OAuth2.0 - Exposing vulnerabilities in the wild

Amrutha Ramesh
University of Illinois at Chicago
Chicago, IL, USA
arames6@uic.edu

Arushi Mishra
University of Illinois at Chicago
Chicago, IL, USA
amishr6@uic.edu

Ayush Chugh
University of Illinois at Chicago
Chicago, IL, USA
achugh4@uic.edu

ABSTRACT

The Web has evolved drastically in the last few decades and so have the security and privacy issues related to it. We rely on websites to perform various daily tasks - both personal and professional, from paying our bills, communicating with friends and family to managing our bank accounts. This means that a user of the Web has multiple online accounts with its respective passwords. This inconvenience was alleviated by the introduction of Single Sign On (SSO) services integrated into these websites. This integration might use one of the available protocols like OAuth2.0, OpenID, OpenID Connect and so on. However, this integration has to be done with the utmost care by developers to avoid privacy risks. In this project, we focus on the potential risks and vulnerabilities of the OAuth2.0 protocol flow. CSRF is the most common vulnerability of OAuth that has already been studied. So we looked into some lesser known vulnerabilities, namely - Redirect URI and Login CSRF. Previous work on these risks are studies and do not involve attacks. We conduct these attacks by designing a malicious web application for the victim and find evidence that such attacks are plausible and not insignificant. A number of these SSO providers are vulnerable and our findings highlight that even seemingly simple vulnerabilities can have significant privacy repercussions.

1. INTRODUCTION

Single Sign On (SSO) is a way of authentication in which the user has the option to login to a website with one of the identity providers like Google, Facebook and so on. This helps the user sign on to a website without going to the trouble of creating a separate username and password. By using the same set of providers for different websites, the need to remember username/password combinations is obviated and user experience becomes better. This integration of SSO providers follow different procedures specific to each website. Every SSO provider also has different protocol implementations for integrating SSO on their website. Some of these protocols are OAuth1.0, OAuth2.0, OpenID, OpenID Connect and so on. Any website integrating an SSO option chooses one of these protocol implementations. OAuth2.0 protocol flow and implementation is the focus of our project. This protocol is an authorization protocol and involves three parties :

Resource Owner: User
The user is responsible for authorization an application to access resources at the resource provider and this access is restricted by the "scope" of the authorization granted. Eg: Read or Write

Resource/Authorization server: API
The resource server is responsible for hosting the user's accounts and the authorization server verifies the user's identity. It then issues access to the application.

Client: Application
The client is the application that wants to access the user's account at the resource server. It first requires authorization from the user and then this is validated by the API.

There are also different flows that can be used to implement OAuth2.0. The implicit flow, the authorization code flow, resource owner password flow and client credentials flow. In our research, we are utilizing the authorization code flow for web server applications. In this flow, the client sends an authorization request to the resource server. The server prompts the user to log in and once the authorization is confirmed the user is redirected to the client with authorization code. This code is then exchanged for an access token by sending a request to the server and the token used for subsequent actions performed.

Cross Site Request Forgery(CSRF) is the most widely studied OAuth2.0 vulnerability. In this, a user's session is hijacked by the attacker and used to perform actions that change some state on the user's account. The defense to this attack is to append a "state" parameter in the client's requests for authorization. This can be any randomly generated value that cannot be guessed by the attacker. Most of the earlier work has been conducted around this vulnerability, therefore we are not focusing on this for our research. There are other lesser known OAuth2.0 vulnerabilities that also have significant privacy impact. Two of them have been explored in this project : Login CSRF and Redirect URI attacks. In the login CSRF attack, the attacker is able to forge login requests with the attacker's credentials that the victim unknowingly uses. This way the attacker gets to know about the user's activities on his account. There are two redirect URI attacks. In the Redirect URI Mismatch attack, by manipulating the redirect URI provided as a parameter to the authorization request, the code and access token can be sent to a URI of the attacker's choice. This occurs because the resource provider does not check for the redirect URI mismatch between the one used for registering the application and the one sent in the authorization request. In the Cover Redirect URI attack, the access token is directly sent to the location hash of the URL which exposes it to any third party content on a client.

The vulnerabilities described have been studied and defenses have been proposed to mitigate them. These de-

fense mechanisms include a CSRF token in the login form presented to the user and checking for redirect URI mismatch on the provider's end before giving the code to the client. Unfortunately, these preventive measures are not being followed by most providers and still remains a significant threat.

