User Access Privacy in OAuth 2.0 and OpenID Connect

1st Wanpeng Li
Department of Computing Science
University of Aberdeen
Aberdeen, UK
wanpeng.li.2013@live.rhul.ac.uk

2nd Chris J Mitchell
Information Security Group
Royal Holloway, University of London
Egham, UK
me@chrismitchell.net

***Abstract*—Currently widely used federated login (single sign-on) systems, notably those based on OAuth 2.0, offer very little privacy for the user, and as a result the identity provider (e.g. Google or Facebook) can learn a great deal about user web behaviour, in particular which sites they access. This is clearly not desirable for privacy reasons, and in particular for privacy-conscious users who wish to minimise the information about web access behaviour that they reveal to third party organisations. In this paper we give a systematic analysis of the user access privacy properties of OAuth 2.0 and OpenID Connect systems, and in doing so describe how simple it is for an identity provider to track user accesses. We also propose possible ways in which these privacy issues could to some extent be mitigated, although we conclude that to make the protocols truly privacy-respecting requires significant changes to the way in which they operate. In particular, it seems impossible to develop simple browser-based mitigations without modifying the protocol behaviour. We also briefly examine parallel research by Hammann et al., who have proposed a means of improving the privacy properties of OpenID Connect.**

***Index Terms*—OAuth 2.0, OpenID Connect, Authentication, Authorization, Privacy**

1. Introduction

Since the OAuth 2.0 authorisation framework was published at the end of 2012 [8], it has been adopted by many websites worldwide as a means of providing federated identity services (what we refer to here as single sign-on (SSO)). By using OAuth 2.0, websites can reduce the burden of password management for their users, as well as saving users the inconvenience of re-entering attributes that are instead stored by identity providers (IdPs) and provided to relying party (RP) websites as required. There is a correspondingly rich infrastructure of IdPs providing identity services using OAuth 2.0; for example, Ghasemisharif et al. [6] identified 65 IdPs among the Alexa top one million websites in 2018. Indeed, some RPs, such as the website USATODAY[1], support as many as six different IdPs.

The security of OAuth 2.0 and OpenID Connect is therefore of critical importance, and it has been widely examined both in theory and in practice [1], [3]–[5], [10]–[12], [14]–[17], [19]–[25]. Previous studies (see, for example, [10]–[12], [21]–[25]) show that, in practice, RPs do not always implement OAuth 2.0 correctly; as a result, many real-world OAuth 2.0 and OpenID Connect systems are vulnerable to attack. A range of mitigations have been proposed for RP developers [2], [13], [15], [25], designed to help secure OAuth 2.0 and OpenID connect systems. However, apart from the work of Hammann et al. [7], discussed in greater detail in Section 4.3, relatively little attention has been paid to the degree to which OAuth 2.0/OpenID Connect protects (or, fails to protect) user privacy. The purpose of this paper is to address this issue, i.e. to thoroughly investigate the user access privacy properties of both OAuth 2.0 and OpenID Connect, covering all the relevant protocol flows.

The remainder of this paper is structured as follows. Section 2 provides an introduction to the operation of OAuth 2.0 and OpenID Connect. In Section 3, we provide a systematic analysis of the user access privacy properties of OAuth 2.0 and OpenID Connect. We discuss the causes of the identified privacy issues and propose possible mitigations in Section 4. Section 5 concludes the paper.

2. Background

2.1. OAuth 2.0

The OAuth 2.0 specification [8] describes a system that allows an application to access resources (typically personal information) protected by a *resource server* on behalf of the *resource owner*, through the consumption of an *access token* issued by an *authorization server*. In support of this system, the OAuth 2.0 architecture involves the following four roles (see Fig. 1).

1) The *Resource Owner* is typically an end user.
2) The *Client* is a server which makes requests on behalf of the resource owner (the *Client* is the RP when OAuth 2.0 is used for SSO).
3) The *Authorization Server* generates access tokens for the client, after authenticating the resource owner and obtaining its authorization.
4) The *Resource Server* stores the protected resources and consumes access tokens provided by an authorization server (this entity and the *Authorization Server* jointly constitute the IdP when OAuth 2.0 is used for SSO).

Fig. 1 summarises the OAuth 2.0 protocol. The client (1) sends an authorization request to the resource owner.

1. https://login.usatoday.com/USAT-GUP/authenticate/?

In response, the resource owner generates an authorization grant (or authorization response) in the form of a *code*, and (2) sends it to the client. After receiving the authorization grant, the client initiates an access token request by authenticating itself to the authorization server and presenting the authorization grant, i.e. the code issued by the resource owner (3). The authorization server issues (4) an access token to the client after successfully authenticating the client and validating the authorization grant. The client makes a protected source request by presenting the access token to the resource server (5). Finally, the resource server sends (6) the protected resources to the client after validating the access token.

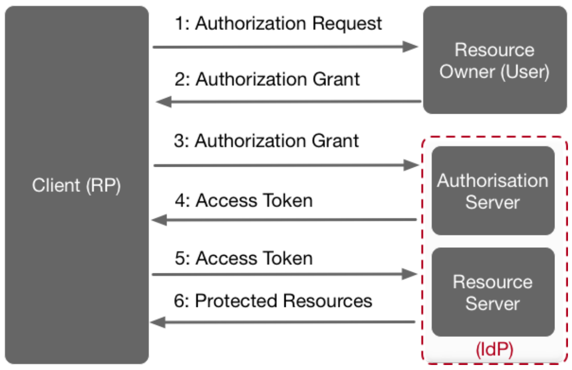


Figure 1. OAuth 2.0 — Protocol Flow

2.1.1. OAuth 2.0 used for SSO. As noted above, in order to use OAuth 2.0 as the basis of an SSO system, the resource server and authorization server together play the IdP role; the client plays the role of the RP, and the resource owner corresponds to the user. OAuth 2.0 and OpenID Connect SSO systems build on user agent (UA) redirections, where a user (U) wishes to access services protected by the RP which consumes the access token generated by the IdP. The UA is typically a web browser. The IdP provides ways to authenticate the user, asks the user to grant permission for the RP to access the user's attributes, and generates an access token on behalf of the user. After receiving the access token, the RP can access the user's attributes using the API provided by the IdP.

