Stream:	Internet Engineering Task Force (IETF)
RFC:	9700
BCP:	240
Updates:	6749, 6750, 6819
Category:	Best Current Practice
Published:	November 2024
ISSN:	2070-1721

Authors:	T. Lodderstedt	J. Bradley	A. Labunets	D. Fett
	SPRIND	*Yubico*	*Independent Researcher*	*Authlete*

RFC 9700
OAuth 2.0 Security Best Current Practice

Abstract

This document describes best current security practice for OAuth 2.0. It updates and extends the threat model and security advice given in RFCs 6749, 6750, and 6819 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. Further, it deprecates some modes of operation that are deemed less secure or even insecure.

Status of This Memo

This memo documents an Internet Best Current Practice.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on BCPs is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9700.

Copyright Notice

Table of Contents

1. Introduction

Since its publication in [RFC6749] and [RFC6750], OAuth 2.0 (referred to as simply "OAuth" in this document) has gained massive traction in the market and became the standard for API protection and the basis for federated login using OpenID Connect [OpenID.Core]. While OAuth is used in a variety of scenarios and different kinds of deployments, the following challenges can be observed:

- OAuth implementations are being attacked through known implementation weaknesses and anti-patterns (i.e., well-known patterns that are considered insecure). Although most of these threats are discussed in the OAuth 2.0 Threat Model and Security Considerations [RFC6819], continued exploitation demonstrates a need for more specific recommendations, easier to implement mitigations, and more defense in depth.
- OAuth is being used in environments with higher security requirements than considered initially, such as open banking, eHealth, eGovernment, and electronic signatures. Those use cases call for stricter guidelines and additional protection.
- OAuth is being 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], [RFC6750], and [RFC6819].

  OAuth initially assumed static relationships between clients, authorization servers, and resource servers. The URLs of the servers were known to the client at deployment time and built an anchor for the trust relationships among those parties. The validation of whether the client is talking to a legitimate server was based on TLS server authentication (see Section 4.5.4 of [RFC6819]). 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 email or OpenID Connect) or serve as a front end to a particular tenant in a multi-tenant environment. Extensions of OAuth, such as the OAuth 2.0 Dynamic Client Registration Protocol [RFC7591] and OAuth 2.0 Authorization Server Metadata [RFC8414] were developed to support the use of OAuth in dynamic scenarios.
- Technology has changed. For example, the way browsers treat fragments when redirecting requests has changed, and with it, the implicit grant's underlying security model.

This document provides updated security recommendations to address these challenges. It introduces new requirements beyond those defined in existing specifications such as OAuth 2.0 [RFC6749] and OpenID Connect [OpenID.Core] and deprecates some modes of operation that are deemed less secure or even insecure. However, this document does not supplant the security advice given in [RFC6749], [RFC6750], and [RFC6819], but complements those documents.

Naturally, not all existing ecosystems and implementations are compatible with the new requirements, and following the best practices described in this document may break interoperability. Nonetheless, it is **RECOMMENDED** that implementers upgrade their implementations and ecosystems as soon as feasible.

OAuth 2.1, under development as [OAUTH-V2.1], will incorporate security recommendations from this document.

1.1. Structure

The remainder of this document is organized as follows: Section 2 summarizes the most important best practices for every OAuth implementor. Section 3 presents the updated OAuth attacker model. Section 4 is a detailed analysis of the threats and implementation issues that can be found in the wild (at the time of writing) along with a discussion of potential countermeasures.

1.2. Conventions and Terminology

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This specification uses the terms "access token", "authorization endpoint", "authorization grant", "authorization server", "client", "client identifier" (client ID), "protected resource", "refresh token", "resource owner", "resource server", and "token endpoint" defined by OAuth 2.0 [RFC6749].

An "open redirector" is an endpoint on a web server that forwards a user's browser to an arbitrary URI obtained from a query parameter.

2. Best Practices

This section describes the core set of security mechanisms and measures that are considered to be best practices at the time of writing. Details about these security mechanisms and measures (including detailed attack descriptions) and requirements for less commonly used options are provided in Section 4.

2.1. Protecting Redirect-Based Flows

When comparing client redirect URIs against pre-registered URIs, authorization servers **MUST** utilize exact string matching except for port numbers in `localhost` redirection URIs of native apps (see Section 4.1.3). This measure contributes to the prevention of leakage of authorization codes and access tokens (see Section 4.1). It can also help to detect mix-up attacks (see Section 4.4).

Clients and authorization servers **MUST NOT** expose URLs that forward the user's browser to arbitrary URIs obtained from a query parameter (open redirectors) as described in Section 4.11. Open redirectors can enable exfiltration of authorization codes and access tokens.

Clients **MUST** prevent Cross-Site Request Forgery (CSRF). In this context, CSRF refers to requests to the redirection endpoint that do not originate at the authorization server, but at a malicious third party (see Section 4.4.1.8 of [RFC6819] for details). Clients that have ensured that the authorization server supports Proof Key for Code Exchange (PKCE) [RFC7636] **MAY** rely on the CSRF protection provided by PKCE. In OpenID Connect flows, the `nonce` parameter provides CSRF protection. Otherwise, one-time use CSRF tokens carried in the `state` parameter that are securely bound to the user agent **MUST** be used for CSRF protection (see Section 4.7.1).

When an OAuth client can interact with more than one authorization server, a defense against mix-up attacks (see Section 4.4) is **REQUIRED**. To this end, clients **SHOULD**

- use the `iss` parameter as a countermeasure according to [RFC9207], or
- use an alternative countermeasure based on an `iss` value in the authorization response (such as the `iss` claim in the ID Token in [OpenID.Core] or in [OpenID.JARM] responses), processing it as described in [RFC9207].

In the absence of these options, clients **MAY** instead use distinct redirect URIs to identify authorization endpoints and token endpoints, as described in Section 4.4.2.

An authorization server that redirects a request potentially containing user credentials **MUST** avoid forwarding these user credentials accidentally (see Section 4.12 for details).

