Open Authentication Protocol
Internet-Draft
Intended status: Best Current Practice
Expires: May 17, 2018

T. Lodderstedt, Ed.
YES.com AG
J. Bradley
Yubico
A. Labunets
Facebook

November 13, 2017

OAuth Security Topics
draft-ietf-oauth-security-topics-04

Abstract

This draft gives a comprehensive overview on open OAuth security topics. It is intended to serve as a working document for the OAuth working group to systematically capture and discuss these security topics and respective mitigations and eventually recommend best current practice and also OAuth extensions needed to cope with the respective security threats.

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. While OAuth was used in a variety of scenarios and different kinds of deployments, the following challenges could be observed:

o 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.

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

o 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 RFC 6749 [RFC6749], RFC 6750 [RFC6749], and RFC 6819 [RFC6819].

OAuth initially assumed a static relationship between client, authorization server and resource servers. The URLs of AS and RS were known to the client at deployment time and built an anchor for the trust relationsship among those parties. The validation whether the client talks to a legitimate server was based on TLS server authentication (see [RFC6819], Section 4.5.4). With the increasing adoption of OAuth, this simple model dissolved and, in several scenarios, was replaced by a dynamic establishment of the relationship between clients on one side and the authorization and resource servers of a particular deployment on the other side. This way the same client could be used to access services of different providers (in case of standard APIs, such as e-Mail or OpenID Connect) or serves as a frontend to a particular tenant in a multi-tenancy. Extensions of OAuth, such as [RFC7591] and [I-D.ietf-oauth-discovery] were developed in order to support the usage of OAuth in dynamic scenarios. As a challenge to the community, such usage scenarios open up new attack angles, which are discussed in this document.

The remainder of the document is organized as follows: The next section gives a summary of the set of security mechanisms and practices, the working group shall consider to recommend to OAuth implementers. This is followed by a section proposing modifications

to OAuth intended to either simplify its usage and to strengthen its security.

The remainder of the draft gives a detailed analyses of the weaknesses and implementation issues, which can be found in the wild today, along with a discussion of potential counter measures.

2. Best Practices

This section describes the set of security mechanisms the authors believe should be taken into consideration by the OAuth working group to be recommended to OAuth implementers.

2.1. Protecting redirect-based flows

Authorization servers shall utilize exact matching of client redirect URIs against pre-registered URIs. This measure contributes to the prevention of leakage of authorization codes and access tokens (depending on the grant type). It also helps to detect mix up attacks.

Clients shall avoid any redirects or forwards, which can be parameterized by URI query parameters, in order to provide a further layer of defence against token leakage. If there is a need for this kind of redirects, clients are advised to implement appropriate counter measures against open redirection, e.g. as described by the OWASP [owasp].

Clients shall ensure to only process redirect responses of the OAuth authorization server they send the respective request to and in the same user agent this request was initiated in. In particular, clients shall implement appropriate XSRF prevention by utilizing one-time use XSRF tokens carried in the STATE parameter, which are securely bound to the user agent. Moreover, the client shall store the authorization server's identity it sends an authorization request to in a transaction-specific manner, which is also bound to the particular user agent. Furthermore, clients should use AS-specific redirect URIs as a means to identify the AS a particular response came from. Matching this with the before mentioned information regarding the AS the client sent the request to helps to detect mix-up attacks.

Note: [I-D.bradley-oauth-jwt-encoded-state] gives advice on how to implement XSRF prevention and AS matching using signed JWTs in the STATE parameter.

Clients shall use PKCE [RFC7636] in order to (with the help of the authorization server) detect attempts to inject authorization codes

into the authorization response. The PKCE challenges must be transaction-specific and securely bound to the user agent, in which the transaction was started.

Note: although PKCE so far was recommended as mechanism to protect native apps, this advice applies to all kinds of OAuth clients, including web applications.

2.2. Token Leakage Prevention

Authorization servers shall use TLS-based methods for sender constraint access tokens as described in section Section 4.7.1.2, such as token binding [I-D.ietf-oauth-token-binding] or Mutual TLS for OAuth 2.0 [I-D.ietf-oauth-mtls]. It is also recommend to use end-to-end TLS whenever possible.