2.1.2. OAuth 2.0 protocol flows. The OAuth 2.0 framework defines four ways in which an RP can obtain an access token, namely *Authorization Code Grant*, *Implicit Grant*, *Resource Owner Password*, and *Client Credentials Grant*. Each of these methods involves a distinct set of protocol flows.

Two of the flows, i.e. *Resource Owner Password* and *Client Credentials Grant*, are not relevant in the context of SSO. The *Resource Owner Password* flow requires the user to reveal his or her IdP account password credentials to the resource owner, i.e. it requires a high level of trust in the resource owner; this severely limits the adoption of this flow. The *Client Credentials Grant* flow only involves the client and the authorization server, i.e. no user interactions are needed in this flow. Also, no browser redirection is performed in these two flows; the client directly talks to authorization server.

As a result, and given our focus on SSO privacy issues, in this paper we restrict our attention to the *Authorization Code Grant* and *Implicit Grant* protocol flows, in which browser redirection *is* required. Note that, in the descriptions below, protocol parameters given in bold font are defined as mandatory in the OAuth 2.0 Authorization Framework [8].

```
+--------+                                   +--------+
|        |                                   |        |
|        |---------(1) AuthN Request-------->|        |
|        |                                   |        |
|        |  +--------+                       |        |
|        |  |        |                       |        |
|        |  |  End-  |<--(2) AuthN & AuthZ-->|        |
|        |  |  User  |                       |        |
|   RP   |  |        |                       |   OP   |
|        |  +--------+                       |  (IdP) |
|        |                                   |        |
|        |<--------(3) AuthN Response--------|        |
|        |                                   |        |
|        |---------(4) UserInfo Request----->|        |
|        |                                   |        |
|        |<--------(5) UserInfo Response-----|        |
|        |                                   |        |
+--------+                                   +--------+
```

Figure 2. OpenId Connect — Protocol Overview

2.2. OpenID Connect

OpenID Connect 1.0 [18] builds an identity layer on top of the OAuth 2.0 protocol. The added functionality enables RPs to verify an end user identity by relying on an authentication process performed by an *OpenID Provider* (*OP*) (for consistency, we refer to an IdP instead of an OP in the remainder of the paper). In order to enable an RP to verify the identity of an end user, OpenID Connect adds a new type of token to OAuth 2.0, namely the *id_token*. This complements the access token and code, which are already part of OAuth 2.0. An *id_token* contains claims about the authentication of an end user by an OP, together with any other claims requested by the RP. OpenID Connect (see Fig. 2) supports three authentication flows [18], i.e. ways in which the system can operate, namely *Hybrid Flow*, *Authorization Code Flow* and *Implicit Flow*.

2.3. Tokens

The tokens used by OAuth 2.0 and OpenID Connect, notably the code, access token and id token, are key to the operation of these protocols and also have major privacy implications. We therefore next describe their structure and content.

- An authorization code is an opaque value which is typically bound to an identifier and a URL of an RP. Its main purpose is as a means of giving the RP authorisation to retrieve other tokens from the IdP. In order to help minimise threats arising from its possible exposure, it has a limited validity period and is typically set to expire shortly after issue to the RP.
- An access token is a credential used to authorise access to protected resources stored at a third party (e.g. the Resource Owner). Its value is an opaque string representing an authorization issued to the RP. It encodes the right for the RP to access data held by a specified third party with a specific scope and duration, granted by the end user and enforced by the RP and the IdP. It is a bearer token; that

is, it can be used by any party that gains access to it.

- An id token contains claims about the authentication of an end user by an IdP together with any other claims requested by the RP. Claims that can be inserted into such a token include: the identity of the IdP that issued it, the user's unique identifier at this IdP, the identity of the intended recipient, the time at which it was issued, and its expiry time. It takes the form of a JSON Web Token [9] and is digitally signed by the IdP.

2.4. RP Registration

The RP must register with the IdP before it can use OAuth 2.0, at which time the IdP gathers security-critical information about the RP, including the RP's redirect URI (***redirect_uri***). This is the URI to which the UA is redirected after the IdP has generated the authorization response for the RA and sent it to the UA (for convenience, we also refer to the redirect URI as the *Google sign-in endpoint*). During registration, the IdP issues the RP with a unique identifier (***client_id***) and, optionally, a secret (*client_secret*). If defined, *client_secret* is used by the IdP to authenticate the RP in the Authorization Code Grant flow.

2.5. Authorization Flows

2.5.1. Authorization Code Grant — OAuth 2.0. The OAuth 2.0 Authorization Code Grant flow is very similar to the OpenID Connect Authorization Code flow (see Section 2.5.2 below). This protocol flow relies on information established during the registration process, as described in Section 2.4. The protocol proceeds as follows.

1) U → RP: The user clicks a login button on the RP website, as displayed by the UA, which causes the UA to send an HTTP request to the RP.
2) RP → UA: The RP produces an OAuth 2.0 authorization request and sends it back to the UA. The authorization request includes ***client_id***, the identifier for the client which the RP registered with the IdP previously; ***response_type=code***, indicating that the Authorization Code Grant method is requested; *redirect_uri*, the URI to which the IdP will redirect the UA after access has been granted; *state*, an opaque value used by the RP to maintain state between the request and the callback (step 6 below); and *scope*, the scope of the requested permission.
3) UA → IdP: The UA redirects the request received in step 2 to the IdP.
4) IdP → UA: The IdP first compares the value of *redirect_uri* it received in step 3 (embedded in the authorization request) with the registered value; if the comparison fails, the process terminates. If the user has already been authenticated by the IdP, then the next step is skipped. If not, the IdP returns a login form which is used to collect the user authentication information.
5) U → UA → IdP: The user completes the login form and grants permission for the RP to access the attributes stored by the IdP.
6) IdP → UA → RP: After (if necessary) using the information provided in the login form to authenticate the user, the IdP generates an authorization response and redirects the UA back to the RP. The authorization response contains ***code***, the authorization code (representing the authorization grant) generated by the IdP; and *state*, the value sent in step 2.
7) RP → IdP: The RP produces an access token request and sends it to the IdP token endpoint directly (i.e. not via the UA). The request includes ***grant_type=authorization_code***, ***client_id***, *client_secret* (if the RP has been issued one), ***code*** (generated in step 6), and the ***redirect_uri***.
8) IdP → RP: The IdP checks the ***code***, ***client_id***, *client_secret* (if present), and ***redirect_uri*** and, if the checks succeed, responds to the RP with *access_token*.
9) RP → IdP: The RP passes ***access_token*** to the IdP via a defined API to request the user attributes.
10) IdP → RP: The IdP checks ***access_token*** (how this works is not specified in the OAuth 2.0 specification) and, if satisfied, sends the requested user attributes to the RP.

