More Guidelines Than Rules: CSRF Vulnerabilities from Noncompliant OAuth 2.0 Implementations

Ethan Shernan[1], Henry Carter[1], Dave Tian[2], Patrick Traynor[2], and Kevin Butler[2]

[1] Georgia Institute of Technology
eshernan3@mail.gatech.edu, carterh@gatech.edu
[2] University of Florida
daveti@ufl.edu, traynor@cise.ufl.edu, butler@cise.ufl.edu

Abstract. OAuth 2.0 provides an open framework for the authorization of users across the web. While the standard enumerates mandatory security protections for a variety of attacks, many embodiments of this standard allow these protections to be optionally implemented. In this paper, we analyze the extent to which one particularly dangerous vulnerability, Cross Site Request Forgery, exists in real-world deployments. We crawl the Alexa Top 10,000 domains, and conservatively identify that 25% of websites using OAuth appear vulnerable to CSRF attacks. We then perform an in-depth analysis of four high-profile case studies, which reveal not only weaknesses in sample code provided in SDKs, but also inconsistent implementation of protections among services provided by the same company. From these data points, we argue that protection against known and sometimes subtle security vulnerabilities can not simply be thrust upon developers as an option, but instead must be strongly enforced by Identity Providers before allowing web applications to connect.

1 Introduction

One of the most significant recent revolutions in web applications is the ability to combine mechanisms and data from disparate domains. So transformative is this change that a number of protocols have been proposed to encourage and facilitate such interaction. None of these protocols have been as widely adopted as OAuth 2.0 [18], an IETF standard for delegated authorization employed as a means of providing secure access to application data and user accounts. Through the use of this highly flexible standard, a wide range of domains can now build extensible tools far surpassing OAuth 2.0's original mandate, including authentication and single-sign on services.

Recent research has shown that noncompliance with the standard has led to vulnerabilities in real-world deployments. For instance, many mobile application developers have struggled to develop secure implementations of OAuth 2.0 because their use in non-web applications conflicts with assumptions made in the standard [10]. Other researchers have demonstrated that a range of implementations fail to correctly implement or apply features, leaving them similarly insecure [39, 29]. In particular, [39] considers a range of OAuth implementation vulnerabilities and provides recommended mitigations for each. However, these studies do not explore the root causes of these vulnerabilities in depth, and as a result advise mitigations that are ineffective in practice.

In this paper, we consider the extent to which a specific documented vulnerability is embodied in real-world implementations. Specifically, the OAuth 2.0 standard explicitly identifies the potential for Cross Site Request Forgery (CSRF) attacks, which may force an unsuspecting user to perform an "authorized" action without their knowledge (e.g., transmit financial information to a malicious third-party, modify sensitive file contents, etc). The OAuth 2.0 standard is *absolutely unambiguous* about the danger that such vulnerabilities pose, and notes that both client and server "MUST implement CSRF protection" [18]. Unfortunately, these specific warnings are not heeded by many deployments of this protocol, leaving the implementation of protections as either a suggested task or simply an unmentioned burden for web application developers. As our experiments demonstrate, this lack of strict adherence to the standard leaves a significant portion of OAuth-enabled web applications vulnerable to CSRF attacks.

We make the following contributions:

- **Analysis of Adherence to Standard:** We evaluate 13 of the most popular OAuth 2.0 Identity Providers, including Facebook, Google, and Microsoft. We show that only four out of thirteen such providers force CSRF protections as part of their APIs, leaving the remaining nine to merely suggest or simply not mention that protection is necessary.
- **Measurement Study:** We develop a crawler and perform a depth-two analysis of the Alexa Top 10,000 websites [1], visiting more than 5.6 million URLs during our evaluation. Of those we detect as offering OAuth 2.0 services, we show that 25% do not implement standard CSRF protections and appear vulnerable to attack.
- **Analysis of Case Studies:** We provide deeper analysis into four different specific instances in which vulnerable uses of OAuth 2.0 APIs are identified. We show that mistakes are the result of a range of factors, ranging from vulnerable sample code published by Identity Providers to inconsistencies between APIs across a large company. These contributing factors all point to Identity Providers as the logical agents to effect change by mandating compliant implementations.

From these observations, we argue that expecting web application developers to understand and implement subtle web security protection mechanisms is a design choice doomed to failure. Specifically, when a known vulnerability identified in the standard can be fixed with a known remediation technique that does not impact performance, it must be a mandatory component of any embodiment of that standard.

The remainder of this paper is organized as follows: Section 2 provides background information on the OAuth 2.0 protocol, and discusses CSRF attacks; Section 3 applies CSRF to OAuth, and demonstrates how the use of a challenge-response string can prevent such attacks; Section 4 offers the results of our web crawl; Section 5 details four case studies; Section 6 provides further insights and discussion; Section 7 examines related work; and Section 8 provides concluding remarks.

2 Background

At its core, the OAuth protocol allows a user to grant a web application authorized access to his data on a different application [28]. Specifically, it allows a user to grant a Relying Party (RP) the ability to perform a set of operations on an account with an

Fig. 1. Examples of popular single-sign on services that use OAuth. The presence of these buttons on a web page indicate that the domain has implemented OAuth as a single-sign on protocol.

Identity Provider (IdP). The set of permissible actions are defined by the IdP in an API. A very common application of distributing authorization via OAuth is Single-Sign On (SSO), which allows users to connect to multiple web services (RPs) using one set of credentials from an IdP [21] (e.g., logging into Urbanspoon with Facebook). However, unlike dedicated SSO protocols such as OpenID [42] (which is built on top of OAuth 2.0), OAuth is capable of functioning as a general-purpose authorization protocol, making the impact of the protocol much more broad than a dedicated SSO application. OAuth 2.0 first authenticates users with a username and password, followed by consent given at a permissions screen for the RP to access the identity stored at the IdP [18]. The protocol relies on the use of access tokens that expire over time to authorize requests made by an RP to access data provided by an IdP on behalf of a user [25]. OAuth 2.0 has multiple forms of authentication flows to allow for use in different scenarios such as native, web, and mobile applications [18]. The significance of the OAuth 2.0 protocol is clear, as research has shown that a growing number of web applications are relying heavily on a small set of IdPs, many of which (e.g., Facebook, Google, Twitter) employ the OAuth 2.0 protocol [43] (Figure 1). In addition, new research is proposing the use of OAuth 2.0 as a critical underlying component for authentication [33, 12].

2.1 Authorization Code Flow

The OAuth 2.0 standard has multiple authorization protocols to allow for flexibility in implementation for both IdPs and RPs. These different "grant types" are meant to enable easy implementations for a wide variety of scenarios. For example: the implicit grant flow is tailored toward client side web applications [18]. This flow involves making a single HTTP request to an IdP's authorization server and retrieving a bearer token that can be used to make subsequent authorized API calls for a specific user's data.

