OAuthGuard: Protecting User Security and Privacy with OAuth 2.0 and OpenID Connect

Wanpeng Li
School of Computing and Mathematics
Manchester Metropolitan University
Manchester, UK
W.Li@mmu.ac.uk

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

Thomas Chen
Department of Electrical & Electronic Engineering
City, University of London
London, UK
Tom.Chen.1@city.ac.uk

ABSTRACT

Millions of users routinely use Google to log in to websites supporting the standardised protocols OAuth 2.0 or OpenID Connect; the security of OAuth 2.0 and OpenID Connect is therefore of critical importance. As revealed in previous studies, in practice RPs often implement OAuth 2.0 incorrectly, and so many real-world OAuth 2.0 and OpenID Connect systems are vulnerable to attack. However, users of such flawed systems are typically unaware of these issues, and so are at risk of attacks which could result in unauthorised access to the victim user's account at an RP. In order to address this threat, we have developed *OAuthGuard*, an OAuth 2.0 and OpenID Connect vulnerability scanner and protector, that works with RPs using Google OAuth 2.0 and OpenID Connect services. It protects user security and privacy even when RPs do not implement OAuth 2.0 or OpenID Connect correctly. We used OAuthGuard to survey the 1000 top-ranked websites supporting Google sign-in for the possible presence of five OAuth 2.0 or OpenID Connect security and privacy vulnerabilities, of which one has not previously been described in the literature. Of the 137 sites in our study that employ Google Sign-in, 69 were found to suffer from at least one serious vulnerability. OAuthGuard was able to protect user security and privacy for 56 of these 69 RPs, and for the other 13 was able to warn users that they were using an insecure implementation.

CCS CONCEPTS

• **Security and privacy** → *Authorization*; *Security protocols*; *Web protocol security*; *Privacy protections*.

KEYWORDS

OAuth 2.0, OpenID Connect, Identity Management

ACM Reference Format:
Wanpeng Li, Chris J. Mitchell, and Thomas Chen. 2019. OAuthGuard: Protecting User Security and Privacy with OAuth 2.0 and OpenID Connect. In *5th Security Standardisation Research Workshop (SSR'19), November 11, 2019, London, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3338500.3360331

SSR'19, November 11, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6832-2/19/11…$15.00
https://doi.org/10.1145/3338500.3360331

1 INTRODUCTION

Since the OAuth 2.0 authorisation framework was published as a draft standard by the IETF at the end of 2012 [12], it has been adopted by a many websites worldwide as a means of providing single sign-on (SSO) services. 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 and provided to relying parties as required. There is a correspondingly rich infrastructure of identity providers (IdPs) providing identity services using OAuth 2.0. Indeed, some relying parties (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. Previous studies 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. This is yet another example of an apparently well-designed standard protocol being vulnerable to attack because of incorrect implementations; encouraging developers to be more careful is obviously not sufficient to ensure end user security, motivating the work described here that is designed to reduce the threat posed by such vulnerable implementations.

Researchers have developed a range of mitigations for RP developers, designed to help secure OAuth 2.0 and OpenID connect systems. However, none of this prior art (with one exception, discussed below) is aimed at protecting users who are (unwittingly) employing an insecure OAuth 2.0 or OpenID Connect implementation. To help close this gap, we have developed *OAuthGuard*, an OAuth 2.0 and OpenID Connect vulnerability scanner and protector, for use with RPs using Google OAuth 2.0 and OpenID Connect services. While we have focussed only on Google sign-in in the work described here, we believe that the same approach can be used to protect user security and privacy when working with other identity providers.

The main contributions of this paper are as follows.

(1) **A new vulnerability**. We identify a new privacy vulnerability which is present in a number of real-world websites.
(2) **OAuthGuard**. We describe the design and implementation of OAuthGuard (see Section 5), which provides real-time protection for users against vulnerabilities arising from poor implementations of OAuth 2.0 and OpenID Connect by web

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

sites (RPs) using the Google SSO service. Despite the ubiquity of these implementation vulnerabilities, this is the first practical help that has been offered to end users. We also outline how we addressed the challenges we faced in making the system work, including the trade-offs we made to ensure that OAuthGuard is compatible with all the RPs in our study.

(3) **A large-scale study**. We ran OAuthGuard on the top 1,000 websites from majestic.com[2] (Section 6). Key results from the study include finding at least one vulnerability in 69 of the 137 RPs that use Google Sign-in (Section 6). We further manually analysed the 109 RPs in the top 1,000 for which OAuthGuard did not detect a CSRF attack threat, and found that 25 of them are nevertheless vulnerable to a CSRF attack. Of the 69 RPs it found to be vulnerable, OAuthGuard is able to protect users against CSRF attacks for 48 of the 53 RPs (91%) which are vulnerable to such an attack; OAuthGuard was also able to upgrade the protocol from HTTP to HTTPS for 8 of the 13 RPs (62%) that erroneously use HTTP to transfer their OAuth 2.0 response. OAuthGuard identified nine RPs that leak user tokens to third party websites, either unintentionally or intentionally, and in total blocked 75 http requests leaking user tokens for these nine RPs. Finally, OAuthGuard generated a warning to users for 13 RPs that are vulnerable to an impersonation attack.

The remainder of this paper is structured as follows. Section 2 provides background on OAuth 2.0 and OpenID Connect. Section 3 describes related work. In Section 4, we describe the five vulnerabilities that OAuthGuard can detect and mitigate, one of which was not previously known. Section 5 specifies the infrastructure of OAuthGuard. In Section 6, we describe a case study on the Google sign-in security of 137 RPs, performed using OAuthGuard. Section 7 discusses the limitations and deployment of OAuthGuard. Section 8 concludes the paper.

2 BACKGROUND

2.1 OAuth 2.0

The OAuth 2.0 specification [12] 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).