3. Recommended Changes to OAuth

This section describes the set of modifications and extensions the authors believe should be taken into consideration by the OAuth working group change and extend OAuth in order to strengthen its security and make it simpler to implement. It also recommends some changes to the OAuth set of specs.

Remove requirement to check actual redirect URI at token endpoint - seems to be complicated to implement properly and could be compromised. The protection goal is achieved even more effective by utilizing PKCE as recommended in Section 2.1.

4. Attacks and Mitigations

4.1. Insufficient redirect URI validation

Some authorization servers allow clients to register redirect URI patterns instead of complete redirect URIs. In those cases, the authorization server, at runtime, matches 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

- by exposing the OAuth credentials to an attacker by utilizing an open redirector at the client in conjunction with the way user agents handle URL fragments.

4.1.1. Attacks on Authorization Code Grant

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

Let's assume the redirect URL pattern "https://*.example.com/*" had been registered for the client "s6BhdRkqt3". This pattern allows redirect URIs 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.example.com%2Fcb HTTP/1.1
Host: server.example.com
```

(1) 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.

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

(3) 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 directly 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 (e.g. by mounting a code injection attack). This kind of injections is covered in Section Code Injection.

4.1.2. Attacks on Implicit Grant

The attack described above works for the implicit grant as well. 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, implicit clients can be subject to a further kind of attacks. It utilizes the fact that 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). The attack described here combines this behavior with the client as an open redirector in order to get access to access tokens. This allows circumvention even of strict redirect URI patterns (but not strict URL matching!).

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
```

(1) 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&...
```

(2) 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
```

(3) 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&...
```

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

4.1.3. Proposed Countermeasures

The complexity of implementing and managing pattern matching correctly obviously causes security issues. This document therefore proposes to simplify the required logic and configuration by using exact redirect URI matching only. This means 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 on whether this feature is really used in deployments today.

o The working group may also consider this change as a step towards improved interoperability for OAuth implementations since RFC 6749 is somewhat vague on redirect URI validation. Notably there are

no rules for pattern matching. One may therefore 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 and foster security?

- 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.
  Question: Does redirect uri validation solve any problem for native apps? Effective against impersonation when used in conjunction with claimed HTTPS redirect URIs only.
  For Windows token broker exact redirect URI matching is important as the redirect URI encodes the app identity. For custom scheme redirects there is a question however it is probably a useful part of defense in depth.

Additional recommendations:

- o Servers on which callbacks are hosted must not expose open redirectors (see respective section).

- o Clients may drop fragments via intermediary URLs with "fix fragments" (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:

- o authenticate client using digital signatures (JAR? https://tools.ietf.org/html/draft-ietf-oauth-jwsreq-09)

4.2. Authorization code leakage via referrer headers

It is 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.

4.2.1. Proposed Countermeasures

There are some means to prevent leakage as described above:

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

- o 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))

- o 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 (don't transport credentials via URL parameters and GET)

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)

4.3. Attacks in the Browser

4.3.1. 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.

Proposed countermeasures: code is one time use, has limited duration, is bound to client id/secret (confidential clients only)

4.3.2. 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

4.3.3. Javascript Code stealing Access Tokens (TBD)

sandboxing using service workers

4.4. Mix-Up

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 countermeasures 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 carries client id and issuer (requires OpenID Connect)

- Clients use AS-specific redirect URIs, for every authorization request store intended AS and compare intention with actual redirect URI where the response was received (no change to OAuth required)

4.5. 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:

- 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.

(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.

4.5.1. Proposed Countermeasures

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

Code-bound 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.

Note: token binding could be used in conjunction with PKCE as an option (https://tools.ietf.org/html/draft-ietf-oauth-token-binding-02#section-4).
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 countermeasures described above if he is able to create or capture the respective secret or code_challenge 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 and 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)?

4.6. 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)

4.7. Access Token Leakage at the Resource Server

4.7.1. Access Token Phishing by Counterfeit Resource Server

An attacker may setup his own resource server and trick a client into sending access tokens to it, which are valid for other resource servers. If the client sends a valid access token to this counterfeit resource server, the attacker in turn may use that token to access other services on behalf of the resource owner.

This attack assumes the client is not bound to a certain resource server (and the respective URL) at development time, but client instances are configured with an resource server's URL at runtime. This kind of late binding is typical in situations, where the client uses a standard API, e.g. for e-Mail, calendar, health, or banking

and is configured by an user or administrator for the standard-based service, this particular user or company uses.

There are several potential mitigation strategies, which will be discussed in the following sections.

4.7.1.1. Metadata

An authorization server could provide the client with additional information about the location where it is safe to use its access tokens.

In the simplest form, this would require the AS to publish a list of its known resource servers, illustrated in the following example using a metadata parameter "resource_servers":

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "issuer":"https://server.example.com",
  "authorization_endpoint":"https://server.example.com/authorize",
  "resource_servers":[
    "email.example.com",
    "storage.example.com",
    "video.example.com"]
  ...
}
```