2.5.2. Authorization Code Flow — OpenID Connect. As previously noted, the Authorization Code flow of OpenID Connect has a similar sequence of steps to that of the OAuth 2.0 Authorization Code Grant. We specify below only those steps where OpenID Connect differs from OAuth 2.0 operation.

2. The RP produces an OpenID Connect authorization request and sends it back to the UA. The authorization request includes ***client_id***, the identifier for the client which the RP registered with the IdP previously; ***response_type=code***, indicating that the Authorization Code Grant method is requested; ***redirect_uri***, the URI to which the IdP will redirect the UA after access has been granted; *state*, an opaque value used by the RP to maintain state between the request and the callback (step 6 below); and ***scope***, the scope of the requested permission.
8. IdP → RP: The IdP checks the ***code***, ***client_id***, *client_secret* (if present), and ***redirect_uri*** and, if the checks succeed, responds to the RP with an ***access_token*** and an ***id_token***.
9. RP → IdP: The RP verifies the validity of the ***id_token***. If it is valid, the RP then passes the ***access_token*** to the IdP to request the desired user attributes.
10. IdP → RP: The IdP checks the ***access_token*** and, if satisfied, sends the requested user attributes to the RP.

2.5.3. Implicit Grant — OAuth 2.0. The OAuth 2.0 Implicit Grant flow is very similar to the OpenID Connect Implicit and Hybrid flows (see Sections 2.5.4 and 2.5.5 below). This flow has a similar sequence of steps to the OAuth 2.0 Authorization Code Grant. We specify below only those steps where Implicit Grant differs from Authorization Code Grant.

2. RP $\rightarrow$ UA: The RP produces an OAuth 2.0 authorization request and sends it back to the UA. The authorization request includes ***client_id***, the identifier for the client which the RP registered with the IdP previously; ***response_type=token***, indicating that the Implicit Grant is requested; *redirect_uri*, the URI to which the IdP will redirect the UA after access has been granted; *state*, an opaque value used by the RP to maintain state between the request and the callback (step 6); and *scope*, the scope of the requested permission.
6. IdP $\rightarrow$ UA $\rightarrow$ RP: After (if necessary) using the information provided in the login form to authenticate the user, the IdP generates an access token and redirects the UA back to the RP using the value of *redirect_uri* provided in step 2. The access token is appended to *redirect_uri* as a URI fragment (i.e. as a suffix to the URI following a # symbol).
7. RP $\rightarrow$ IdP: The RP passes ***access_token*** to the IdP via a defined API to request the user attributes.
8. IdP $\rightarrow$ RP: The IdP checks ***access_token*** (how this works is not stated in the OAuth 2.0 specification) and, if satisfied, sends the requested user attributes to the RP.

As URI fragments are not sent in HTTP requests, the access token is not immediately transferred when the UA is redirected to the RP. Instead, the RP returns a web page (typically an HTML document with an embedded script) capable of accessing the full redirection URI, including the fragment retained by the UA, and extracting the access token (and other parameters) contained in the fragment; the retrieved access token is returned to the RP. The RP can now use this access token to retrieve data stored at the IdP.

2.5.4. Implicit Flow — OpenID Connect. As previously noted, the Implicit Flow of OpenID Connect has a similar sequence of steps to the OAuth 2.0 Implicit Grant. We specify below only those steps where OpenID Connect differs from OAuth 2.0 operation.

2. RP $\rightarrow$ UA: The RP produces an OpenID Connect authorization request and sends it back to the UA. The authorization request includes ***client_id***, the identifier for the client which the RP registered with the IdP previously; ***response_type=****token* ***id_token*** (***id_token*** is always returned in this flow), indicating that the Implicit Flow is requested; ***redirect_uri***, the URI to which the IdP will redirect the UA after access has been granted; *state*, an opaque value used by the RP to maintain state between the request and the callback (step 6 below); ***scope***, the scope of the requested permission; and ***nonce***, the value used to associate a client session with an id token to mitigate replay attacks.
6. IdP $\rightarrow$ UA $\rightarrow$ RP: After (if necessary) using the information provided in the login form to authenticate the user, the IdP generates an ***id_token*** and an *access_token* if requested and redirects the UA back to the RP using the value of ***redirect_uri*** provided in step 2. The id token and access token are appended to ***redirect_uri*** as a URI fragment (i.e. as a suffix to the URI following a # symbol).
7. RP $\rightarrow$ IdP: The RP verifies the validity of the ***id_token***. If it is valid, the RP then passes the ***access_token*** to the IdP to request the desired user attributes.
8. IdP $\rightarrow$ RP: The IdP checks the ***access_token*** and, if satisfied, sends the requested user attributes to the RP.

2.5.5. Hybrid Flow — OpenID Connect. Again as noted above, the Hybrid Flow of OpenID Connect has a similar sequence of steps to the OpenID Connect Implicit Flow. We specify only those steps where Hybrid Flow differs from Implicit Flow.

2. RP $\rightarrow$ UA: The RP produces an OpenID Connect authorization request and sends it back to the UA. The authorization request includes ***client_id***, the identifier for the client which the RP registered with the IdP previously; ***response_type=****token id_token* ***code*** (***code*** is always returned in this flow, other tokens are returned only when requested), indicating that the Implicit Flow is requested; ***redirect_uri***, the URI to which the IdP will redirect the UA after access has been granted; *state*, an opaque value used by the RP to maintain state between the request and the callback (step 6 below); ***scope***, the scope of the requested permission; and ***nonce***, the value used to associate a client session with an id token to mitigate replay attacks.
6. IdP $\rightarrow$ UA $\rightarrow$ RP: After (if necessary) using the information provided in the login form to authenticate the user, the IdP generates a ***code***, an *id_token* or an *access_token* if requested, and redirects the UA back to the RP using the value of ***redirect_uri*** provided in step 2. The code, id token and access token are appended to ***redirect_uri*** as a URI fragment (i.e. as a suffix to the URI following a # symbol).