[2]https://majestic.com/reports/majestic-million

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

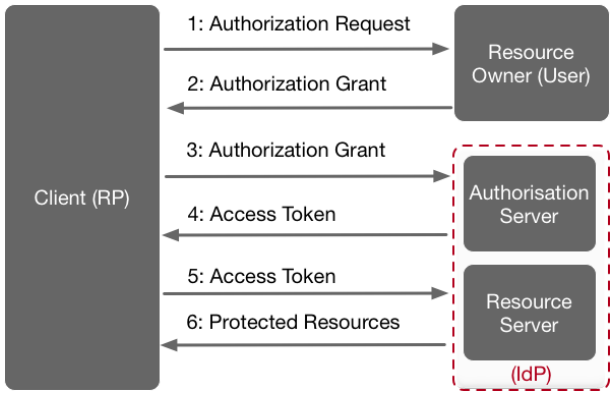

Figure 1: OAuth 2.0 Protocol Flow

The OAuth 2.0 framework defines four ways for RPs to obtain access tokens, namely *Authorization Code Grant*, *Implicit Grant*, *Resource Owner Password*, and *Client Credentials Grant*. In this paper we are only concerned with the *Authorization Code Grant* and *Implicit Grant* protocol flows. Note that, in the descriptions below, protocol parameters given in bold font are defined as required (i.e. mandatory) in the OAuth 2.0 Authorization Framework [12].

2.2 OpenID Connect

OpenID Connect 1.0 [21] 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)*. 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 supports three authentication flows [21], i.e. ways in which the system can operate, namely *Hybrid Flow*, *Authorization Code Flow* and *Implicit Flow*.

2.3 OAuth 2.0 used for SSO

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.

2.3.1 RP Registration. The RP must register with the IdP before it can use OAuth 2.0, during which the IdP gathers security-critical information about the RP, including the RP's redirect URI (***redirect_uri***), i.e. the URI to which the UA is redirected after the IdP has generated the authorization response and sent it to the RP via 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.

2.3.2 Authorization Code Grant. The OAuth 2.0 Authorization Code Grant is very similar to the OpenID Connect Authorization Code Flow; for simplicity, we only give the description of the OAuth 2.0 Authorization Code Grant. It relies on information established during the registration process, as described in Section 2.3.1. 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 ***client_id***, *client_secret* (if present), ***code*** 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.

2.3.3 Implicit Grant. The OAuth 2.0 Implicit Grant is very similar to the OpenID Connect Implicit Flow and Hybrid Flow; for simplicity, we only give the description of the OAuth 2.0 Implicit Grant. This flow has a similar sequence of steps to Authorization Code Grant. We specify below only those steps where the Implicit Grant flow differs from the Authorization Code Grant flow.

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=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 below); and *scope*, the scope of the requested permission.
6. IdP → UA → 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).

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.

3 ANALYSING THE SECURITY OF OAUTH 2.0

OAuth 2.0 has been analysed using formal methods. Pai et al. [20] confirmed a security issue described in the OAuth 2.0 Threat Model [19] using the Alloy Framework [13]. Chari et al. analysed OAuth 2.0 in the Universal Composability Security framework [6] and showed that OAuth 2.0 is secure if all the communications links are SSL-protected. Frostig and Slack [23] discovered a cross-site request forgery attack in the Implicit Grant flow of OAuth 2.0, using the Murphi framework [8]. Bansal et al. [1] analysed the security of OAuth 2.0 using the WebSpi [2] and ProVerif models [4]. Fett et al. [10] performed a formal security analysis of OpenID Connect. However, all this work is based on abstract models, and so delicate implementation details are ignored.

The security properties of real-world OAuth 2.0 implementations have also been examined. Wang et al. [25] examined deployed SSO systems, focusing on a logic flaw present in many such systems, including OpenID. In parallel, Sun and Beznosov [24] also studied deployed OAuth 2.0 systems. Later, Li and Mitchell [14] examined the security of deployed OAuth 2.0 systems providing services in Chinese. In parallel, Zhou and Evans [28] conducted a large scale study of the security of Facebook's OAuth 2.0 implementation. Chen et al. [7], and Shehab and Mohsen [22] have looked at the security of OAuth 2.0 implementations on mobile platforms. Li and Mitchell [15] conducted an empirical study of the security of the OpenID Connect-based SSO service provided by Google.

Fett et al. [9] proposed an IdP Mix-Up attack against RPs that support multiple IdPs. In their attack, a network attack is needed to modify the http or https messages generated by the RP in step 1 (see Section 2.3.2). Li and Mitchell [16] argued that the IdP Mix-Up attack would not be a genuine threat to the security of OAuth 2.0 if IdP implementations were strictly following the standard. Li and Mitchell [18] proposed a Partial Redirection URI Manipulation attack against RPs that support multiple IdPs. A 2016 study conducted by Yang et al. [27] revealed that 61% of 405 websites using OAuth 2.0 (chosen from the 500 top-ranked US and Chinese websites) did not implement CSRF countermeasures; even worse, for those RPs which support the state parameter, 55% of them are still vulnerable to CSRF attacks because of misuse/mishandling of the state parameter. They also disclosed four scenarios where the state parameter can be misused by RP developers. Most recently, Yang, Lau and Shi [26] conducted a large scale study of Android OAuth 2.0-based SSO systems. They found three previously unknown security flaws among first-tier identity providers and a large number of popular third party apps.

These practical studies suggest that in practice many real-world OAuth 2.0 and OpenID Connect systems contain security vulnerabilities, often because of implementation errors made by RP developers. In some cases these errors result from a lack of clear guidance from IdPs. Regardless of the causes, these vulnerabilities pose a significant threat to end users, and addressing this threat has motivated the work described in this paper.