This paper focuses specifically on a vulnerability contained within the authorization code flow of the OAuth 2.0 standard [18]. This flow is widely implemented by IdPs and is designed for web server applications. We describe the authorization code flow when used by a client application (relying party) *MyApp* to access user data provided by an IdP (identity provider) *Bookface*. A graphical representation can be found in Figure 2a.

1. *MyApp* sends an HTTP GET request to *Bookface's* OAuth 2.0 endpoint containing - at minimum - the following parameters, specified in the OAuth 2.0 RFC [18]:
   (a) `client_id`: a value that determines the specific RP application utilizing the IdP's services (*MyApp* in this example)
   (b) `scope`: an IdP defined set of descriptors that describe what data the RP has access to on behalf of its users. The RP can set a combination of these descriptors to describe the scope of authorization a user must consent to.

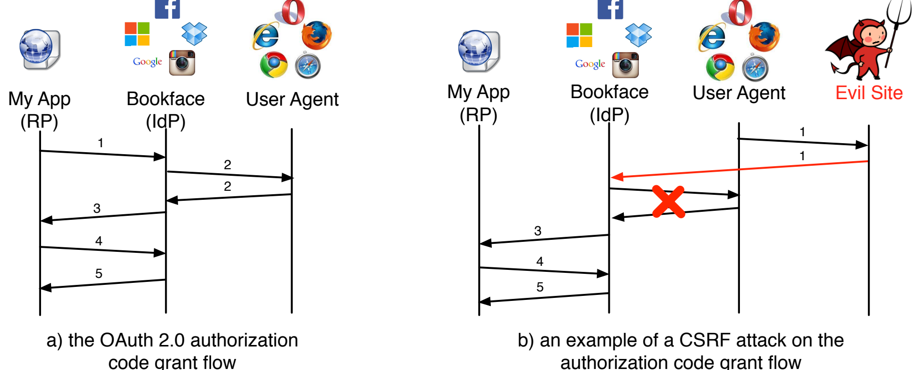


Fig. 2. The OAuth authorization code grant flow compared to an OAuth CSRF attack. In a normal execution, the user initiates authorization (1), grants the RP access to his IdP account (2), then forwards this approval to the RP (3). The protocol concludes with the RP authenticating and receiving and access token (4-5). In the CSRF attack, the adversary tricks the user into initiating the protocol. If the user has recently authenticated to the IdP, step 2 of the grant flow is skipped and the user agent automatically forwards the request embedded in the evil site to the IdP.

   (c) `grant_type`: set to `code` for the authorization code grant
   (d) `redirect_uri`: the URI the IdP will redirect the *user agent* (a web browser or similar application) to where it will receive an *authorization code*. This should be a URI that the RP application can handle requests from.
2. *Bookface* responds to this request by prompting the current user of *MyApp* for their *Bookface* credentials and consent to allow *MyApp* access to their data based on the scope of the authorization. This is done by redirecting the user agent to a login page specific to the IdP.
3. If the user provides consent, *Bookface* sends an *authorization code* to the `redirect_uri` specified in Step 2.
4. *MyApp* receives the code at the `redirect_uri` and makes a POST request to *Bookface* to exchange this code for the final *authorization token* that can be used to make authorized API calls to *Bookface*. This request should include - at minimum - the following parameters in its POST data [18]:
   (a) `client_id`: the same value as used in the request in step 1
   (b) `client_secret`: a password to authenticate *MyApp*
   (c) `redirect_uri`: the URI to be redirected to with the *authorization token*
   (d) `grant_type`: set to `code` for the authorization code grant
   (e) `code`: the authorization code received with the result of step 3
5. *Bookface* exchanges the authorization code with an *authorization token* and delivers it to the redirect_uri.

The *authorization token* can now be used by *MyApp* to make authorized and authenticated requests to *Bookface's* APIs to retrieve specific user data. This authorization protocol differs from the OAuth 2.0 implicit grant flow in that it grants an authorization code before the authorization token, while the implicit grant flow directly grants the

authorization token in step 3. This ensures that only the RP authenticated in step 4 is granted access to the API. By contrast, the implicit grant flow allows *any* party holding the authorization token to access the API. It should be noted that in both steps 2 and 5 there are optional parameters that may be included according to the standard, but not required for implementations of OAuth 2.0 [18]. One parameter that is critical for CSRF prevention is the `state` parameter. This parameter can hold any optional state required by the RP application. The value of this parameter is then echoed back to the RP by the IdP in any response in which it was included in the initial HTTP request.

2.2 Cross Site Request Forgery

A cross site request forgery attack (CSRF) is a common form of *confused deputy* attack where a user's previous session data is used by an attacker to make a malicious request on behalf of the victim [19, 27]. CSRF attacks involve an attacker performing an HTTP request on behalf of a victim who logs into a website using stored session data (usually a cookie). CSRF attacks leverage the commonly used paradigm of storing session data about a user to make HTTP requests as if the victim actually authorized the request [26]. CSRF vulnerabilities were long ignored by web developers, and prominent websites such as The New York Times and YouTube have had significant CSRF vulnerabilities in the past [49]. While developers have historically considered this a low-risk vulnerability, the security community has considered the attack as a serious threat. Allowing non-expert software developers to make security assumptions can potentially lead to unexpected system behavior and vulnerabilities, so understanding CSRF and developing better means for enforcement remains a critical research area.

The malicious URL used in a CSRF attack is often embedded within an `<img>` HTML tag on an innocent looking web page so that a web browser will automatically perform a GET request to the URL without user consent. However, an attack requiring POST data can also be performed by tricking a victim into submitting form data to a maliciously formed URL on an honest web site. If a user has previously logged into the honest web application the attacker is targeting, a session cookie is automatically sent along with the malicious HTTP request, thereby authenticating the user. If the user is authorized to make the request, the honest web application processes the request as normal, even if the victim's user agent has actually just been tricked into making it.

A concrete example of this attack is described as follows: Alice logs into her bank website `bank.example.com`. This site allows users to make transfers by performing a GET request to the following endpoint: `https://bank.example.com/transfer?src=alice&dest=bob&amt=500`. The `amt` parameter is the amount to transfer, the `src` and `dest` parameters are the source and destination of the transfer.

An attacker, Eve, can perform a CSRF on Alice by embedding a malicious form of the previous URL in an `<img>` tag on her website (e.g., `<img src="https://bank.example.com/transfer?src=alice&dest=eve&amt=1000">`)

If Eve can get Alice to visit her website, Alice's web browser will automatically perform a GET request to the transfer endpoint at `bank.example.com`. If Alice has previously logged in with `bank.example.com`, her cookie will automatically be sent along with this request that authenticates her, and as long as she is authorized to make a transfer, `bank.example.com` will process the request as normal.

