OAuth by Sakurity

OAuth by Sakurity is significantly more secure and simple authorization protocol than official one. It fixes huge security and usability gaps in design (read the section below how OAuth2 was vulnerable to every possible attack) and only takes 5 minutes to understand how it works. We do encourage everyone to use this and drop OAuth 2 support.

1. The user clicks "Connect Provider" button on the client (a website or a mobile app). The click initiates a POST form submission (protected with CSRF token) to `/connect/provider` or navigates with GET to `/connect/provider?csrf_token=TOKEN`. The initial step MUST be protected from CSRF, it shouldn't be possible to trigger connection from other origins, there must be an explicit confirmation from the user.

2. On the server side `/connect/provider` endpoint should verify that csrf_token is equal the token from the session, then generate state token (separate CSRF token to ensure OAuth flow integrity) and store it in the session under `provider_name-state`. Then redirect to provider.com/oauth/authorize?client_id=CLIENT_ID&redirect_uri=CLIENT/callback&state=STATE. scope parameter is optional (some critical scopes should only be allowed to verified clients).

3. Provider checks that given redirect_uri is inside of whitelist of redirect URIs. The provider must store two whitelists of redirect_uris: one for hash (analog of response_type=token, implicit flow) and one for query (analog for response_type=code). It MUST NOT be a parameter, the type of response (# or ?) is based solely on in what list supplied redirect_uri was found.

4. The state parameter MUST be required by the provider and be at least 10 letters (despite spec says it is optional - for most developers it is synonim for "not needed"). It may end up as &state=1234567890 for many, but it will force the developers to pay a little attention to what this attribute is for and give us a better chance they protect CLIENT-PROVIDER-CLIENT chain of redirects from CSRF.

5. The user sees Approve/Deny buttons on provider domain along with list of scopes the client asks for. After clicking Approve POST form is submitted to the same URL, protected with CSRF token.

6. For query type of redirect, send the user agent to CLIENT/callback?access_token=TOKEN&state=STATE (all tokens are stored on central server and every API request is signed with appsecret_proof), for hash redirect to CLIENT/callback#access_token=TOKEN&state=STATE (tokens are stored in local applications, appsecret_proof is not required). In both cases (websites and mobile apps) clients MUST check the state is equal provider-state from the session.

7. Final step: the token received on implicit type redirect_uri (a JS app or a mobile app) is good to be used with API without any signatures. The token received using server side flow (formerly response_type=code) can also be used right away and the provider MUST require clients to send appsecret_proof along (HMAC of the access_token using client_secret). No need for `code` nor `refresh_token` because each request is authenticated with valid client_secret and access_token alone is not enough. In both cases access_token never expires.

Why access_token never expires?

With refresh token attackers get a window when they can use access_tokens to send spam or download social graph of all victims. Only when access_tokens expire, according to spec, clients should use refresh_token and client_secret to get a new token. Instead of this security theater, we are going to require appsecret_proof every time.

Every request to Provider MUST have access_token and `appsecret_proof=hash_hmac('sha256', $access_token, $client_secret);` if the token was returned with response_type=query. Now if the attacker leaks access_token, tokens are worthless. Even if they steal your client_secret, you can simply rotate it.

And there's no round trip to exchange code for token?

Yes, that request was simply an equivalent of refresh_token request, just using another parameter and extra value in your database. Since all tokens are authenticated, there's no need for `code` too.

Wait, no more response_type in initial request?

Right, we don't need it. Every redirect_uri is already whitelisted, and has a corresponding response_type. This way an XSS on client website wouldn't be able to get implicit kind of access_token (not requiring client_secret) by simply passing a response_type=token.

Are you sure state should be a required parameter?

Absolutely. The provider must raise an exception if one isn't supplied. The web isn't a game where you can routinely protect or not protect an interaction based on redirects from spoofing.

How two lists of URIs work?

For example two comma separated lists, one for query response `https://client/callback` and one for hash response `my_app_on_ios8765://callback`. There's no need for domain whitelisting. In fact, having website X you can use same userbase for Y without making them authenticate again, just add new redirect_uri to the list.

What about signed_request?

Signed_request is a great way to securely transfer provider uid to the client without extra requests to the provider, but it was never intended to be part of this authorization protocol. Feel free to make your access_tokens look like signed_request.

Furthermore, to preven One Token to Rule Them all attack, you can store all the data under explicit `{"client_id_123345":{"uid":"UID","token":"TOKEN"}}` so the check that token was issued for your client will be built-in by design.

OAuth Security Cheatsheet

This document describes common OAuth2/Single Sign On/OpenID-related vulnerabilities. Many cross-site interactions are vulnerable to different kinds of leakings and hijackings. None of these issues is present in the OAuth by Sakurity specification presented above.