In recent work conducted in parallel to that described here[3], Calzavara et al. [5] proposed WPSE, a web browser security monitor for OAuth 2.0. OAuthGuard and WPSE have some similar functionalities, e.g. being able to detect some common OAuth 2.0 attacks and provide mitigations for the user (see Table 1). One major advantage of OAuthGuard is that it is able to detect and provide protections for five common vulnerabilities for users, whereas WPSE can only detect three. A more detailed comparison of OAuthGuard with WPSE is provided in Section 7.

4 VULNERABILITIES

The design of OAuthGuard was motivated by the work of Li and Mitchell [15] and Yang et al. [27]. They examined the security of real-world OAuth 2.0 and OpenID Connect implementations and identified a range of vulnerabilities; they also proposed mitigations designed to enable RPs to make their OAuth 2.0 and OpenID connect systems secure. However, none of these mitigations help protect users who are employing an insecure OAuth 2.0 or OpenID Connect implementation. OAuthGuard is intended to help meet this need.

OAuthGuard can detect five classes of OAuth 2.0 or OpenID Connect vulnerabilities — four of these vulnerabilities have previously been discussed (see, for example, Li and Mitchell, [15]) — the only vulnerability not previously discussed is the 'privacy leak' issue, i.e. the fifth in the list below. Impersonation attacks only apply to the Implicit Grant flow, as described in Section 2.3.3; the other four attacks affect both flows, as defined in Section 2.3.

- **CSRF Attack Threat Detection**. CSRF attacks against the OAuth 2.0 *redirect_uri* [19] can allow an attacker to obtain authorization to access OAuth-protected resources without the consent of the user. Such an attack is possible for both the Authorization Code Grant Flow and the Implicit Grant Flow.
  One possible CSRF attack involves an attacker engaging with the target RP using its own device, and acquiring a *code*, *access_token* or *id_token* for the attacker's own resources. The attacker then aborts the redirect flow back to the RP, and, by means of a CSRF, instead causes the victim user to send the (attacker's) redirect flow back to the target RP. The target RP receives the redirect, fetches the (attacker's) attributes from the IdP, and associates the victim user's RP session with the attacker's resources accessed using the tokens. The victim user then accesses resources on behalf of the attacker. The impact of such an attack depends on the resources accessed. For example, the user might upload private data to the RP, thinking it is uploading information to its own profile, and this data will subsequently be available to the attacker. Alternatively, as described by Li and Mitchell [14], an attacker can use a CSRF attack to control a victim user's RP account without knowing the user's username and password.
  A 2016 study conducted by Yang et al. [27] revealed that 61% of 405 websites using OAuth 2.0 (chosen from the 500 top-ranked US and Chinese websites) did not implement the 'standard' CSRF countermeasures, notably including use of the state parameter; even worse, of those RPs which did support the state parameter, 55% were still vulnerable to CSRF attacks because of incorrect use of this parameter. They also described four scenarios in which the state parameter can be misused by RP developers. Given these variations in incorrect implementations, it is difficult to devise a universally applicable method to automatically detect a CSRF attack threat. As discussed in greater detail in the next section, the technique OAuthGuard uses to detect this threat is simply to check whether a *state* parameter is present in an OAuth 2.0 response. If no such parameter is present, then OAuthGuard reports that the RP is vulnerable to a CSRF attack. Thus OAuthGuard is not able to detect all RPs that are vulnerable to CSRF attacks, e.g. arising from incorrect use of the parameter.
- **Impersonation attacks**. This vulnerability stems from confusion about authentication and authorization. In OAuth 2.0, an *access_token* is intended for authorization purposes, and it is not tied to any specific RP. As the *access_token* is a bearer

[3]The source code of OAuthGuard[4] was first made available at github.com in February 2018.

token, it can be used by any RP that gains access to it. If an RP submits only an *access_token* to their Google sign-in endpoint, a malicious RP can submit a victim user's *access_token*, issued to the malicious RP by Google, to the RP's Google sign-in endpoint. The RP can use this *access_token* to get victim user information from Google, and then get full access to the victim user's account at the RP.

- **Authorization Flow Misuse**. As described in Section 2, OAuth 2.0 has four authorization flows and OpenID Connect has three authentication flows. RP developers must choose an appropriate flow and implement the OAuth 2.0 or OpenID Connect protocol correctly. According to the OAuth 2.0 and OpenID Connect standards, only a *code* should be submitted back to the RP's Google sign-in endpoint as evidence that the user has been authenticated. However, in reality, many RPs submit a combination of *code*, *access_token* and *id_token* back to their Google sign-in endpoint. As discussed by Li and Mitchell [15], this can lead to serious vulnerabilities.
- **Unsafe Token Transfers**. The main purpose of OAuth 2.0 and OpenID Connect is to allow an RP to access user information stored at an IdP without giving the RP the user's credentials for the IdP. This is achieved using a *code*, *access_token* or *id_token*. These tokens are vitally important, and hence they need to be protected when transferred between the RP and Google (e.g. using HTTPS). However, as has been discussed by Li and Mitchell [15], many RPs do not use HTTPS to protect the Google sign-in data transfers.
- **Privacy Leaks**. When a user uses the Google service to authenticate to an RP website, the user's *code*, *access_token* or *id_token*, retrieved by the RP from Google, should not be revealed to any other parties. We consider two cases where such a token may be revealed to a third party, which we refer to as a *privacy leak*; we further distinguish between *intentional privacy leaks* and *referer (unintentional) privacy leaks*, depending on whether the RP is aware of the leak or not. An unintentional privacy leak might occur when an RP includes third party content in its Google sign-in endpoint; an intentional privacy leak occurs when an RP deliberately sends user tokens to a third party.