The AS could also return the URL(s) an access token is good for in the token response, illustrated by the example return parameter "access_token_resource_server":

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "access_token_resource_server":"https://hostedresource.example.com/path1",
...
}
```

This mitigation strategy would rely on the client to enforce the security policy and to only send access tokens to legitimate destinations. Results of OAuth related security research (see for example [oauth_security_ubc] and [oauth_security_cmu]) indicate a

large portion of client implementations do not or fail to properly implement security controls, like state checks. So relying on clients to detect and properly handle access token phishing is likely to fail as well. Moreover given the ratio of clients to authorization and resource servers, it is considered the more viable approach to move as much as possible security-related logic to those entities. Clearly, the client has to contribute to the overall security. But there are alternative counter measures, as described in the next sections, which provide a better balance between the involved parties.

4.7.1.2. Sender Constrained Access Tokens

As the name suggests, sender constraint access token scope the applicability of an access token to a certain sender. This sender is obliged to demonstrate knowledge of a certain secret as prerequisite for the acceptance of that token at a resource server.

A typical flow looks like this:

1. The authorization server associates data with the access token, which bind this particular token to a certain client. The binding can utilize the client identity, but in most cases the AS utilizes key material (or data derived from the key material) known to the client.

2. This key material must be distributed somehow. Either the key material already exists before the AS creates the binding or the AS creates ephemeral keys. The way pre-existing key material is distributed varies among the different approaches. For example, X.509 Certificates can be used in which case the distribution happens explicitly during the enrollment process. Or the key material is created and distributed at the TLS layer, in which case it might automatically happens during the setup of a TLS connection.

3. The RS must implement the actual proof of possession check. This is typically done on the application level, it may utilize capabilities of the transport layer (e.g. TLS). Note: replay detection is required as well!

There exists several proposals to demonstrate the proof of possession in the scope of the OAuth working group:

o [I-D.ietf-oauth-token-binding]: In this approach, an access tokens is, via the so-called token binding id, bound to key material representing a long term association between a client and a certain TLS host. Negotiation of the key material and proof of

possession in the context of a TLS handshake is taken care of by the TLS stack. The client needs to determine the token binding id of the target resource server and pass this data to the access token request. The authorization server than associates the access token with this id. The resource server checks on every invocation that the token binding id of the active TLS connection and the token binding id of associated with the access token match. Since all crypto-related functions are covered by the TLS stack, this approach is very client developer friendly. As a prerequisite, token binding as described in [I-D.ietf-tokbind-https] (including federated token bindings) must be supported on all ends (client, authorization server, resource server).

o [I-D.ietf-oauth-mtls]: The approach as specified in this document allow use of mutual TLS for both client authentication and sender constraint access tokens. For the purpose of sender constraint access tokens, the client is identified towards the resource server by the fingerprint of its public key. During processing of an access token request, the authorization server obtains the client's public key from the TLS stack and associates its fingerprint with the respective access tokens. The resource server in the same way obtains the public key from the TLS stack and compares its fingerprint with the fingerprint associated with the access token.

o [I-D.ietf-oauth-signed-http-request] specifies an approach to sign HTTP requests. It utilizes [I-D.ietf-oauth-pop-key-distribution] and represents the elements of the signature in a JSON object. The signature is built using JWS. The mechanism has built-in support for signing of HTTP method, query parameters and headers. It also incorporates a timestamp as basis for replay detection.

