Web Authorization Protocol
Internet-Draft
Intended status: Best Current Practice
Expires: 1 July 2019

T. Lodderstedt
yes.com
J. Bradley
Yubico
A. Labunets
Facebook
D. Fett
yes.com

28 December 2018

# OAuth 2.0 Security Best Current Practice
# draft-ietf-oauth-security-topics-11

## Abstract

This document describes best current security practice for OAuth 2.0. It updates and extends the OAuth 2.0 Security Threat Model to incorporate practical experiences gathered since OAuth 2.0 was published and covers new threats relevant due to the broader application of OAuth 2.0.

## Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 1 July 2019.

Table of Contents



## 1. Introduction

It's been a while since OAuth has been published in [RFC6749] and [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 [OpenID], identity providing. While OAuth was used in a variety of scenarios and different kinds of deployments, the following challenges could be observed:

* OAuth implementations are being attacked through known implementation weaknesses and anti-patterns (CSRF, referrer header). Although most of these threats are discussed in the OAuth 2.0 Threat Model and Security Considerations [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 [RFC6749], [RFC6749], and [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 relationship 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 [RFC8414] 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 summarizes the most important recommendations of the OAuth working group for every OAuth implementor. Afterwards, a detailed analysis of the threats and implementation issues which can be found in the wild today is given along with a discussion of potential countermeasures.

## 2. Recommendations

This section describes the set of security mechanisms the OAuth working group recommendeds to OAuth implementers.

## 2.1. Protecting Redirect-Based Flows

Authorization servers MUST 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 SHOULD avoid forwarding the user's browser to a URI obtained from a query parameter since such a function could be utilized to exfiltrate authorization codes and access tokens. If there is a strong need for this kind of redirects, clients are advised to implement appropriate countermeasures against open redirection, e.g., as described by the OWASP [owasp].

Clients MUST prevent CSRF and ensure that each authorization response is only accepted once. One-time use CSRF tokens carried in the "state" parameter, which are securely bound to the user agent, SHOULD be used for that purpose.

In order to prevent mix-up attacks, clients MUST only process redirect responses of the OAuth authorization server they send the respective request to and from the same user agent this authorization request was initiated with. Clients MUST memorize which authorization server they sent an authorization request to and bind this information to the user agent and ensure any sub-sequent messages are sent to the same authorization server. Clients SHOULD use AS-specific redirect URIs as a means to identify the AS a particular response came from.

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

### 2.1.1. Authorization Code Grant

Clients utilizing the authorization grant type MUST use PKCE [RFC7636] in order to (with the help of the authorization server) detect and prevent attempts to inject (replay) 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. OpenID Connect clients MAY use the "nonce" parameter of the OpenID Connect authentication request as specified in [OpenID] in conjunction with the corresponding ID Token claim for the same purpose.

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

including web applications.

Authorization servers MUST bind authorization codes to a certain client and authenticate it using an appropriate mechanism (e.g. client credentials or PKCE).

Authorization servers SHOULD furthermore consider the recommendations given in [RFC6819], Section 4.4.1.1, on authorization code replay prevention.

### 2.1.2. Implicit Grant

The implicit grant (response type "token") and other response types causing the authorization server to issue access tokens in the authorization response are vulnerable to access token leakage and access token replay as described in Section 3.1, Section 3.2, Section 3.3, and Section 3.6.

Moreover, no viable mechanism exists to cryptographically bind access tokens issued in the authorization response to a certain client as it is recommended in Section 2.2. This makes replay detection for such access tokens at resource servers impossible.

In order to avoid these issues, clients SHOULD NOT use the implicit grant (response type "token") or any other response type issuing access tokens in the authorization response, such as "token id_token" and "code token id_token", unless the issued access tokens are sender-constrained and access token injection in the authorization response is prevented.

A sender constrained access token scopes 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 the recipient (e.g., a resource server).

Clients SHOULD instead use the response type "code" (aka authorization code grant type) as specified in Section 2.1.1 or any other response type that causes the authorization server to issue access tokens in the token response. This allows the authorization server to detect replay attempts and generally reduces the attack surface since access tokens are not exposed in URLs. It also allows the authorization server to sender-constrain the issued tokens.

## 2.2. Token Replay Prevention

Authorization servers SHOULD use TLS-based methods for sender constrained access tokens as described in Section 3.8.1.2, such as token binding [I-D.ietf-oauth-token-binding] or Mutual TLS for OAuth 2.0 [I-D.ietf-oauth-mtls] in order to prevent token replay. It is also recommended to use end-to-end TLS whenever possible.

