Open Authentication Protocol
Internet-Draft
Intended status: Informational
Expires: May 16, 2017

T. Lodderstedt, Ed.
Deutsche Telekom AG
J. Bradley
Ping Identity
A. Labunets
Facebook

November 12, 2016

OAuth Security Topics
draft-lodderstedt-oauth-security-topics-00

Abstract

This draft gives a comprehensive overview on open OAuth security topics. It is intended to serve as a tool for the OAuth working group to systematically address these open security topics, recommending mitigations, and potentially also defining OAuth extensions needed to cope with the respective security threats. This draft will potentially become a BCP over time.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

It's been a while since OAuth has been published in RFC 6749 [RFC6749] and RFC 6750 [RFC6750]. Since publication, OAuth 2.0 has gotten massive traction in the market and became the standard for API protection and, as foundation of OpenID Connect, identity providing.

- OAuth implementations are being attacked through known implementation weaknesses and anti-patterns (XSRF, referrer header). Although most of these threats are discussed in RFC 6819 [RFC6819], continued exploitation demonstrates there may be a need for more specific recommendations or that the existing mitigations are too difficult to deploy.

- Technology has changed, e.g. the way browsers treat fragments in some situations, which may change the implicit grant's underlying security model.

- OAuth is used in much more dynamic setups than originally anticipated, creating new challenges with respect to security. Those challenges go beyond the original scope of both RFC 6749 [RFC6749] and RFC 6819 [RFC6819].

This remainder of the document is organized as follows: The next section describes various scenarios how OAuth credentials (namely access tokens and authorization codes) may be disclosed to attackers and proposes countermeasures. Afterwards, the document discusses attacks possible with captured credential and how they may be prevented. The last sections discuss additional threats.

2. OAuth Credentials Leakage

2.1. Redirect URI validation of authorization requests

The following implementation issue has been observed: Some authorization servers allow clients to register redirect URI patterns instead of complete redirect URIs. In those cases, the authorization servers, at runtime, match the actual redirect URI parameter value at the authorization endpoint against this pattern. This approach allows clients to encode transaction state into additional redirect URI parameters or to register just a single pattern for multiple redirect URIs. As a downside, it turned out to be more complex to implement and error prone to manage than exact redirect URI matching. Several successful attacks have been observed in the wild, which utilized flaws in the pattern matching implementation or concrete configurations. Such a flaw effectively breaks client identification or authentication (depending on grant and client type) and allows the attacker to obtain an authorization code or access token, either

- by directly sending the user agent to a URI under the attackers control or

- usually via the client as open redirector in conjunction with fragment handling (implicit grant) carrying the response including the respective OAuth credentials.

2.1.1. Authorization Code Grant

For a public client using the grant type code, an attack would look as follows:

Let's assume the pattern "https://*.example.com/*" had been registered for the client "s6BhdRkqt3". This pattern allows redirect URI from any host residing in the domain example.com. So if an attacker manager to establish a host or subdomain in "example.com" he

can impersonate the legitimate client. Assume the attacker sets up the host "evil.example.com".

(1) The attacker needs to trick the user into opening a tampered URL in his browser, which launches a page under the attacker's control, say "https://www.evil.com".

(2) This URL initiates an authorization request with the client id of a legitimate client to the authorization endpoint. This is the example authorization request (line breaks are for display purposes only):

```
GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=xyz
  &redirect_uri=https%3A%2F%2Fevil.client.example.com%2Fcb HTTP/1.1
Host: server.example.com
```

(3) The authorization validates the redirect URI in order to identify the client. Since the pattern allows arbitrary domains host names in "example.com", the authorization request is processed under the legitimate client's identity. This includes the way the request for user consent is presented to the user. If auto-approval is allowed (which is not recommended for public clients according to RFC 6749), the attack can be performed even easier.

(4) If the user does not recognize the attack, the code is issued and directly sent to the attacker's client.

(5) Since the attacker impersonated a public client, it can directly exchange the code for tokens at the respective token endpoint.