o [I-D.sakimura-oauth-jpop]: this draft describes different ways to constrain access token usage, namely TLS or request signing. Note: Since the authors of this draft contributed the TLS-related proposal to [I-D.ietf-oauth-mtls], this document only considers the request signing part. For request signing, the draft utilizes [I-D.ietf-oauth-pop-key-distribution] and RFC 7800 [RFC7800]. The signature data is represented in a JWT and JWS is used for signing. Replay detection is provided by building the signature over a server-provided nonce, client-provided nonce and a nonce counter.

[I-D.ietf-oauth-mtls] and [I-D.ietf-oauth-token-binding] are built on top of TLS and this way continue the successful OAuth 2.0 philosophy to leverage TLS to secure OAuth wherever possible. Both mechanisms

allow prevention of access token leakage in a fairly client developer friendly way.

There are some differences between both approaches: To start with, in [I-D.ietf-oauth-token-binding] all key material is automatically managed by the TLS stack whereas [I-D.ietf-oauth-mtls] requires the developer to create and maintain the key pairs and respective certificates. Use of self-signed certificates, which is supported by the draft, significantly reduce the complexity of this task. Furthermore, [I-D.ietf-oauth-token-binding] allows to use different key pairs for different resource servers, which is a privacy benefit. On the other hand, [I-D.ietf-oauth-mtls] only requires widely deployed TLS features, which means it might be easier to adopt in the short term.

Application level signing approaches, like [I-D.ietf-oauth-signed-http-request] and [I-D.sakimura-oauth-jpop] have been debated for a long time in the OAuth working group without a clear outcome.

As one advantage, application-level signing allows for end-to-end protection including non-repudiation even if the TLS connection is terminated between client and resource server. But deployment experiences have revealed challenges regarding robustness (e.g. reproduction of the signature base string including correct URL) as well as state management (e.g. replay detection).

This document therefore recommends implementors to consider one of TLS-based approaches wherever possible.

4.7.1.3. Audience Restricted Access Tokens

An audience restriction essentially restricts the resource server a particular access token can be used at. The authorization server associates the access token with a certain resource server and every resource server is obliged to verify for every request, whether the access token send with that request was meant to be used at the particular resource server. If not, the resource server must refuse to serve the respective request. In the general case, audience restrictions limit the impact of a token leakage. In the case of a counterfeit resource server, it may (as described see below) also prevent abuse of the phished access token at the legitimate resource server.

The audience can basically be expressed using logical names or physical addresses (like URLs). In order to detect phishing, it is necessary to use the actual URL the client will send requests to. In the phishing case, this URL will point to the counterfeit resource

server. If the attacker tries to use the access token at the legitimate resource server (which has a different URL), the resource server will detect the mismatch (wrong audience) and refuse to serve the request.

In deployments where the authorization server knows the URLs of all resource servers, the authorization server may just refuse to issue access tokens for unknown resource server URLs.

The client needs to tell the authorization server, at which URL it will use the access token it is requesting. It could use the mechanism proposed [I-D.campbell-oauth-resource-indicators] or encode the information in the scope value.

Instead of the URL, it is also possible to utilize the fingerprint of the resource server's X.509 certificate as audience value. This variant would also allow to detect an attempt to spoof the legit resource server's URL by using a valid TLS certificate obtained from a different CA. It might also be considered a privacy benefit to hide the resource server URL from the authorization server.

Audience restriction seems easy to use since it does not require any crypto on the client side. But since every access token is bound to a certain resource server, the client also needs to obtain different RS-specific access tokens, if it wants to access several resource services. [I-D.ietf-oauth-token-binding] has the same property, since different token binding ids must be associated with the access token. [I-D.ietf-oauth-mtls] on the other hand allows a client to use the access token at multiple resource servers.