If `bank.example.com` instead had the transfer operation as a POST request, a similar attack could be made but with Eve tricking Alice into submitting a form that points to the transfer endpoint of `bank.example.com`.

CSRF attacks can be prevented in different ways [6, 31]. One simple protection scheme uses a randomly generated token that synchronizes a specific request with a specific user session [2]. Requests are disallowed unless a token is included and matches the user's current session token as remembered by the server. In the above example, `bank.example.com` could include on their website a hidden HTML field that includes a token that is randomly generated each time Alice visits their website. This token can then be included along with every HTTP request and identifies Alice's session with the sequence of requests. When Alice visits Eve's website, the malicious HTTP request is made as before. However this request will not include the same synchronizing token that `bank.example.com` now requires, and the request is rejected even if Alice was previously authenticated. As long as the token cannot be guessed by Eve, a CSRF attack will not be possible [49].

3 Attack

In this section, we describe a CSRF attack that can be launched on an incorrectly-implemented OAuth 2.0 connection. This attack is well-known and discussed in the OAuth RFC, but does not appear in a majority of live developer documentation. We close the section with a discussion of documented mitigation techniques and other mitigations that appear in practice.

3.1 CSRF in OAuth

The CSRF attack on the authorization code grant flow of OAuth 2.0 involves four parties, as shown in Figure 2b. In this scenario, a victim user has accessed an RP at some point in the past, and has granted that RP access to his account on an IdP. While the user may have logged out of the RP application, we assume that he is still logged into his account with the IdP. While both the RP and the IdP are honest players, the RP must have a vulnerable implementation of the authorization code grant flow. The adversary is assumed to have no control over the RP or the IdP, but is capable of launching requests from the user agent. This action could be performed in practice by luring the user to click a malicious link in a phishing email or as a part of a clickjacking attack. This link could then load a malicious HTML `<img>` tag or other malicious code on a web page.

The attack proceeds as follows: If the adversary recognizes a vulnerable OAuth URL in an RP application, she can initiate the authorization code grant flow by luring the user to load the URL from their user agent (step 1 in Figure 2b). When the victim loads this URL, their browser will automatically submit a GET request to this vulnerable OAuth URL. Because the victim is still logged in with that OAuth IdP, the pop-up request for user authorization in step 2 will be bypassed, and the IdP will proceed to issue the authorization token to the RP.

The result of the attack is that the RP now has authorization to access the user's IdP account *without the user having granted consent for this session*. The OAuth authoriza-

Provider	CSRF Protection	Provider	CSRF Protection	Provider	CSRF Protection
Battle.net	Forced	**Dropbox**	Suggested	**AOL**	No Mention
Github	Forced	**Facebook**	Suggested	**Microsoft**	No Mention
LinkedIn	Forced	**Google**	Suggested	**Salesforce.com**	No Mention
Reddit	Forced	**Instagram**	Suggested		
Amazon	Suggested	**PayPal**	Suggested		

Table 1. A table of major OAuth Identity Providers. Regarding the prevention of CSRF attacks, this table describes if the IdP forces CSRF prevention, suggests it, or makes no mention of implementing it. Note that only 4 of the 13 IdPs require that RPs implement protections that are mandated by the OAuth 2.0 standard.

tion token allows an RP to execute any operation in the IdPs API, potentially accessing and modifying private user information.

The RFC defining OAuth 2.0 provides an entire subsection detailing the potential for CSRF attack in the authorization code grant flow. After detailing how the attack proceeds, the RFC expressly states that:

> *The client MUST implement CSRF protection for its redirection URI...The authorization server MUST implement CSRF protection for its authorization endpoint...* [18]

The specification clearly requires that CSRF protection be implemented to prevent the attack described above. However, upon investigating the OAuth policy and implementation of 13 major IdPs, we discovered that CSRF prevention tools are only required by 4 of the 13 IdPs (Table 1). The documented policies of the rest of the IdPs either recommend but do not require CSRF defenses, or completely ignore them. This lack of proper enforcement of the OAuth 2.0 standard indicates that there is a dangerous potential for RP developers to implement OAuth 2.0 requests in a vulnerable manner that will still be accepted by the IdP.

3.2 Developer implementation problems

The OAuth 2.0 standard contains an entire section entitled "Security Considerations" that is dedicated to specific security concerns when implementing OAuth [18]. Given that developers appear to be ignoring the subsection regarding CSRF, we examined the other considerations to determine if the problem of noncompliant implementation is widespread or if CSRF presents a unique challenge in correct implementation.

The section covers 16 different security concerns and mitigations for each attack. Many of these mitigations are standard good security practice, with examples including not storing or sending passwords in plaintext, sanitizing all data fields, and securing connections with TLS. A smaller subset of considerations deal with server back-end implementations, such as preventing brute-force login attempts and choosing secure cryptographic values. Finally, there are a few considerations warning against using deprecated flows and encouraging proper user education on strong password creation and identifying phishing attacks.

The most important take away from the documentation is that throughout the entire section, the client (i.e., the RP initiating the request) is only tasked with implementing three security mechanisms. Those mechanisms are:

- Validating the server TLS certificate
- Sanitizing all data fields received during the protocol
- Implementing CSRF protection

While many server-based applications have been found to improperly validate TLS certificates, these problems are largely a result of errors and poor design in the underlying TLS libraries, not application code itself [13]. Likewise, sanitizing data is a well-studied security problem with extensive documentation on mitigation techniques to check the format of incoming data. However, implementing CSRF protection mechanisms according to the OAuth standard slightly increases the complexity of application code. This increase in complexity appears to have the strong side effect of discouraging developers from implementing protections at all. Even when developers are only tasked with managing a handful of security concerns, requiring even a small amount of security expertise appears to be an impediment to proper implementation.

3.3 Mitigation

The OAuth specification explicitly identifies the use of the OAuth `state` parameter in the section defining necessary CSRF protection mechanisms:

> *This is typically accomplished by requiring any request sent to the redirection URI endpoint to include a value that binds the request to the user-agent's authenticated state (e.g., a hash of the session cookie used to authenticate the user-agent). The client SHOULD utilize the "state" request parameter to deliver this value to the authorization server when making an authorization request.*

By requiring a non-guessable value that binds the request to a specific, authenticated state, an adversary is prevented from constructing a valid malicious request to then lure the victim into launching through his user agent.

Given this advised policy, we next examined the CSRF prevention techniques employed by a selection of common RPs, shown in Table 2. We found a variety of different CSRF prevention techniques exist in practical deployments. The varying CSRF prevention techniques implemented by RPs tended to correspond with the CSRF policy of the IdP they were accessing.