3. User Access Privacy Analysis of OAuth 2.0 and OpenID Connect

3.1. Scope of Analysis

In this analysis we consider only what an ‘honest but curious’ IdP learns about the web access behaviour of a single user, i.e. we do not address privacy issues relating to *use* of websites. In particular we consider how readily the IdP can determine the identity of the RPs with which a user is interacting, simply by performing its legitimate role in the protocols. Of course, revealing this information may not be harmful for the user if the IdP is honest and trustworthy; nevertheless, an honest IdP could still use this information for website personalisation, e.g. to deliver targeted advertisements. For example, if a user uses Google Sign-in to log in to a music band’s web page, then

Google could use this information to deliver advertising for new albums by this band.

The primary design goal of OAuth 2.0 and OpenID Connect is to enable an RP to gain limited access to an HTTP service either on behalf of the user or for the purposes of the RP itself; there are no specific privacy objectives of these protocols. In order to achieve the design goals, the RP has to retrieve user data from the IdP server; thus the IdP is able to track user activity at an RP by monitoring its resource server. Moreover, in the OpenID Connect protocol flows, as the id token contains attributes of an end user, two colluding RPs could use the id tokens they receive to link interactions belonging to the same user, even without the help of the IdP.

In the absence of specific privacy objectives, it is hardly surprising that hiding the RP identity from the IdP was not a goal for the protocol designers. Indeed, it is probably the case that many IdPs like this feature, since it helps them build user profiles. Nonetheless, it is clearly a privacy risk for the user, and this is why we have chosen to investigate ways in which the advantages of OAuth 2.0 and OpenID Connect could be preserved whilst limiting what an IdP learns about the identities of the websites visited by a user. We now give an analysis of the user access privacy issues that arise in a range of scenarios.

3.2. Privacy Goals

As mentioned previously, our focus on privacy is restricted to considering what an IdP can learn about the identity of the websites which a user visits. This seems a reasonable focus, given that the main objective of the SSO systems we examine is to enable authenticated sessions between a user browser and a website, and the IdPs are typically not involved once such sessions are established.

More precisely, in this paper, we are concerned with the following two user privacy goals.

- **User login unlinkability with respect to the IdP**. Given two honest RPs, RP_1 and RP_2, the IdP cannot distinguish which of them the user has chosen to log-in to. Meeting this goal could prevent an IdP from building a website activity profile for a user who uses the IdP to log-in to multiple RPs.
- **User information retrieval unlinkability with respect to the IdP**. Given two honest RPs, RP_1 and RP_2, the IdP cannot distinguish which of them is accessing the user information stored at the IdP.

3.3. Privacy Issues in Registration

Both OAuth 2.0 and OpenID Connect require that the RP registers with the IdP before it can use OAuth 2.0 or OpenID Connect services. During registration the IdP gathers information about the RP, including the RP's redirect URI (***redirect_uri***). The IdP also issues the RP with a unique identifier (***client_id***) and a secret (*client_secret*). Any one of these three values can be used to identify the RP. Thus if a user's activity can be linked to any of these values then the IdP immediately knows the RP with which the user is interacting, endangering both the privacy goals enumerated above. In particular, if a code or access token containing any of these three values is transferred to the IdP, then the IdP can immediately determine the identity of the RP.

3.4. Privacy Issues in Authorization Code Grant

As described in Section 2.5.1, the OAuth 2.0 Authorization Code Grant is very similar to the OpenID Connect Authorization Code Flow. The privacy issues described below thus apply to both OAuth 2.0 and OpenID Connect.

There are several points within the protocol flow at which an IdP could learn the identity of the RP with which the user is interacting, as listed below.

1) Whenever an RP sends an OAuth 2.0 authorization request (step 2 of Section 2.5.1) to the IdP using the Authorization Code Grant flow, the IdP can learn which RP the user is trying to access by checking the *client_id* and *redirect_uri* of the authorization request. After authenticating the user, the IdP redirects the user to the RP (step 6). If a user uses the IdP to log-in to multiple RPs, the IdP will learn the identities of all the RPs that the user is accessing.
2) After the RP receives the authorization response and the embedded code, it has to exchange the code for an access token with the IdP (step 7). During the exchange process, the RP has to reveal information which reveals its identity, such as its *client_id* and *client_secret*, to the IdP. The IdP can then use this information to connect the user's activity to this specific RP, enabling the IdP to track the user.
3) The access token used in both OAuth 2.0 and OpenID Connect is a bearer token which encodes the user's id, its expiry time, and its intended audience (the RP). IdPs such as Facebook[2] and Google[3] (see Fig 3) provide APIs for RPs to check the access token information. When an RP retrieves the user's attributes from the IdP's resource server (step 9) using an access token, the IdP will learn both the RP identity and the user attributes the RP is trying to access. The IdP could use this information to track the user's accesses to the RP without the user being aware. This is because the RP can retrieve user information from the IdP using the access token, even when the user is offline. When this occurs, the IdP is obviously aware of the information retrieval, and hence knows which RP is retrieving which user's information.
4) When using OpenID Connect, both an *id_token* and *access_token* are issued to the RP in step 8. As described in Section 2.2, an *id_token* contains claims about the authentication of an end user by an IdP, together with any other claims requested by the RP. It also contains the issuer and audience of the token, which means the IdP is aware of the RP identity when generating the *id_token*. That

2. https://developers.facebook.com/docs/facebook-login/manually-build-a-login-flow#checktoken
3. https://developers.google.com/identity/protocols/OAuth2UserAgent#validate-access-token

is, when an RP relies on an *id_token* to grant user access to its services, the IdP is still able to track the access behaviour of the user.

All four of the cases considered above threaten both of the privacy goals described in Section 3.2.