Note: This attack will not work for confidential clients, since the code exchange requires authentication with the legitimate client's secret. The attacker will need to utilize the legitimate client to redeem the code. This and other kinds of injections are covered in Section OAuth Credentials Injection.

2.1.2. Implicit Grant

The attack described above for grant type authorization code works similarly for the implicit grant. If the attacker is able to send the authorization response to a URI under his control, he will directly get access to the fragment carrying the access token.

Additionally, it is possible to conduct an attack utilizing the way user agents treat fragments in case of redirects. User agents re-attach fragments to the destination URL of a redirect if the location header does not contain a fragment (see [RFC7231], section 9.5). In

this attack this behavior is combined with the client as an open redirector in order to get access to access tokens. This allows circumvention even of strict redirect URI patterns.

Assume the pattern for client "s6BhdRkqt3" is "https://client.example.com/cb?*", i.e. any parameter is allowed for redirects to "https://client.example.com/cb". Unfortunately, the client exposes an open redirector. This endpoint supports a parameter "redirect_to", which takes a target URL and will send the browser to this URL using a HTTP 302.

(1) Same as above, the attacker needs to trick the user into opening a tampered URL in his browser, which launches a page under the attacker's control, say "https://www.evil.com".

(2) The URL initiates an authorization request, which is very similar to the attack on the code flow. As differences, it utilizes the open redirector by encoding "redirect_to=https://client.evil.com" into the redirect URI and it uses the response type "token" (line breaks are for display purposes only):

```
GET /authorize?response_type=token&client_id=s6BhdRkqt3&state=xyz
  &redirect_uri=https%3A%2F%2Fclient.example.com%2Fcb%26redirect_to
  %253Dhttps%253A%252F%252Fclient.evil.com%252Fcb HTTP/1.1
Host: server.example.com
```

(3) Since the redirect URI matches the registered pattern, the authorization server allows the request and sends the resulting access token with a 302 redirect (some response parameters are omitted for better readability)

```
HTTP/1.1 302 Found
  Location: https://client.example.com/cb?
  redirect_to%3Dhttps%3A%2F%2Fclient.evil.com%2Fcb
  #access_token=2YotnFZFEjr1zCsicMWpAA&...
```

(4) At the example.com, the request arrives at the open redirector. It will read the redirect parameter and will issue a HTTP 302 to the URL "https://evil.example.com/cb".

```
HTTP/1.1 302 Found
     Location: https://client.evil.com/cb
```

(5) Since the redirector at example.com does not include a fragment in the Location header, the user agent will re-attach the original fragment

"#access_token=2YotnFZFEjr1zCsicMWpAA&..." to the URL and will navigate to the following URL:

https://client.evil.com/cb#access_token=2YotnFZFEjr1zCsicMWpAA&...

(6) The attacker's page at client.evil.com can access the fragment and obtain the access token.

2.1.3. Countermeasure: exact redirect URI matching

Since the cause of the implementation and management issues is the complexity of the pattern matching, this document proposes to recommend general use of exact redirect URI matching instead, i.e. the authorization server shall compare the two URIs using simple string comparison as defined in [RFC3986], Section 6.2.1..

This would cause the following impacts:

o This change will require all OAuth clients to maintain the transaction state (and XSRF tokens) in the "state" parameter. This is a normative change to RFC 6749 since section 3.1.2.2 allows for dynamic URI query parameters in the redirect URI. In order to assess the practical impact, the working group needs to collect data whether this feature is used in deployed reality today.

o The working group might also consider this change as a step towards improved interoperability for OAuth implementations since RFC 6749 is somehow vague on redirect URI validation. There is especially no rule for pattern matching. So one may assume all clients utilizing pattern matching will do so in a deployment specific way. On the other hand, RFC 6749 already recommends exact matching if the full URL had been registered.

o Clients with multiple redirect URIs need to register all of them explicitly.
Note: clients with just a single redirect URI would not even need to send a redirect URI with the authorization request. Does it make sense to emphasize this option? Would that further simplify use of the protocol?

o Exact redirect matching does not work for native apps utilizing a local web server due to dynamic port numbers - at least wild cards for port numbers are required.
Note: Does redirect uri validation solve any problem for native apps? Effective against impersonation when used in conjunction with claimed HTTPS redirect URIs only.