The four IdPs listed in Table 1 as forcing CSRF protection all required that connecting RPs implement protection using the `state` parameter as documented in the OAuth standard. RPs sending connection requests to these services without the `state` parameter defined are denied authorization. This strict adherence to the standard forces developers to implement protection correctly, preventing the CSRF attack described above when accessing these IdPs. For example, Battle.net provides a comprehensive developer guide to properly generating and using the `state` variable [7].

However, a small selection of IdPs that do not explicitly require CSRF protection offered non-standard protection mechanisms. As an example, the RPs listed under "secure

Technique and App	Web/Desktop	IdP	Details
CSRF Token			
Spotify	Desktop	Facebook	Extra CSRF token
TheVerge.com	Web	Google, Facebook	Google suggests state parameter, Facebook suggests the Facebook SDK
Secure Protocol			
OneNote	Desktop	Microsoft	Uses the implicit OAuth 2.0 flow
Google	Desktop (OS X)	Google	Uses the (deprecated) ClientLogin flow
OneDrive	Desktop	Microsoft	Uses the implicit OAuth 2.0 flow
LinkedIn	Desktop (OS X)	LinkedIn	Uses an undocumented method in OAuth 1.0
Other			
Spotify	Web	Facebook	Uses protection mechanism included in the Facebook SDK
OneNote	Web	Microsoft	Uses proprietary Microsoft authentication
Huffington Post	Web	Facebook	Facebook SDK mechanism
Hulu	Web	Facebook	Facebook SDK mechanism
ESPN.com	Web	Facebook	Facebook SDK mechanism
MLB.com	Web	Facebook	Facebook SDK mechanism
Facebook	Desktop (OS X)	Facebook	Uses the auth.login method from the Facebook API

Table 2. A listing of RPs and their means of preventing CSRF attacks. Without a single mandatory technique for CSRF prevention, RPs have developed a wide variety of protection schemes.

protocol" use a modified OAuth flow or completely different protocol for authorizing IdP access. In particular, the use of the OAuth 2.0 implicit grant flow by OneNote and OneDrive prevents this particular CSRF attack by requiring more specific user agent interaction that is difficult for an adversary to initiate without the user's knowledge. Other IdPs provided proprietary authentication or non-standard variables to prevent CSRF attacks. Unfortunately, because these mechanisms do not conform to the CRSF protection specified in the standard, they may not offer the necessary protection against attack. Ultimately, the lax and variable standards for implementing CSRF protection in a majority of the IdPs leaves extreme potential for developers to bypass or incorrectly implement CSRF protection in their OAuth applications.

4 CSRF in the wild

While the attack described in Section 3 is well understood with known mitigation strategies, our goal is to learn how often mitigations are correctly implemented in practice. To achieve this goal, we first performed an analysis of the Alexa top 10,000 websites to ascertain the breadth of the problem. We then performed in-depth analyses of four high-profile case studies to learn the underlying issues that cause developers to incorrectly implement OAuth 2.0.

4.1 Web Crawler Design and Implementation

Popular web crawler frameworks such as Scrapy [37] and Crawler4j [41] have been developed to provide robust web scraping functionality. However, these frameworks are heavy-weight compared to our needs for OAuth URL detection, and only provide limited URL crawling front-end capability. For these reasons, we implemented our own web crawler, which we entitled the OAuth Detector (OAD).

OAD is a light-weight, multi-threaded, high-performance web crawler, with an OAuth-specific data collection back-end. We implemented the crawler using the Beautiful Soup library [36]. OAD supports raw URL crawling using Python `urllib2` [35]. For our application, raw URL crawling offers the best speed. Once the list of websites to crawl is loaded, OAD divides these sites according to the target thread requested. Any remaining sites are distributed to different threads in an round-robin fashion, balancing the workload across all threads. As our goal is to determine if the website contains vulnerable OAuth URLs, OAD allocates the minimum memory used to store these URLs for each website. If additional analytic information is needed, it is easy to extend the current data structure to save extra information. The OAD source code is available at `https://bitbucket.org/uf_sensei/oadpublic`.

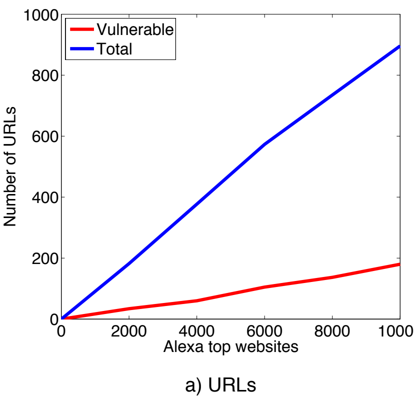
4.2 Data Collection Setup

To approximate the number of occurrences of this vulnerability in the wild, we used OAD to analyze the Alexa top 10,000 websites. For each site, we analyzed two points: First, we checked whether the website makes any OAuth requests. Second, we checked to see if the website implemented the OAuth authorization code grant flow correctly (i.e., implemented CSRF protection in the `state` parameter). To do this, our crawler iteratively examined links found in each site's source code. To identify an OAuth URL, we checked that the `client_id` and `grant_type` parameters were defined. If an OAuth URL was found, it was examined to identify the grant flow, determine whether the vulnerability existed, and logged. The metric used to determine vulnerability was the existence or absence of a `state` variable in the URL. While it is possible for developers to implement CSRF protection outside of the `state` variable, this constitutes a non-standard CSRF protection technique not advised by the OAuth standard. If the link was not an OAuth URL, the crawler followed the link, and the process recursively repeated on the next page. Each link on the main page was followed up to depth-two, inspecting all links on the main page and the links on each page linked from the main page. In the Alexa top 10,000 websites, we crawled a total of 5,671,022 URLs.

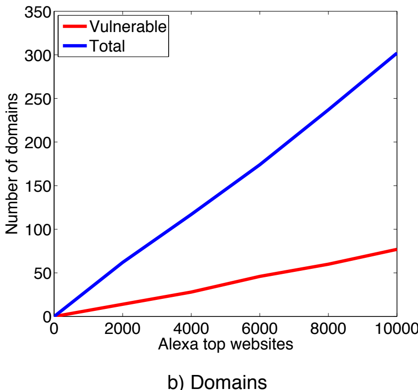
Our results provide a conservative count for OAuth use in the Alexa top 10,000. Because OAD scans URLs in the page source, is not capable of finding URLs that are produced after executing Javascript on each page. As a result, it is possible that more domains exist that are vulnerable to CSRF or implement a non-standard CSRF protection mechanism. This makes our vulnerability estimate conservative.