3.5. Privacy Issues in Implicit Grant

The OAuth 2.0 Implicit Grant flow shares almost all the privacy issues we have identified in OAuth 2.0 Authorization Code Grant. The only difference is that no code is exchanged during OAuth 2.0 Implicit Grant. Thus the second privacy issue described in Section 3.4 does not apply to the Implicit Grant flow.

4. Discussion

4.1. Main Issues

Both OAuth 2.0 and OpenID Connect require that a RP registers with the IdP before using the service. At this point the IdP collects privacy-related information (e.g. the *redirect_uri*) which could be used to learn the identity of the RP with which a user is interacting; it additionally distributes information (e.g. the *client_id* and *client_secret*) that could also be used to learn which RPs a user is accessing.

All the (relevant) authorization flows of OAuth 2.0 and OpenID Connect rely on user agent redirections. During the authorization process, in order to deliver the response to the RP, the IdP needs to know the web location (the *redirect_uri*) to which it must deliver the response. This *redirect_uri* immediately reveals the RP identity to the IdP. This is the main reason why OAuth 2.0 and OpenID Connect do not support the user access privacy goals enumerated in Section 3.2.

Also, both the access token and *id_token* contain the RP's identity. The IdP is aware of the token audience when issuing such a token, and can use this information to determine the number of users that have logged in to the RP using its identity service.

At this point it is important to observe that a range of middleware [2], [15] has been published to try to mitigate both the vulnerabilities and the privacy threats posed by incorrect OAuth 2.0 and OpenID Connect implementations. However, the privacy focus of these schemes is very limited; they all restrict their attention to mitigating the privacy threats that can be caused by leaking the *id_token* or the *access_token* to unauthorised parties, and rely on user agent redirection to deliver the authorization request. As a result, none of them is able to address the privacy issues we described above. That is, they are not concerned with user access privacy *with respect to the IdP*, which is the main focus of this paper.

Given the fundamental nature of the privacy issues we have observed, and in particular their reliance on browser redirections, it seems impossible to enhance the privacy properties of OAuth 2.0 (and OpenID Connect) without making some fairly fundamental changes to the way these systems work. That is, while it would clearly be desirable to introduce user-privacy-enhancing obfuscations into the protocol flows at the UA without changing the ways in which IdPs and RPs interact, as we clarify below all such efforts seem to be blocked by the design of the protocol flows. Indeed, it could even be that this is a deliberate choice by the protocol designers. We therefore conclude this paper by considering possible ways in which these systems could be redesigned to enable greater user privacy protection without significantly affecting the simplicity of protocol operation.

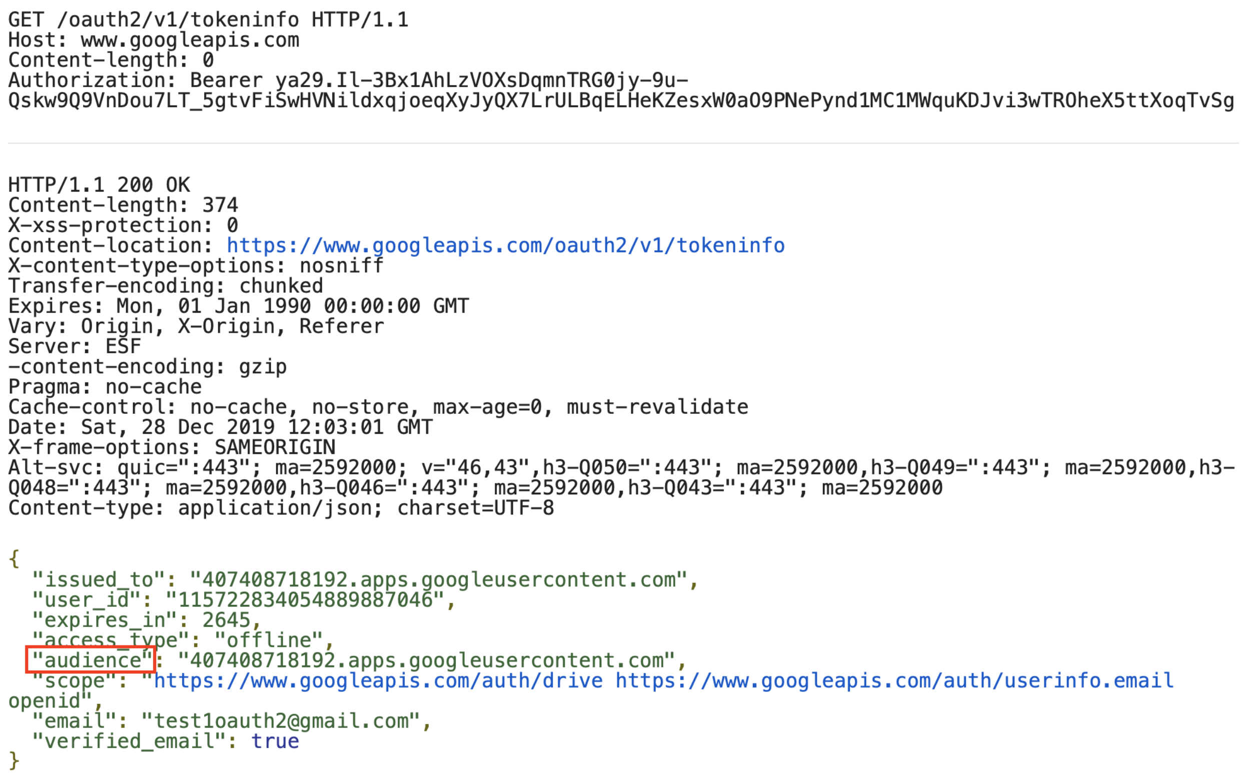
4.2. Possible Mitigations

As described above, the main reasons why OAuth 2.0 and OpenID Connect are not user-access-privacy-preserving is because they both rely on user agent redirection and the IdP has to exchange privacy-breaching information with the RP. In order to attempt to mitigate these privacy issues, we have the following recommendations for possible modifications to the design and/or implementation of these systems.

4.2.1. Protocol Changes. The following two changes to the protocols are required to address the issues we have identified.

