
Prof. Wing C. Lau
Joint-work with Pili Hu, Ronghai Yang
Department of Information Engineering
The Chinese University of Hong Kong
for
APAC Innovation Summit
Dec 2, 2014
What is OAuth?

- OAuth = Open Authentication Protocol
- The most commonly used Single-Sign-On Protocol on the Web
- Widely used for User Authentication and App Authorization
- Adopted by a wide range of Online Social Networks and Cloud Service Providers
- Currently deployed version: OAuth 2.0
  - Security analysis: RFC6819, Jan 2013
Who are using OAuth 2.0?
Outline

- OAuth Background
  - Authorization
  - Authentication
- Authorization
  - App Impersonation
- Authentication
  - User Impersonation
- OAuthTester: Large-scale, Automated Testing of OAuth2.0 Implementations in the Wild
Three Different Parties under the OAuth Framework

- **Cloud-based Service Provider (e.g. Facebook)**
  - ![Social Media Logos] ...

- **User (e.g. User)**
  - Register user account on Provider
  - Operate various data objects

- **App (e.g. WeChat)**
  - Register developer account on Provider
  - Get data objects access permission from
    - Provider: via application/approval
    - User: via OAuth
  - AppID, **AppSecret** (shared b/w App & Provider)
Basic Interaction among App, User and Provider

Let me post status for you

Let App post status for me

Here’s the proof for App to do this

Post status “Wing is in APAC’2015”. Here’s the proof

Depending on different modes in OAuth, the proof can be a “Code” and/or an “Access Token”

- Authorization: post status, get friend-list etc.
- Authentication: retrieve user’s identity data
1) User enter the App’s webpage

Hyperlink ↔ Facebook

Connect your friends from Facebook

Connect Facebook Now

As always, we won’t do anything that you wouldn’t do. Please see our pledge of privacy to learn more.
2) App redirects to User to Provider

3a) User enters authentication info (username + password)
3b) User explicitly authorizes the App (after reviewing the scope to be granted)
Hyperlink → facebook

Your authorization for facebook is succeeded. This window will be automatically closed.
Detail of OAuth Authorization Code Grant Call flow

1. Enter App
2. Redirect to Provider
3. Authorization
4. Code
5. Code (via redirect)
6. Code + AppSec
7. Access Token (AT)

Request: Parameters + AT + AppSec (optional)

Response
An Alternative Call-flow for OAuth: Implicit Grant

Implicit Grant Flow

1. App front page
2. Redirect to Provider
3. Authorization
4. Access Token
5. Access Token (AT)

Request: Parameters + AT + AppSec (optional)
Response
OAuth Background

Two Authorization Flow Types

Authorization code grant flow

Implicit grant flow

Key difference in Implicit Grant flow:
- Access Token (AT) is returned via User
Authorization:
  ● Use AccessToken to make operations on behalf of the user
    o Post status/photo
    o retrieve user’s data, e.g., friend-list, posts, check-in
    o ...

Authentication:
  ● Use AccessToken to retrieve user’s profile to assume user’s identity
    o https://graph.facebook.com/v2.2/me
    o The ID of user account, the email, ...
Outline

- OAuth Background
  - Authorization
  - Authentication
- Authorization
  - App Impersonation
- Authentication
  - User Impersonation
- OAuthTester: Large-scale, Automated Testing of OAuth 2.0 Implementations in the Wild
App Impersonation Attack
Key Ideas

Key idea:
- Attacker (Any User) Gets/Uses AccessToken without AppSecret
- AccessToken gives the privilege of “App +User” or “App”

How is this possible?
App Impersonation Attack
Key Ideas:

- Access token is returned **through** the User
- No App Secret is required
App Impersonation Attack: Made Possible by OAuth 2.0

OAuth 2.0 allows an App User to:

- Get AccessToken without AppSecret: because of most providers allow “Implicit grant flow” regardless of whether the App actually needs it!

- Use AccessToken without AppSecret: as long as the provider supports the “Bearer token” mode of operation

Once the above conditions are satisfied,
⇒ Any user can impersonate an App without knowing the shared secret (AppSecret) between App and Provider
App Impersonation Attack Illustration

Go to normal authorization page

Hyperlink will receive the following info: your public profile, friend list, email address, News Feed, birthday, status updates, checkins, website and personal description and your friends' birthdays, status updates, checkins, websites and personal descriptions.
App Impersonation Attack Illustration

Change `response_type` to "token"

Hyperlink will receive the following info: your public profile, friend list, email address, News Feed, birthday, status updates, checkins, website and personal description and your friends' birthdays, status updates, checkins, websites and personal descriptions.
App Impersonation Attack Illustration

Access token obtained!