2.1.1. Authorization Code Grant

Clients **MUST** prevent authorization code injection attacks (see Section 4.5) and misuse of authorization codes using one of the following options:

- Public clients **MUST** use PKCE [RFC7636] to this end, as motivated in Section 4.5.3.1.
- For confidential clients, the use of PKCE [RFC7636] is **RECOMMENDED**, as it provides strong protection against misuse and injection of authorization codes as described in Section 4.5.3.1. Also, as a side effect, it prevents CSRF even in the presence of strong attackers as described in Section 4.7.1.
- With additional precautions, described in Section 4.5.3.2, confidential OpenID Connect [OpenID.Core] clients **MAY** use the `nonce` parameter and the respective Claim in the ID Token instead.

In any case, the PKCE challenge or OpenID Connect `nonce` **MUST** be transaction-specific and securely bound to the client and the user agent in which the transaction was started. Authorization servers are encouraged to make a reasonable effort at detecting and preventing the use of constant values for the PKCE challenge or OpenID Connect `nonce`.

Note: Although PKCE was designed as a mechanism to protect native apps, this advice applies to all kinds of OAuth clients, including web applications.

When using PKCE, clients **SHOULD** use PKCE code challenge methods that do not expose the PKCE verifier in the authorization request. Otherwise, attackers that can read the authorization request (cf. attacker (A4) in Section 3) can break the security provided by PKCE. Currently, `S256` is the only such method.

Authorization servers **MUST** support PKCE [RFC7636].

If a client sends a valid PKCE `code_challenge` parameter in the authorization request, the authorization server **MUST** enforce the correct usage of `code_verifier` at the token endpoint.

Authorization servers **MUST** mitigate PKCE downgrade attacks by ensuring that a token request containing a `code_verifier` parameter is accepted only if a `code_challenge` parameter was present in the authorization request; see Section 4.8.2 for details.

Authorization servers **MUST** provide a way to detect their support for PKCE. It is **RECOMMENDED** for authorization servers to publish the element `code_challenge_methods_supported` in their Authorization Server Metadata [RFC8414] containing the supported PKCE challenge methods (which can be used by the client to detect PKCE support). Authorization servers **MAY** instead provide a deployment-specific way to ensure or determine PKCE support by the authorization server.

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 Sections 4.1, 4.2, 4.3, and 4.6.

Moreover, no standardized method for sender-constraining exists to bind access tokens to a specific client (as recommended in Section 2.2) when the access tokens are issued in the authorization response. This means that an attacker can use the leaked or stolen access token at a resource endpoint.

In order to avoid these issues, clients **SHOULD NOT** use the implicit grant (response type "token") or other response types issuing access tokens in the authorization response, unless access token injection in the authorization response is prevented and the aforementioned token leakage vectors are mitigated.

Clients **SHOULD** instead use the response type `code` (i.e., 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, such as the `code id_token` response type. This allows the authorization server to detect replay attempts by attackers 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 (see Section 2.2.

2.2. Token Replay Prevention

2.2.1. Access Tokens

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 a prerequisite for the acceptance of that token at the recipient (e.g., a resource server).

Authorization and resource servers **SHOULD** use mechanisms for sender-constraining access tokens, such as Mutual TLS for OAuth 2.0 [RFC8705] or OAuth 2.0 Demonstrating Proof of Possession (DPoP) [RFC9449] (see Section 4.10.1), to prevent misuse of stolen and leaked access tokens.

2.2.2. Refresh Tokens

Refresh tokens for public clients **MUST** be sender-constrained or use refresh token rotation as described in Section 4.14. [RFC6749] already mandates that refresh tokens for confidential clients can only be used by the client for which they were issued.

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 help to reduce the impact of access token leakage.

In particular, access tokens **SHOULD** be audience-restricted to a specific resource server or, if that is not feasible, to a small set of resource servers. 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 it was not, the resource server **MUST** refuse to serve the respective request. The `aud` claim as defined in [RFC9068] **MAY** be used to audience-restrict access tokens. Clients and authorization servers **MAY** utilize the parameters `scope` or `resource` as specified in [RFC6749] and [RFC8707], 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] and `authorization_details` as specified in [RFC9396] to determine those resources and/or actions.

2.4. Resource Owner Password Credentials Grant

The resource owner password credentials grant [RFC6749] **MUST NOT** be used. This grant type insecurely exposes the credentials of the resource owner to the client. Even if the client is benign, this results in an increased attack surface (credentials can leak in more places than just the authorization server) and users are trained to enter their credentials in places other than the authorization server.

Furthermore, the resource owner password credentials grant is not designed to work with two-factor authentication and authentication processes that require multiple user interaction steps. Authentication with cryptographic credentials (cf. WebCrypto [W3C.WebCrypto], WebAuthn [W3C.WebAuthn]) may be impossible to implement with this grant type, as it is usually bound to a specific web origin.

2.5. Client Authentication

Authorization servers **SHOULD** enforce client authentication if it is feasible, in the particular deployment, to establish a process for issuance/registration of credentials for clients and ensuring the confidentiality of those credentials.

It is **RECOMMENDED** to use asymmetric cryptography for client authentication, such as mutual TLS (mTLS) [RFC8705] or signed JWTs ("Private Key JWT") in accordance with [RFC7521] and [RFC7523] (in [OpenID.Core] defined as the client authentication method `private_key_jwt`). When asymmetric cryptography for client authentication is used, authorization servers do not need to store sensitive symmetric keys, making these methods more robust against leakage of keys.

2.6. Other Recommendations

The use of OAuth Authorization Server Metadata [RFC8414] can help to improve the security of OAuth deployments:

- It ensures that security features and other new OAuth features can be enabled automatically by compliant software libraries.
- It reduces chances for misconfigurations -- for example, misconfigured endpoint URLs (that might belong to an attacker) or misconfigured security features.
- It can help to facilitate rotation of cryptographic keys and to ensure cryptographic agility.

It is therefore **RECOMMENDED** that authorization servers publish OAuth Authorization Server Metadata according to [RFC8414] and that clients make use of this Authorization Server Metadata (when available) to configure themselves.