5 OAUTHGUARD

OAuthGuard, a JavaScript Chrome browser extension, which is freely available via the Chrome web store[5], contains three main components: the *OAuth 2.0 Detector*, the *Vulnerability Analyser*, and the *Vulnerability Protector* (see Fig. 2). The OAuth 2.0 Detector monitors every HTTP request and extracts the OAuth 2.0 request or response metadata (see Listing 1 in Appendix) if the request is an OAuth 2.0 request or response. The Vulnerability Analyser analyses the OAuth 2.0 request and response reported by the OAuth 2.0 Detector, with the goal of identifying the possible vulnerabilities described in Section 4. Once a vulnerability has been detected by the Vulnerability Analyser, the Vulnerability Protector is triggered and appropriate mitigations are executed.

[5]https://chrome.google.com/webstore/detail/oauthguard/phamalogfapdjegegmghgcihhpabocfn

5.1 Vulnerability Mitigation

OAuthGuard protects against all five of the vulnerabilities in Section 4. We next describe how OAuthGuard mitigates these vulnerabilities.

- **CSRF Attack Protection**. OAuthGuard is designed to mitigate CSRF attacks even if the RP does not implement any countermeasures against such attacks (e.g. if it does not include a state parameter in the authorization response). To achieve this, OAuthGuard uses the CSRF countermeasures recently proposed by Li and Mitchell [17]. The idea is that, when used correctly, the referer header of the OAuth 2.0 response should point to either the RP domain or the IdP domain; this can be used to detect CSRF attacks. However, one limitation of this approach is that, if the *redirect_uri* of the RP uses HTTP, the referer header will be suppressed by the user agent [11] (i.e. the necessary domain information will be removed). Thus, OAuthGuard can only be used to mitigate CSRF attacks for RPs that use HTTPS to transfer their OAuth 2.0 response.
  To implement this mitigation, OAuthGuard first checks to see whether HTTPS has been used to transfer the OAuth 2.0 response; if not, it simply ignores the OAuth 2.0 response for compatibility reasons (so as not to block RPs using HTTP to transfer their OAuth 2.0 response). That is, OAuthGuard accepts all requests for RPs that use HTTP to deliver their OAuth 2.0 response. Otherwise, i.e. if HTTPS is used, it checks whether the HTTP referer header of the OAuth 2.0 response points to either the Google domain or the RP's domain; if not then OAuthGuard knows it is a CSRF attack against the RP's Google sign-in endpoint, drops the message, and notifies the user that it has blocked a CSRF attack attempt.
  This technique works with most RPs, although a few RPs use a proxy service (e.g. gigya) to implement support for Google sign-in, or use a domain other than the domain registered with Google as their Google sign-in endpoint. We whitelisted these RP domains so that, in such cases, OAuthGuard will not block the OAuth 2.0 response. In summary, OAuthGuard implements strict Referer validation [3] to protect against CSRF attacks for RPs that use HTTPS to deliver their OAuth 2.0 response; that is, OAuthGuard blocks all HTTPS requests whose Referer header has an incorrect value (e.g. an empty referer header).
- **Impersonation Attack Warning**. OAuthGuard is able to discover the RP's Google sign-in endpoint, and can also extract all three types of token from an OAuth 2.0 Response HTTP message. If only an *access_token* is submitted to the PR's Google sign-in endpoint, then OAuthGuard notifies the user that the RP's website might be vulnerable to an impersonation attack, and that the user is recommended to stop using Google sign-in with that RP.
- **Authorization Flow Misuse Warning**. OAuthGuard is able to detect an Authorization Flow Misuse vulnerability, as described in Section 4. However, it cannot determine which token is used by the RP to authenticate the user. As a result,

OAuthGuard does not implement any active mitigations, but simply generates a warning message to the user.

- **Unsafe Token Transfer Protection**. OAuthGuard is able to extract the protocol message used to transfer an OAuth 2.0 response. If HTTP is used, OAuthGuard attempts to redirect the response using HTTPS before the response leaves the user's browser. Of course, this measure only works if HTTPS is available at the RP.
- **Privacy Leak Protection**. As discussed in Section 4, if either of the two types of privacy leak is detected, OAuthGuard blocks the transfers and notifies the users that it has blocked an attempted privacy leak. Another possible mitigation would be to remove the tokens from the referer header instead of blocking the entire request. We chose to block the request because we want to discourage users from using the Google sign-in service for an RP that leaks user data to a third party.

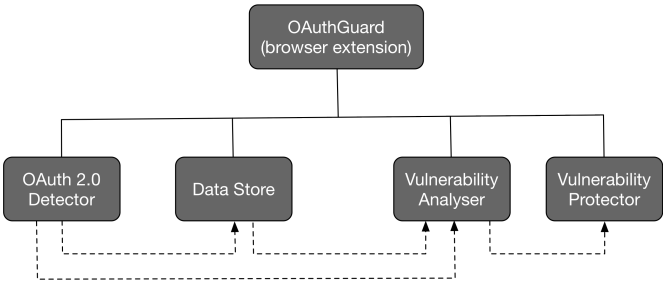

Figure 2: OAuthGuard Components

5.2 OAuth 2.0 Detector

Figure 3 shows the workflow of the OAuth 2.0 Detector. The OAuth Detector first examines every received HTTP request to check whether it is an OAuth 2.0 request; this involves scanning the url of the request for the keywords *oauth* and *redirect_uri*. If both keywords are present, the HTTP request is deemed to be an OAuth 2.0 request; the OAuth 2.0 Detector then extracts the OAuth 2.0 request metadata (see Listing 1 in Appendix) from the HTTP request and saves this metadata to the extension's localStorage[6] using RPDomain as key.

Otherwise, i.e. if these keywords are not both present, the OAuth 2.0 Detector scans the HTTP request for a *code*, *access_token* or *id_token*; if one of these tokens is identified, the HTTP request is deemed to be an OAuth 2.0 response. In this case the OAuth 2.0 Detector extracts the OAuth 2.0 response metadata from the HTTP request and saves it to localStorage using RPDomain as key.