Hyperlink ↔ facebook

Unfortunately, the authorization fails.
App Impersonation Attack
Illustration

```
%cat post-status-fb.sh
#!/bin/bash

access_token="CAAEdrgfH...

curl -F "access_token=$access_token" \
    -F 'message=Test post from curl' \
    https://graph.facebook.com/me/feed

%./post-status-fb.sh
{"id":"100002175400771_682335645182276"}
```

Can be done fully in browser if the endpoint uses GET method.
Or with the help of some browser extensions/developer tools.
How vulnerable are the Major Service Providers?

Providers we studied:

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Registered Users</th>
<th>Alexa Rank*</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Provider 1</td>
<td>&gt;300,000,000</td>
<td>≤ 10</td>
</tr>
<tr>
<td>P2</td>
<td>Provider 2</td>
<td>&gt;200,000,000</td>
<td>≤ 1000</td>
</tr>
<tr>
<td>P3</td>
<td>Provider 3</td>
<td>&gt;300,000,000</td>
<td>≤ 20</td>
</tr>
<tr>
<td>P4</td>
<td>Provider 4</td>
<td>&gt;100,000,000</td>
<td>≤ 50</td>
</tr>
<tr>
<td>P5</td>
<td>Provider 5</td>
<td>&gt;200,000,000</td>
<td>≤ 10</td>
</tr>
<tr>
<td>P6</td>
<td>Provider 6</td>
<td>&gt;300,000,000</td>
<td>≤ 10</td>
</tr>
<tr>
<td>P8</td>
<td>Provider 8</td>
<td>Not Found</td>
<td>≤ 20</td>
</tr>
<tr>
<td>P7</td>
<td>Provider 7</td>
<td>&gt;5,000,000</td>
<td>≤ 200</td>
</tr>
<tr>
<td>P9</td>
<td>Provider 9</td>
<td>&gt;200,000,000</td>
<td>≤ 10</td>
</tr>
<tr>
<td>P10</td>
<td>Provider 10</td>
<td>&gt;200,000,000</td>
<td>≤ 10</td>
</tr>
<tr>
<td>P11</td>
<td>Provider 11</td>
<td>&gt;200,000,000</td>
<td>≤ 20</td>
</tr>
<tr>
<td>P12</td>
<td>Provider 12</td>
<td>&gt;300,000,000</td>
<td>≤ 10</td>
</tr>
</tbody>
</table>
Summary Findings

Properties of 12 major Online Service Providers:

<table>
<thead>
<tr>
<th>Property</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
<th>P8</th>
<th>P9</th>
<th>P10</th>
<th>P11</th>
<th>P12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implicit Grant Flow</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Implicit Grant Flow Opt-Out</td>
<td>Y¹</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Authorization Code Grant Flow</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Bearer Token</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Bearer Token Opt-Out</td>
<td>Y¹</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>MAC Token</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Scope Dimensions</td>
<td>3D</td>
<td>2D</td>
<td>3D</td>
<td>2D</td>
<td>3D</td>
<td>3D</td>
<td>2D</td>
<td>3D</td>
<td>0D</td>
<td>3D</td>
<td>2D</td>
<td>2D</td>
</tr>
<tr>
<td>Scope Constraint</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Rate Control (/IP)</td>
<td>Y²</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Rate Control (/User /App)</td>
<td>Y²</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Rate Control (/App)</td>
<td>N²</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Rate Control (/User)</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>App Differentiation</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>

- Provide implicit-grant-flow and bearer token w/o opt-out → App Impersonation
- 8 out of 12 providers under studied are vulnerable to App Impersonation
What are the Consequences?
Case 1: App Reputation Attack

As the demo of App Impersonation on FB:

- Available on: Provider 1, Provider 2, Provider 3, Provider 6, Provider 9, Provider 10 and Provider 11
- More useful on open OSN (e.g. micro-blogging type)
Case 2: Privilege Escalation and Rate Amplification

- **Rate Amplification:**
  - Get more API access rate than entry level allowed
  - Provider 2, Provider 3 and Provider 9

- **Privilege Escalation:**
  - Get more scopes that are separately approved
  - Provider 3, Provider 4, Provider 6, Provider 10 and Provider 11

**Note:**
- Submission/Approval on some OSN is tedious
- App impersonation makes it cost effective to access much more resources
Case 3: Massive Crawling