Under the conditions described in Section 4.15.1, authorization servers **SHOULD NOT** allow clients to influence their `client_id` or any claim that could cause confusion with a genuine resource owner.

It is **RECOMMENDED** to use end-to-end TLS according to [BCP195] between the client and the resource server. If TLS traffic needs to be terminated at an intermediary, refer to Section 4.13 for further security advice.

Authorization responses **MUST NOT** be transmitted over unencrypted network connections. To this end, authorization servers **MUST NOT** allow redirect URIs that use the `http` scheme except for native clients that use loopback interface redirection as described in Section 7.3 of [RFC8252].

If the authorization response is sent with in-browser communication techniques like postMessage [WHATWG.postmessage_api] instead of HTTP redirects, both the initiator and receiver of the in-browser message **MUST** be strictly verified as described in Section 4.17.

To support browser-based clients, endpoints directly accessed by such clients including the Token Endpoint, Authorization Server Metadata Endpoint, `jwks_uri` Endpoint, and Dynamic Client Registration Endpoint **MAY** support the use of Cross-Origin Resource Sharing (CORS) [WHATWG.CORS]. However, CORS **MUST NOT** be supported at the authorization endpoint, as the client does not access this endpoint directly; instead, the client redirects the user agent to it.

3. The Updated OAuth 2.0 Attacker Model

In [RFC6819], a threat model is laid out that describes the threats against which OAuth deployments must be protected. While doing so, [RFC6819] makes certain assumptions about attackers and their capabilities, i.e., it implicitly establishes an attacker model. In the following, this attacker model is made explicit and is updated and expanded to account for the potentially dynamic relationships involving multiple parties (as described in Section 1), to include new types of attackers, and to define the attacker model more clearly.

The goal of this document is to ensure that the authorization of a resource owner (with a user agent) at an authorization server and the subsequent usage of the access token at a resource server is protected, as well as practically possible, at least against the following attackers.

(A1) Web attackers that can set up and operate an arbitrary number of network endpoints (besides the "honest" ones) including browsers and servers. Web attackers may set up websites that are visited by the resource owner, operate their own user agents, and participate in the protocol.

In particular, web attackers may operate OAuth clients that are registered at the authorization server, and they may operate their own authorization and resource servers that can be used (in parallel to the "honest" ones) by the resource owner and other resource owners.

It must also be assumed that web attackers can lure the user to navigate their browser to arbitrary attacker-chosen URIs at any time. In practice, this can be achieved in many ways, for example, by injecting malicious advertisements into advertisement networks or by sending legitimate-looking emails.

Web attackers can use their own user credentials to create new messages as well as any secrets they learned previously. For example, if a web attacker learns an authorization code of a user through a misconfigured redirect URI, the web attacker can then try to redeem that code for an access token.

They cannot, however, read or manipulate messages that are not targeted towards them (e.g., sent to a URL controlled by a non-attacker-controlled authorization server).

(A2) Network attackers that additionally have full control over the network over which protocol participants communicate. They can eavesdrop on, manipulate, and spoof messages, except when these are properly protected by cryptographic methods (e.g., TLS). Network attackers can also block arbitrary messages.

While an example for a web attacker would be a customer of an internet service provider, network attackers could be the internet service provider itself, an attacker in a public (Wi-Fi) network using ARP spoofing, or a state-sponsored attacker with access to internet exchange points, for instance.

The aforementioned attackers (A1) and (A2) conform to the attacker model that was used in formal analysis efforts for OAuth [arXiv.1601.01229]. This is a minimal attacker model. Implementers **MUST** take into account all possible types of attackers in the environment of their OAuth implementations. For example, in [arXiv.1901.11520], a very strong attacker model is used that includes attackers that have full control over the token endpoint. This models effects of a possible misconfiguration of endpoints in the ecosystem, which can be avoided by using authorization server metadata as described in Section 2.6. Such an attacker is therefore not listed here.

However, previous attacks on OAuth have shown that the following types of attackers are relevant in particular:

(A3) Attackers that can read, but not modify, the contents of the authorization response (i.e., the authorization response can leak to an attacker).

Examples of such attacks include open redirector attacks; insufficient checking of redirect URIs (see Section 4.1); problems existing on mobile operating systems (where different apps can register themselves on the same URI); mix-up attacks (see Section 4.4), where the client is tricked into sending credentials to an attacker-controlled authorization server; and the fact that URLs are often stored/logged by browsers (history), proxy servers, and operating systems.

(A4) Attackers that can read, but not modify, the contents of the authorization request (i.e., the authorization request can leak, in the same manner as above, to an attacker).

(A5) Attackers that can acquire an access token issued by an authorization server. For example, a resource server may be compromised by an attacker, an access token may be sent to an attacker-controlled resource server due to a misconfiguration, or social engineering may be used to get a resource owner to use an attacker-controlled resource server. Also see Section 4.9.2.

(A3), (A4), and (A5) typically occur together with either (A1) or (A2). Attackers can collaborate to reach a common goal.

Note that an attacker (A1) or (A2) can be a resource owner or act as one. For example, such an attacker can use their own browser to replay tokens or authorization codes obtained by any of the attacks described above at the client or resource server.

This document focuses on threats resulting from attackers (A1) to (A5).

4. Attacks and Mitigations

This section gives a detailed description of attacks on OAuth implementations, along with potential countermeasures. Attacks and mitigations already covered in [RFC6819] are not listed here, except where new recommendations are made.

This section further defines additional requirements (beyond those defined in Section 2) for certain cases and protocol options.

4.1. Insufficient Redirect URI Validation

Some authorization servers allow clients to register redirect URI patterns instead of complete redirect URIs. The authorization servers then match the redirect URI parameter value at the authorization endpoint against the registered patterns at runtime. This approach allows clients to encode transaction state into additional redirect URI parameters or to register a single pattern for multiple redirect URIs.