5.3 Vulnerability Analyser

Figure 4 shows the workflow of the Vulnerability Analyser. Whenever an OAuth 2.0 response it reported by the OAuth 2.0 Detector, the Vulnerability Analyser is triggered. It first retrieves the OAuth 2.0 request (if any) using the RPDomain from the OAuth 2.0 response.

[6]https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API#localStorage

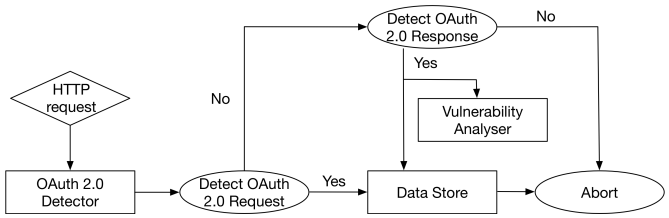

Figure 3: OAuthGuard OAuth 2.0 Detector Overview

- If no OAuth 2.0 request is retrieved, then the OAuth 2.0 response might be from an RP that is either using a proxy service (e.g. gigya) to implement Google sign-in, or a domain other than the domain registered with Google as its Google sign-in endpoint. Such RPs and proxy services are whitelisted in OAuthGuard. If neither of these two cases applies, i.e. the domain is not in the whitelist, the OAuth 2.0 response is deemed to be an intentional privacy leak.
- If an OAuth 2.0 request is retrieved, the Vulnerability Analyser uses the OAuth 2.0 request and response to identify possible vulnerabilities as follows.

(1) **Detection of CSRF Threats**. If a *state* parameter is not present in an OAuth 2.0 response, then OAuthGuard reports that the RP is vulnerable to a CSRF attack.
(2) **Detection of an Impersonation attack**. If only an *access_token* is detected in the OAuth 2.0 response, OAuthGard reports a possible Impersonation attack.
(3) **Detection of Authorization Flow Misuse**. If a combination of *code*, *access_token* and *id_token* is detected in the OAuth 2.0 response, OAuthGuard reports an Authorization Flow Misuse.
(4) **Detection of Unsafe Token Transfer**. OAuthGuard checks whether the RP is using HTTP or HTTPS to transfer the OAuth 2.0 response. If HTTP is detected, it reports an Unsafe Token Transfer threat.
(5) **Detection of Privacy Leaks**. OAuthGuard uses a specific Referer Leakage Detection module to detect Unintentional Privacy Leak vulnerabilities. This module first looks for a *code*, *access_token* or *id_token* in a referer header (if present in an HTTP request). If any of these tokens are identified, the module extracts the domains of the referer header and the HTTP request, and checks whether they are the same. If not, it reports that an Unintentional Privacy Leak vulnerability has been detected.

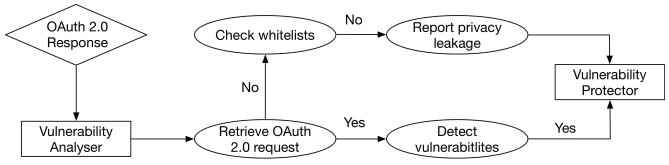

Figure 4: OAuthGuard Vulnerability Analyser Overview

5.4 Vulnerability Protector

Depending on which type of vulnerability has been reported by the Vulnerability Analyser, the Vulnerability Protector executes the following actions:

- it blocks an HTTP request if a Privacy Leak vulnerability is detected;
- it tries to redirect an OAuth 2.0 response using HTTPS if an Unsafe Token Transfer vulnerability is detected;
- it warns the user if the RP is vulnerable to an Impersonation attack;
- it blocks an OAuth 2.0 response if a CSRF attack is detected.

6 A CASE STUDY

We used OAuthGuard to help understand the degree to which RPs using the Google OAuth service are vulnerable to known threats. This involved manually running OAuthGuard against the top-ranked 1,000 websites from majestic.com[7] as of 12 December 2017. 137 of these 1,000 websites support Google sign-in. We used a Macbook Pro (late 2013) running macOS High Sierra 10.13.1 and Chrome browser version 63.0.3239.132.

As discussed earlier, OAuthGuard detects CSRF attack threats by checking whether a state parameter is present in an OAuth 2.0 response. To supplement the automated threat detection, we also manually looked through all the RPs for which a CSRF threat was not reported by OAuthGuard to discover whether these RPs are actually vulnerable to a CSRF attack. While OAuthGuard reported CSRF attack threats for 28 of the 137 RPs, we manually identified a further 25 that are vulnerable to CSRF attacks.

Figure 5 divides the 1,000 sites we examined into groups of 100 (starting with those ranked highest), and for each group of 100 indicates (a) the percentage supporting Google SSO, and (b) of those that do support Google SSO which were found to possess at least one vulnerability. The graph suggests that the more popular sites are a little more likely to support Google sign-in and also slightly more likely to possess implementation vulnerabilities. Whilst the former result is not surprising, the latter is somewhat alarming, since one might expect popular sites to have more resources to devote to ensuring site security.

Unsurprisingly, we got similar results to those of the 2016 Yang et al. study [27]. We can summarise our findings as follows (noting that in each case they apply to the 137 RPs that support Google SSO).

- 53 RPs (39%) are vulnerable to a CSRF attack against their OAuth 2.0 *redirect_uri* endpoint;
- 21 RPs (15%) misuse authorization flows, of which 13 are vulnerable to an impersonation attack;
- 9 RPs leak tokens though referer headers; of these, two explicitly send user tokens to third party websites;
- 13 RPs did not implement https to protect the transfer of user tokens.
- A total of 69 RPs (50%) possessed at least one vulnerability of the types discussed in Section 4.