Additional recommendations:

- It is also advisable that the domains on which callbacks are hosted should not expose open redirectors (see respective section).

- As a further recommendation, clients may drop fragments via intermediary URL with fix fragment (e.g. https://developers.facebook.com/blog/post/552/) to prevent the user agent from appending any unintended fragments.

Alternatives to exact redirect URI matching: authenticate client using digital signatures (JAR? https://tools.ietf.org/html/draft-ietf-oauth-jwsreq-09), ...

2.2. Authorization code leakage via referrer headers

The section above already discussed use of the referrer header for one kind of attack to obtain OAuth credentials. It is also possible authorization codes are unintentionally disclosed to attackers, if a OAuth client renders a page containing links to other pages (ads, faq, ...) as result of a successful authorization request.

If the user clicks onto one of those links and the target is under the control of an attacker, it can get access to the response URL in the referrer header.

It is also possible that an attacker injects cross-domain content somehow into the page, such as <img> (f.e. if this is blog web site etc.): the implication is obviously the same - loading this content by browser results in leaking referrer with a code.

2.2.1. Countermeasures

There are some means to prevent leakage as described above:

- Use of the HTML link attribute rel="noreferrer" (Chrome 52.0.2743.116, FF 49.0.1, Edge 38.14393.0.0, IE/Win10)

- Use of the "referrer" meta link attribute (possible values e.g. noreferrer, origin, ...) (cf. https://w3c.github.io/webappsec-referrer-policy/ - work in progress (seems Google, Chrome and Edge support it))

- Redirect to intermediate page (sanitize history) before sending user agent to other pages
  Note: double check redirect/referrer header behavior

o Use form post mode instead of redirect for authorization response

Note: There shouldn't be a referer header when loading HTTP content from a HTTPS -loaded page (e.g. help/faq pages)

Note: This kind of attack is not applicable to the implicit grant since fragments are not be included in referrer headers (cf. https://tools.ietf.org/html/rfc7231#section-5.5.2)

2.3. Code in browser history (TBD)

When browser navigates to "client.com/redirection_endpoint?code=abcd" as a result of a redirect from a provider's authorization endpoint.

Proposal for counter-measures: code is one time use, has limited duration, is bound to client id/secret (confidential clients only)

2.4. Access token in browser history (TBD)

When a client or just a web site which already has a token deliberately navigates to a page like provider.com/ get_user_profile?access_token=abcdef.. Actually RFC6750 discourages this practice and asks to transfer tokens via a header, but in practice web sites often just pass access token in query

When browser navigates to client.com/ redirection_endpoint#access_token=abcef as a result of a redirect from a provider's authorization endpoint.

Proposal: replace implicit flow with postmessage communication

2.5. Access token on bad resource servers (TBD)

In the beginning, the basic assumption of OAuth 2.0 was that the OAuth client is implemented for and tightly bound to a certain deployment. It therefore knows the URLs of the authorization and resource servers upfront, at development/deployment time. So well-known URLs to resource servers serve as trust anchor. The validation whether the client talks to a legitimate resource server is based on TLS server authentication (see [RFC6819], Section 4.5.4).

As OAuth clients nowadays more and more bind dynamically at runtime to authorization and resource servers, there need to be alternative solutions to ensure, client do not deliver access tokens to bad resource servers.

...

Potential mitigations:

- PoP
- Token Binding
- AS-provided Meta Data
- ...

2.6. Mix-Up (TBD)

Mix-up is another kind of attack on more dynamic OAuth scenarios (or at least scenarios where a OAuth client interacts with multiple authorization servers). The goal of the attack is to obtain an authorization code or an access token by tricking the client into sending those credentials to the attacker (which acts as MITM between client and authorization server)

A detailed description of the attack and potential counter-measures is given in cf. https://tools.ietf.org/html/draft-ietf-oauth-mix-up-mitigation-01.

Potential mitigations:

- AS returns client_id and its iss in the response. Client compares this data to AS it believed it sent the user agent to.
- ID token (so requires OpenID Connect) carries client id and issuer
- register AS-specific redirect URIs, bind transaction to AS
- ...