This approach turned out to be more complex to implement and more error-prone to manage than exact redirect URI matching. Several successful attacks exploiting flaws in the pattern-matching implementation or concrete configurations have been observed in the wild (see, e.g., [research.rub2]). Insufficient validation of the redirect URI 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 attacker's 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.

These attacks are shown in detail in the following subsections.

4.1.1. Redirect URI Validation Attacks on Authorization Code Grant

For a client using the grant type `code`, an attack may work as follows:

Assume the redirect URL pattern `https://*.somesite.example/*` is registered for the client with the client ID `s6BhdRkqt3`. The intention is to allow any subdomain of `somesite.example` to be a valid redirect URI for the client, for example, `https://app1.somesite.example/redirect`. However, a naive implementation on the authorization server might interpret the wildcard `*` as "any character" and not "any character valid for a domain name". The authorization server, therefore, might permit `https://attacker.example/.somesite.example` as a redirect URI, although `attacker.example` is a different domain potentially controlled by a malicious party.

The attack can then be conducted as follows:

To begin, the attacker needs to trick the user into opening a tampered URL in their browser that launches a page under the attacker's control, say, `https://www.evil.example` (see attacker A1 in Section 3).

This URL initiates the following authorization request with the client ID of a legitimate client to the authorization endpoint (line breaks for display only):

```
GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=9ad67f13
    &redirect_uri=https%3A%2F%2Fattacker.example%2F.somesite.example
    HTTP/1.1
Host: server.somesite.example
```

The authorization server validates the redirect URI and compares it to the registered redirect URL patterns for the client `s6BhdRkqt3`. The authorization request is processed and presented to the user.

If the user does not see the redirect URI or does not recognize the attack, the code is issued and immediately sent to the attacker's domain. If an automatic approval of the authorization is enabled (which is not recommended for public clients according to [RFC6749]), the attack can be performed even without user interaction.

If the attacker impersonates a public client, the attacker can exchange the code for tokens at the respective token endpoint.

This attack will not work as easily for confidential clients, since the code exchange requires authentication with the legitimate client's secret. However, the attacker can use the legitimate confidential client to redeem the code by performing an authorization code injection attack; see Section 4.5.

It is important to note that redirect URI validation vulnerabilities can also exist if the authorization server handles wildcards properly. For example, assume that the client registers the redirect URL pattern `https://*.somesite.example/*` and the authorization server interprets this as "allow redirect URIs pointing to any host residing in the domain `somesite.example`". If an attacker manages to establish a host or subdomain in `somesite.example`, the attacker can impersonate the legitimate client. For example, this could be caused by a subdomain takeover attack [research.udel], where an outdated CNAME record (say,

`external-service.somesite.example`) points to an external DNS name that no longer exists (say, `customer-abc.service.example`) and can be taken over by an attacker (e.g., by registering as `customer-abc` with the external service).

4.1.2. Redirect URI Validation 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 an attacker-controlled URI, the attacker will directly get access to the fragment carrying the access token.

Additionally, implicit grants (and also other grants when using `response_mode=fragment` as defined in [OAuth.Responses]) can be subject to a further kind of attack. The attack utilizes the fact that user agents reattach fragments to the destination URL of a redirect if the location header does not contain a fragment (see Section 17.11 of [RFC9110]). The attack described here combines this behavior with the client as an open redirector (see Section 4.11.1) in order to obtain access tokens. This allows circumvention even of very narrow redirect URI patterns, but not of strict URL matching.

Assume the registered URL 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.

The attack can now be conducted as follows:

To begin, as above, the attacker needs to trick the user into opening a tampered URL in their browser that launches a page under the attacker's control, say, `https://www.evil.example`.

Afterwards, the website initiates an authorization request that is very similar to the one in the attack on the code flow. Different to above, it utilizes the open redirector by encoding `redirect_to=https://attacker.example` into the parameters of the redirect URI, and it uses the response type "token" (line breaks for display only):

```
GET /authorize?response_type=token&state=9ad67f13
    &client_id=s6BhdRkqt3
    &redirect_uri=https%3A%2F%2Fclient.somesite.example
     %2Fcb%26redirect_to%253Dhttps%253A%252F
     %252Fattacker.example%252F HTTP/1.1
Host: server.somesite.example
```

Then, since the redirect URI matches the registered pattern, the authorization server permits the request and sends the resulting access token in a 303 redirect (some response parameters omitted for readability):

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

At client.somesite.example, the request arrives at the open redirector. The endpoint will read the redirect parameter and will issue an HTTP 303 Location header redirect to the URL `https://attacker.example/`.

```
HTTP/1.1 303 See Other
Location: https://attacker.example/
```

Since the redirector at client.somesite.example does not include a fragment in the Location header, the user agent will reattach the original fragment `#access_token=2YotnFZFEjr1zCsicMWpAA&...` to the URL and will navigate to the following URL:

```
https://attacker.example/#access_token=2YotnFZFEjr1z...
```

The attacker's page at `attacker.example` can then access the fragment and obtain the access token.

4.1.3. Countermeasures

The complexity of implementing and managing pattern matching correctly obviously causes security issues. This document therefore advises simplifying the required logic and configuration by using exact redirect URI matching. This means the authorization server **MUST** ensure that the two URIs are equal; see Section 6.2.1 of [RFC3986], Simple String Comparison, for details. The only exception is native apps using a `localhost` URI: In this case, the authorization server **MUST** allow variable port numbers as described in Section 7.3 of [RFC8252].

Additional recommendations:

- Web servers on which redirect URIs are hosted **MUST NOT** expose open redirectors (see Section 4.11).
- Browsers reattach URL fragments to Location redirection URLs only if the URL in the Location header does not already contain a fragment. Therefore, servers **MAY** prevent browsers from reattaching fragments to redirection URLs by attaching an arbitrary fragment identifier, for example `#_`, to URLs in Location headers.
- Clients **SHOULD** use the authorization code response type instead of response types that cause access token issuance at the authorization endpoint. This offers countermeasures against the reuse of leaked credentials through the exchange process with the authorization server and against token replay through sender-constraining of the access tokens.