## 2.3. Access Token Privilege Restriction

The privileges associated with an access token SHOULD be restricted to the minimum required for the particular application or use case. This prevents clients from exceeding the privileges authorized by the resource owner. It also prevents users from exceeding their privileges authorized by the respective security policy. Privilege restrictions also limit the impact of token leakage although more effective counter-measures are described in Section 2.2.

In particular, access tokens SHOULD be restricted to certain resource servers, preferably to a single resource server. To put this into effect, the authorization server associates the access token with certain resource servers and every resource server is obliged to verify for every request, whether the access token sent with that request was meant to be used for that particular resource server. If not, the resource server MUST refuse to serve the respective request. Clients and authorization servers MAY utilize the parameters "scope" or "resource" as specified in [RFC6749] and [I-D.ietf-oauth-resource-indicators], respectively, to determine the resource server they want to access.

Additionally, access tokens SHOULD be restricted to certain resources and actions on resource servers or resources. To put this into effect, the authorization server associates the access token with the respective resource and actions and every resource server is obliged to verify for every request, whether the access token sent with that request was meant to be used for that particular action on the particular resource. If not, the resource server must refuse to serve the respective request. Clients and authorization servers MAY utilize the parameter "scope" as specified in [RFC6749] to determine those resources and/or actions.

## 3. Attacks and Mitigations

This section gives a detailed description of attacks on OAuth implementations, along with potential countermeasures. This section complements and enhances the description given in [RFC6819].

### 3.1. 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.

#### 3.1.1. 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://_.somesite.example/_" had been registered for the client "s6BhdRkqt3". This pattern allows redirect URIs pointing to any host residing in the domain somesite.example. So if an attacker manages to establish a host or subdomain in somesite.example he can impersonate the legitimate client. Assume the attacker sets up the host "evil.somesite.example".

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.example" (https://www.evil.example").

   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.somesite.example%2Fcb HTTP/1.1
Host: server.somesite.example
```

2. The authorization server validates the redirect URI in order to identify the client. Since the pattern allows arbitrary domains host names in "somesite.example", 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 [RFC6749]), the attack can be performed even easier.

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

   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 impersonate or utilize the legitimate client to redeem the code (e.g., by performing a code injection attack). This kind of injections is covered in Section 3.5.

### 3.1.2. 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.somesite.example/cb?*", i.e., any parameter is allowed for redirects to "https://client.somesite.example/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 an HTTP Location header redirect 303.

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.example".

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.example" 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.somesite.example%2Fcb%26redirect_to
          %253Dhttps%253A%252F%252Fclient.evil.example%252Fcb HTTP/1.1
   Host: server.somesite.example
```

3. Since the redirect URI matches the registered pattern, the authorization server allows the request and sends the resulting access token with a 303 redirect (some response parameters are

omitted for better readability)

```
HTTP/1.1 303 See Other
Location: https://client.somesite.example/cb?
          redirect_to%3Dhttps%3A%2F%2Fclient.evil.example%2Fcb
          #access_token=2YotnFZFEjr1zCsicMWpAA&...
```

4. At example.com, the request arrives at the open redirector. It will read the redirect parameter and will issue an HTTP 303 Location header redirect to the URL "https://client.evil.example/cb".

```
HTTP/1.1 303 See Other
Location: https://client.evil.example/cb
```

5. Since the redirector at client.somesite.example 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.example/cb#access_token=2YotnFZFEjr1zCsicMWpAA&...

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

### 3.1.3. 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 must compare the two URIs using simple string comparison as defined in [RFC3986], Section 6.2.1..

Additional recommendations:

* Servers on which callbacks are hosted must not expose open redirectors (see (#Open.Redirection)).

* Clients MAY drop fragments via intermediary URLs with "fix fragments" (see [fb_fragments]) to prevent the user agent from appending any unintended fragments.

* Clients SHOULD use the authorization code response type instead of response types causing access token issuance at the authorization endpoint. This offers countermeasures against reuse of leaked credentials through the exchange process with the authorization server and token replay through certificate binding of the access tokens.

As an alternative to exact redirect URI matching, the AS could also

authenticate clients, e.g., using [I-D.ietf-oauth-jwsreq].