- Provider 2: A Facebook-like (not Facebook) OSN with >100 million users
- Inconsistent access permissions:
  - Scope:
    - "read_status" ==
    - "read_self_status" +
    - "read_friend_status" +
    - "read_other_status"
Case 3: Massive Crawling by Impersonating a Privileged App

Apps are differentiated on Provider 2:
- Normal App: 200 Queries/hour
- Some higher level App: 900 Queries/hour
  \( \Rightarrow \) Takes years to collect the data even if it’s “public”

Happen to find one App with:
1 million queries/hour
  \( \Rightarrow \) only 100 hours to complete the crawl
  \( \Rightarrow \) < US$100 cost on Amazon Web Service
Case 4: Send Push Notifications with Embedded URL to Million Strangers

- Make possible by another Design problem of this Provider in mixing up:
  - User operations:
    - Implicit-flow-grant
    - Authorization-code-grant
  - App operations:
    - Client App Credentials Grant (or variants)
Case 5: Massive Connection Establishment for Sybil Account

- Social promotion campaign: more eye balls in form of followers
- Leverage reciprocal following behaviour (See Kaggle 2012 FB challenge)
- Follow/unfollow other users to acquire a lot of followers
- Rate is a bottleneck (usually more restricted than “read” operations)
Suggested Immediate Fixes for App Impersonation Attack

- Provider let each App Opt-out/ Opt-in for Implicit Grant flow
- Provider let each App Opt-out/ Opt-in for Bearer Token type
- Review “scope” design in Access Control Matrix
- Review Rate Limiting/Control mechanism/ granularity: not only a per App quota
- Review policy for, and Closely monitor Privileged Apps
Outline

- OAuth Background
  - Authorization
  - Authentication
- Authorization
  - App Impersonation
- Authentication
  - User Impersonation
- OAuthTester: Large-scale, Automated Testing of OAuth2.0 Implementations in the Wild
Workflow for User Impersonation via CSRF Attack: Without STATE

User (Victim)
1). Victim logs into App with username and password

2). Login Successfully
   set-cookie: user=Victim

App
3). Attacker visits App
   4). redirect_uri

Attacker
5). authentication & authorization
   6). Access Token

Provider

7). User visits the malicious website hosted by the Attacker

8). Send malicious page <img src=https://skype? token=attacker’s Access Token&...

   cookie: user=Victim

10). https://Graph.facebook.com/v2.2/me?token=attacker’s Access Token

11). Attacker’s account information: email, username, etc, got bound to Victim’s App account

Attacker can log into the application as the Victim & Bind User’s App account to Attacker’s account @ Provider
Two Required conditions for User Impersonation (w/o STATE)

2 Required Conditions:
- User has already logged into App before visiting the Attacker’s malicious website
- App supports the so-called “Identity federation”

How to ensure Step 9 is not a CSRF attack?

use the **STATE** parameter in OAuth
User Impersonation
Identity Federation

User can login App via two ways:
- OAuth service provided by Provider
  - IdP-managed account
- Local login system
  - App-managed account

Hi Ronghai Yang, you’re one step away
Signed in with Facebook account. Change account?

Already using Skype?
Link to your existing Skype account to access your credit and all your contacts in one place.

New to Skype?
Welcome! It’s quick and easy to join.

I have a Skype account
I’m new to Skype
Thwarting User Impersonation with the OAuth STATE parameter

1. User visits App
2. redirect_uri+STATE
3. authentication & authorization
4. Access Token+STATE
5. redirection endpoint: Access Token+STATE
5. forged request: attacker’s Access Token+attacker’s STATE

Session cookies

hash (cookies || nonce)

STATE =? hash (cookies || nonce)
User Impersonation Attack Call flow

The attacker MUST get the **STATE** value of the victim.

User (Victim) → App → Attacker → Provider

1). Victim logs into App with username and password

2). Login Successfully
   
   set-cookie: user=Victim ;
   
   victim’s STATE parameter

3). Attacker visits App

4). redirect_uri+ STATE

5). authentication & authorization

6). Access Token+STATE

7). User visits the malicious website hosted by the Attacker

8). Send malicious page `<img src=https://skype? token=attacker’s Access Token&...`

9). CSRF: https://skype.xxx?
   
   token=attacker’s Access Token

   &state= victim’s STATE&...

   cookie: user=Victim

10). https://graph.facebook.com/v2.2/me?token=attacker’s Access Token

11). Attacker’s account information: email, username, etc

User Impersonation becoming much more difficult with the use of the STATE parameter in OAuth
User Impersonation Attack Call flow