Both hackers and developers can benefit from reading it.

OAuth is a critical functionality. It is responsible for access to sensitive user data, authentication and authorization. **Poorly implemented OAuth is a reliable way to take over an account**. Unlike XSS, it is easy to exploit, but hard to mitigate for victims (NoScript won't help, JavaScript is not required).

Because of OAuth many startups including Soundcloud, Foursquare, Airbnb, About.me, Bit.ly, Pinterest, Digg, Stumbleupon, Songkick had an account hijacking vulnerability. And a lot of websites are still vulnerable. **Our motivation is to make people aware of "Social login" risks, and we encourage you to use OAuth very carefully.**

The cheatsheet **does not** explain how OAuth flows work, please look for it on the official website.

Authorization Code flow

Client account hijacking by connecting attacker's provider account.

Also known as The Most Common OAuth2 Vulnerability. In other words, CSRF.

Provider returns `code` by redirecting user-agent to `SITE/oauth/callback?code=CODE` Now the client must send `code` along with client credentials and `redirect_uri` to obtain `access_token`.

If the client implementation doesn't use `state` parameter to mitigate CSRF, we can easily connect **our provider account** to **the victim's client account**.

It works for clients with social login and ability to add a login option to existing master account (screenshots of pinterest.com below).

Remediation: Before sending user to the provider generate a random nonce and save it in cookies or session. When user is back make sure `state` you received is equal one from cookies.

State fixation bug: It was possible to fixate `state` in omniauth because of legacy code which utilized user supplied `/connect?state=user_supplied` instead of generating a random one.

This is another OAuth design issue - sometimes developers want to use `state` for own purposes. Although you can send both values concatenated `state=welcome_landing.random_nonce`, but no doubt it looks ugly. A neat solution is to use JSON Web Token as state

Client account hijacking through abusing session fixation on the provider

Even if the client properly validates the `state` we are able to replace auth cookies on the provider with the attacker's account: using CSRF on login (VK, Facebook), header injection, or cookie forcing or tossing.

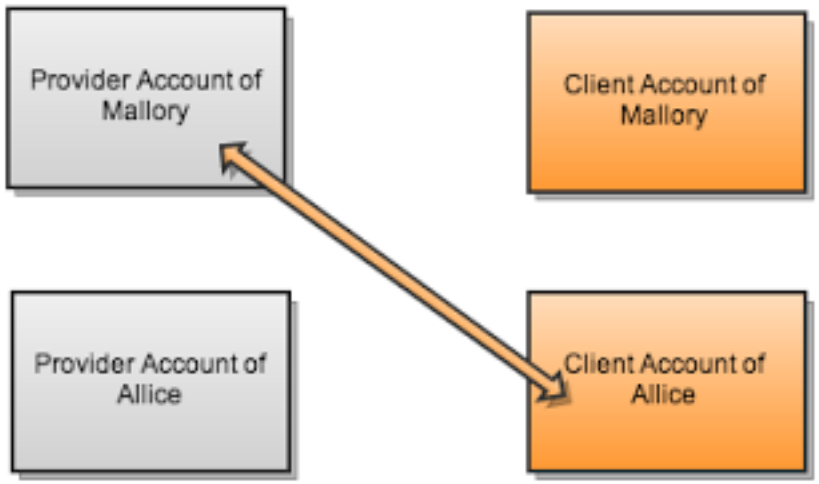

Figure 1:

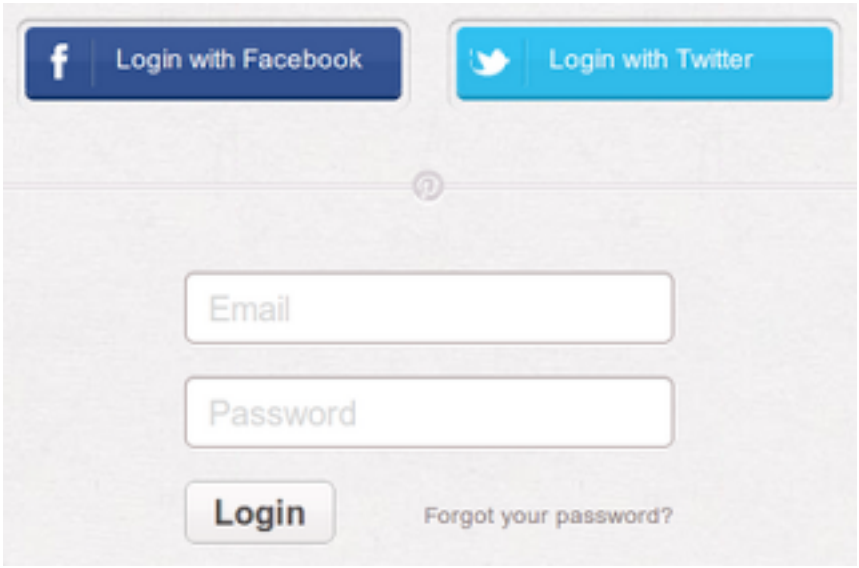

Figure 2:

