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 authorizing 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 defense 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 user's 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 user's 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.

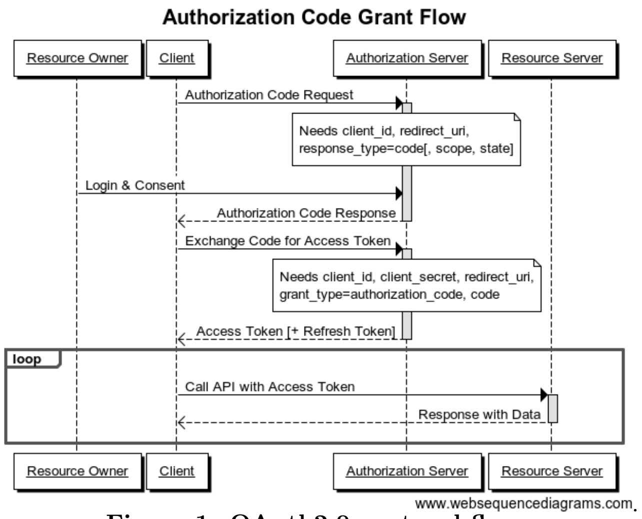

Authorization Code Grant Flow
Resource Owner
Client
Authorization Server
Resource Server
Authorization Code Request
Needs client_id, redirect_uri, response_type=code[, scope, state]
Login & Consent
Authorization Code Response
Exchange Code for Access Token
Needs client_id, client_secret, redirect_uri, grant_type=authorization_code, code
Access Token [+ Refresh Token]
loop
Call API with Access Token
Response with Data
Resource Owner
Client
Authorization Server
Resource Server
www.websequencediagrams.com

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.com

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. They are :

- Google
- Yammer
- VK
- Stripe
- Basecamp
- Zendesk
- Box
- Formstack
- Github

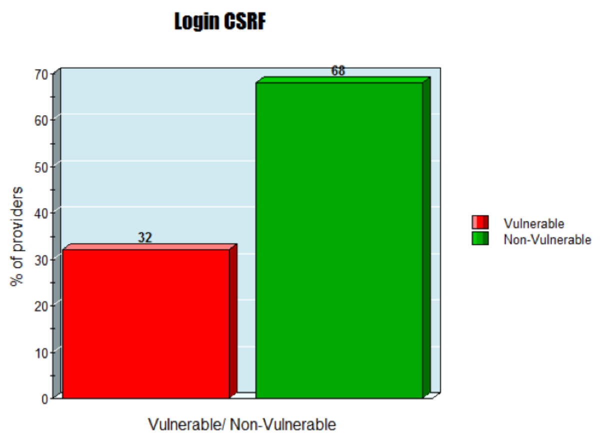


Figure 2: Login CSRF

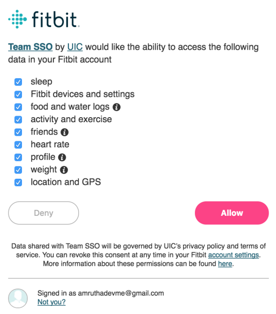
- Reddit
- Yandex
- Twitch
- Instagram
- Foursquare
- Fitbit
- Imgur
- Linkedin
- Salesforce
- Strava
- Dropbox
- Battle.net
- Yahoo

We present the different results below.

4.1 Login CSRF

Out of the 22 providers examined, 7 were vulnerable to this attack and the rest were found non vulnerable. For the vulnerability, we tested two kinds of scenarios. These scenarios depend on the login form presented to the user as part of the OAuth2.0 flow and another as part of the normal login page that a user might land on when he/she enters the website's login page address in the address bar. This is important because for some of the providers the login form presented to the user during the OAuth flow might be different from the normal login page he/she is usually redirected to. In such a case, the two login forms might not have the protection enabled in both and hence can be exploited.

(A) *Vulnerable:* The following were vulnerable to the attack :



fitbit

Team SSO by UIC would like the ability to access the following data in your Fitbit account

- ✓ sleep
- ✓ Fitbit devices and settings
- ✓ food and water logs
- ✓ activity and exercise
- ✓ friends
- ✓ heart rate
- ✓ profile
- ✓ weight
- ✓ location and GPS

Deny Allow

Data shared with Team SSO will be governed by UIC's privacy policy and terms of service. You can revoke this consent at any time in your Fitbit account settings. More information about these permissions can be found here.