### 3.2. Credential Leakage via Referrer Headers

Authorization codes or values of "state" can unintentionally be disclosed to attackers through the referrer header, by leaking either from a client's web site or from an AS's web site. Note: even if specified otherwise in [RFC2616], section 14.36, the same may happen to access tokens conveyed in URI fragments due to browser implementation issues as illustrated by Chromium Issue 168213 [bug.chromium].

#### 3.2.1. Leakage from the OAuth client

This requires that the client, as a result of a successful authorization request, renders a page that

* contains links to other pages under the attacker's control (ads, faq, ...) and a user clicks on such a link, or

* includes third-party content (iframes, images, etc.) for example if the page contains user-generated content (blog).

As soon as the browser navigates to the attacker's page or loads the third-party content, the attacker receives the authorization response URL and can extract "code", "access token", or "state".

#### 3.2.2. Leakage from the Authorization Server

In a similar way, an attacker can learn "state" if the authorization endpoint at the authorization server contains links or third-party content as above.

#### 3.2.3. Consequences

An attacker that learns a valid code or access token through a referrer header can perform the attacks as described in Section 3.1.1, Section 3.5, and Section 3.6. If the attacker learns "state", the CSRF protection achieved by using "state" is lost, resulting in CSRF attacks as described in [RFC6819], Section 4.4.1.8..

#### 3.2.4. Proposed Countermeasures

The page rendered as a result of the OAuth authorization response and the authorization endpoint SHOULD not include third-party resources or links to external sites.

The following measures further reduce the chances of a successful attack:

* Bind authorization code to a confidential client or PKCE

challenge. In this case, the attacker lacks the secret to request the code exchange.

* Authorization codes SHOULD be invalidated by the AS after their first use at the token endpoint. For example, if an AS invalidated the code after the legitimate client redeemed it, the attacker would fail exchanging this code later. (This does not mitigate the attack if the attacker manages to exchange the code for a token before the legitimate client does so.)

* The "state" value SHOULD be invalidated by the client after its first use at the redirection endpoint. If this is implemented, and an attacker receives a token through the referrer header from the client's web site, the "state" was already used, invalidated by the client and cannot be used again by the attacker. (This does not help if the <spanx style="verb">state</spanx> leaks from the AS's web site, since then the <spanx style="verb">state</spanx> has not been used at the redirection endpoint at the client yet.)

* Suppress the referrer header by adding the attribute "rel="noreferrer"" to HTML links or by applying an appropriate Referrer Policy [webappsec-referrer-policy] to the document (either as part of the "referrer" meta attribute or by setting a Referrer-Policy header).

* Use authorization code instead of response types causing access token issuance from the authorization endpoint. This provides countermeasures against leakage on the OAuth protocol level through the code exchange process with the authorization server.

* Additionally, one might use the form post response mode instead of redirect for authorization response (see [oauth-v2-form-post-response-mode]).

## 3.3. Attacks through the Browser History

Authorization codes and access tokens can end up in the browser's history of visited URLs, enabling the attacks described in the following.

### 3.3.1. Code in Browser History

When a browser navigates to "client.example/redirection_endpoint?code=abcd" as a result of a redirect from a provider's authorization endpoint, the URL including the authorization code may end up in the browser's history. An attacker with access to the device could obtain the code and try to replay it.

Proposed countermeasures:

* Authorization code replay prevention as described in [RFC6819],

Section 4.4.1.1, and Section 3.5

* Use form post response mode instead of redirect for authorization response (see [oauth-v2-form-post-response-mode])

### 3.3.2. Access Token in Browser History

An access token may end up in the browser history if a 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 parameters.

In case of implicit grant, a URL like "client.example/redirection_endpoint#access_token=abcdef" may also end up in the browser history as a result of a redirect from a provider's authorization endpoint.

Proposed countermeasures:

* Replace implicit flow with postmessage communication or the authorization code grant

* Never pass access tokens in URL query parameters

## 3.4. Mix-Up

Mix-up is an attack on scenarios where an OAuth client interacts with multiple authorization servers, as is usually the case when dynamic registration is used. 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 instead of using them at the respective endpoint at the authorization/resource server.

### 3.4.1. Attack Description

For a detailed attack description, refer to [arXiv.1601.01229] and [I-D.ietf-oauth-mix-up-mitigation]. The description here closely follows [arXiv.1601.01229], with variants of the attack outlined below.