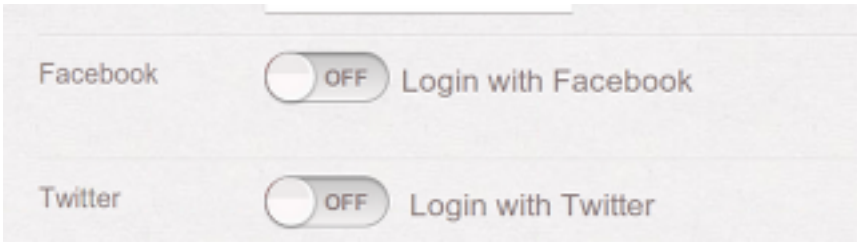

Figure 3: connect options

Then we just load a GET request triggering connect (`/user/auth/facebook` in omniauth), Facebook will respond with the attacker's user info (uid=attacker's uid) and it will eventually connect the attacker's provider account to the victim's client account.

Remediation: make sure that adding a new social connection requires a valid csrf_token, so it is not possible to trigger the process with CSRF. Ideally, use POST instead of GET.

Facebook refused to fix CSRF on login from their side, and many libraries remain vulnerable. **Do not expect providers to give you reliable authentication data**.

Account hijacking by leaking authorization code.

OAuth documentation makes it clear that providers must check the first `redirect_uri` is equal `redirect_uri` the client uses to obtain `access_token`. We didn't really check this because it looked too hard to get it wrong. Surprisingly **many** providers got it wrong: Foursquare (reported), VK (report, in Russian), Github (could be used to leak tokens to private repos), and a lot of "home made" Single Sign Ons.

The attack is straightforward: find a leaking page on the client's domain, insert cross domain image or a link to your website, then use this page as `redirect_uri`. When your victim will load crafted URL it will send him to `leaking_page?code=CODE` and victim's user-agent will expose the code in the Referrer header.

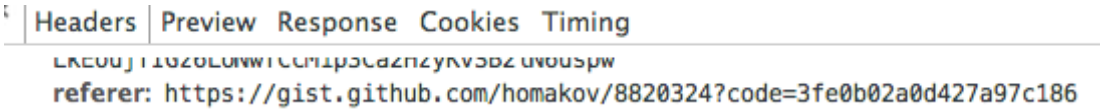

Figure 4:

Now you can re-use leaked authorization code on the actual `redirect_uri` to log in the victim account.

Remediation: flexible `redirect_uri` is a bad practise. But if you need it, store redirect_uri for every code you issue and verify it on access_token creation.

Implicit flow

Leaking access_token/signed_request with an open redirect.

It was a media hype called "covert redirect" but in fact it was known for years. You 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,signed_request`. 302 redirect will preserve #fragment, and attacker's Javascript code will have access to location.hash.

Leaked `access_token` can be used for spam and ruining your privacy. Furthermore, leaked `signed_request` is even more sensitive data. By finding an open redirect on the client you compromise Login with Facebook completely.

Remediation: whitelist only one redirect_uri in app's settings:

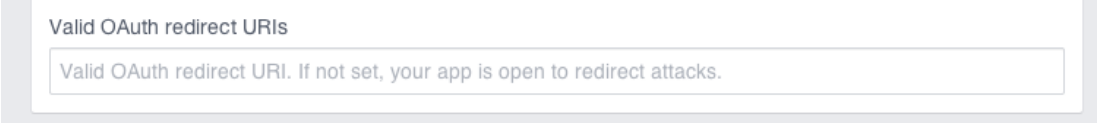

Figure 5:

Account hijacking by using access_token issued for the attacker's client.

Also known as One Token to Rule Them All. This bug is relevant to mobile and client-side apps, because they often use access_token directly supplied by the user.

Imagine, user has many "authorization rings" and gives a ring to every new website where he wants to log in. A malicious website admin can use rings of its users to log in other websites his customers have accounts on.