To illustrate the potential risks in real-world websites, we next give an example of vulnerabilities detected by OAuthGuard. Ranked

[7] https://majestic.com/reports/majestic-million

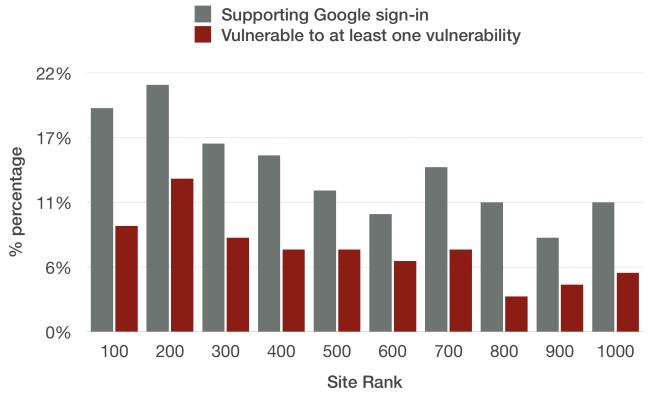

Figure 5: Google Sign-in integration results by site rank

61st on the list, issuu.com is the world's largest electronic publishing platform. OAuthGuard detected that issuu.com is vulnerable to Referer token leaks, CSRF attacks, as well as Impersonation attacks. After the user has used Google sign-in to log in to issuu.com, it sends an *access_token* directly to its Google sign-in endpoint without implementing any CSRF countermeasures. Moreover, its Google sign-in endpoint contains content (e.g. gif and JavaScript files) from third-party websites, including optimizely.com, bing.com, licdn.com, and quantserver.com. When the browser retrieves this content, it sends the *access_token* in the Referer header to these third-party sites. The permissions issuu.com requests from Google include access to the user's profile information and email address, so that possession of the *access_token* gives access to this information without user consent.

7 DISCUSSION

7.1 Implementation challenges

The main design goal of OAuthGuard is to protect user security and privacy when using Google sign-in. Since each RP implements its own Google sign-in endpoint, it is hard to devise a solution that will work for all RPs. We next describe some of the difficulties we encountered in designing OAuthGuard, and the trade-offs we made to enable it to operate.

CSRF Protection As discussed in Section 5.1, OAuthGuard implements strict referer validation [3] to protect users against CSRF attacks for RPs using HTTPS to deliver an OAuth 2.0 response. This works as in this case the referer header domain should be either the IdP's domain or the RP's domain. However, some RPs use a proxy service (e.g. gigya) to implement Google sign-in, or use a domain other than the domain registered with Google as their Google sign-in endpoint. For example, chicagotribune.com registers https://signin.chicagotribune.com/GS/GSLogin.aspx? as its Google sign-in *redirect_uri*, but uses the domain https://ssor.tribdss.com/assets/sso_popup.html to display its Google sign-in button. If no other checks were implemented, OAuthGuard would incorrectly report a CSRF attack on chicagotribune.com, as the referer header domain tribdss.com does not equal either google.com or chicagotribune.com. In order to make OAuthGuard compatible

with such RPs, we chose to whitelist all such domain names in the OAuthGuard source code. In total we whitelisted 11 domains (8%) from the set of 137 RPs. Even given these difficulties, OAuthGuard can protect user security for 48 of the 53 RPs (91%) which we found to be vulnerable to CSRF attacks. The other five RPs use HTTP to deliver the OAuth 2.0 response, and as a result the OAuthGuard CSRF countermeasure does not work.

Privacy Protection As described in Section 6, OAuthGuard identified nine RPs that leak user tokens to third party websites, either intentionally or unintentionally. OAuthGuard blocked all the token-leaking HTTP requests for these nine RPs. In total, OAuthGuard blocked 75 HTTP requests that leak user tokens for these nine RPs. Blocking third party requests that leak tokens might prevent users from using Google sign-in to log in to the relevant RPs. However, we decided to block these requests as it will discourage users from using insecure Google sign-in implementations; most importantly it prevents unauthorised token disclosure, which could have a serious negative impact on user privacy.

Impersonation Attack Warnings It is up to the RP to decide which types of tokens it should submit back to its Google sign-in endpoint. If tokens are used inappropriately, the only thing OAuthGuard can do is to warn users that an RP is vulnerable to an impersonation attack, and suggest that users should not employ Google sign-in at these RPs.

HTTPS Upgrade If OAuthGuard detects an OAuth 2.0 response transferred using HTTP, it attempts to redirect it using HTTPS. Of course, this protection only works with RPs that implement HTTPS on their website. In our study, OAuthGuard was able to upgrade the protocol to HTTPS for 8 of the 13 RPs (62%) that use HTTP to transfer an OAuth 2.0 response (in each case the HTTPS upgrade resulted in a successful login). For RPs not supporting HTTPS, OAuthGuard will by default make the Google sign-in service unavailable; to give the user flexibility in which sites they are able to use, OAuthGuard enables users to turn off the HTTPS upgrade function.

7.2 Comparison with WPSE

The OAuthGuard approach to protecting against CSRF attacks is more efficient than that employed by WPSE, because WPSE blocks any OAuth 2.0 response which does not contain a state parameter (see [5], Section 4.1.1); Yang et al. [27] found that 61% of 405 websites using OAuth 2.0 (chosen from the 500 top-ranked US and Chinese websites) did not implement CSRF countermeasures; this means that WPSE would block the OAuth 2.0 response for all these websites. Our approach to mitigating CSRF attacks is to use the CSRF countermeasures recently proposed by Li and Mitchell [17]. These build on the observation that, when used correctly, the referer header of the OAuth 2.0 response should point to either the RP domain or the IdP domain; this can be used to detect CSRF attacks. Using this approach, OAuthGuard can protect users against CSRF attacks even when RPs do not implement any CSRF countermeasures (including the 61% of RPs in Yang's study [27]).

7.3 Limitations

OAuthGuard detects vulnerabilities by analysing HTTP messages. However, this approach cannot be used to detect vulnerabilities that can only be found by deep server-side application scanning.