Preconditions: For the attack to work, we assume that

* the implicit or authorization code grant are used with multiple AS of which one is considered "honest" (H-AS) and one is operated by the attacker (A-AS),

* the client stores the AS chosen by the user in a session bound to the user's browser and uses the same redirection endpoint URI for each AS, and

* the attacker can manipulate the first request/response pair from a user's browser to the client (in which the user selects a certain AS and is then redirected by the client to that AS).

Some of the attack variants described below require different preconditions.

In the following, we assume that the client is registered with H-AS (URI: "https://honest.as.example" (https://honest.as.example"), client id: 7ZGZldHQ) and with A-AS (URI: "https://attacker.example" (https://attacker.example"), client id: 666RVZJTA).

Attack on the authorization code grant:

1. The user selects to start the grant using H-AS (e.g., by clicking on a button at the client's website).

2. The attacker intercepts this request and changes the user's selection to "A-AS".

3. The client stores in the user's session that the user selected "A-AS" and redirects the user to A-AS's authorization endpoint by sending the following response:

```
HTTP/1.1 303 See Other
Location: https://attacker.example/authorize?
response_type=code&client_id=666RVZJTA
```

4. Now the attacker intercepts this response and changes the redirection such that the user is being redirected to H-AS. The attacker also replaces the client id of the client at A-AS with the client's id at H-AS, resulting in the following response being sent to the browser:

```
HTTP/1.1 303 See Other
Location: https://honest.as.example/authorize?
response_type=code&client_id=7ZGZldHQ
```

5. Now, the user authorizes the client to access her resources at H-AS. H-AS issues a code and sends it (via the browser) back to the client.

6. Since the client still assumes that the code was issued by A-AS, it will try to redeem the code at A-AS's token endpoint.

7. The attacker therefore obtains code and can either exchange the code for an access token (for public clients) or perform a code injection attack as described in Section 3.5.

Variants:

* *Implicit Grant*: In the implicit grant, the attacker receives an

access token instead of the code; the rest of the attack works as above.

* *Mix-Up Without Interception*: A variant of the above attack works even if the first request/response pair cannot be intercepted (for example, because TLS is used to protect these messages): Here, we assume that the user wants to start the grant using A-AS (and not H-AS). After the client redirected the user to the authorization endpoint at A-AS, the attacker immediately redirects the user to H-AS (changing the client id "7ZGZldHQ"). (A vigilant user might at this point detect that she intended to use A-AS instead of H-AS.) The attack now proceeds exactly as in step <xref format="counter" target="list_mixup_acg_after_authep"/> of the attack description above. <!-- I think this counter is not working properly! -->

* *Per-AS Redirect URIs*: If clients use different redirect URIs for different ASs, do not store the selected AS in the user's session, and ASs do not check the redirect URIs properly, attackers can mount an attack called "Cross-Social Network Request Forgery". Refer to [oauth_security_jcs_14] for details.

* *OpenID Connect*: There are several variants that can be used to attack OpenID Connect. They are described in detail in [arXiv.1704.08539], Appendix A, and [arXiv.1508.04324v2], Section 6 ("Malicious Endpoints Attacks").

### 3.4.2. Countermeasures

In scenarios where an OAuth client interacts with multiple authorization servers, clients MUST prevent mix-up attacks.

Potential countermeasures:

* Configure authorization servers to return an AS identitifier ("iss") and the "client_id" for which a code or token was issued in the authorization response. This enables clients to compare this data to their own client id and the "iss" identifier of the AS it believed it sent the user agent to. This mitigation is discussed in detail in [I-D.ietf-oauth-mix-up-mitigation]. In OpenID Connect, if an ID token is returned in the authorization response, it carries client id and issuer. It can be used for this mitigation.

* As it can be seen in the preconditions of the attacks above, clients can prevent mix-up attack by (1) using AS-specific redirect URIs with exact redirect URI matching, (2) storing, for each authorization request, the intended AS, and (3) comparing the intended AS with the actual redirect URI where the authorization response was received.

### 3.5. Authorization 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. Examples are:

* The attacker wants to access certain functions in this particular client. As an example, the attacker wants to impersonate his victim in a certain app or on a certain web site.

* 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.

* The authorization or resource servers are limited to certain networks, the attackers is unable to access directly.

How does an attack look like?

1. The attacker obtains an authorization code by performing any of the attacks described above.

2. It performs a regular 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 (see [RFC6749]).

6. If all checks succeed, the authorization server issues access and other tokens to the client, so now the attacker is able to impersonate the legitimate user.