1) As described in Sections 3.3 and 4.1, the registration processes for OAuth 2.0 and OpenID Connect involve equipping the IdP with information about the RP, notably the redirect_uri, the client_id and client_secret, use of any of which will immediately reveal the identity of the RP. If these are not used during operation of the protocols, then there is no point in giving them to the IdP during registration. This process therefore needs to be redesigned to make sure that the information exchanged by the IdP and RP (and then used in protocol operation) will not subsequently threaten user privacy. How this could be achieved is not immediately clear, and remains an issue for further research.
2) The audience attribute in both the *id_token* and *access_token* needs to be removed from the token itself. This will require the design of the protocol to be modified to ensure that no security issues will arise after removing this attribute. In particular, if the audience attribute is removed, a malicious RP could use collected *access_token* and *id_token* values to impersonate a victim user to another RP.
   One possible mitigation to this security problem would be to use a challenge-response-based approach, e.g. requiring an RP to generate a random value for each authorization request (functioning just like the state parameter in OAuth 2.0), and include this random value in the user's RP session as well as the *access_token* and *id_token*; this would enable the receiving RP to verify that the token was generated for it. As the value is generated randomly, the IdP should not be able to use it to build a profile of the RP.

4.2.2. Client-based Mitigations. Client-based middleware (e.g. a browser extension) could be used to redirect the OAuth 2.0 authorization requests and responses



```
GET /oauth2/v1/tokeninfo HTTP/1.1
Host: www.googleapis.com
Content-length: 0
Authorization: Bearer ya29.Il-3Bx1AhLzVOXsDqmnTRG0jy-9u-
Qskw9Q9VnDou7LT_5gtvFiSwHVNildxqjoeqXyJyQX7LrULBqELHeKZesxW0aO9PNePynd1MC1MWquKDJvi3wTROheX5ttXoqTvSg
```

```
HTTP/1.1 200 OK
Content-length: 374
X-xss-protection: 0
Content-location: https://www.googleapis.com/oauth2/v1/tokeninfo
X-content-type-options: nosniff
Transfer-encoding: chunked
Expires: Mon, 01 Jan 1990 00:00:00 GMT
Vary: Origin, X-Origin, Referer
Server: ESF
-content-encoding: gzip
Pragma: no-cache
Cache-control: no-cache, no-store, max-age=0, must-revalidate
Date: Sat, 28 Dec 2019 12:03:01 GMT
X-frame-options: SAMEORIGIN
Alt-svc: quic=":443"; ma=2592000; v="46,43",h3-Q050=":443"; ma=2592000,h3-Q049=":443"; ma=2592000,h3-
Q048=":443"; ma=2592000,h3-Q046=":443"; ma=2592000,h3-Q043=":443"; ma=2592000
Content-type: application/json; charset=UTF-8

{
  "issued_to": "407408718192.apps.googleusercontent.com",
  "user_id": "115722834054889887046",
  "expires_in": 2645,
  "access_type": "offline",
  "audience": "407408718192.apps.googleusercontent.com",
  "scope": "https://www.googleapis.com/auth/drive https://www.googleapis.com/auth/userinfo.email
openid",
  "email": "test1oauth2@gmail.com",
  "verified_email": true
}
```

Figure 3. Google OAuth 2.0 Token Information Endpoint

transferred between the RP and IdP. Introducing such middleware could prevent the IdP from learning where the authorization request is coming from and where the authorization response is sent to, thus helping to reduce the privacy threats to users. Such a design change could have the helpful by-product of largely mitigating the threat of phishing attacks, a well-known threat vector for both OAuth 2.0 and OpenID Connect.

Again, designing such middleware to ensure privacy without introducing new security vulnerabilities is a non-trivial issue, requiring further research.

A further possibility might be to introduce a new browser API specifically for SSO, that would enable RPs to start protocol runs with an IdP of their choice. This could address the issues caused by browser redirections. Such a change could be supplementary to the standard protocol, or be integrated into a modified protocol.

4.3. Privacy-Preserving OpenID Connect

The above proposals clearly need further development in order to try to develop a completely privacy-friendly SSO system. In particular, the precise security and privacy properties of the modified protocols will need careful analysis. We conclude this discussion by considering a very recent proposal for protocol modifications which, although intended to address the privacy issues we have discussed, unfortunately fails to do so effectively.

4.3.1. Operation of scheme. In work conducted concurrently with that described here, Hammann et al. [7] very recently proposed a privacy-preserving modification to OpenID Connect Implicit Flow (see Figure 4), which has some features in common with the mitigations proposed above. They made two changes to the standard OpenID Connect Implicit Flow, and deleted the authentication process from the 'standard' OpenID Connect Implicit Flow (see step 5 in Section 2.5.1). The operation of the modified scheme is summarised in Fig. 4.

Two changes are of particular significance.

- **Masking the *client_id*.** The original *client_id* is replaced by an (opaque) hash value $H(client\_id||rp\_nonce||u\_nonce)$, where H is a cryptographic hash function, *u_nonce* is a cryptographically secure, unpredictable, random value generated by the user agent, and *rp_nonce* is a nonce generated by the RP for replay protection.
- **User consent in user agent.** A new signed token *client_id_binding* needs to be generated during registration. This token is used to present the consent page to the user. It contains the RP's *client_id*; *client_name*; the name of the RP in a form that can be understood and checked by the user; and the RP's set of registered *redirect_uris*, i.e. valid end-points to which the *private_id_token* can be sent.

Note that the difference between the *private_id_token* and the original *id_token* is that the *private_id_token* contains a *private_aud* field which equals to $H(client\_id||rp\_nonce||u\_nonce)$ and replaces the *aud* field in the orignal *id_token*.