Signed in as amruthadevme@gmail.com
Not you?

Figure 3: Login CSRF : Screenshot of Fitbit

- **Reddit.** An attacker has to just forge a simple form with "user" and "passwd" and submit it to the Reddit login endpoint. Here the user is redirected to the same generic login page and hence only one form is involved.
- **Formstack.** This site involves two login forms as explained above. The form in the OAuth flow is vulnerable to attack as it involves sending only the user email and password. However in the normal login page is not vulnerable because it contains a hidden CSRF token field that is sent along with user email and password form data.
- **Basecamp.** This site is vulnerable in both scenarios. There is an authenticity token being sent as part of the login request but it is not being checked by the provider consequently allowing the attacker to forge a login request.
- **Yandex.** This site is vulnerable because it does not contain any kind of a CSRF token and only user credentials are sent as part of the login request.
- **Yammer.** This site is interesting because it needs a organizational email to login. When we tested with our uic.edu email ID it redirected us to the UIC login which did not contain a CSRF token and hence could be exploited to login into Yammer.
- **Fitbit.** This site is vulnerable because it does not contain any kind of a CSRF token and only user credentials are sent as part of the login request.
- **Imgur.** This site is vulnerable because it does not contain any kind of a CSRF token and only user credentials are sent as part of the login request.

(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.

Note : Please see Appendix10 for the screenshots of other providers vulnerable to login CSRF.

4.2 Redirect URI Mismatch

We redirected the user-agent to the authorization server, including information identifying 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. We have registered the application with all the providers with the https://team-sso.appspot.com/<callbackname> and we are trying to redirect the code to an unregistered subdirectory https://team-sso.appspot.com/<callbackname>/test or a different directory https://team-sso.appspot.com/maliciousapp.

(A) *Vulnerable :* A few providers did not check for the redirect URI mismatch. This led them to send the authorization code to another directory or subdirectory of the registered redirect URI. This code could then be exchanged by the attacker for the access token. Using the access token the attacker could make API requests from the user's account and access whatever services he wants from the provider on behalf of the user. The providers who leaked and token to a subdirectory are :

- Box
- Github
- Formstack
- Foursquare
- Imgur
- Yahoo
- Strava

Among these the providers who leaked and token to another malicious directory are :

- Yahoo
- Strava

(B) *Non Vulnerable :* The other providers implemented the check for mismatch of redirect URI and did not leak the code to another URL other than the exact one registered with the provider. They all gave a general redirect URI mismatch error while attempting the attack.

Note : Please see Appendix10 for the screenshots of other providers vulnerable to redirect URI mismatch.

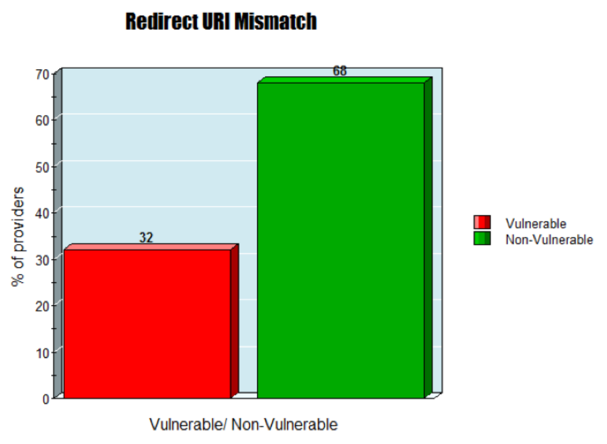


Figure 4: Redirect URI Mismatch

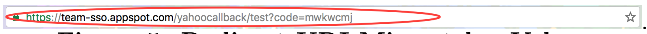

https://team-sso.appspot.com/yahoocallback/test?code=mwkwcmj

Figure 5: Redirect URI Mismatch : Yahoo

4.3 Cover redirect

The access token which is exchanged for the code should never be requested directly i.e. it should never be a direct part of the request parameters. However, many providers allowed for this to be a valid scenario. We attempted the attack by changing the response type parameter in the request to token instead of code which should ideally be invalid. We were able to get the token in the fragment of the URI which could be accessed by any third party on the page by JavaScript using location.hash and this would give them access to make API calls on behalf of the user to any provider service. This combined with the redirect URI mismatch would be even more dangerous because if the access token is given in the URL for a malicious client it can further be accessed by all the malicious third parties present on their page as well the malicious client itself.