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. The check will also fail if the authorization code was already redeemed by the legitimate user and was one-time use only.

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:

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

* 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.

This document therefore recommends to instead bind every authorization code to a certain client instance on a certain device (or in a certain user agent) in the context of a certain transaction.

### 3.5.1. Proposed Countermeasures

There are multiple technical solutions to achieve this goal:

* *Nonce*: OpenID Connect's existing "nonce" parameter could be 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. The assumption is that an attacker cannot get hold of the user agent state on the victims device, where he has stolen the respective authorization code. The main advantage of this option is that Nonce is an existing feature used in the wild. On the other hand, leveraging Nonce by the broader OAuth community would require AS and client to adopt ID Tokens.

* *Code-bound State*: The "state" parameter as specified in [RFC6749] could be used similarly to what is described above. This would require to add a further parameter "state" to the code exchange token endpoint request. The authorization server would then compare 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. The advantage of this approach would be to utilize an existing OAuth parameter. But it would also mean to re-interpret the purpose of "state" and to extend the token endpoint request.

* *PKCE*: The PKCE parameter "challenge" along with the corresponding "verifier" as specified in [RFC7636] 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. PKCE is a deployed OAuth feature, even though it is used today to secure native apps, only.

* *Token Binding*: Token binding [I-D.ietf-oauth-token-binding] could also be used. In this case, the code would need to be bound to two legs, between user agent and AS and the user agent and the client. This requires further data (extension to response) to manifest binding id for particular code. Token binding is promising as a secure and convenient mechanism (due to its browser integration). As a challenge, it requires broad browser support and use with native apps is still under discussion.

* *per instance client id/secret*: One could use per instance "client_id" and secrets and bind the code to the respective "client_id". Unfortunately, this does not fit into the web application programming model (would need to use per user client ids). </list>

PKCE seems to be the most obvious solution for OAuth clients as it available and effectively used today for similar purposes for OAuth

native apps whereas "nonce" is appropriate for OpenId Connect clients.

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. Attacks on native apps using custom schemes or redirect URIs on localhost cannot be prevented this way, except if the AS enforces one-time use for PKCE verifier or "nonce" values.

### 3.6. Access Token Injection

In such an attack, the adversary attempts to inject a stolen access token into a legitimate client on a device under his control. This will typically happen if the attacker wants to utilize a leaked access token to impersonate a user in a certain client.

To conduct the attack, the adversary starts an OAuth flow with the client and modifies the authorization response by replacing the access token issued by the authorization server or directly makes up an authorization server response including the leaked access token. Since the response includes the state value generated by the client for this particular transaction, the client does not treat the response as a CSRF and will use the access token injected by the attacker.

#### 3.6.1. Proposed Countermeasures

There is no way to detect such an injection attack on the OAuth protocol level, since the token is issued without any binding to the transaction or the particular user agent.

The recommendation is therefore to use the authorization code grant type instead of relying on response types issuing acess tokens at the authorization endpoint. Code injection can be detected using one of the countermeasures discussed in Section 3.5.

### 3.7. Cross Site Request Forgery

An attacker might attempt to inject a request to the redirect URI of the legitimate client on the victim's device, e.g., to cause the client to access resources under the attacker's control.

#### 3.7.1. Proposed Countermeasures

Standard CSRF defenses should be used to protect the redirection endpoint, for example:

* *CSRF Tokens*: Use of CSRF tokens which are bound to the user agent and passed in the "state" parameter to the authorization server.

* *Origin Header*: The Origin header can be used to detect and prevent CSRF attacks. Since this feature, at the time of writing, is not consistently supported by all browsers, CSRF tokens should be used in addition to Origin header checking.

For more details see [owasp_csrf].

### 3.8. Access Token Leakage at the Resource Server

Access tokens can leak from a resource server under certain circumstances.