The goal of this research was to investigate the weaknesses of OAuth2.0 implementation in real time. To this end, we developed a malicious application that exploits these implementation flaws to carry out the different attacks. The contributions of this research are :

- First large scale study on the various OAuth2.0 providers and analyzing their security loopholes.
- Performed the login CSRF, redirect URI mismatch and cover redirect attacks to identify the incorrect implementation scenarios.
- We experimentally evaluate the attacks against 22 of the 33 OAuth2.0 providers and discovered different incorrect implementation patterns. This is described in detail in Section 4

The previous work lacks replication of these vulnerabilities in the wild.

2. BACKGROUND

2.0.1 OAuth2.0

(A) Overview
OAuth is an authorization protocol where clients are granted access to the userâĂŹs data at the resource provider. So OAuth exposes the resources and services at the resource provider with permission from the user so that, for a duration of time, the client has limited access to the resources on the userâĂŹs behalf. So when you let an app sign in with Google or LinkedIn etc., and when it asks the user for permission, this app will have access to your email id, contacts etc. You provide the permission to access your resources hosted by the resource provider.

(B) Protocol flow
Any client that wants to implement the OAuth flow has to first to register itself as an application at the resource provider. It gets assigned a unique Client ID and secret which is kept confidential. The general workflow for obtaining the access token is the following:

- The client redirects the user to the authorization page at the resource provider with client ID, redirect URI, scope, response type and state. The response type has to be set to "code" in this step. The redirect URI should match the one specified in the application registration.
- The resource server prompts the user for its credentials and also permission to allow/deny access to the client
- Once the user authorizes the client, the authorization code is sent to the redirect URI specified in step 1.

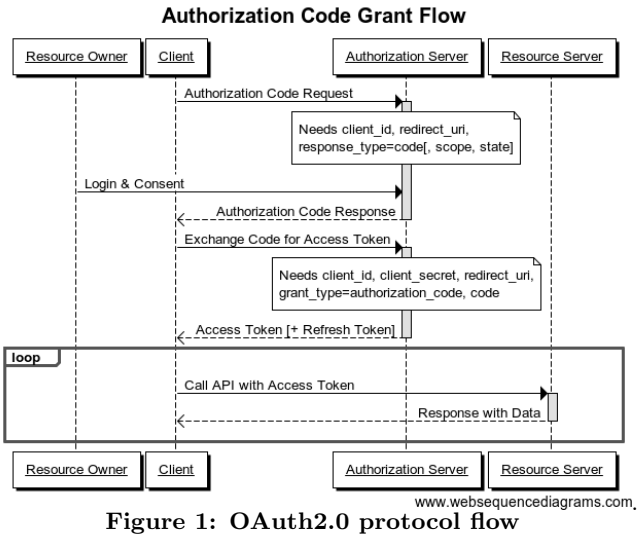

Figure 1: OAuth2.0 protocol flow

- The client then sends a POST request with the authorization code, client ID and secret, grant type set to "authorization code" and redirect URI to exchange the code for the access token. This redirect URI should ideally match the one used when requesting for the code. This token can then be used to make API calls to the resource server.

The authorization code flow can be seen in Figure 1. OAuth2.0 uses two different credentials :
Access token: It represents access provided by the user. This token can be used by the client to make API calls to the resources at the provider. For example, a client can access the files in Google drive by using the access token obtained during authorization. It eventually expires but it can also last for a long time.

Code: The code is used to get an access token. This request as mentioned should include the client secret. The code is not long lived and expires almost immediately.

2.1 Threat Model

Attackers: The attackers can be any individual or organization that intends to hijack the user's account resources and perform malicious activities on his behalf as well as passively obtaining user information by logging him in through the attacker's account. The capabilities of such an attacker need not be tremendous. He/she can be a simple weak attacker who is capable of luring the victim to a malicious application. A good understanding of the OAuth2.0 code flow can enable the attacker to tweak parts of the code to obtain information he/she requires.

3. SYSTEM OVERVIEW

For our attacks, we designed a malicious client. The application is a web server application that implements various OAuth2.0 Single sign ons. It has the following components :

- A Python web server that handles the requests and responses

- A front end templating library called Jinja2 to serve the HTML templates resolved by the server
- Deployed on Google App Engine at https://team-sso.appspot

In order to integrate the 22 single sign ons, we registered the application at the different resource providers depending on the various authorization flows that each implemented. The client ID and secret were collected and stored for all of them which were used for the various requests. The application contains SSO buttons that trigger the authorization process.