If the origin and integrity of the authorization request containing the redirect URI can be verified, for example, when using [RFC9101] or [RFC9126] with client authentication, the authorization server **MAY** trust the redirect URI without further checks.

4.2. Credential Leakage via Referer Headers

The contents of the authorization request URI or the authorization response URI can unintentionally be disclosed to attackers through the Referer HTTP header (see Section 10.1.3 of [RFC9110]), by leaking from either the authorization server's or the client's website, respectively. Most importantly, authorization codes or `state` values can be disclosed in this way. Although specified otherwise in Section 10.1.3 of [RFC9110], the same may happen to access tokens conveyed in URI fragments due to browser implementation issues, as illustrated by a (now fixed) issue in the Chromium project [bug.chromium].

4.2.1. Leakage from the OAuth Client

Leakage from the OAuth client 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 and a user clicks on such a link, or
- includes third-party content (advertisements in 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` or `state` (and potentially `access_token`).

4.2.2. Leakage from the Authorization Server

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

4.2.3. Consequences

An attacker that learns a valid code or access token through a Referer header can perform the attacks as described in Sections 4.1.1, 4.5 and 4.6. If the attacker learns `state`, the CSRF protection achieved by using `state` is lost, resulting in CSRF attacks as described in Section 4.4.1.8 of [RFC6819].

4.2.4. 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:

- Suppress the Referer header by applying an appropriate Referrer Policy [W3C.webappsec-referrer-policy] to the document (either as part of the "referrer" meta attribute or by setting a Referrer-Policy header). For example, the header `Referrer-Policy: no-referrer` in the

response completely suppresses the Referer header in all requests originating from the resulting document.

- Use authorization code instead of response types causing access token issuance from the authorization endpoint.
- Bind the authorization code to a confidential client or PKCE challenge. In this case, the attacker lacks the secret to request the code exchange.
- As described in Section 4.1.2 of [RFC6749], authorization codes **MUST** be invalidated by the authorization server after their first use at the token endpoint. For example, if an authorization server invalidated the code after the legitimate client redeemed it, the attacker would fail to exchange 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. Therefore, [RFC6749] further recommends that, when an attempt is made to redeem a code twice, the authorization server **SHOULD** revoke all tokens issued previously based on that code.
- 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 Referer header from the client's website, the `state` was already used, invalidated by the client and cannot be used again by the attacker. (This does not help if the `state` leaks from the authorization server's website, since then the `state` has not been used at the redirection endpoint at the client yet.)
- Use the form post response mode instead of a redirect for the authorization response (see [OAuth.Post]).

4.3. Credential Leakage via 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.

4.3.1. Authorization 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.

Countermeasures:

- Authorization code replay prevention as described in Section 4.4.1.1 of [RFC6819], and Section 4.5.
- Use the form post response mode instead of redirect for the authorization response (see [OAuth.Post]).

4.3.2. Access Token in Browser History

An access token may end up in the browser history if a client or a website that already has a token deliberately navigates to a page like `provider.com/get_user_profile?access_token=abcdef`. [RFC6750] discourages this practice and advises transferring tokens via a header, but in practice websites often pass access tokens in query parameters.

In the 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.

Countermeasures:

- Clients **MUST NOT** pass access tokens in a URI query parameter in the way described in Section 2.3 of [RFC6750]. The authorization code grant or alternative OAuth response modes like the form post response mode [OAuth.Post] can be used to this end.

4.4. Mix-Up Attacks

Mix-up attacks are in scenarios where an OAuth client interacts with two or more authorization servers and at least one authorization server is under the control of the attacker. This can be the case, for example, if the attacker uses dynamic registration to register the client at their own authorization server or if an authorization server becomes compromised.

The goal of the attack is to obtain an authorization code or an access token for an uncompromised authorization server. This is achieved by tricking the client into sending those credentials to the compromised authorization server (the attacker) instead of using them at the respective endpoint of the uncompromised authorization/resource server.

4.4.1. Attack Description

The description here follows [arXiv.1601.01229], with variants of the attack outlined below.

Preconditions: For this variant of the attack to work, it is assumed that

- the implicit or authorization code grant is used with multiple authorization servers of which one is considered "honest" (H-AS) and one is operated by the attacker (A-AS), and
- the client stores the authorization server chosen by the user in a session bound to the user's browser and uses the same redirection endpoint URI for each authorization server.

In the following, it is further assumed that the client is registered with H-AS (URI: `https://honest.as.example`, client ID: `7ZGZldHQ`) and with A-AS (URI: `https://attacker.example`, client ID: `666RVZJTA`). URLs shown in the following example are shortened for presentation to include only parameters relevant to the attack.

Attack on the authorization code grant:

1. The user selects to start the grant using A-AS (e.g., by clicking on a button on the client's website).
2. The client stores in the user's session that the user selected "A-AS" and redirects the user to A-AS's authorization endpoint with a Location header containing the URL `https://attacker.example/authorize?response_type=code&client_id=666RVZJTA`.
3. When the user's browser navigates to the attacker's authorization endpoint, the attacker immediately redirects the browser to the authorization endpoint of H-AS. In the authorization request, the attacker replaces the client ID of the client at A-AS with the client's ID at H-AS. Therefore, the browser receives a redirection (303 See Other) with a Location header pointing to `https://honest.as.example/authorize?response_type=code&client_id=7ZGZldHQ`
4. The user authorizes the client to access their resources at H-AS. (Note that a vigilant user might at this point detect that they intended to use A-AS instead of H-AS. The first attack variant listed below avoids this.) H-AS issues a code and sends it (via the browser) back to the client.
5. 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.
6. The attacker therefore obtains code and can either exchange the code for an access token (for public clients) or perform an authorization code injection attack as described in Section 4.5.