1. Victim logs into App with username and password
2. Login successfully
   \[\text{set-cookie: user=Victim}\]
3. Attacker visits App
4. redirect_uri+ STATE
5. authentication & authorization
6. Access Token+STATE
7. User visits the malicious website hosted by the Attacker
8. Send malicious page
   \[\text{<img src=https://skype?token=attacker’s Access Token&…}\]
9. CSRF: https://skype,xxx?
   \[\text{token=attacker’s Access Token}\]
   \[\text{cookie: user=Victim}\]
10. https://graph.facebook.com/v2.2/me?
    \[\text{token=attacker’s Access Token}\]
    \[\text{state= victim’s STATE&…}\]
11. Attacker’s account information: email, username, etc

User Impersonation becomes Possible again if the STATE parameter is NOT checked or implemented properly.
Examples of Failing to use STATE properly:

- Lack of Validation
- STATE parameter is missing
- Applications do not bind STATE to individual user and recognize all the STATE parameters generated by itself.

- STATE Replay Attack
  - STATE can be used for multiple times until User logs the App again
  - STATE can be reused as long as User uses the same browser
  - STATE parameter is a constant number
# Statistics on Flawed OAuth Implementations in the Wild

<table>
<thead>
<tr>
<th>IdPs (No. of Apps)</th>
<th>No Validation %</th>
<th>STATE is missing %</th>
<th>STATE do not bind to individual user %</th>
<th>State Replay</th>
<th>Summary %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>State multiple use</td>
<td>Same State One Browser</td>
</tr>
<tr>
<td>Provider A (79)</td>
<td>12.50%</td>
<td>14.29%</td>
<td>21.42%</td>
<td>5.37%</td>
<td>14.29%</td>
</tr>
<tr>
<td>Provider B (182)</td>
<td>13.32%</td>
<td>13.32%</td>
<td>59.97%</td>
<td>38.66%</td>
<td>10.68%</td>
</tr>
<tr>
<td>Provider C (68)</td>
<td>18.17%</td>
<td>9.09%</td>
<td>30.31%</td>
<td>18.19%</td>
<td>12.12%</td>
</tr>
<tr>
<td>Provider D (46)</td>
<td>12.50%</td>
<td>12.50%</td>
<td>12.50%</td>
<td>37.50%</td>
<td>12.50%</td>
</tr>
<tr>
<td>Average (405)</td>
<td>13.09%</td>
<td>11.85%</td>
<td>37.53%</td>
<td>18.52%</td>
<td>11.36%</td>
</tr>
</tbody>
</table>
OAuth Problems Outline

- OAuth Background
  - Authorization
  - Authentication
- Authorization
  - App Impersonation
- Authentication
  - User Impersonation
- OAuthTester: Large-scale, Automated Testing of OAuth2.0 Implementations in the Wild
OAuthTester: Motivations

Many attacks due to Improper Implementation/Deployment of OAuth2.0 have been discovered, e.g.:

- Covert redirect: open redirector
- Social CSRF: Get the victim’s Code/AccessToken to login App as the victim
- Session fixation: cookie same origin policy violation and much more...

Too easy for App/Provider to implement OAuth correctly

=> Need large-scale, Automated Testing
**OAuth Tester Threat Model**

- Alice follows normal instructions to complete OAuth2.0 protocol
- Eve can send arbitrary request to any party of OAuth
# User Impersonation Statistics of misuse cases

<table>
<thead>
<tr>
<th>IdPs (No. of Apps)</th>
<th>No Validation</th>
<th>STATE is missing</th>
<th>STATE do not bind to individual user</th>
<th>State Replay</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>State multiple use</td>
<td>Same State one Browser</td>
</tr>
<tr>
<td>Provider A (79)</td>
<td>12.50%</td>
<td>14.29%</td>
<td>21.42%</td>
<td>5.37%</td>
<td>14.29%</td>
</tr>
<tr>
<td>Provider B (182)</td>
<td>13.32%</td>
<td>13.32%</td>
<td>59.97%</td>
<td>38.66%</td>
<td>10.68%</td>
</tr>
<tr>
<td>Provider C (68)</td>
<td>18.17%</td>
<td>9.09%</td>
<td>30.31%</td>
<td>18.19%</td>
<td>12.12%</td>
</tr>
<tr>
<td>Provider D (46)</td>
<td>12.50%</td>
<td>12.50%</td>
<td>12.50%</td>
<td>37.50%</td>
<td>12.50%</td>
</tr>
<tr>
<td>Average (405)</td>
<td>13.09%</td>
<td>11.85%</td>
<td>37.53%</td>
<td>18.52%</td>
<td>11.36%</td>
</tr>
</tbody>
</table>
Other problems discovered by OAuthTester