Login CSRF attack:

In the first step, when a victim lands on the homepage, he/she will see the various sign on option buttons. Once he/she clicks on a button representing a provider of their choice, they are redirected to the next page. In this transition, we forge a login request to the target provider and log in with the attacker's credentials in the background. The user now sees another login button which is clicked and he/she is taken directly to the permissions page of the OAuth flow. This way the user does not ever log in with his/her credentials. Now when the user performs subsequent actions at the provider, he/she is unknowingly passing information about their activity to the attacker. This attack can happen in any form that does not have a CSRF token in it. We found that sometimes for the normal login scenario (where you reach the login page by typing the address) and in the OAuth scenario (user is redirected to a login form), different forms are used in which case one might be safe and one might be vulnerable.

Redirect URI attacks:

(A) Redirect mismatch : The user logs in with his/her credentials. But the attacker has replaced the redirect URI in the code request to a URI of his choice. The request is sent with this URI and the response code is sent to this new redirect URI. Due to this the code is leaked to a URI other than the one registered.

(B) Cover redirect : The user logs in with his/her credentials. But the attacker has changed the response type to access token straightaway. This leaks the access token in the hash of the URL which can be seen by any third party in the page.

4. EXPERIMENTAL EVALUATION

We have carried out the attacks on 22 OAuth2.0 providers. We present the different results below.

4.1 Login CSRF

(A) *Vulnerable:* Out of the 22 we examined, 7 were vulnerable to this attack. Two scenarios for the attack : normal login page form and OAuth login form(if different forms). They are:

- **Reddit.** A form with "user" and "passwd" is sent in the form submission.
- **Formstack.** It uses different forms the two scenarios. OAuth scenario form is vulnerable to attack and normal login is non vulnerable because it contains a CSRF token field.
- **Basecamp.** It is vulnerable in both scenarios even though there is a token being generated and sent.

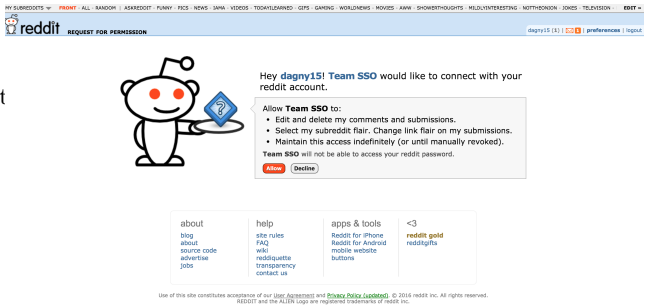

Figure 2: Login CSRF : Screenshot of Reddit

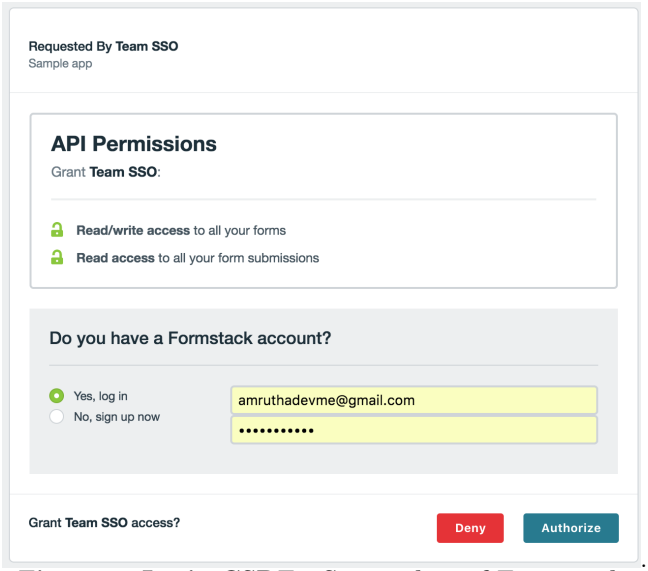

Figure 3: Login CSRF : Screenshot of Formstack

- **Yandex.** This does not have a CSRF token at all. So is vulnerable.
- **Yammer.** It is vulnerable because it redirects to an organization endpoint which has no CSRF token.
- **Fitbit.** This is vulnerable because it does not have a CSRF token at all. So request can be submitted with just credentials.
- **Imgur.** It has no CSRF token at all therefore vulnerable.