Variants:

- Mix-Up with Interception: This variant works only if the attacker can intercept and manipulate the first request/response pair from a user's browser to the client (in which the user selects a certain authorization server and is then redirected by the client to that authorization server), as in Attacker (A2) (see Section 3). This capability can, for example, be the result of a attacker-in-the-middle attack on the user's connection to the client. In the attack, the user starts the flow with H-AS. The attacker intercepts this request and changes the user's selection to A-AS. The rest of the attack proceeds as in Step 2 and following above.
- Implicit Grant: In the implicit grant, the attacker receives an access token instead of the code in Step 4. The attacker's authorization server receives the access token when the client makes either a request to the A-AS userinfo endpoint or a request to the attacker's resource server (since the client believes it has completed the flow with A-AS).
- Per-AS Redirect URIs: If clients use different redirect URIs for different authorization servers, clients do not store the selected authorization server in the user's session, and authorization servers do not check the redirect URIs properly, attackers can mount an attack called "Cross-Social Network Request Forgery". These attacks have been observed in practice. Refer to [research.jcs_14] for details.
- OpenID Connect: Some variants can be used to attack OpenID Connect. In these attacks, the attacker misuses features of the OpenID Connect Discovery [OpenID.Discovery] mechanism or replays access tokens or ID Tokens to conduct a mix-up attack. The attacks are described

in detail in Appendix A of [arXiv.1704.08539] and Section 6 of [arXiv.1508.04324v2] ("Malicious Endpoints Attacks").

4.4.2. Countermeasures

When an OAuth client can only interact with one authorization server, a mix-up defense is not required. In scenarios where an OAuth client interacts with two or more authorization servers, however, clients **MUST** prevent mix-up attacks. Two different methods are discussed below.

For both defenses, clients **MUST** store, for each authorization request, the issuer they sent the authorization request to and bind this information to the user agent. The issuer serves, via the associated metadata, as an abstract identifier for the combination of the authorization endpoint and token endpoint that are to be used in the flow. If an issuer identifier is not available (for example, if neither OAuth Authorization Server Metadata [RFC8414] nor OpenID Connect Discovery [OpenID.Discovery] is used), a different unique identifier for this tuple or the tuple itself can be used instead. For brevity of presentation, such a deployment-specific identifier will be subsumed under the issuer (or issuer identifier) in the following.

It is important to note that just storing the authorization server URL is not sufficient to identify mix-up attacks. An attacker might declare an uncompromised authorization server's authorization endpoint URL as "their" authorization server URL, but declare a token endpoint under their own control.

4.4.2.1. Mix-Up Defense via Issuer Identification

This defense requires that the authorization server sends its issuer identifier in the authorization response to the client. When receiving the authorization response, the client **MUST** compare the received issuer identifier to the stored issuer identifier. If there is a mismatch, the client **MUST** abort the interaction.

There are different ways this issuer identifier can be transported to the client:

- The issuer information can be transported, for example, via a separate response parameter `iss`, defined in [RFC9207].
- When OpenID Connect is used and an ID Token is returned in the authorization response, the client can evaluate the `iss` claim in the ID Token.

In both cases, the `iss` value **MUST** be evaluated according to [RFC9207].

While this defense may require deploying new OAuth features to transport the issuer information, it is a robust and relatively simple defense against mix-up.

4.4.2.2. Mix-Up Defense via Distinct Redirect URIs

For this defense, clients **MUST** use a distinct redirect URI for each issuer they interact with.

Clients **MUST** check that the authorization response was received from the correct issuer by comparing the distinct redirect URI for the issuer to the URI where the authorization response was received on. If there is a mismatch, the client **MUST** abort the flow.

While this defense builds upon existing OAuth functionality, it cannot be used in scenarios where clients only register once for the use of many different issuers (as in some open banking schemes) and due to the tight integration with the client registration, it is harder to deploy automatically.

Furthermore, an attacker might be able to circumvent the protection offered by this defense by registering a new client with the "honest" authorization server using the redirect URI that the client assigned to the attacker's authorization server. The attacker could then run the attack as described above, replacing the client ID with the client ID of their newly created client.

This defense **SHOULD** therefore only be used if other options are not available.

4.5. Authorization Code Injection

An attacker who has gained access to an authorization code contained in an authorization response (see Attacker (A3) in Section 3) can try to redeem the authorization code for an access token or otherwise make use of the authorization code.

In the case that the authorization code was created for a public client, the attacker can send the authorization code to the token endpoint of the authorization server and thereby get an access token. This attack was described in Section 4.4.1.1. of [RFC6819].

For confidential clients, or in some special situations, the attacker can execute an authorization code injection attack, as described in the following.

In an authorization code injection attack, the attacker attempts to inject a stolen authorization code into the attacker's own session with the client. The aim is to associate the attacker's session at the client with the victim's resources or identity, thereby giving the attacker at least limited access to the victim's resources.

Besides circumventing the client authentication of confidential clients, other use cases for this attack include:

- The attacker wants to access certain functions in this particular client. As an example, the attacker wants to impersonate their victim in a certain app or on a certain website.
- The authorization or resource servers are limited to certain networks that the attacker is unable to access directly.

Except in these special cases, authorization code injection is usually not interesting when the code is created for a public client, as sending the code to the token endpoint is a simpler and more powerful attack, as described above.

4.5.1. Attack Description

The authorization code injection attack works as follows:

1. The attacker obtains an authorization code (see Attacker (A3) in Section 3). For the rest of the attack, only the capabilities of a web attacker (A1) are required.

2. From the attacker's device, the attacker starts a regular OAuth authorization process with the legitimate client.
3. In the response of the authorization server to the legitimate client, the attacker replaces the newly created authorization code with the stolen authorization code. Since this response is passing through the attacker's device, the attacker can use any tool that can intercept and manipulate the authorization response to this end. The attacker does not need to control the network.
4. The legitimate client sends the code to the authorization server's token endpoint, along with the `redirect_uri` and the client's client ID and client secret (or other means of client authentication).
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. All checks succeed and the authorization server issues access and other tokens to the client. The attacker has now associated their session with the legitimate client with the victim's resources and/or identity.

4.5.2. Discussion

Obviously, the check-in step (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 manipulated redirect URI 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.