4.3 Results

On the websites crawled, OAD found 302 domains that implement OAuth 2.0 in some form. Of those domains, 77 implemented at least one OAuth connection without CSRF



a) URLs



b) Domains

Fig. 3. CDFs showing the number of vulnerable URLs and domains in the top Alexa web pages. These plots show that vulnerable domains and URLs are evenly distributed throughout the top 10,000 websites.

protection. This result conservatively shows that as many as 25% of RPs do not correctly implement CSRF protection. To determine how these vulnerabilities are distributed among domains, we broke the top 10,000 sites into five groups of 2,000 sites. When we considered the number of vulnerable domains placed in these buckets of 2,000 sites each, the CDF in Figure 3b shows that the vulnerable implementations are evenly distributed across the top 10,000. To further determine how many vulnerable connections each domain implements, we also analyzed the total number of vulnerable URLs. Figure 3a shows that the vulnerable URLs roughly follow the same distribution as the vulnerable domains, indicating that these domains are erroneously implementing OAuth 2.0 at a similar rate.

To statistically demonstrate that this vulnerability is occurring consistently regardless of the popularity of the website, we performed a statistical analysis on the CDFs in Figure 3. We divided the vulnerable URLs and domains into two sets, the lower and upper 5,000 websites of the Alexa top 10,000. We then applied Fisher's exact test to the hypothesis "website ranking correlates with the likelihood of website vulnerability", where our null hypothesis is that the two statistics are not correlated. For the vulnerable URLs, we demonstrated that the correlation is statistically not significant with $p = 0.065 > 0.05$. While this value appears to approach significance, we found even lower correlation when we applied the same test to the vulnerable domains, with $p = 0.229 > 0.05$. Both experiments strongly imply the null hypothesis, that vulnerabilities occur at a rate that does not correlate with the Alexa ranking of the website. This clearly demonstrates that correctly implementing CSRF protection is uniformly problematic for developers of both low-traffic and high-profile web applications.

In addition to examining the distribution of vulnerabilities among RPs, we divided the number of vulnerable URLs down by which IdP they connect to. Figure 4 shows that of the vulnerable URLs, the most common IdP accessed with these vulnerable connections is Facebook. In addition, the problem is evident across IdPs from differ-

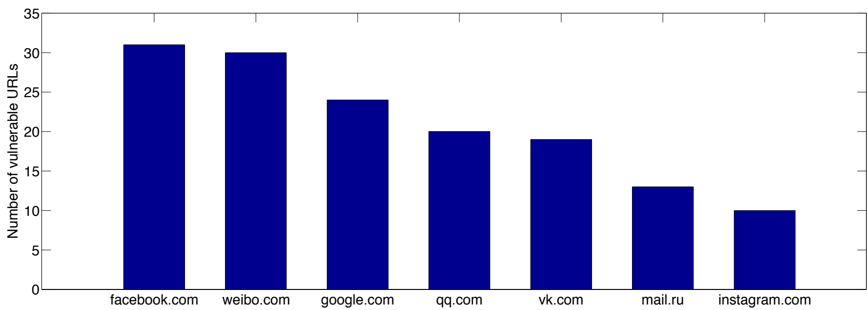


Fig. 4. The most common IdPs contacted using vulnerable OAuth URLs with the number of vulnerable URLs accessing each.

ent countries as well. IdPs serving Europe (vk.com), China (weibo.com, qq.com), and Russia (mail.ru) all have RPs that are improperly implementing OAuth connections.

5 Case Studies

To identify the reasons why this well-documented vulnerability still exists at such a scale in live deployments, we selected four examples from our crawl data that represent high-profile services. We selected these examples for two reasons. First, they represent high profile web applications with significant corporate support, and should be expected to have high quality production code. Second, each subsequent example demonstrates that incremental increases in support for developers still allow for significant failures in implementation security, and that nothing short of IdPs mandating compliance will completely repair the vulnerability. We have notified each IdP that allows vulnerable connections of this potential attack prior to publishing this work.

5.1 Missing Documentation

If This Then That (IFTTT) is a popular website that connects Web APIs together to create interactions between two distinct services. This service leverages third party web APIs like Microsoft's OneNote, Salesforce.com's Chatter and Instagram to create unique alert combinations for users. For example, a user can allow IFTTT to automatically save New York Times articles to their OneNote notebook [22]. In the background, it uses OAuth 2.0 to authenticate itself with these third party identity providers provided IFTTT users give consent.

IFTTT allows connections to a range of IdPs, and accurately follows the documentation of each IdP with regards to using OAuth. We observed that if the IdP provided documentation or code to implement CSRF protection (e.g., the Facebook SDK), IFTTT correctly and securely implemented OAuth connections to those IdPs. However, IdPs like Microsoft, Salesforce.com and Instagram do not require the use of the `state` variable and do not provide developer tools for securely implementing OAuth 2.0 protocols

```
/**
 * Obtain a Live Connect authorization endpoint URL based on configuration.
 * @returns {string} The authorization endpoint URL
 */
this.getAuthUrl = function () {
    var scopes = ['wl.signin', 'wl.basic', 'wl.offline_access', 'office.onenote_create'];
    var query = toQueryString({
        'client_id': clientId,
        'scope': scopes.join(' '),
        'redirect_uri': redirectUrl,
        'display': 'page',
        'locale': 'en',
        'response_type': 'code'
    });
    return oauthAuthorizeUrl + "?" + query;
};
```


Fig. 5. Example code for connecting to Microsoft's Live Connect Services. Note that there is no random token to help protect against a CSRF attack made on the URL generated by this function.

for preventing CSRF attacks [34]. As a result, IFTTT implements OAuth connections to all three of these services that were vulnerable to CSRF attacks, allowing an attacker to authorize IFTTT to access a user's account on these services without their consent.

This example of the OAuth CSRF attack indicates that developers will simply omit CSRF protection if the implementation is left to their discretion. However, developers connecting to IdPs that provide CSRF documentation are much more likely to build their web applications correctly.

5.2 Incorrect Code Samples

The previous case study showed that without proper documentation or requiring CSRF protection, developers are prone to implementing OAuth 2.0 in an insecure manner. Our second case study revealed that this problem is often exacerbated by IdPs who provide tools that encourage insecure implementations.

To assist developers with implementing OAuth connections, it is common for IdPs to provide sample code demonstrating how to properly use their OAuth 2.0 implementations. These code samples are meant to show complete and correct examples for connecting to the IdP's API. Microsoft and AOL are two such IdPs that post sample code for developers to reference. AOL provides sample PHP code [3] hosted on their own developer website, and Microsoft publishes sample code [32] in a variety of languages on Github. Figure 5 shows a snippet of Javascript code provided by Microsoft, and Figure 6 shows a PHP code sample provided by AOL. The Javascript function provided by Microsoft is used to build an OAuth 2.0 URL to request access to Microsoft's Live Connect services. A user accessing the URL generated by this function will be prompted for their username and password. They will then be asked for access to the security scopes provided by the `scopes` list on line 28. Given this client-side information, the function fills in the necessary URL fields to make a valid OAuth request to access the Microsoft API. If access is granted, the RP will receive an access token (see Section 2) that is used to access the user's protected resources. The PHP code provided by AOL

```
else{//for simplicity force thru to authorization but often flows would use
something like the HTML example with a popup
    $authorizationReq = $aolAuthorizeUrl."?
client_id=".$clientId."&response_type=code&redirect_uri=".$callbackUrl;
    header("Location: $authorizationReq");
    die();
}
```

no "state" parameter

Fig. 6. Example code for connecting to AOL's OAuth services. Notice that like Microsoft, there is no random token to protect against CSRF.

serves the same purpose. This code builds up the parameter list of the URL used to initiate AOL's OAuth 2.0 authorization code flow. An RP implementing this code would simply need to specify its `client_id` (registered with AOL) and the `redirect_url` to return to after the authentication is complete.