It shall be noted that audience restrictions, or generally speaking an indication by the client to the authorization server where it wants to use the access token, has additional benefits beyond the scope of token leakage prevention. It allows the authorization server to create different access token whose format and content is specifically minted for the respective server. This has huge functional and privacy advantages in deployments using structured access tokens.

4.7.2. Compromised Resource Server

An attacker may compromise a resource server in order to get access to its resources and other resources of the respective deployment. Such a compromise may range from partial access to the system, e.g. its logfiles, to full control of the respective server.

If the attacker was able to take over full control including shell access it will be able to circumvent all controls in place and access

resources without access control. It will also get access to access tokens, which are sent to the compromised system and which potentially are valid for access to other resource servers as well. Even if the attacker "only" is able to access logfiles or databases of the server system, it may get access to valid access tokens.

Preventing and detecting server breaches by way of hardening and monitoring server systems is considered a standard operational procedure and therefore out of scope of this document. This section will focus on the impact of such breaches on OAuth-related parts of the ecosystem, which is the replay of captured access tokens on the compromised resource server and other resource servers of the respective deployment.

The following measures shall be taken into account by implementors in order to cope with access token replay:

o The resource server must treat access tokens like any other credentials. It is considered good practice to not log them and not to store them in plain text.

o Sender constraint access tokens as described in Section 4.7.1.2 will prevent the attacker from replaying the access tokens on other resource servers. Depending on the severity of the penetration, it will also prevent replay on the compromised system.

o Audience restriction as described in Section 4.7.1.3 may be used to prevent replay of captured access tokens on other resource servers.

4.8. Refresh Token Leakage (TBD)

mitm, log files on the device, ...

refresh token rotation, mutual TLS authentication at the token endpoint

4.9. Open Redirection (TBD)

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

Using the Client as Open Redirector

4.10. TLS Terminating Reverse Proxies

A common deployment architecture for HTTP applications is to have the application server sitting behind a reverse proxy, which terminates the TLS connection and dispatches the incoming requests to the respective application server nodes.

This section highlights some attack angles of this deployment architecture, which are relevant to OAuth, and give recommendations for security controls.

In some situations, the reverse proxy needs to pass security-related data to the upstream application servers for further processing. Examples include the IP address of the request originator, token binding ids and authenticated TLS client certificates.

If the reverse proxy would pass through any header sent from the outside, an attacker could try to directly send the faked header values through the proxy to the application server in order to circumvent security controls that way. For example, it is standard practice of reverse proxies to accept "forwarded_for" headers and just add the origin of the inbound request (making it a list). Depending on the logic performed in the application server, the attacker could simply add a whitelisted IP address to the header and render a IP whitelist useless. A reverse proxy must therefore sanitize any inbound requests to ensure the authenticity and integrity of all header values relevant for the security of the application servers.

If an attacker would be able to get access to the internal network between proxy and application server, it could also try to circumvent security controls in place. It is therefore important to ensure the authenticity of the communicating entities. Furthermore, the communication link between reverse proxy and application server must therefore be protected against tapping and injection (including replay prevention).

4.11. Other Topics

o redirect via status code 307 - use 302

o why to rotate refresh tokens

o how to support multi AS per RS

o differentiate native, JS and web clients

o do not put sensitive data in URL/GET parameters (Jim Manico)