Section 4.1.3 of [RFC6749] requires the authorization server 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 in Section 4.5.1, 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 attacker's page). So, the authorization server would detect the attack and refuse to exchange the code.

This check could also detect attempts to inject an authorization code that had been obtained from another instance of the same client on another device if certain conditions are fulfilled:

- the redirect URI itself contains a nonce or another kind of one-time use, secret data and
- the client has bound this data to this particular instance of the client.

But, this approach conflicts with the idea of enforcing 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 specification.

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. However, this kind of check obviously does not fulfill the intent of the specification, since the tampered redirect URI is not considered. So, any attempt to inject an authorization code obtained using the `client_id` of a legitimate client or by utilizing the legitimate client on another device will not be detected in the respective deployments.

It is also assumed that the requirements defined in Section 4.1.3 of [RFC6749] increase client implementation complexity as clients need to store or reconstruct the correct redirect URI for the call to the token endpoint.

Asymmetric methods for client authentication do not stop this attack, as the legitimate client authenticates at the token endpoint.

This document therefore recommends instead binding 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 using one of the mechanisms described next.

4.5.3. Countermeasures

There are two good technical solutions to binding authorization codes to client instances, as follows.

4.5.3.1. PKCE

The PKCE mechanism specified in [RFC7636] can be used as a countermeasure (even though it was originally designed to secure native apps). When the attacker attempts to inject an authorization code, the check of the `code_verifier` fails: the client uses its correct verifier, but the code is associated with a `code_challenge` that does not match this verifier.

PKCE not only protects against the authorization code injection attack but also protects authorization codes created for public clients: PKCE ensures that an attacker cannot redeem a stolen authorization code at the token endpoint of the authorization server without knowledge of the `code_verifier`.

4.5.3.2. Nonce

OpenID Connect's existing `nonce` parameter can protect against authorization code injection attacks. The `nonce` value is one-time use and is created by the client. The client is supposed to bind it to the user agent session and send it with the initial request to the OpenID Provider (OP). The OP puts the received `nonce` value into the ID Token that is issued as part of the code exchange at the token endpoint. If an attacker injects an authorization code in the authorization response, the nonce value in the client session and the `nonce` value in the ID Token received from

the token endpoint will not match, and the attack is detected. The assumption is that an attacker cannot get hold of the user agent state on the victim's device (from which the attacker has stolen the respective authorization code).

It is important to note that this countermeasure only works if the client properly checks the `nonce` parameter in the ID Token obtained from the token endpoint and does not use any issued token until this check has succeeded. More precisely, a client protecting itself against code injection using the `nonce` parameter

1. **MUST** validate the `nonce` in the ID Token obtained from the token endpoint, even if another ID Token was obtained from the authorization response (e.g., `response_type=code+id_token`), and
2. **MUST** ensure that, unless and until that check succeeds, all tokens (ID Tokens and the access token) are disregarded and not used for any other purpose.

It is important to note that `nonce` does not protect authorization codes of public clients, as an attacker does not need to execute an authorization code injection attack. Instead, an attacker can directly call the token endpoint with the stolen authorization code.

4.5.3.3. Other Solutions

Other solutions like binding `state` to the code, sender-constraining the code using cryptographic means, or per-instance client credentials are conceivable, but lack support and bring new security requirements.

PKCE is the most obvious solution for OAuth clients, as it is available at the time of writing, while `nonce` is appropriate for OpenID Connect clients.

4.5.4. Limitations

An attacker can circumvent the countermeasures described above if they can modify the `nonce` or `code_challenge` values that are used in the victim's authorization request. The attacker can modify these values to be the same ones as those chosen by the client in their own session in Step 2 of the attack above. (This requires that the victim's session with the client begins after the attacker started their session with the client.) If the attacker is then able to capture the authorization code from the victim, the attacker will be able to inject the stolen code in Step 3 even if PKCE or `nonce` are used.

This attack is complex and requires a close interaction between the attacker and the victim's session. Nonetheless, measures to prevent attackers from reading the contents of the authorization response still need to be taken, as described in Sections 4.1, 4.2, 4.3, 4.4, and 4.11.

4.6. Access Token Injection

In an access token injection attack, the attacker attempts to inject a stolen access token into a legitimate client (that is not under the attacker's 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 attacker starts an OAuth flow with the client using the implicit grant and modifies the authorization response by replacing the access token issued by the authorization server or directly making 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 attack and uses the access token injected by the attacker.

4.6.1. Countermeasures

There is no way to detect such an injection attack in pure-OAuth flows since the token is issued without any binding to the transaction or the particular user agent.

In OpenID Connect, the attack can be mitigated, as the authorization response additionally contains an ID Token containing the `at_hash` claim. The attacker therefore needs to replace both the access token as well as the ID Token in the response. The attacker cannot forge the ID Token, as it is signed or encrypted with authentication. The attacker also cannot inject a leaked ID Token matching the stolen access token, as the `nonce` claim in the leaked ID Token will contain (with a very high probability) a different value than the one expected in the authorization response.

Note that further protection, like sender-constrained access tokens, is still required to prevent attackers from using the access token at the resource endpoint directly.

The recommendations in Section 2.1.2 follow from this.

4.7. 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. This is a variant of an attack known as Cross-Site Request Forgery (CSRF).

4.7.1. Countermeasures

The long-established countermeasure is that clients pass a random value, also known as a CSRF Token, in the `state` parameter that links the request to the redirect URI to the user agent session as described. This countermeasure is described in detail in Section 5.3.5 of [RFC6819]. The same protection is provided by PKCE or the OpenID Connect `nonce` value.

When using PKCE instead of `state` or `nonce` for CSRF protection, it is important to note that:

- Clients **MUST** ensure that the authorization server supports PKCE before using PKCE for CSRF protection. If an authorization server does not support PKCE, `state` or `nonce` **MUST** be used for CSRF protection.
- If `state` is used for carrying application state, and the integrity of its contents is a concern, clients **MUST** protect `state` against tampering and swapping. This can be achieved by binding the contents of state to the browser session and/or signed/encrypted state values. One example of this is discussed in the expired Internet-Draft [JWT-ENCODED-STATE].