Neither of these code samples contains example code for properly implementing CSRF protection. Given that these code samples can be cut and pasted into a developer's implementation and will produce a functioning OAuth connection back to the IdP, it is not surprising that developers consistently implement the example code as is, without CSRF protection. Until IdPs remedy the vulnerabilities in their example code, these same vulnerabilities will continue to propagate into live RP implementations.

5.3 Inconsistent Requirements

While some IdPs provide insecure examples of OAuth code, there are examples of IdPs that provide helpful developer tools for correctly implementing OAuth 2.0. One example of this is Facebook, which provides a comprehensive Javascript SDK that contains built-in protection against CSRF attacks. However, other services offered by the same company fail to provide any developer assistance in proper OAuth implementation. Now owned by Facebook for nearly three years, Instagram also provides its own API for single sign on, allowing websites to access their services. One common use of this API is to build third-party "web viewers" for Instagram accounts, such as `ink361.com` [23]. A user can sign onto this service using either their Instagram or Facebook account, then access their Instagram albums for online viewing. This authorization and authentication is handled by the OAuth 2.0 protocol. Upon examining the OAuth URL used to connect to these IdPs, we discovered that the Facebook connection was securely implemented using the Facebook SDK, while the Instagram implementation was vulnerable to CSRF attack.

After investigating the Instagram OAuth 2.0 documentation [24], we found that the IdP recommends CSRF protection with the following warning: "You may provide an optional `state` parameter to carry through any server-specific state you need to, for example, protect against CSRF issues." However, unlike Facebook, Instagram does not provide any tools for correctly implementing the `state` parameter, nor documentation for how to do so. Without proper enforcement from the OAuth specification itself,

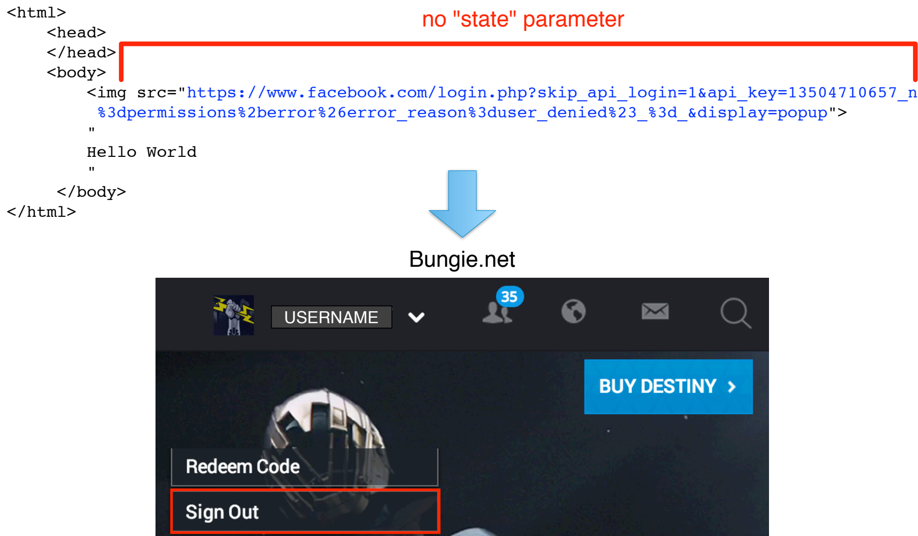


Fig. 7. An example of malicious HTML and the resulting CSRF attack. The code sample above contains a vulnerable OAuth URL. When a victim loads the URL, they are logged into the RP Bungie.net without being prompted for consent by the IdP, Facebook.

IdPs are unable to consistently document and implement correct OAuth implementation tools, *even within the same managing corporation.*

5.4 Lack of Enforcement

Given that our previous case studies showed that many IdPs are poorly documenting proper OAuth implementation or not providing developers with usable tools for implementing CSRF protection, Facebook appeared to have solved the issue by providing a comprehensive SDK. However, their implementation does not force any CSRF protection to be implemented. Our crawl revealed that of the vulnerable OAuth URLs found, almost 20% connected to the Facebook API. One of these vulnerable URLs is implemented on the website of popular video game developer Bungie. Bungie is best known for the development of the Halo franchise and most recently the first-person shooter Destiny. They provide an online service for collecting and presenting in-game data and providing opportunities for social interactions between players, with single-sign on options for connecting to Facebook. Bungie uses OAuth 2.0 to connect to Facebook, implementing the authorization code grant flow. However, instead of using the Facebook SDK and implementing the built-in CSRF protection mechanism, Bungie implemented the OAuth protocol from scratch. Their ability to implement a functioning OAuth implementation *without* CSRF protection indicates that while Facebook provides developer tools for implementing CSRF protection, they do not require connection requests to actually use this protection. The resulting CSRF attack is illustrated in Figure 7.

This example shows that even when developer tools are provided to assist in correct, secure implementation, developers may still choose to build their own insecure implementations. Our crawl results show that, in fact, a large number of developers are doing this. Ultimately, if correct security protection mechanisms are not mandated by the protocol, developers will only meet the minimum functional standards, even if these standards are not secure.

5.5 Recommended approaches to mitigation

To effectively mitigate the problem of CSRF, these case studies all point to the IdPs as the logical party to effect change. While previous studies have recommended mitigation techniques that expect RPs to implement a range of security constructions [39], our results show that even when RPs are given effective developer tools, they *still fail to correctly implement constructions as simple as a random token*. Without the ability to entrust the RPs with correctly managing their implementations, the onus lies on the IdPs to force proper implementation through a) providing *correct* and *complete* developer tools for implementing OAuth, and more importantly b) forcing correct RP implementations by rejecting OAuth requests that do not contain all necessary authenticating tokens. While it is possible that an RP could include these tokens and willingly not verify their validity after the IdP responds, our observations of the four IdPs that force CSRF protection using the `state` parameter show this is not a common case. When these IdPs mandate the use of the `state` parameter and provide proper documentation, RPs largely provide correct and secure implementations and effectively mitigate CSRF attacks. However, until more IdPs begin enforcing this level of strict adherence to the OAuth specification, RPs will continue to produce vulnerable implementations and expose their customers to attack.