#### 3.8.1. 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.

##### 3.8.1.1. 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.somesite.example",
  "authorization_endpoint":
"https://server.somesite.example/authorize",
  “resource_servers”:[
“email.somesite.example”,
”storage.somesite.example”,
”video.somesite.example”]
  ...
}
```

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.somesite.example/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 prevent 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 countermeasures, as described in the next sections, which provide a better balance between the involved parties.

#### 3.8.1.2. Sender Constrained Access Tokens

As the name suggests, sender constrained 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 binds 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 prevention is required as well!

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

* [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).

* [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.

* [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 prevention.

* [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 [RFC7800]. The signature data is represented in a JWT and JWS is used for signing. Replay prevention 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 prevention).

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

#### 3.8.1.3. 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 sent 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 prevent 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.ietf-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.

### 3.8.2. 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 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 should be taken into account by implementors in order to cope with access token replay:

* 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.

* Sender constraint access tokens as described in Section 3.8.1.2 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.

* Audience restriction as described in Section 3.8.1.3 may be used to prevent replay of captured access tokens on other resource servers.

## 3.9. Open Redirection

The following attacks can occur when an AS or client has an open redirector, i.e., a URL which causes an HTTP redirect to an attacker-controlled web site.

### 3.9.1. Authorization Server as Open Redirector

Attackers could try to utilize a user's trust in the authorization server (and its URL in particular) for performing phishing attacks.

[RFC6749], Section 4.1.2.1, already prevents open redirects by stating the AS MUST NOT automatically redirect the user agent in case of an invalid combination of client_id and redirect_uri.

However, as described in [I-D.ietf-oauth-closing-redirectors], an attacker could also utilize a correctly registered redirect URI to perform phishing attacks. It could for example register a client via dynamic client [RFC7591] registration and intentionally send an erroneous authorization request, e.g., by using an invalid scope value, to cause the AS to automatically redirect the user agent to its phishing site.

The AS MUST take precautions to prevent this threat. Based on its risk assessment the AS needs to decide whether it can trust the redirect URI or not and SHOULD only automatically redirect the user agent, if it trusts the redirect URI. If not, it MAY inform the user that it is about to redirect her to the another site and rely on the user to decide or MAY just inform the user about the error.

### 3.9.2. Clients as Open Redirector

Client MUST NOT expose URLs which could be utilized as open redirector. Attackers may use an open redirector to produce URLs which appear to point to the client, which might trick users to trust the URL and follow it in her browser. Another abuse case is to produce URLs pointing to the client and utilize them to impersonate a client with an authorization server.

In order to prevent open redirection, clients should only expose such a function, if the target URLs are whitelisted or if the origin of a request can be authenticated.

## 3.10. 307 Redirect

At the authorization endpoint, a typical protocol flow is that the AS prompts the user to enter her credentials in a form that is then submitted (using the HTTP POST method) back to the authorization server. The AS checks the credentials and, if successful, redirects the user agent to the client's redirection endpoint.

In [RFC6749], the HTTP status code 302 is used for this purpose, but "any other method available via the user-agent to accomplish this redirection is allowed". However, when the status code 307 is used for redirection, the user agent will send the form data (user credentials) via HTTP POST to the client since this status code does not require the user agent to rewrite the POST request to a GET request (and thereby dropping the form data in the POST request body). If the relying party is malicious, it can use the credentials

to impersonate the user at the AS.

In the HTTP standard [RFC6749], only the status code 303 unambigiously enforces rewriting the HTTP POST request to an HTTP GET request. For all other status codes, including the popular 302, user agents can opt not to rewrite POST to GET requests and therefore to reveal the user credentials to the client. (In practice, however, most user agents will only show this behaviour for 307 redirects.)

