] 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.

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

[5] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and M. L. Tobarra. Formal Analysis of SAML 2.0 Web Browser Single Sign-on: Breaking the SAML-based Single Sign-on for Google Apps. In *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, FMSE 2008*, pages 1–10. ACM, 2008.

[6] M. Balduzzi, C. T. Gimenez, D. Balzarotti, and E. Kirda. Automated Discovery of Parameter Pollution Vulnerabilities in Web Applications. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011.

[7] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Keys to the Cloud: Formal Analysis and Concrete Attacks on Encrypted Web Storage. In *Principles of Security and Trust - Second International Conference, POST 2013*, volume 7796 of *Lecture Notes in Computer Science*, pages 126–146. Springer, 2013.

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

[9] L. Bauer, S. Cai, L. Jia, T. Passaro, M. Stroucken, and Y. Tian. Run-time Monitoring and Formal Analysis of Information Flows in Chromium. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.

[10] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for a browser model. In *5th International Conference on Network and System Security, NSS 2011, Milan, Italy, September 6-8, 2011*, pages 97–104. IEEE, 2011.

[11] A. Bohannon and B. C. Pierce. Featherweight Firefox: formalizing the core of a web browser. In *Proceedings of the 2010 USENIX conference on Web application development*, pages 11–11. USENIX Association, 2010.

[12] J. Bradley. The problem with OAuth for Authentication. Blog post. Jan. 2012. http://www.thread-safe.com/2012/01/problem-with-oauth-for-authentication.html.

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

[14] M. Bugliesi, S. Calzavara, R. Focardi, and W. Khan. CookiExt: Patching the browser against session hijacking attacks. *Journal of Computer Security*, 23(4):509–537, 2015.

[15] M. Bugliesi, S. Calzavara, R. Focardi, W. Khan, and M. Tempesta. Provably Sound Browser-Based Enforcement of Web Session Integrity. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, pages 366–380. IEEE Computer Society, 2014.

[16] S. Calzavara, R. Focardi, N. Grimm, and M. Maffei. Micro-policies for Web Session Security. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 179–193. IEEE Computer Society, 2016.

[17] Y. Cao, Y. Shoshitaishvili, K. Borgolte, C. Krügel, G. Vigna, and Y. Chen. Protecting Web-Based Single Sign-on Protocols against Relying Party Impersonation Attacks through a Dedicated Bi-directional Authenticated Secure Channel. In *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, volume 8688 of *Lecture Notes in Computer Science*, pages 276–298. Springer, 2014.

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

[19] 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.

[20] 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*, volume 9326 of *Lecture Notes in Computer Science*, pages 43–65. Springer, 2015.

[21] 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 (CCS 2015), Denver, CO, USA, October 12-6, 2015*, pages 1358–1369. ACM, 2015.

[22] D. Fett, R. Küsters, and G. Schmitz. A Comprehensive Formal Security Analysis of OAuth 2.0. In *Proceedings of the 23nd ACM SIGSAC Conference on Computer and Communications Security (CCS 2016)*, pages 1204–1215. ACM, 2016.

[23] D. Fett, R. Küsters, and G. Schmitz. The Web SSO Standard OpenID Connect: In-Depth Formal Analysis and Security Guidelines. Technical Report arXiv:1704.08539, arXiv, 2017. Available at http://arxiv.org/abs/1704.08539.

[24] T. Groß, B. Pfitzmann, and A. Sadeghi. Browser Model for Security Analysis of Browser-Based Protocols. In *Computer Security - ESORICS 2005, 10th European Symposium on Research in Computer Security, Milan, Italy, September 12-14, 2005, Proceedings*, volume 3679 of *Lecture Notes in Computer Science*, pages 489–508. Springer, 2005.

[25] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified Security for Browser Extensions. In *32nd IEEE Symposium on Security and*

Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA, pages 115–130. IEEE Computer Society, 2011.

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

[27] D. Hedin, L. Bello, and A. Sabelfeld. Information-flow security for JavaScript and its APIs. *Journal of Computer Security*, 24(2):181–234, 2016.

[28] M. Jones, J. Bradley, and B. Campbell. OAuth 2.0 Token Binding – draft-ietf-oauth-token-binding-01. IETF. Mar. 2016. https://tools.ietf.org/html/draft-ietf-oauth-token-binding-01.

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

[30] M. Jones, J. Bradley, and N. Sakimura. RFC7519 – JSON Web Token (JWT). IETF. May 2015. https://tools.ietf.org/html/rfc7519.

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

[32] 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.

[33] 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.

[34] W. Li and C. J. Mitchell. Analysing the Security of Google's Implementation of OpenID Connect. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, volume 9721, pages 357–376, 2016.

[35] 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.

[36] C. Mainka, V. Mladenov, J. Schwenk, and T. Wich. SoK: Single Sign-On Security – An Evaluation of OpenID Connect. In *IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, 2017.

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

[38] G. Pellegrino, O. Catakoglu, D. Balzarotti, and C. Rossow. Uses and Abuses of Server-Side Requests. In *Research in Attacks, Intrusions, and Defenses - 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings*, volume 9854 of *Lecture Notes in Computer Science*, pages 393–414. Springer, 2016.

[39] 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.

[40] 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.

[41] 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.

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

[43] 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.

[44] M. West. Content Security Policy Level 3 – W3C Working Draft, 13 September 2016. W3C. Sep. 2016. https://www.w3.org/TR/2016/WD-CSP3-20160913/.

[45] S. Yoshihama, T. Tateishi, N. Tabuchi, and T. Matsumoto. Information-Flow-Based Access Control for Web Browsers. *IEICE Transactions*, 92-D(5):836–850, 2009.

[46] 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.