6 Discussion

6.1 Comparison to HTTPS use

Another possible conclusion for why CSRF protection is not correctly implemented in a large number of RPs is that developers are cutting corners in an attempt to develop a more efficient authorization connection. To further explore this possibility, we examined another security feature that is sometimes optionally implemented: HTTPS. Upon examining the 180 vulnerable OAuth URLs found by OAD, we discovered that 146 (81%) of the URLs connected to the IdP over an HTTPS connection. The process of establishing a secure channel for communication requires significantly more time and bandwidth than generating and transmitting a short random token to maintain state. However, our results show that the overwhelming majority of RPs implementing CSRF-vulnerable connections *do not* cut corners when it comes to using HTTPS. This seems to indicate that the lack of CSRF protection is not an attempt to preserve efficiency.

Upon examining which IdPs were contacted over the insecure HTTP connections, we noted that 15 connected to `qq.com`, 8 connected to `vk.com`, 1 connected to Facebook, and the rest to small IdPs. Since most of the large IdPs represented in our study

were never connected to over an insecure connection, this would seem to indicate that strong IdP policy ensures that developers implement HTTPS. Were the same strong enforcement on CSRF protection in place, we hypothesize that the occurrence of vulnerable OAuth connections would also appear at a similarly low rate.

6.2 OAuth 1.0

The OAuth 1.0 protocol is described in RFC 5849 [17]. This RFC was originally released in 2010 and is the original incarnation of the OAuth protocol. We elected not to focus on this particular version as it has been deemed "obsolete" by the OAuth 2.0 RFC [18]. The OAuth 1.0 RFC mentions in section 4.13 that CSRF attacks are possible on OAuth URIs, and specific preventions are "beyond the scope of this specification."

7 Related Work

Single Sign-On systems have been extensively studied in general, and have been shown to exhibit a variety of vulnerabilities. Like our work, some of these vulnerabilities stem from poor implementation and unclear developer guidelines [5, 45], while others are a result of flawed protocols [38, 44]. In addition, a variety of automated tools exist to analyze the security of SSO implementations in the wild. However, the tools that are currently available either target more generic vulnerabilities [4, 46] or vulnerabilities of a different nature [50] than the CSRF attack that we describe here.

Since its release, OAuth 2.0 has received mixed evaluations on the security of the protocol. Formal analysis of the protocol has shown that under specific assumptions and correct implementation, OAuth 2.0 provides secure authentication. However, the same analyses point out that the prioritization of simple implementation over security has left significant potential for incorrectly implementing the protocol [9, 14, 20, 16, 11]. In practice, relying parties have been found to have implementations that fail to protect against common web attacks such as cross site scripting and click-jacking [39]. In addition, OAuth 2.0 has also been shown to be susceptible to attacks relating to all forms of SSO [40, 45], such as phishing or eavesdropping on an unprotected communication channel [48, 15, 8]. Finally, mobile applications of OAuth 2.0 have been found to be particularly vulnerable due to developer confusion [10].

Previous studies have shown that CSRF vulnerabilities could exist in OAuth 2.0 [47, 29, 39, 5]. The official threat model contains two sections dedicated to potential implementation vulnerabilities in the implicit grant and authentication code flows [30]. These sections recommend that the `state` parameter of the OAuth 2.0 protocol be used as a pseudorandom token (as recommended in [49]). However, neither the threat model nor the standard requires the use of this parameter as a synchronizer token. Other studies have been done to exploit these documented CSRF vulnerabilities in OAuth 2.0 on the relying party side [39]. However, the extent to which these vulnerabilities exist in the wild has not yet been shown. Furthermore, there has not been a study on the variations of implementations of identity providers and how these differences can actually cause these critical vulnerabilities. This paper aims to show that because the OAuth 2.0 standard does not require certain precautions such as a synchronizer token, IdPs are providing vulnerable implementations of "the standard" to unknowing relying parties.

8 Conclusion

As an increasing number of web applications require the exchange of user data between services, OAuth 2.0 provides a convenient framework for authorization between accounts. While the standard contains explicit instructions for implementing security protection mechanisms, these standards do not always translate into real-world implementations. This work demonstrates that the documented CSRF vulnerability in particular is often not defended against in practical deployments. Our OAD crawler was able to show that 25% of the domains using OAuth in the Alexa top 10,000 do not properly implement CSRF protection mechanisms. After examining four high-profile vulnerabilities, we demonstrate that the reason for this lack of compliance is because IdPs do not require developers to implement the protocol with CSRF protection according to the RFC. This lack of enforcement shows that we cannot rely on developers to properly implement optional security controls, but must strongly enforce their use within the protocol and developer tools.

Acknowledgements This work is based upon work supported by the U.S. National Science Foundation under grant numbers CNS-1118046, CNS-1245198, and CNS-1464087.

References

1. Alexa Internet, inc.: Alexa top sites. http://www.alexa.com/ (2014)
2. Alur, D., Crupi, J., Malks, D.: Core j2ee patterns: Best practices and design strategies. http://www.corej2eepatterns.com/Design/PresoDesign.htm (2001)
3. AOL inc.: Php sample. http://identity.aol.com/documentation/start/oauth2/web-site-integration/php-sample/ (2014)
4. Bai, G., Lei, J., Meng, G., Venkatraman, S.S., Saxena, P., Sun, J., Liu, Y., Dong, J.S.: Authscan: Automatic extraction of web authentication protocols from implementations. In: Proceedings of the Network and Distributed System Security Symposium (2013)
5. Bansal, C., Bhargavan, K., Maffeis, S.: Discovering concrete attacks on website authorization by formal analysis. In: Proceedings of the IEEE Computer Security Foundations Symposium (2012)
6. Barth, A., Jackson, C., Mitchell, J.C.: Robust defenses for cross-site request forgery. In: Proceedings of the ACM Conference on Computer and Communications Security (2008)
7. Blizzard Entertainment, Inc.: Using OAuth. https://dev.battle.net/docs/read/oauth (2014)
8. Cao, Y., Shoshitaishvili, Y., Borgolte, K., Kruegel, C., Vigna, G., Chen, Y.: Protecting web-based single sign-on protocols against relying party impersonation attacks through a dedicated bi-directional authenticated secure channel. In: Research in Attacks, Intrusions and Defenses, Lecture Notes in Computer Science, vol. 8688, pp. 276–298. Springer International Publishing (2014)
9. Chari, S., Jutla, C., Roy, A.: Universally composable security analysis of OAuth v2.0. Cryptology ePrint Archive, Report 2011/526 (2011), http://eprint.iacr.org/
10. Chen, E., Pei, Y., Chen, S., Tian, Y., Kotcher, R., Tague, P.: OAuth demystified for mobile application developers. In: Proceedings of the ACM Conference on Computer and Communications Security (2014)