	OAuthGuard	WPSE
CSRF Attacks	x	x
Impersonation Attacks	x	
Privacy Leaks	x	x
Authorization Flow Misuse	x	
Unsafe Token Transfers	x	x
Mix-IdP Attack		x

Table 1: Comparison between OAuthGuard and WPSE

For example, the IdP Mix-Up attack revealed by Fett et al. [9] could be detected by RP developers using program analysis techniques, but cannot be detected by an external tool with no awareness of the site's implementation details or internal state. Also, since OAuthGuard blocks HTTP messages that leak user tokens to third-party websites, it could make the Google sign-in service unavailable for some RPs.

7.4 Disclosure

We reported our findings to seven RPs that are vulnerable to the impersonation attacks described in Section 4. We contacted them either by email or by submitting a website form. The responses were disappointing. The lack of response is perhaps explained by the fact that the vulnerabilities we identified are primarily in consumer-oriented RP sites, who may not have dedicated security teams or ways of effectively addressing security issues. So far we have only received responses from two RPs in which they acknowledge our reports and are working on fixing the vulnerability; of course, we may receive more responses in the future — we certainly hope so.

7.5 Testing and Deployment

OAuthGuard has been informally tested by the authors and their colleagues; no significant usability issues have so far been detected. Of course, this is hardly a thorough test, consistent with the fact that OAuthGuard is primarily intended as a prototype and proof-of-concept. If it is to be very widely deployed, then further development work will be required to ensure that the whitelist is expanded to cover all well-used sites that would otherwise fail the checks. Nonetheless, our informal tests reveal that OAuthGuard as it is offers an enhanced level of user security and privacy protection.

OAuthGuard is freely available via the Chrome web store[8], and the source code is available at github[9]. We hope that researchers and developers can help to further develop the tool, as well as enabling support for other OAuth 2.0 systems, such as those of Facebook and Microsoft.

Apart from end user deployment, OAuthGuard can also be used by RP developers to check Google sign-in implementations. After the usual development testing, and before launching support for Google sign in, developers could usefully run OAuthGuard to detect any residual vulnerabilities.

[8] https://chrome.google.com/webstore/detail/oauthguard/phamalogfapdjegegmghgcihhpabocfn
[9] https://github.com/oauthguard/OAuthGuard

8 CONCLUSION

We have described OAuthGuard, an OAuth 2.0 and OpenID Connect vulnerability scanner and protector for RPs using Google OAuth 2.0 and OpenID Connect. It can be used to protect user security and privacy even if RPs have not implemented OAuth 2.0 or OpenID Connect correctly. We used OAuthGuard to check the security and privacy properties of the 1,000 top-ranked websites supporting Google sign-in; in particular OAuthGuard checked for five OAuth 2.0 or OpenID Connect vulnerabilities. Of the 137 sites (from the 1000) that employ Google Sign-in, 69 were found to suffer from at least one serious vulnerability in their implementation of OAuth 2.0 or OpenID Connect. OAuthGuard is able to protect user security and privacy for 56 of these 69 vulnerable RPs, and provide a warning to users of the other 13.

REFERENCES

[1] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. 2014. Discovering concrete attacks on website authorization by formal analysis. *Journal of Computer Security* 22, 4 (2014), 601–657. https://doi.org/10.3233/JCS-140503

[2] Chetan Bansal, Karthikeyan Bhargavan, and S. Maffeis. 2011. WebSpi and web application models. (2011). http://prosecco.gforge.inria.fr/webspi/CSF/.

[3] Adam Barth, Collin Jackson, and John C Mitchell. 2008. Robust defenses for cross-site request forgery. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, Peng Ning, Paul F. Syverson, and Somesh Jha (Eds.). ACM, 75–88.

[4] Bruno Blanchet and Ben Smyth. [n.d.]. ProVerif: Cryptographic protocol verifier in the formal model. ([n. d.]). http://prosecco.gforge.inria.fr/personal/bblanche/proverif/.

[5] Stefano Calzavara, Riccardo Focardi, Matteo Maffei, Clara Schneidewind, Marco Squarcina, and Mauro Tempesta. 2018. WPSE: Fortifying Web Protocols via Browser-Side Security Monitoring. In *27th USENIX Security Symposium (USENIX Security 18)*. 1493–1510.

[6] Suresh Chari, Charanjit S Jutla, and Arnab Roy. 2011. Universally Composable Security Analysis of OAuth v2.0. *IACR Cryptology ePrint Archive* 2011 (2011), 526.

[7] Eric Y. Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. 2014. OAuth Demystified for Mobile Application Developers. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM, 892–903. https://doi.org/10.1145/2660267.2660323

[8] David L Dill. 1996. The Mur*phi* Verification System. In *Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, July 31 – August 3, 1996, Proceedings (Lecture Notes in Computer Science)*, Rajeev Alur and Thomas A. Henzinger (Eds.), Vol. 1102. Springer, 390–393.

[9] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2016. A Comprehensive Formal Security Analysis of OAuth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 1204–1215. https://doi.org/10.1145/2976749.2978385

[10] Daniel Fett, Ralf Küsters, and Guido Schmitz. 2017. The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines. *arXiv preprint arXiv:1704.08539* (2017).

[11] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. 1999. RFC 2616: Hypertext transfer protocol–HTTP/1.1. https://tools.ietf.org/html/rfc2616.

[12] Dick Hardt (editor). 2012. RFC 6749: The OAuth 2.0 Authorization Framework. (October 2012). http://tools.ietf.org/html/rfc6749.

[13] Daniel Jackson. 2010. Alloy 4.1. (2010). http://alloy.mit.edu/community/.