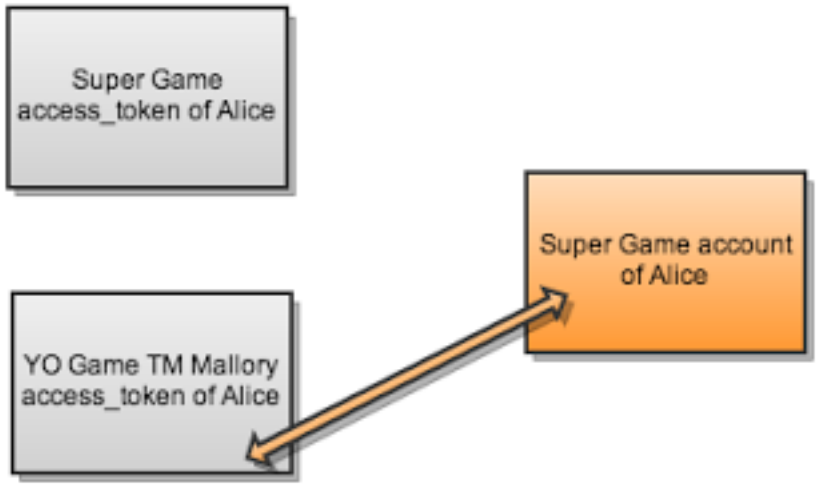

Figure 6:

Remediation: Before accepting user supplied access_token check if it was issued for your `client_id` at `https://graph.facebook.com/app?fields=id&access_token=TOKEN`

Transport and JS SDK bugs

For better support of client-side applications (or browser apps) some of OAuth 2 providers added an out-of-specs custom variant of implicit flow, involving a proxy/relay page as a sort of router for access tokens. This proxy communicates with the client page of application either through standard HTML5 postMessage API, or using legacy ways: manipulating window name and url (known as Fragment transport, rmr) and Adobe Flash LocalConnection API.

Regardless of the selected method, it is important to ensure proxy can exchange messages only with the target origin, to disallow malicious page from impersonating the application

page, and finally to verify these client-side checks are consistent with server-side checks. Fail properly secure client-side transport usually results in two kinds of problems: * leaking data to an attacker (authorization and authentication bypass [1], [2]) * trusting data from an attacker (session fixation [1], [2])

Additionally, modern web-applications tend to have rich javascript frontend, and unsafe client-side communication may lead to even more serious attacks beyond OAuth itself (DOM XSS on Facebook through OAuth 2 logical flaws)

Remediation: * Completely disable all legacy client-side communication methods (Flash or Fragment), use postMessage * Check origins of all incoming and outgoing messages, allow only the target application domain * Ensure that all client-side origin checks are consistent with server-side origin checks * Use a cryptographic nonce validation before starting data transmission with another party

Extra

Leaked client credentials threat

Client credetials are not as important as it sounds. All you can do is using leaking pages to leak auth code, then manually getting an access_token for them (providing leaking redirect_uri instead of actual). Even this threat can be mitigated when providers use static redirect_uri.

Session fixation (OAuth1.0)

Main difference between OAuth2 and 1 is the way you transfer parameters to providers. In the first version you send all parameters to the provider and obtain according request_token. Then you navigate user to `provider?request_token=TOKEN` and after authorization user is redirected back to `client/callback?request_token=SAME_TOKEN`.

The idea of fixation here is we can trick user into accepting Token1 supplied by us which was issued for us, then re-use Token1 on the client's callback.

This is not a severe vulnerability because it is mostly based on phishing. FYI Paypal express checkout has this bug

Provider In The Middle.

Many startups have Facebook Connect, and at the same time they are providers too. Being providers, they must redirect users to 3rd party websites, and those are "open redirects" you just cannot fix. It makes this chain possible: Facebook -> Middleware Provider -> Client's callback, leading to FB token leakage.

To fix this problem Facebook adds `#_=_` in the end of callback URLs. Your startup should "kill" fragment to prevent leaking. Redirect this way:

```
Location: YOUR_CLIENT/callback?code=code#
```

Tricks to bypass redirect_uri validation

If you are allowed to set subdirectory here are path traversal tricks:

1. /old/path/../../new/path
2. /old/path/%2e%2e/%2e%2e/new/path
3. /old/path/%252e%252e/%252e%252e/new/path
4. /new/path///../../old/path/
5. /old/path/.%0a./.%0d./new/path

Replay attack.

`code` is sent via GET and potentionally will be stored in the logs. Providers must delete it after use or expire in 5 minutes.

Leaking a code with an open redirect

Usually you need a referrer-leaking page to leak ?query parameters. There are two tricks to do it with an open redirect though: * When redirect uses `<meta>` tag instead of 302 status and Location header. It will leak redirecting page's referrer to in the next request. * When you managed to add `%23`(#) in the end of `redirect_uri`. It will result in sending the code in the fragment `Location: https://CLIENT/callback/../open_redirect?to=evil#&code=CODE`