- o Incorporate Christian Mainka's feedback
- o WPAD attack - https://www.blackhat.com/docs/us-16/materials/us-16-Kotler-Crippling-HTTPS-With-Unholy-PAC.pdf

5. Acknowledgements

We would like to thank Jim Manico, Phil Hunt, and Brian Campbell 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. References

8.1. 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, <https://www.rfc-editor.org/info/rfc3986>.

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework", RFC 6749, DOI 10.17487/RFC6749, October 2012, <https://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, <https://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, <https://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, <https://www.rfc-editor.org/info/rfc7231>.

[RFC7591] Richer, J., Ed., Jones, M., Bradley, J., Machulak, M., and P. Hunt, "OAuth 2.0 Dynamic Client Registration Protocol", RFC 7591, DOI 10.17487/RFC7591, July 2015, <https://www.rfc-editor.org/info/rfc7591>.

8.2. Informative References

[I-D.bradley-oauth-jwt-encoded-state]
Bradley, J., Lodderstedt, T., and H. Zandbelt, "Encoding claims in the OAuth 2 state parameter using a JWT", draft-bradley-oauth-jwt-encoded-state-07 (work in progress), March 2017.

[I-D.campbell-oauth-resource-indicators]
Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", draft-campbell-oauth-resource-indicators-02 (work in progress), November 2016.

[I-D.ietf-oauth-discovery]
Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", draft-ietf-oauth-discovery-07 (work in progress), September 2017.

[I-D.ietf-oauth-mtls]
Campbell, B., Bradley, J., Sakimura, N., and T. Lodderstedt, "Mutual TLS Profile for OAuth 2.0", draft-ietf-oauth-mtls-05 (work in progress), November 2017.

[I-D.ietf-oauth-pop-key-distribution]
Bradley, J., Hunt, P., Jones, M., and H. Tschofenig, "OAuth 2.0 Proof-of-Possession: Authorization Server to Client Key Distribution", draft-ietf-oauth-pop-key-distribution-03 (work in progress), February 2017.

[I-D.ietf-oauth-signed-http-request]
Richer, J., Bradley, J., and H. Tschofenig, "A Method for Signing HTTP Requests for OAuth", draft-ietf-oauth-signed-http-request-03 (work in progress), August 2016.

[I-D.ietf-oauth-token-binding]
Jones, M., Campbell, B., Bradley, J., and W. Denniss, "OAuth 2.0 Token Binding", draft-ietf-oauth-token-binding-05 (work in progress), October 2017.

[I-D.ietf-tokbind-https]
Popov, A., Nystrom, M., Balfanz, D., Langley, A., Harper, N., and J. Hodges, "Token Binding over HTTP", draft-ietf-tokbind-https-10 (work in progress), July 2017.

[I-D.sakimura-oauth-jpop]
Sakimura, N., Li, K., and J. Bradley, "The OAuth 2.0 Authorization Framework: JWT Pop Token Usage", draft-sakimura-oauth-jpop-04 (work in progress), March 2017.

[oauth_security_cmu]
Carnegie Mellon University, Carnegie Mellon University, Microsoft Research, Carnegie Mellon University, Carnegie Mellon University, and Carnegie Mellon University, "OAuth Demystified for Mobile Application Developers", November 2014.

[oauth_security_ubc]
University of British Columbia and University of British Columbia, "The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems", October 2012, <http://passwordresearch.com/papers/paper267.html>.

[owasp] "Open Web Application Security Project Home Page", <https://www.owasp.org/>.

[RFC7636] Sakimura, N., Ed., Bradley, J., and N. Agarwal, "Proof Key for Code Exchange by OAuth Public Clients", RFC 7636, DOI 10.17487/RFC7636, September 2015, <https://www.rfc-editor.org/info/rfc7636>.

[RFC7800] Jones, M., Bradley, J., and H. Tschofenig, "Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)", RFC 7800, DOI 10.17487/RFC7800, April 2016, <https://www.rfc-editor.org/info/rfc7800>.

Appendix A. Document History

[[To be removed from the final specification]]

-04

o Restructured document for better readability

o Added best practices on Token Leakage prevention

-03

o Added section on Access Token Leakage at Resource Server

o incorporated Brian Campbell's findings

-02

- Folded Mix up and Access Token leakage through a bad AS into new section for dynamic OAuth threats

- reworked dynamic OAuth section

-01

- Added references to mitigation methods for token leakage

- Added reference to Token Binding for Authorization Code

- incorporated feedback of Phil Hunt

- fixed numbering issue in attack descriptions in section 2

-00 (WG document)

- turned the ID into a WG document and a BCP

- Added federated app login as topic in Other Topics

Authors' Addresses

Torsten Lodderstedt (editor)
YES.com AG

Email: torsten@lodderstedt.net

John Bradley
Yubico

Email: ve7jtb@ve7jtb.com

Andrey Labunets
Facebook

Email: isciurus@fb.com