4.3.2. Analysis. As shown in Fig. 4, the modified protocol does not include the step in which the IdP authenticates the

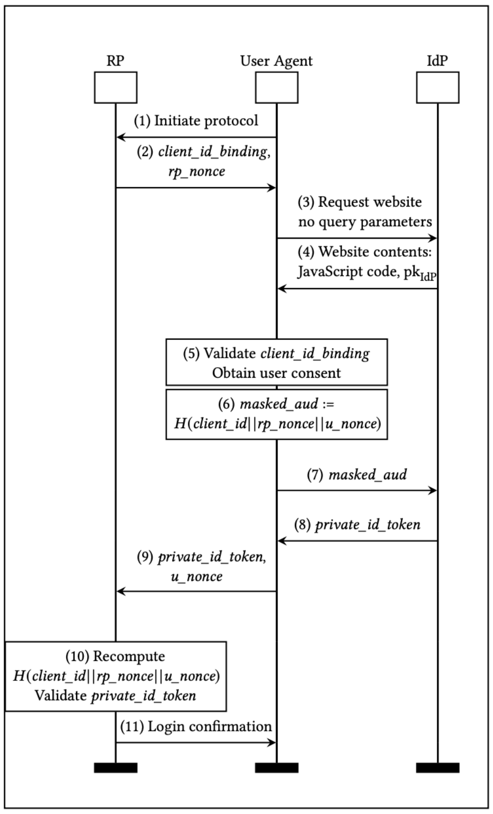

RP
User Agent
IdP
(1) Initiate protocol
(2) *client_id_binding*, *rp_nonce*
(3) Request website no query parameters
(4) Website contents: JavaScript code, pk_{IdP}
(5) Validate *client_id_binding* Obtain user consent
(6) *masked_aud* := $H(client\_id||rp\_nonce||u\_nonce)$
(7) *masked_aud*
(8) *private_id_token*
(9) *private_id_token*, *u_nonce*
(10) Recompute $H(client\_id||rp\_nonce||u\_nonce)$ Validate *private_id_token*
(11) Login confirmation

Figure 4. Privacy-preserving OpenID Connect of Hammann et al.

end user (see step 5 in 2.5.1), which will cause problems in practice. By putting the *client_id_binding* and *rp_nonce* into the URI as fragments, they implicitly assume that the user has already been authenticated by the IdP, and the IdP acts as if it has authenticated the user and generates tokens for the user.

This looks like a good idea at first sight. However, omitting the standard authentication process from the protocol makes the proposed scheme unworkable, since the IdP will not know the identity of the user for which it is generating tokens. As a result, there needs to be an authentication process in the protocol which collects user credentials, and this process[4] should occur just after step 3 in Figure 4.

However, as the authentication process needs to collect user credentials (e.g. using a login form), normally user agent redirections need to be involved after form submission. The user will be redirected to the user consent page after being authenticated, and the user consent page will have no access to the URI fragment parameters in step 2. This would render steps 4–9 unworkable. In order to make their scheme work, a dedicated authentication process should be designed for the protocol using JavaScript APIs (e.g. `XMLHTTPRequest/fetch` or iframes that communicate via `postMessage`). However, Hammann et al. claim that 'This authentication step is, however, not explicitly part of the OpenID Connect protocol, and thus is not part of privacy-preserving OpenID Connect either' (see section 4.3 in [7]).

In summary, because of the removal of the user authentication process, the Hammann et al. scheme will work only if the user is already authenticated at the IdP and this may pose usability concerns. Moreover, their scheme only focuses on privacy issues caused by the *id_token* (used for authentication in OpenID Connect); the privacy issues caused by the *access_token* (used for authorization in OpenID Connect) are not addressed.

Additionally we point out that the Hammann et al. attack model is somewhat unrealistic. It assumes that the IdP is honest but curious (i.e. exactly the model we also assume), and the proposed modified version of the OpenID Connect protocol appears to be based on the assumption that client-side JavaScript code could be run independently, i.e. without the involvement of the IdP. Specifically, in steps 4–9 of the modified OpenID Connect protocol (see Section 4.3 of [7]), the website which supplies the JavaScript that executes at the user agent is under the control of the IdP. This means that a 'slightly dishonest' IdP could arrange for the JavaScript to send back to the IdP every parameter involved in steps 4–9. This includes the *client_id*, which is the value that the modified protocol is designed to hide from the IdP, and the *redirect_uri*, which is used to deliver the authorization response to the RP. We use the term 'slightly dishonest' here, as the JavaScript would conduct all steps in the protocol honestly, but would simply send back to its owner information it learns. That is, we suggest that there is little difference in practice between information which inevitably passes through the IdP and that which passes through a proxy under the control of the IdP.

Of course, in principle the JavaScript code could be analysed, and any monitoring behaviour would be revealed — this could be very damaging for the reputation of the IdP. However, in practice such JavaScript is likely to be large and complex, and extremely difficult to analyse. Also, any transfer of data to the IdP could be packaged with necessary data transfers and hence would be invisible to anyone monitoring the behaviour of the JavaScript.

This demonstrates the difficulty in devising modifications that effectively meet the privacy goals without making very major changes to protocol operation.

5. Conclusions

In this paper, we have described serious user access privacy issues that affect both OAuth 2.0 and OpenID Connect. We also observed that it seems impossible to develop simple browser-based mitigations that significantly enhance user privacy with respect to the IdP without changing the underlying protocols; that is, changes are required to both the browsers and the protocols to make OAuth 2.0 and OpenID Connect user-access-privacy-friendly. We have therefore also outlined possible modifications to the operation of OAuth 2.0 and OpenID Connect that could help mitigate the main privacy issues. However, these ideas are still at an early stage of development and need considerable further work before detailed changes to the protocols can be proposed. We also critically examined parallel research by Hammann et

4. If the authentication process takes place before step 3, the IdP needs to record the URL from step 2 in the URL parameters and redirect the user to that URL after authentication is complete; thus the IdP will learn the value *client_id_binding*.

al., who have proposed a means of improving the privacy properties of OpenID Connect.

Acknowledgements

The authors would like to thank Mauro Tempesta and the anonymous referees for their valuable comments and suggestions which have significantly improved the paper.

References

[1] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Discovering concrete attacks on website authorization by formal analysis. *Journal of Computer Security*, 22(4):601–657, 2014.

[2] Stefano Calzavara, Riccardo Focardi, Matteo Maffei, Clara Schneidewind, Marco Squarcina, and Mauro Tempesta. WPSE: Fortifying web protocols via browser-side security monitoring. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1493–1510, 2018.