(B) *Non vulnerable:* The rest of the providers were not vulnerable because most of them included some kind of login CSRF token in their forms. VK had three randomly generated values that is sent with every login form submission instead of a CSRF token. Stripe sent a dynamically generated token value in the request but that was not present in the form itself. Salesforce had a two step verification process and sent a code request was sent in a message to your email id whenever a login was attempted. Dropbox and Battlenet had a very complex request JSON that could not have been replicated.

4.2 Redirect URI

Redirect URI mismatch. We redirected the user-agent to the authorization server, including information identify-

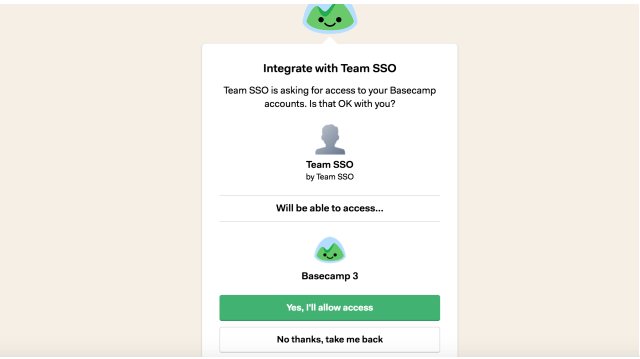

Figure 4: Login CSRF : Screenshot of Basecamp

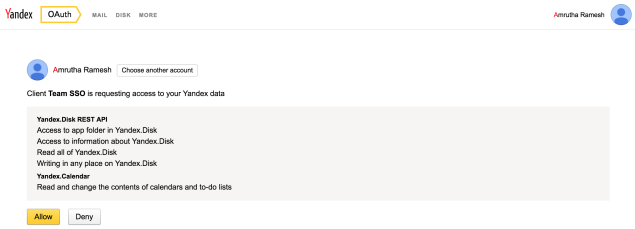

Figure 5: Login CSRF : Screenshot of Yandex

ing itself (a client id), the request (scope - the permissions being requested), and a URL pointing back to the client (here we used a redirect URL that was different from the one we used for registering the application).Ideally, the authorization server should give a redirect URI mismatch error (which we got in most cases) However, there were some providers which did not check for the mismatch and the authorization server authorized the resource owner, and performed authentication such as username/password verification, and confirmation of the action requested. On success, it directed the user-agent back to the client through the provided redirect URL, adding an authorization code to the URL. **Cover redirect.**We simply need to find an open redirect on the client's domain or its subdomains, send it as redirect uri and replace response type with token. We preserve the fragment which the attacker's JavaScript code will have access to in the location hash. Leaked access token can be used for spam and ruining your privacy.

Following are the sites which are susceptible :

(A) *Vulnerable:* The following leak the code information to a subdirectory of the registered URI :

- Box
- Formstack
- Github

  The following leak both code and token to a subdirectory :

- Foursquare
- Imgur

  The following leak code to a different malicious directory : Strava and Yahoo were found to leak the authorization code to a completely different malicious directory. New URI : https://team-sso.appspot.com/maliciousapp

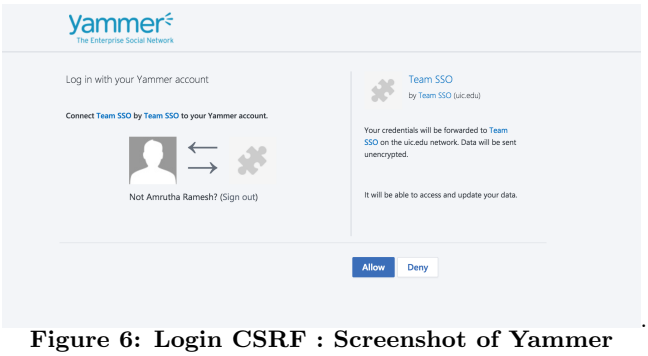

Figure 6: Login CSRF : Screenshot of Yammer

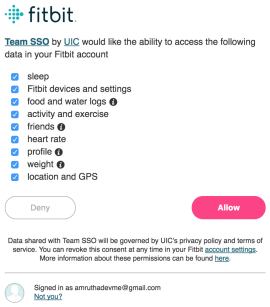

Figure 7: Login CSRF : Screenshot of Fitbit

- *Non Vulnerable:* All other providers were found to be non vulnerable.