3. OAuth Credentials Injection

Credential injection means an attacker somehow obtained a valid OAuth credential (code or token) and is able to utilize this to impersonate the legitimate resource owner or to cause a victim to access resources under the attacker's control (XSRF).

3.1. Code Injection

In such an attack, the adversary attempts to inject a stolen authorization code into a legitimate client on a device under his control. In the simplest case, the attacker would want to use the code in his own client. But there are situations where this might not be possible or intended. Example are:

- o The code is bound to a particular confidential client and the attacker is unable to obtain the required client credentials to redeem the code himself and/or

- o The attacker wants to access certain functions in this particular client. As an example, the attacker potentially wants to impersonate his victim in a certain app.

- o Another example could be that access to the authorization and resource servers is some how limited to networks, the attackers is unable to access directly.

How does an attack look like?

(1) The attacker obtains an authorization code by executing any of the attacks described above (OAuth Credentials Leakage).

(2) It performs an OAuth authorization process with the legitimate client on his device.

(3) The attacker injects the stolen authorization code in the response of the authorization server to the legitimate client.

(4) The client sends the code to the authorization server's token endpoint, along with client id, client secret and actual redirect_uri.

(5) The authorization server checks the client secret, whether the code was issued to the particular client and whether the actual redirect URI matches the redirect_uri parameter.

(6) If all checks succeed, the authorization server issues access and other tokens to the client.

(7) The attacker just impersonated the victim.

Obviously, the check in step (5) will fail, if the code was issued to another client id, e.g. a client set up by the attacker.

An attempt to inject a code obtained via a malware pretending to be the legitimate client should also be detected, if the authorization server stored the complete redirect URI used in the authorization request and compares it with the redirect_uri parameter.

[RFC6749], Section 4.1.3, requires the AS to ... "ensure that the "redirect_uri" parameter is present if the "redirect_uri" parameter was included in the initial authorization request as described in Section 4.1.1, and if included ensure that their values are

identical." In the attack scenario described above, the legitimate client would use the correct redirect URI it always uses for authorization requests. But this URI would not match the tampered redirect URI used by the attacker (otherwise, the redirect would not land at the attackers page). So the authorization server would detect the attack and refuse to exchange the code.

Note: this check could also detect attempt to inject a code, which had been obtained from another instance of the same client on another device, if certain conditions are fulfilled:

- o the redirect URI itself needs to contain a nonce or another kind of one-time use, secret data and

- o the client has bound this data to this particular instance

But this approach conflicts with the idea to enforce exact redirect URI matching at the authorization endpoint. Moreover, it has been observed that providers very often ignore the redirect_uri check requirement at this stage, maybe, because it doesn't seem to be security-critical from reading the spec.

Other providers just pattern match the redirect_uri parameter against the registered redirect URI pattern. This saves the authorization server from storing the link between the actual redirect URI and the respective authorization code for every transaction. But this kind of check obviously does not fulfill the intent of the spec, since the tampered redirect URI is not considered. So any attempt to inject a code obtained using the client_id of a legitimate client or by utilizing the legitimate client on another device won't be detected in the respective deployments.

It is also assumed that the requirements defined in [RFC6749], Section 4.1.3, increase client implementation complexity as clients need to memorize or re-construct the correct redirect URI for the call to the tokens endpoint.

The authors therefore propose to the working group to drop this feature in favor of more effective and (hopefully) simpler approaches to code injection prevention as described in the following section.