AS which redirect a request that potentially contains user credentials therefore MUST not use the HTTP 307 status code for redirection. If an HTTP redirection (and not, for example, JavaScript) is used for such a request, AS SHOULD use HTTP status code 303 "See Other".

### 3.11. 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).

## 3.12. Refresh Token Protection

Refresh tokens are a convenient and UX-friendly way to obtain new access tokens after the expiration of older access tokens. Refresh tokens also add to the security of OAuth since they allow the authorization server to issue access tokens with a short lifetime and reduced scope thus reducing the potential impact of access token leakage.

Refresh tokens themself are an attractive target for attackers since they represent the overall grant a resource owner delegated to a certain client. If an attacker is able to exfiltrate and successfully replay a refresh token, it will be able to mint access tokens and use them to access resource servers on behalf of the resource server.

[RFC6749] already provides robust base protection by requiring

* confidentiality of the refresh tokens in transit and storage,

* the transmission of refresh tokens over TLS-protected connections between authorization server and client,

* the authorization server to maintain and check the binding of a refresh token to a certain client_id,

* authentication of this client_id during token refresh, if possible, and

* that refresh tokens cannot be generated, modified, or guessed.

[RFC6749] also lays the foundation for further (implementation specific) security measures, such as refresh token expiration and revocation as well as refresh token rotation by defining respective error codes and response behavior.

This draft gives recommendations beyond the scope of [RFC6749] and clarifications.

Authorization servers MUST determine based on their risk assessment whether to issue refresh tokens to a certain client. If the authorization server decides not to issue refresh tokens, the client may refresh access tokens by utilizing other grant types, such as the authorization code grant type. In such a case, the authorization server may utilize cookies and persistents grants to optimize the user experience.

If refresh tokens are issued, those refresh tokens MUST be bound to the scope and resource servers as consented by the resource owner. This is to prevent privilege escalation by the legit client and reduce the impact of refresh tokens leakage.

Authorization server MUST utilize one of the methods listed below to detect refresh token replay for public clients:

* Sender constrained refresh tokens: the authorization server cryptographically binds the refresh token to a certain client instance by utilizing [I-D.ietf-oauth-token-binding] or [I-D.ietf-oauth-mtls].

* Refresh token rotation: the authorization issues a new refresh token with every access token refresh response. The previous refresh token is invalidated but information about the relationship is retained by the authorization server. If a refresh token is compromised and subsequently used by both the attacker and the legitimate client, one of them will present an invalidated refresh token, which will inform the authorization server of the breach. The authorization server cannot determine which party submitted the invalid refresh token, but it can revoke the active refresh token. This stops the attack at the cost of forcing the legit client to obtain a fresh authorization grant.

  Implementation note: refresh tokens belonging to the same grant may share a common id. If any of those refresh tokens is used at the authorization server, the authorization server uses this common id to look up the currently active refresh token and can revoke it.

Authorization servers may revoke refresh tokens automatically in case of a security event, such as:

* password change

* logout at the authorization server

Refresh tokens SHOULD expire if the client has been inactive for some time,i.e. the refresh token has not been used to obtain fresh access tokens for some time. The expiration time is at the discretion of the authorization server. It might be a global value or determined based on the client policy or the grant associated with the refresh token (and its sensitivity).

## 4. Acknowledgements

We would like to thank Jim Manico, Phil Hunt, Nat Sakimura, Christian Mainka, Doug McDorman, Johan Peeters, Joseph Heenan, Brock Allen, Vittorio Bertocci, David Waite, Nov Matake, Tomek Stojecki, Dominick Baier, Neil Madden, William Dennis, Dick Hardt, Petteri Stenius, Annabelle Richard Backman, Aaron Parecki, George Fletscher, and Brian Campbell for their valuable feedback.

## 5. IANA Considerations

This draft includes no request to IANA.

## 6. Security Considerations

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

## 7. Normative References


[arXiv.1508.04324v2]
Schwenk, J., "On the security of modern Single Sign-On Protocols: Second-Order Vulnerabilities in OpenID Connect", 7 January 2016.

[arXiv.1601.01229]
Schmitz, G., "A Comprehensive Formal Security Analysis of OAuth 2.0", 6 January 2016.

[arXiv.1704.08539]
Schmitz, G., "The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines", 27 April 2017.

[bug.chromium]
"Referer header includes URL fragment when opening link using New Tab", December 2018.

[fb_fragments]
"Facebook Developer Blog", December 2018.

[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-09 (work in progress), 4 November 2018, <https://www.ietf.org/archive/id/draft-bradley-oauth-jwt-encoded-state-09>.

[I-D.ietf-oauth-closing-redirectors]