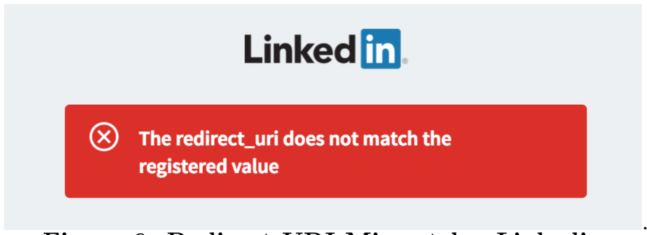
(A) *Vulnerable :* Most providers allowed this scenario for the registered redirect URI at the provider. These providers were :

- Yammer
- Yahoo
- Formstack
- VK
- Yandex
- Twitch
- Salesforce
- Dropbox
- Imgur
- Foursquare

Among these some which leaked the token in the URL for even unregistered directories and sub-directories were :

- Imgur
- Foursquare
- Formstack
- Yahoo

(B) *Non Vulnerable :* The other providers that were non-vulnerable to this attack gave a common error of bad request which indicated that the response type was invalid and token could not be requested directly. This is the correct implementation which all providers should adopt.


LinkedIn

The redirect_uri does not match the registered value

Figure 6: Redirect URI Mismatch : Linkedin

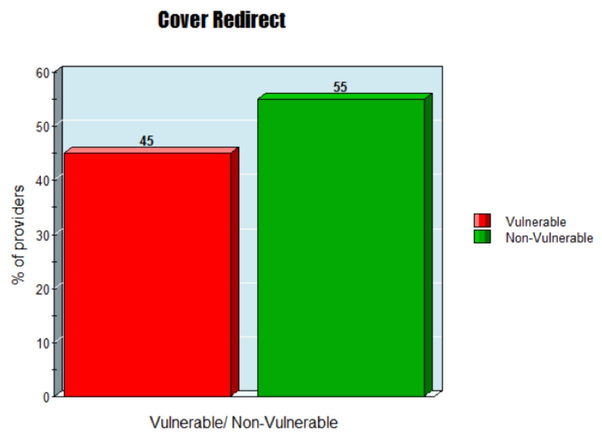


Figure 7: Cover Redirect : Salesforce

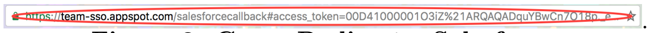
Note : Please see Appendix10 for the screenshots of other providers vulnerable to cover redirect.

In the end, we found that 32 percent of the providers were vulnerable to our redirect URI mismatch and Login CSRF attacks whereas 45 percent were vulnerable to the Cover redirect attack. Overall, 73 percent of the providers were vulnerable to one or the attack and 27 percent were not vulnerable to any.

5. DEFENSES

There are a few rules /precautionary measures which can be followed by the resource providers such that they do not give way to such vulnerable implementations. These defenses are :

- Put a hard check on redirect URI mismatch
- The only safe validation method for redirect URI is exact matching
- Do not allow direct request to access token
- Include a login CSRF token check in the request



https://team-sso.appspot.com/salesforcecallback#access_token=00D41000001O3iZ%21ARQAQADquYBwCn7O18p_e

Figure 8: Cover Redirect : Salesforce

```
{"code": 403, "error_type": "OAuthForbiddenException", "error_message": "Implicit authentication is disabled"}
```

Figure 9: Cover Redirect : Instagram

- Include some uniquely generated value in the request which prevents request forgery
- Include a two-step verification mechanism

6. DISCUSSION

The attacks done as part of this research are evidence that there are both providers that take security seriously and those who don't. 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 have them enabled. The possible solutions could be following an exact redirect URI matching approach, incorporating two-step verification, and including a login CSRF token in the request.

7. RELATED WORK

The previous work in the field of SSO vulnerabilities comprises of studies which have exposed the weaknesses in the SSO system but has not covered any experiments on the existing providers. [5] They have only explored the flaws and suggested mechanisms to mitigate the same. [4] Also, the previous researchers have suggested the techniques to mitigate the vulnerabilities. Hence, most of the work is theoretical and doesn't focus on the real-time attacks. Inspiring from this fact, we have done a first large study of the OAuth 2.0 providers.