5. DISCUSSION

The attacks done as part of this research are evidence that there are both providers that take security seriously and ones who donâĂŹt so much. By identifying through real time attacks about the ones which do not implement the defenses, we can prove that there are both popular and lesser known providers alike in the mix. One might be tempted to argue that the login CSRF attacks described are not very plausible for some of the providers which openly show who is logged in and with what email ID. But imagine an attacker who has the resources to have a mail server at his disposal who can create mail IDs that are generic and do not give away information about the logged in user. This way the victim will be made to believe that it is a guest account and nothing else. For the redirect URI attacks, by leaking code and access token information to a subdirectory of a URI can pose significant risks if the said subdirectory is part of some website like a blog that might hold user content. The defenses for these attacks must be taken care of on the resource provider side. Most of the sites have the protection enabled, but there are still websites which do not and which must.

6. RELATED WORK

The previous work in this field has only studies about the various vulnerabilities but has not covered any experiments on the existing providers. They have only explored the flaws and suggested methods for the same. The previous researchers have suggested the techniques to mitigate

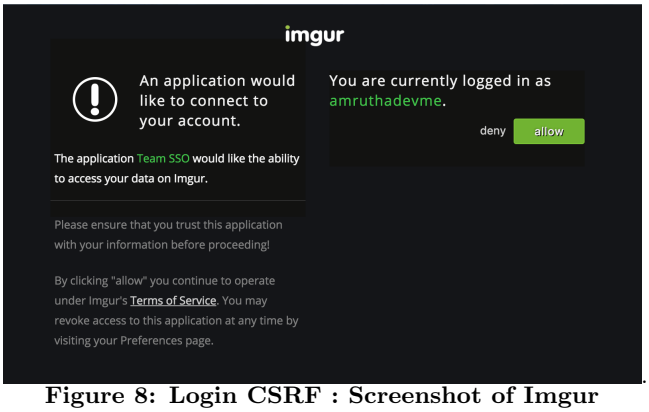

Figure 8: Login CSRF : Screenshot of Imgur

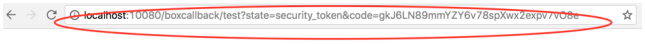

Figure 9: Screenshot of Box

the vulnerabilities. The methods suggested include keeping a check on redirect-uri mismatch and including CSRF token generally as well as for logins. These are some of the mitigation techniques that we also suggest and hope that more providers will follow the rules correctly.

7. CONCLUSIONS

In conclusion, we have performed attacks to showcase the vulnerabilities in real time. Our work gives solid evidence of the providers and the loopholes they have with respect to the various attacks. These attacks can be prevented by diligent measures on the resource provider's side.

8. REFERENCES

Figure 10: Screenshot of Formstack

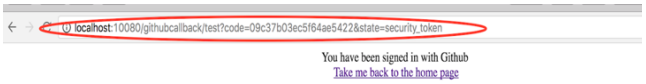

Figure 11: Screenshot of Github

Figure 12: Screenshot of Foursquare

Figure 13: Screenshot of Foursquare

Figure 14: Screenshot of Imgur

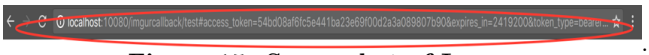

Figure 15: Screenshot of Imgur

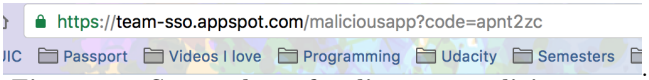

Figure 16: Screenshot of redirect to malicious app

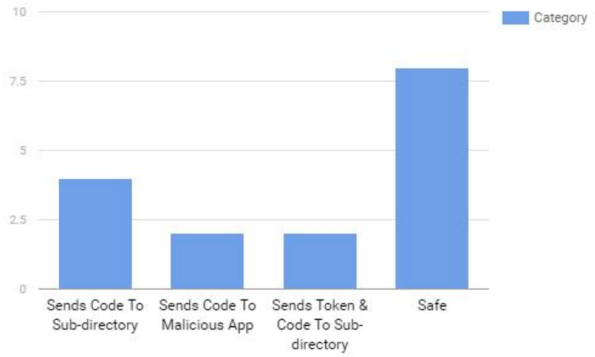

Figure 17: Statistics of redirect attack