Bradley, J., Sanso, A., and H. Tschofenig, "OAuth 2.0 Security: Closing Open Redirectors in OAuth", draft-ietf-oauth-closing-redirectors-00 (work in progress), 4 February 2016, <https://www.ietf.org/archive/id/draft-ietf-oauth-closing-redirectors-00>.

[I-D.ietf-oauth-jwsreq]
Sakimura, N. and J. Bradley, "The OAuth 2.0 Authorization Framework: JWT Secured Authorization Request (JAR)", draft-ietf-oauth-jwsreq-17 (work in progress), 21 October 2018, <https://www.ietf.org/archive/id/draft-ietf-oauth-jwsreq-

17>.

[I-D.ietf-oauth-mix-up-mitigation]
Jones, M., Bradley, J., and N. Sakimura, "OAuth 2.0 Mix-Up Mitigation", draft-ietf-oauth-mix-up-mitigation-01 (work in progress), 7 July 2016, <https://www.ietf.org/archive/id/draft-ietf-oauth-mix-up-mitigation-01>.

[I-D.ietf-oauth-mtls]
Campbell, B., Bradley, J., Sakimura, N., and T. Lodderstedt, "OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens", draft-ietf-oauth-mtls-12 (work in progress), 18 October 2018, <https://www.ietf.org/archive/id/draft-ietf-oauth-mtls-12>.

[I-D.ietf-oauth-pop-key-distribution]
Bradley, J., Hunt, P., Jones, M., Tschofenig, H., and M. Mihaly, "OAuth 2.0 Proof-of-Possession: Authorization Server to Client Key Distribution", draft-ietf-oauth-pop-key-distribution-04 (work in progress), 23 October 2018, <https://www.ietf.org/archive/id/draft-ietf-oauth-pop-key-distribution-04>.

[I-D.ietf-oauth-resource-indicators]
Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", draft-ietf-oauth-resource-indicators-01 (work in progress), 19 October 2018, <https://www.ietf.org/archive/id/draft-ietf-oauth-resource-indicators-01>.

[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), 8 August 2016, <https://www.ietf.org/archive/id/draft-ietf-oauth-signed-http-request-03>.

[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-08 (work in progress), 19 October 2018, <https://www.ietf.org/archive/id/draft-ietf-oauth-token-binding-08>.

[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-18 (work in progress), 26 June 2018, <https://www.ietf.org/archive/id/draft-ietf-tokbind-https-

18>.

[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), 27 March 2017, <https://www.ietf.org/archive/id/draft-sakimura-oauth-jpop-04>.

[oauth-v2-form-post-response-mode]
"OAuth 2.0 Form Post Response Mode", 27 April 2015.

[oauth_security_jcs_14]
Maffeis, S., "Discovering concrete attacks on website authorization by formal analysis", 23 April 2014.

[OpenID] "OpenID Connect Core 1.0 incorporating errata set 1", 8 November 2014.

[owasp] "Open Web Application Security Project Home Page", December 2018.

[owasp_csrf]
"Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet", December 2018.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, DOI 10.17487/RFC2616, June 1999, <https://www.rfc-editor.org/info/rfc2616>.

[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>.

[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>.

[RFC8414] Jones, M., Sakimura, N., and J. Bradley, "OAuth 2.0 Authorization Server Metadata", RFC 8414, DOI 10.17487/RFC8414, June 2018, <https://www.rfc-editor.org/info/rfc8414>.

[webappsec-referrer-policy] "Referrer Policy", 20 April 2017.

## Appendix A. Document History

[[ To be removed from the final specification ]]

-11

* Adapted section 2.1.2 to outcome of consensus call

* more text on refresh token inactivity and implementation note on refres token replay detection via refresh token rotation

-10

* incorporated feedback by Joseph Heenan

* changed occurrences of SHALL to MUST

* added text on lack of token/cert binding support tokens issued in the authorization response as justification to not recommend issuing tokens there at all

* added requirement to authenticate clients during code exchange (PKCE or client credential) to 2.1.1.

* added section on refresh tokens

* editorial enhancements to 2.1.2 based on feedback

-09

* changed text to recommend not to use implicit but code

* added section on access token injection

* reworked sections 3.1 through 3.3 to be more specific on implicit grant issues

-08

* added recommendations re implicit and token injection

* uppercased key words in Section 2 according to RFC 2119

-07

* incorporated findings of Doug McDorman

* added section on HTTP status codes for redirects

* added new section on access token privilege restriction based on comments from Johan Peeters

-06

* reworked section 3.8.1

* incorporated Phil Hunt's feedback

* reworked section on mix-up

* extended section on code leakage via referrer header to also cover state leakage

* added Daniel Fett as author

* replaced text intended to inform WG discussion by recommendations to implementors

* modified example URLs to conform to RFC 2606

-05

* Completed sections on code leakage via referrer header, attacks in browser, mix-up, and CSRF

* Reworked Code Injection Section

* Added reference to OpenID Connect spec

*  removed refresh token leakage as respective considerations have been given in section 10.4 of RFC 6749

*  first version on open redirection

*  incorporated Christian Mainka's review feedback

-04

*  Restructured document for better readability

*  Added best practices on Token Leakage prevention

-03

*  Added section on Access Token Leakage at Resource Server

*  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
yes.com

Email: torsten@lodderstedt.net

John Bradley
Yubico

Email: ve7jtb@ve7jtb.com

Andrey Labunets
Facebook

Email: isciurus@fb.com

Daniel Fett
yes.com

Email: mail@danielfett.de