8. CONCLUSIONS

In conclusion, we have performed a first large scale study on 22 OAuth 2.0 providers and analyed the security holes prevalent in them. We have developed three attacks viz. Login CSRF, redirect mismatch, and cover redirect to attack the providers in the wild and found that 32 percent providers are vulnerable to redirect URI mismatch, 45 percent to cover redirect, and 32 percent to login CSRF. Overall, 73 percent of the providers are vulnerable to the one or the other attacks. Our work gives solid evidence of the provider's behavior with respect to various attacks. We believe that this work is one of its kind and it can be further extended to develop concrete mitigation policies by the OAuth providers.

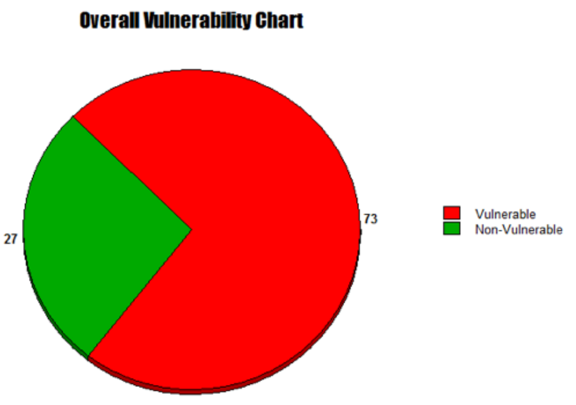
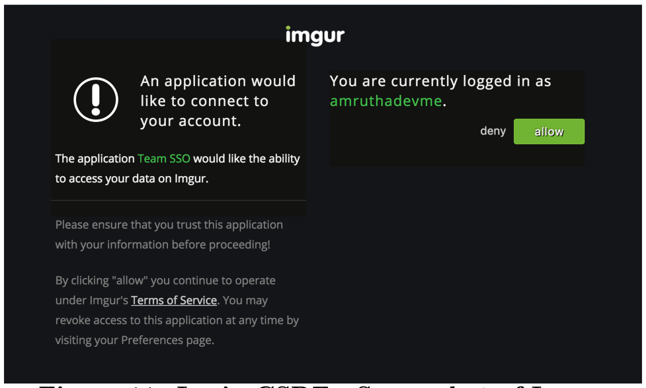


Figure 10: Overall Vulnerability Chart



imgur

An application would like to connect to your account.

The application Team SSO would like the ability to access your data on Imgur.

Please ensure that you trust this application with your information before proceeding!

By clicking "allow" you continue to operate under Imgur's Terms of Service. You may revoke access to this application at any time by visiting your Preferences page.

You are currently logged in as amruthadevme.

Figure 11: Login CSRF : Screenshot of Imgur

9. REFERENCES

[1] OAuth. Available at https://hueniverse.com/oauth/.

[2] OAuth Security Cheat Sheet. Available at http://www.oauthsecurity.com/.

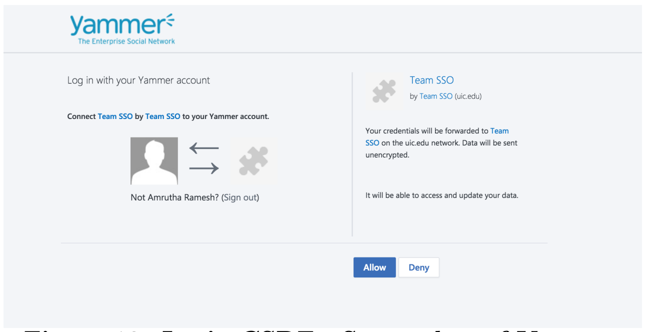
[3] Top 10 OAuth 2 Implementation Vulnerabilities. Available at http://blog.intothesymmetry.com/2015/12/top-10-oauth-2-implementation.html.

[4] A. Barth, C. Jackson, and J. C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *ACM CCS 2008.*

[5] B. Prabakaran, G. Athisenbagam, and K. T. Ganesh. Identifying Robust Defenses for Login CSRF. Available at https://www.cs.uic.edu/~bprabaka/LoginCSRF.pdf.

[6] E. Shernan, H. Carter, D. Tian, P. Traynor, and K. Butler. More Guidelines Than Rules: CSRF Vulnerabilities from Noncompliant OAuth 2.0 Implementations. In *Detection of Intrusions and Malware, and Vulnerability Assessment, pp.239-260.*

10. APPENDIX



Log in with your Yammer account

Connect **Team SSO** by **Team SSO** to your Yammer account.