11. Cherrueau, R.A., Douence, R., Royer, J.C., Südholt, M., De Oliveira, A.S., Roudier, Y., Dell'Amico, M.: Reference monitors for security and interoperability in OAuth 2.0. In: International Workshop on Autonomous and Spontaneous Security (2013)
12. Ferreira, H.G.C., de Sousa Jnior, R.T., de Deus, F.E.G., Canedo, E.D.: Proposal of a secure, deployable and transparent middleware for internet of things. In: Proceedings of the Iberian Conference on Information Systems & Technologies (CISTI) (2014)
13. Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., Shmatikov, V.: The most dangerous code in the world: Validating SSL certificates in non-browser software. In: Proceedings of the ACM Conference on Computer and Communications Security (2012)
14. Gibbons, K., Raw, J.O.: Security evaluation of the OAuth 2.0 framework. Information Management and Computer Security 22(3) (2014)
15. Hammer, E.: OAuth 2.0 (without signatures) is bad for the web. http://hueniverse.com/2010/09/15/oauth-2-0-without-signatures-is-bad-for-the-web/ (2010)
16. Hammer, E.: OAuth 2.0 and the road to hell. http://hueniverse.com/2012/07/26/oauth-2-0-and-the-road-to-hell/ (2012)
17. Hammer-Lahav, E.: The OAuth 1.0 protocol. RFC 5849, RFC Editor (April 2010), http://tools.ietf.org/html/rfc5849
18. Hardt, D.: The OAuth 2.0 authorization framework. RFC 6749, RFC Editor (October 2012), http://tools.ietf.org/html/rfc6749
19. Hardy, N.: The confused deputy: (or why capabilities might have been invented). SIGOPS Operating Systems Review 22(4), 36–38 (1988)
20. Homakov, E.: OAuth1, OAuth2, OAuth...? http://homakov.blogspot.jp/2013/03/oauth1-oauth2-oauth.html (2013)
21. Hhnlein, D., Wich, T., Schmlz, J., Haase, H.M.: The evolution of identity management using the example of web-based applications. Information Technology 56(3), 134–140 (2014)
22. IFTTT Inc.: If this then that. https://ifttt.com/ (2014)
23. INK361: Instagram web viewer – ink361. http://ink361.com/ (2014)
24. Instagram: Authentication. http://instagram.com/developer/authentication/ (2014)
25. Jones, M., Hardt, D.: The OAuth 2.0 authorization framework: Bearer token usage. RFC 6750, RFC Editor (October 2012), http://tools.ietf.org/html/rfc6750
26. Jovanovic, N., Kirda, E., Kruegel, C.: Preventing cross site request forgery attacks. In: Proceedings of the International Conference on Security and Privacy in Communication Networks (Securecomm) (2006)
27. Käfer, K.: Cross site request forgery. http://dump.kkaefer.com/csrf-paper.pdf (2008)
28. Kaur, G., Aggarwal, D.: A survey paper on social sign-on protocol OAuth 2.0. Journal of Engineering, Computers, & Applied Sciences 2(6), 93–96 (2013)
29. Li, W., Mitchell, C.J.: Security issues in OAuth 2.0 SSO implementations. In: Proceedings of the Information Security Conference (ISC) (2014)
30. Lodderstedt, T., McGloin, M., Hunt, P.: OAuth 2.0 threat model and security considerations. RFC 6819, RFC Editor (January 2013), http://tools.ietf.org/html/rfc6819
31. Mao, Z., Li, N., Molloy, I.: Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In: Proceedings of the International Conference on Financial Cryptography and Data Security (2009)
32. Microsoft: liveconnect-client.js. https://github.com/OneNoteDev/OneNoteAPISampleNodejs/blob/master/lib/liveconnect-client.js (2014)

33. Nauman, M., Khan, S., Othman, A.T., Musa, S.u., Rehman, N.U.: POAuth: Privacy-aware open authorization for native apps on smartphone platforms. In: Proceedings of the International Conference on Ubiquitous Information Management and Communication (2012)
34. Patterson, P.: Digging deeper into OAuth 2.0 on force.com. https://developer.salesforce.com/page/Digging_Deeper_into_OAuth_2.0_on_Force.com (2014)
35. Python Software Foundation: urllib2. https://docs.python.org/2/library/urllib2.html (2015)
36. Richardson, L.: Beautiful soup. http://www.crummy.com/software/BeautifulSoup/ (2014)
37. Scrapinghub: Scrapy. http://scrapy.org/ (2015)
38. Somorovsky, J., Mayer, A., Schwenk, J., Kampmann, M., Jensen, M.: On breaking saml: Be whoever you want to be. In: Proceedings of the USENIX Security Symposium (2012)
39. Sun, S.T., Beznosov, K.: The devil is in the (implementation) details: An empirical analysis of OAuth SSO systems. In: Proceedings of the ACM Conference on Computer and Communications Security (2012)
40. Sun, S.T., Pospisil, E., Muslukhov, I., Dindar, N., Hawkey, K., Beznosov, K.: Investigating users' perspectives of web single sign-on: Conceptual gaps and acceptance model. ACM Trans. Internet Technology 13(1), 2:1–2:35 (2013)
41. The crawler4j community: crawler4j. https://code.google.com/p/crawler4j/ (2015)
42. The OpenID Foundation: OpenID. http://openid.net/ (2015)
43. Vapen, A., Carlsson, N., Mahanti, A., Shahmehri, N.: Third-party identity management usage on the web. In: Passive and Active Measurement, Lecture Notes in Computer Science, vol. 8362, pp. 151–162. Springer International Publishing (2014)
44. Wang, R., Chen, S., Wang, X.: Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In: Proceedings of the IEEE Symposium on Security and Privacy (2012)
45. Wang, R., Zhou, Y., Chen, S., Qadeer, S., Evans, D., Gurevich, Y.: Explicating SDKs: Uncovering assumptions underlying secure authentication and authorization. In: Proceedings of the USENIX Security Symposium (2013)
46. Xing, L., Chen, Y., Wang, X., Chen, S.: Integuard: Toward automatic protection of third-party web service integrations. In: Proceedings of the Network and Distributed System Security Symposium (2013)
47. Yang, F., Manoharan, S.: A security analysis of the OAuth protocol. In: IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM) (2013)
48. Yue, C.: The devil is phishing: Rethinking web single sign-on systems security. In: Proceedings of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET) (2013)
49. Zeller, W., Felten, E.W.: Cross-site request forgeries: Exploitation and prevention. Tech. rep., Princeton University (2008)
50. Zhou, Y., Evans, D.: SSOScan: Automated testing of web applications for single sign-on vulnerabilities. In: Proceedings of the USENIX Security Symposium (2014)