[14] Wanpeng Li and Chris J. Mitchell. 2014. Security Issues in OAuth 2.0 SSO Implementations. In *Information Security — 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings (Lecture Notes in Computer Science)*, Sherman S. M. Chow, Jan Camenisch, Lucas Chi Kwong Hui, and Siu-Ming Yiu (Eds.), Vol. 8783. Springer, 529–541. https://doi.org/10.1007/978-3-319-13257-0_34

[15] Wanpeng Li and Chris J. Mitchell. 2016. Analysing the Security of Google's Implementation of OpenID Connect. In *Detection of Intrusions and Malware, and Vulnerability Assessment — 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings (Lecture Notes in Computer Science)*, Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez (Eds.), Vol. 9721. Springer, 357–376. https://doi.org/10.1007/978-3-319-40667-1_18

[16] Wanpeng Li and Chris J. Mitchell. 2016. Does the IdP Mix-Up attack really work? (2016). https://infsec.uni-trier.de/download/oauth-workshop-2016/OSW2016_paper_1.pdf.

[17] Wanpeng Li, Chris J. Mitchell, and Thomas Chen. 2018. Mitigating CSRF attacks on OAuth 2.0 Systems. In *16th Annual Conference on Privacy, Security and Trust, PST 2018, Belfast, Northern Ireland, Uk, August 28-30, 2018*, Kieran McLaughlin, Ali A. Ghorbani, Sakir Sezer, Rongxing Lu, Liqun Chen, Robert H. Deng, Paul Miller, Stephen Marsh, and Jason Nurse (Eds.). IEEE, 1–5. https://doi.org/10.1109/PST.2018.8514180

[18] Wanpeng Li, Chris J. Mitchell, and Thomas Chen. 2018. Your Code Is My Code: Exploiting a Common Weakness in OAuth 2.0 Implementations. In *Security Protocols XXVI - 26th International Workshop, Cambridge, UK, March 19-21, 2018, Revised Selected Papers (Lecture Notes in Computer Science)*, Vashek Matyás, Petr Svenda, Frank Stajano, Bruce Christianson, and Jonathan Anderson (Eds.), Vol. 11286. Springer, 24–41. https://doi.org/10.1007/978-3-030-03251-7_3

[19] Torsten Lodderstedt, Mark McGloin, and Phil Hunt. 2013. RFC 6819: OAuth 2.0 Threat Model and Security Considerations. (2013). http://tools.ietf.org/html/rfc6819.

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

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

[22] Mohamed Shehab and Fadi Mohsen. 2014. Securing OAuth implementations in smart phones. In *Fourth ACM Conference on Data and Application Security and Privacy, CODASPY'14, San Antonio, TX, USA — March 03 - 05, 2014*, Elisa Bertino, Ravi S. Sandhu, and Jaehong Park (Eds.). ACM, 167–170. https://doi.org/10.1145/2557547.2557588

[23] Quinn Slack and Roy Frostig. 2011. Murphi Analysis of OAuth 2.0 Implicit Grant Flow. (2011). http://www.stanford.edu/class/cs259/WWW11/.

[24] San-Tsai Sun and Konstantin Beznosov. 2012. The Devil is in the (Implementation) details: An Empirical Analysis of OAuth SSO Systems. In *the ACM Conference on Computer and Communications Security, CCS '12, Raleigh, NC, USA, October 16-18, 2012*, Ting Yu, George Danezis, and Virgil D. Gligor (Eds.). ACM, 378–390.

[25] Rui Wang, Shuo Chen, and XiaoFeng Wang. 2012. 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*. IEEE Computer Society, 365–379.

[26] Ronghai Yang, Wing Cheong Lau, and Shangcheng Shi. 2017. Breaking and Fixing Mobile App Authentication with OAuth2.0-based Protocols. In *Applied Cryptography and Network Security - 15th International Conference, ACNS 2017, Kanazawa, Japan, July 10-12, 2017, Proceedings (Lecture Notes in Computer Science)*, Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi (Eds.), Vol. 10355. Springer, 313–335. https://doi.org/10.1007/978-3-319-61204-1_16

[27] Ronghai Yang, Guanchen Li, Wing Cheong Lau, Kehuan Zhang, and Pili Hu. 2016. Model-based Security Testing: An Empirical Study on OAuth 2.0 Implementations. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 – June 3, 2016*, Xiaofeng Chen, XiaoFeng Wang, and Xinyi Huang (Eds.). ACM, 651–662. https://doi.org/10.1145/2897845.2897874

[28] Yuchen Zhou and David Evans. 2014. SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 495–510. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/zhou

THE APPENDIX

```
// An OAuth 2.0 request metadata
// the requestURL and state in the request are trimmed for readability
IdP: "https://accounts.google.com"
IdPProtocol: "https:"
RP: "www.dropbox.com"
RPDomain: "dropbox.com"
RPProtocol: "https:"
clientID: "801668726815.apps.googleusercontent.com"
origin: null
redirectURI: "https://www.dropbox.com/google/authcallback"
referer: "https://www.dropbox.com/"
requestURL: "https://accounts.google.com/o/oauth2/auth"
responseType: "code"
scope: "https://www.google.com/m8/feeds email profile"
```

```
state: "ABAm_Lg53XmdhkeMTOmFKH5RULv2egJHsRXl9KHhp6Tazub"

// an OAuth 2.0 response metadata
// the referer, responseURL and state in the response are
     trimmed for readability
IdP: "google.com"
RPDomain: "dropbox.com"
RPHost: "www.dropbox.com"
RPProtocol: "https:"
access_token: ""
code: "4/gKfVUfaN5n-9tmo3RYnYActwrYWIXAwnsXRA7fcUl6E"
cookie: ""
data: ""
id_token: ""
method: "GET"
referer: "https://accounts.google.com/signin/oauth/
    oauthchooseaccount?"
responseURL: "https://www.dropbox.com/google/authcallback
    ?"
state: "ABAm_Lg53XmdhkeMTOmFKH5RULv2egJHsRXl9KHhp6Tazub"
```

Listing 1: The OAuth 2.0 Request and Response metadata