Not Amrutha Ramesh? (Sign out)

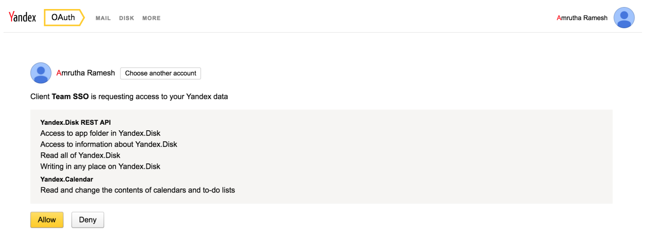
Team SSO

by Team SSO (uic.edu)

Your credentials will be forwarded to Team SSO on the uic.edu network. Data will be sent unencrypted.

It will be able to access and update your data.

Figure 12: Login CSRF : Screenshot of Yammer

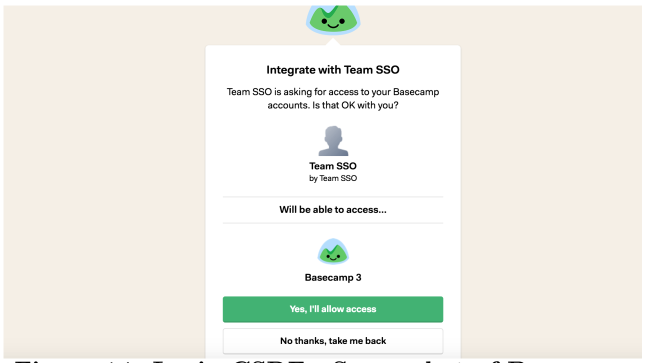


Amrutha Ramesh Choose another account

Client **Team SSO** is requesting access to your Yandex data

Yandex.Disk REST API
Access to app folder in Yandex.Disk
Access to information about Yandex.Disk
Read all of Yandex.Disk
Writing in any place on Yandex.Disk
Yandex.Calendar
Read and change the contents of calendars and to-do lists

Figure 13: Login CSRF : Screenshot of Yandex



Integrate with Team SSO

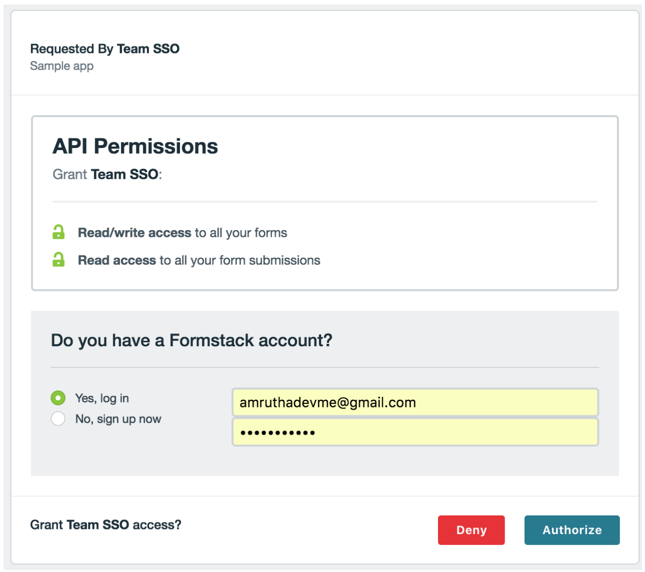
Team SSO is asking for access to your Basecamp accounts. Is that OK with you?

Team SSO
by Team SSO

Will be able to access...

Basecamp 3

Figure 14: Login CSRF : Screenshot of Basecamp



Requested By **Team SSO**
Sample app

API Permissions

Grant **Team SSO**:

Read/write access to all your forms

Read access to all your form submissions

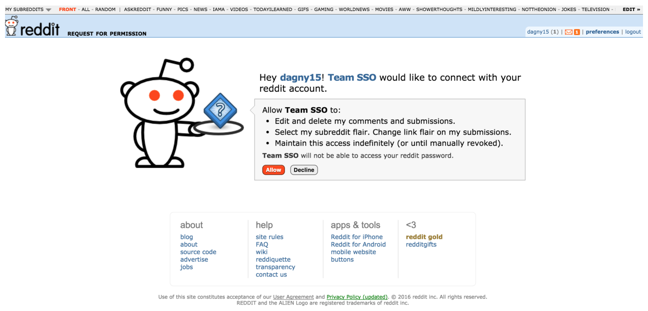
Do you have a Formstack account?