- Failure to revoke authorization
- Failure to re-authenticate App
- Provider Leaking App Secret
- No TLS protection of STATE parameter
- Multiple OAuth flows with Dual-role Providers
## Summary of Findings

<table>
<thead>
<tr>
<th>IdPs (No. of Apps)</th>
<th>STATE Misuse</th>
<th>Multiple Authorization flow</th>
<th>No TLS of STATE (IdP/App)</th>
<th>Fail to Revoke Authorization</th>
<th>Fail to Re-authenticate App</th>
<th>App Secret Leak</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provider A (79)</td>
<td>35.72%</td>
<td>N</td>
<td>N/ 12.15%</td>
<td>N</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>Provider B (182)</td>
<td>74.67%</td>
<td>Y</td>
<td>N/ 69.33%</td>
<td>Y</td>
<td>N</td>
<td>0</td>
</tr>
<tr>
<td>Provider C (68)</td>
<td>65.13%</td>
<td>Y</td>
<td>Y / 51.51%</td>
<td>Y</td>
<td>N</td>
<td>0</td>
</tr>
<tr>
<td>Provider D (46)</td>
<td>50.00%</td>
<td>N</td>
<td>N / 87.50%</td>
<td>N.A</td>
<td>Y</td>
<td>0</td>
</tr>
<tr>
<td>Average (405)</td>
<td>55.31%</td>
<td>/</td>
<td>41.32%</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
</tbody>
</table>
Reflections

- OAuth 2.0 has diverse implementations that differ from specification
- New attacking surface: App Impersonation
- App Impersonation combined with other flaws can result in serious exploits
- Protecting App is a MUST when designing the next generation of the OAuth protocol
- Large-scale, Automated Testing for OAuth implementations is needed
Related Publications

- P.Hu and W.C. Lau, “How to Leak a 100-million-node Social Graph in just one week?” Black Hat USA, Las Vegas, Aug. 2014.
Thanks & Q/A

Visit our Project Homepage at:

http://mobitec.ie.cuhk.edu.hk/oauth/

for more details
Backup Slides
OAuthTester
A State Machine

A state of the system:
- current status of the system
  - Alice Status
  - Eve Status
  - App_IdP
- state transition
  - enabled conditions: get all possible states
  - choose one state based on coverage criterion
- normal response of state transition
  - normal behavior profile
Fail to Revoke Authorization once it is granted to an App

Provider X and Provider Y automatically issue a new access token (code) to applications without User’s authorization and awareness.

**Consequence:** App can quietly access User’s data without User’s confirmation and awareness.
Failure to Re-authenticate App

- Access token supposes to be valid only for a short duration
- But App can use App secret and refresh token to exchange a fresh access token
- However, Provider Z does not authenticate the App (App secret) before granting a refresh!
No TLS Protection of STATE Parameter by Provider

- Provider Y in Step 3

CSRF: https://imdb.xxx?code=attacker’s Access Token&state=victim’s state&...

https://graph.xxx/oauth/grant?client_id=xxx&redirect_uri=xxx&state=8378&...

http://graph.xxx/oauth/grant?client_id=xxx&redirect_uri=xxx&state=8378&...
No TLS Protection of STATE Parameter by App

- Provider Y in the Step 3
- Apps in the redirection endpoint
  - not confidential if implement well (App revokes the STATE)
  - State Relay attack

![Diagram showing the flow of user visits App, followed by a redirect with URI and STATE, then a redirection endpoint with code and STATE.](image)
App Secret Leakage

- example:
  https://www.xxx.com/xxx?
  client_id=xxx&client_secret=c80cc85...&redirect_uri=https://xxx.com

- Security Impact:
  make operations on behalf the App, such as sending notification as the App, get User’s data, update the application settings, etc.

Multiple OAuth Flows with Dual-role Providers

- Dual-role Provider
  - On one hand it is a Provider of OAuth service
  - On the other hand, it plays the role of an App for other OAuth Providers

Two Phase Attack:
- Log into Dual-role Provider as the Attacker
  - Misuse of the STATE parameter
  - CSRF attack on the Login form
  - ...
- Bind the Attacker’s Dual-role Provider account with the Victim’s App account
  - Work even if Victim’s App is implemented correctly
Multiple OAuth Flows with Dual-role Providers

- 10 out of 13 providers are Dual-role IdPs
- 8 providers are vulnerable.