[3] Suresh Chari, Charanjit S Jutla, and Arnab Roy. Universally composable security analysis of OAuth v2.0. http://eprint.iacr.org/2011/526, September 2011. IACR Cryptology ePrint Archive: Report 2011/526.

[4] Daniel Fett, Ralf Küsters, and Guido Schmitz. A comprehensive formal security analysis of OAuth 2.0. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1204–1215. ACM, 2016.

[5] Daniel Fett, Ralf Küsters, and Guido Schmitz. The web SSO standard OpenID Connect: In-Depth formal security analysis and security guidelines. arXiv:1704.08539 [cs.CY], http://arxiv.org/abs/1704.08539, April 2017.

[6] Mohammad Ghasemisharif, Amrutha Ramesh, Stephen Checkoway, Chris Kanich, and Jason Polakis. O single sign-off, where art thou? An empirical analysis of single sign-on account hijacking and session management on the web. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1475–1492, Baltimore, MD, August 2018. USENIX Association.

[7] Sven Hammann, Ralf Sasse, and David Basin. Privacy-preserving OpenID Connect. https://people.inf.ethz.ch/rsasse/pub/poidc-asiaccs.pdf. To appear in Proc. Asia CCS 2020, ACM Press.

[8] Dick Hardt (editor). RFC 6749: The OAuth 2.0 authorization framework. October 2012. http://tools.ietf.org/html/rfc6749.

[9] Michael Jones, Nat Sakimura, and John Bradley. RFC 7519: JSON Web Token (JWT). 2015. https://tools.ietf.org/html/rfc7519.

[10] Wanpeng Li and Chris J. Mitchell. Security issues in OAuth 2.0 SSO implementations. In Sherman S. M. Chow, Jan Camenisch, Lucas Chi Kwong Hui, and Siu-Ming Yiu, editors, *Information Security - 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings*, volume 8783 of *Lecture Notes in Computer Science*, pages 529–541. Springer, 2014.

[11] Wanpeng Li and Chris J. Mitchell. Addressing threats to real-world identity management systems. In Helmut Reimer, Norbert Pohlmann, and Wolfgang Schneider, editors, *ISSE 2015 - Highlights of the Information Security Solutions Europe 2015 Conference, Berlin, Germany, November 1-2, 2015*, pages 251–259. Springer, 2015.

[12] Wanpeng Li and Chris J. Mitchell. Analysing the security of Google's implementation of OpenID Connect. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment — 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*, volume 9721 of *Lecture Notes in Computer Science*, pages 357–376. Springer, 2016.

[13] Wanpeng Li, Chris J. Mitchell, and Thomas Chen. Mitigating CSRF attacks on OAuth 2.0 systems. In Kieran McLaughlin, Ali A. Ghorbani, Sakir Sezer, Rongxing Lu, Liqun Chen, Robert H. Deng, Paul Miller, Stephen Marsh, and Jason R. C. Nurse, editors, *16th Annual Conference on Privacy, Security and Trust, PST 2018, Belfast, Northern Ireland, UK, August 28–30, 2018*, pages 1–5. IEEE Computer Society, 2018.

[14] Wanpeng Li, Chris J. Mitchell, and Thomas Chen. Your code is my code: Exploiting a common weakness in OAuth 2.0 implementations. In Vashek Matyás, Petr Svenda, Frank Stajano, Bruce Christianson, and Jonathan Anderson, editors, *Security Protocols XXVI — 26th International Workshop, Cambridge, UK, March 19–21, 2018, Revised Selected Papers*, volume 11286 of *Lecture Notes in Computer Science*, pages 24–41. Springer, 2018.

[15] Wanpeng Li, Chris J. Mitchell, and Thomas Chen. OAuthGuard: Protecting User Security and Privacy with OAuth 2.0 and OpenID Connect. In *Proceedings of the 5th ACM Workshop on Security Standardisation Research Workshop*, SSR'19, pages 35–44, New York, NY, USA, 2019. ACM.

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

[17] Suhas Pai, Yash Sharma, Sunil Kumar, Radhika M Pai, and Sanjay Singh. Formal verification of OAuth 2.0 using Alloy framework. In *Proceedings of the International Conference on Communication Systems and Network Technologies (CSNT), 2011*, pages 655–659. IEEE, 2011.

[18] Nat Sakimura, John Bradley, Michael Jones, Breno de Medeiros, and Mortimore Chuck. OpenID Connect Core 1.0. 2014. http://openid.net/specs/openid-connect-core-1_0.html.

[19] Mohamed Shehab and Fadi Mohsen. Securing OAuth implementations in smart phones. In Elisa Bertino, Ravi S. Sandhu, and Jaehong Park, editors, *Fourth ACM Conference on Data and Application Security and Privacy, CODASPY'14, San Antonio, TX, USA — March 03 - 05, 2014*, pages 167–170. ACM, 2014.

[20] Quinn Slack and Roy Frostig. Murphi analysis of OAuth 2.0 implicit grant flow. 2011. http://www.stanford.edu/class/cs259/WWW11/.

[21] San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: An empirical analysis of OAuth SSO systems. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS '12, Raleigh, NC, USA, October 16-18, 2012*, pages 378–390. ACM, 2012.

[22] Rui Wang, Shuo Chen, and XiaoFeng Wang. Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 365–379. IEEE Computer Society, 2012.

[23] Ronghai Yang, Wing Cheong Lau, and Shangcheng Shi. Breaking and fixing mobile app authentication with oauth2.0-based protocols. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *Applied Cryptography and Network Security - 15th International Conference, ACNS 2017, Kanazawa, Japan, July 10-12, 2017, Proceedings*, volume 10355 of *Lecture Notes in Computer Science*, pages 313–335. Springer, 2017.

[24] Ronghai Yang, Guanchen Li, Wing Cheong Lau, Kehuan Zhang, and Pili Hu. Model-based security testing: An empirical study on oauth 2.0 implementations. In Xiaofeng Chen, XiaoFeng Wang, and Xinyi Huang, editors, *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 – June 3, 2016*, pages 651–662. ACM, 2016.

[25] Yuchen Zhou and David Evans. SSOScan: Automated testing of web applications for Single Sign-On vulnerabilities. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, pages 495–510. USENIX Association, 2014.