[✓ Yes, log in]
No, sign up now

amruthadevme@gmail.com
•••••••••••

Grant **Team SSO** access?

Figure 15: Login CSRF : Screenshot of Formstack


MY SUBREDDITS FRONT - ALL - RANDOM | ASKREDDIT - FUNNY - PICS - NEWS - IAMA - VIDEOS - TODAYILEARNED - GIFS - GAMING - WORLDNEWS - MOVIES - AWW - SHOWERTHOUGHTS - MILDLYINTERESTING - NOTTHEONION - JOKES - TELEVISION - EDIT »
reddit REQUEST FOR PERMISSION
dagny15 (1) | 1 | preferences | logout

Hey **dagny15**! **Team SSO** would like to connect with your reddit account.

Allow **Team SSO** to:

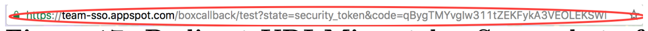
- Edit and delete my comments and submissions.
- Select my subreddit flair. Change link flair on my submissions.
- Maintain this access indefinitely (or until manually revoked).

Team SSO will not be able to access your reddit password.

Allow
Decline
about
blog
about
source code
advertise
jobs
help
site rules
FAQ
wiki
reddiquette
transparency
contact us
apps & tools
Reddit for iPhone
Reddit for Android
mobile website
buttons
<3
reddit gold
redditgifts

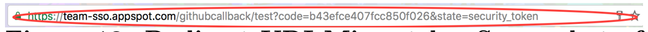
Use of this site constitutes acceptance of our User Agreement and Privacy Policy (updated). © 2016 reddit inc. All rights reserved.
REDDIT and the ALIEN Logo are registered trademarks of reddit inc.

Figure 16: Login CSRF : Screenshot of Reddit



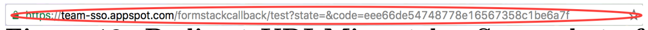
https://team-sso.appspot.com/boxcallback/test?state=security_token&code=qBygTMYvgIw311tZEKFykA3VEOLEKSWI

Figure 17: Redirect URI Mismatch : Screenshot of Box



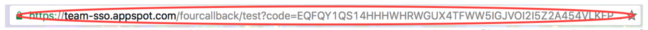
https://team-sso.appspot.com/githubcallback/test?code=b43efce407fcc850f026&state=security_token

Figure 18: Redirect URI Mismatch : Screenshot of Github



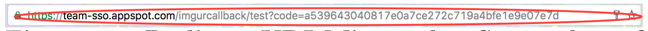
https://team-sso.appspot.com/formstackcallback/test?state=&code=eee66de54748778e16567358c1be6a7f

Figure 19: Redirect URI Mismatch : Screenshot of Formstack



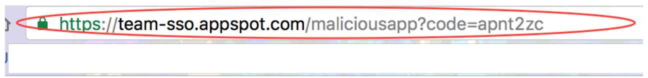
https://team-sso.appspot.com/fourcallback/test?code=EQFQY1QS14HHHWHRWGUX4TFWW5IGJVOI2I5Z2A454VLKEP

Figure 20: Redirect URI Mismatch : Screenshot of Foursquare



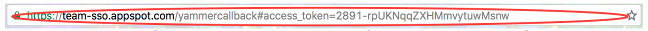
https://team-sso.appspot.com/imgurcallback/test?code=a539643040817e0a7ce272c719a4bfe1e9e07e7d

Figure 21: Redirect URI Mismatch : Screenshot of Imgur



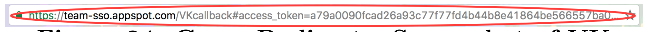
https://team-sso.appspot.com/maliciousapp?code=apnt2zc

Figure 22: Redirect URI Mismatch : Screenshot



https://team-sso.appspot.com/yammercallback#access_token=2891-rpUKNqqZXHMmvytuwMsnw

Figure 23: Cover Redirect : Screenshot of Yammer



https://team-sso.appspot.com/VKcallback#access_token=a79a0090fcad26a93c77f77fd4b44b8e41864be566557ba0...

Figure 24: Cover Redirect : Screenshot of VK