3.1.1. Proposed Counter Measures

The general proposal is to bind every particular authorization code to a certain client on a certain device (or in a certain user agent) in the context of a certain transaction. There are multiple technical solutions to achieve this goal:

Nonce   OpenID Connect's existing "nonce" parameter is used for this purpose. The nonce value is one time use and created by the client. The client is supposed to bind it to the user agent session and sends it with the initial request to the OpenId Provider (OP). The OP associates the nonce to the authorization code and attests this binding in the ID token, which is issued as part of the code exchange at the token endpoint. If an attacker injected an authorization code in the authorization response, the nonce value in the client session and the nonce value in the ID token will not match and the attack is detected. assumption: attacker cannot get hold of the user agent state on the victims device, where he has stolen the respective authorization code.
pro:
- existing feature, used in the wild
con:
- OAuth does not have an ID Token - would need to push that down the stack

State   It has been discussed in the security workshop in December to use the OAuth state value much similar in the way as described above. In the case of the state value, the idea is to add a further parameter state to the code exchange request. The authorization server then compares the state value it associated with the code and the state value in the parameter. If those values do not match, it is considered an attack and the request fails. Note: a variant of this solution would be send a hash of the state (in order to prevent bulky requests and DoS).
pro:
- use existing concept
con:
- state needs to fulfil certain requirements (one time use, complexity)
- new parameter means normative spec change

PKCE   Basically, the PKCE challenge/verifier could be used in the same way as Nonce or State. In contrast to its original intention, the verifier check would fail although the client uses its correct verifier but the code is associated with a challenge, which does not match.
pro:
- existing and deployed OAuth feature
con:
- currently used and recommended for native apps, not web apps

Token Binding Code must be bind to UA-AS and UA-Client legs -
requires further data (extension to response) to manifest
binding id for particular code.
pro:
- highly secure
con:
- highly sophisticated, requires browser support, will it
work for native apps?

per instance client id/secret ...

Note on pre-warmed secrets: An attacker can circumvent the counter-measures described above if he is able to create the respective secret on a device under his control, which is then used in the victim's authorization request.
Exact redirect URI matching of authorization requests can prevent the attacker from using the pre-warmed secret in the faked authorization transaction on the victim's device.
Unfortunately it does not work for all kinds of OAuth clients. It is effective for web and JS apps, for native apps with claimed URLs.
What about other native apps? Treat nonce or PKCE challenge as replay detection tokens (needs to ensure cluster-wide one-time use)?

3.1.2. Access Token Injection (TBD)

Note: An attacker in possession of an access token can access any resources the access token gives him the permission to. This kind of attacks simply illustrates the fact that bearer tokens utilized by OAuth are reusable similar to passwords unless they are protected by further means.
(where do we treat access token replay/use at the resource server? https://tools.ietf.org/html/rfc6819#section-4.6.4 has some text about it but is it sufficient?)

The attack described in this section is about injecting a stolen access token into a legitimate client on a device under the adversaries control. The attacker wants to impersonate a victim and cannot use his own client, since he wants to access certain functions in this particular client.

Proposal: token binding, hybrid flow+nonce(OIDC), other cryptographical binding between access token and user agent instance

3.1.3. XSRF (TBD)

injection of code or access token on a victim's device (e.g. to cause client to access resources under the attacker's control)

mitigation: XSRF tokens (one time use) w/ user agent binding (cf. https://www.owasp.org/index.php/ CrossSite_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet)

4. Other Attacks

Using the AS as Open Redirector - error handling AS (redirects) (draft-ietf-oauth-closing-redirectors-00)

Using the Client as Open Redirector

redirect via status code 307 - use 302

5. Acknowledgements

We would like to thank ... for their valuable feedback.

6. IANA Considerations

This draft includes no request to IANA.

7. Security Considerations

All relevant security considerations have been given in the functional specification.

8. Normative References

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <http://www.rfc-editor.org/info/rfc3986>.

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <http://www.rfc-editor.org/info/rfc6749>.

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", RFC 6750, DOI 10.17487/RFC6750, October 2012, <http://www.rfc-editor.org/info/rfc6750>.

[RFC6819] Lodderstedt, T., Ed., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, DOI 10.17487/RFC6819, January 2013, <http://www.rfc-editor.org/info/rfc6819>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <http://www.rfc-editor.org/info/rfc7231>.

Authors' Addresses

Torsten Lodderstedt (editor)
Deutsche Telekom AG

Email: torsten@lodderstedt.net

John Bradley
Ping Identity

Email: ve7jtb@ve7jtb.com

Andrey Labunets
Facebook

Email: isciurus@fb.com