The authorization server therefore **MUST** provide a way to detect their support for PKCE. Using Authorization Server Metadata according to [RFC8414] is **RECOMMENDED**, but authorization servers **MAY** instead provide a deployment-specific way to ensure or determine PKCE support.

PKCE provides robust protection against CSRF attacks even in the presence of an attacker that can read the authorization response (see Attacker (A3) in Section 3). When `state` is used or an ID Token is returned in the authorization response (e.g., `response_type=code+id_token`), the attacker either learns the `state` value and can replay it into the forged authorization response, or can extract the `nonce` from the ID Token and use it in a new request to the authorization server to mint an ID Token with the same `nonce`. The new ID Token can then be used for the CSRF attack.

4.8. PKCE Downgrade Attack

An authorization server that supports PKCE but does not make its use mandatory for all flows can be susceptible to a PKCE downgrade attack.

The first prerequisite for this attack is that there is an attacker-controllable flag in the authorization request that enables or disables PKCE for the particular flow. The presence or absence of the `code_challenge` parameter lends itself for this purpose, i.e., the authorization server enables and enforces PKCE if this parameter is present in the authorization request, but it does not enforce PKCE if the parameter is missing.

The second prerequisite for this attack is that the client is not using `state` at all (e.g., because the client relies on PKCE for CSRF prevention) or that the client is not checking `state` correctly.

Roughly speaking, this attack is a variant of a CSRF attack. The attacker achieves the same goal as in the attack described in Section 4.7: The attacker injects an authorization code (and with that, an access token) that is bound to the attacker's resources into a session between their victim and the client.

4.8.1. Attack Description

1. The user has started an OAuth session using some client at an authorization server. In the authorization request, the client has set the parameter `code_challenge=hash(abc)` as the PKCE code challenge (with the hash function and parameter encoding as defined in [RFC7636]). The client is now waiting to receive the authorization response from the user's browser.
2. To conduct the attack, the attacker uses their own device to start an authorization flow with the targeted client. The client now uses another PKCE code challenge, say, `code_challenge=hash(xyz)`, in the authorization request. The attacker intercepts the request and removes the entire `code_challenge` parameter from the request. Since this step is performed on the attacker's device, the attacker has full access to the request contents, for example, using browser debug tools.
3. If the authorization server allows for flows without PKCE, it will create a code that is not bound to any PKCE code challenge.

4. The attacker now redirects the user's browser to an authorization response URL that contains the code for the attacker's session with the authorization server.
5. The user's browser sends the authorization code to the client, which will now try to redeem the code for an access token at the authorization server. The client will send `code_verifier=abc` as the PKCE code verifier in the token request.
6. Since the authorization server sees that this code is not bound to any PKCE code challenge, it will not check the presence or contents of the `code_verifier` parameter. It will issue an access token (which belongs to the attacker's resource) to the client under the user's control.

4.8.2. Countermeasures

Using `state` properly would prevent this attack. However, practice has shown that many OAuth clients do not use or check `state` properly.

Therefore, authorization servers **MUST** mitigate this attack.

Note that from the view of the authorization server, in the attack described above, a `code_verifier` parameter is received at the token endpoint although no `code_challenge` parameter was present in the authorization request for the OAuth flow in which the authorization code was issued.

This fact can be used to mitigate this attack. [RFC7636] already mandates that

- an authorization server that supports PKCE **MUST** check whether a code challenge is contained in the authorization request and bind this information to the code that is issued; and
- when a code arrives at the token endpoint, and there was a `code_challenge` in the authorization request for which this code was issued, there must be a valid `code_verifier` in the token request.

Beyond this, to prevent PKCE downgrade attacks, the authorization server **MUST** ensure that if there was no `code_challenge` in the authorization request, a request to the token endpoint containing a `code_verifier` is rejected.

Authorization servers that mandate the use of PKCE (in general or for particular clients) implicitly implement this security measure.

4.9. Access Token Leakage at the Resource Server

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

4.9.1. Access Token Phishing by Counterfeit Resource Server

An attacker may set up their own resource server and trick a client into sending access tokens to it that are valid for other resource servers (see Attackers (A1) and (A5) in Section 3). 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 one specific resource server (and its URL) at development time, but client instances are provided with the resource server URL at runtime. This kind of late binding is typical in situations where the client uses a service implementing a standardized API (e.g., for email, calendar, health, or banking) and where the client is configured by a user or administrator for a service that this user or company uses.

4.9.2. Compromised Resource Server

An attacker may compromise a resource server to gain access to the resources of the respective deployment. Such a compromise may range from partial access to the system, e.g., its log files, to full control over the respective server, in which case all controls can be circumvented and all resources can be accessed. The attacker would also be able to obtain other access tokens held on the compromised system that would potentially be valid to access other resource servers.

Preventing server breaches by hardening and monitoring server systems is considered a standard operational procedure and, therefore, out of the scope of this document. This section focuses on the impact of OAuth-related breaches and the replaying of captured access tokens.

4.9.3. Countermeasures

The following measures should be taken into account by implementers in order to cope with access token replay by malicious actors:

- Sender-constrained access tokens, as described in Section 4.10.1, **SHOULD** be used to prevent the attacker from replaying the access tokens on other resource servers. If an attacker has only partial access to the compromised system, like a read-only access to web server logs, sender-constrained access tokens may also prevent replay on the compromised system.
- Audience restriction as described in Section 4.10.2 **SHOULD** be used to prevent replay of captured access tokens on other resource servers.
- The resource server **MUST** treat access tokens like other sensitive secrets and not store or transfer them in plaintext.

The first and second recommendations also apply to other scenarios where access tokens leak (see Attacker (A5) in Section 3).

4.10. Misuse of Stolen Access Tokens

Access tokens can be stolen by an attacker in various ways, for example, via the attacks described in Sections 4.1, 4.2, 4.3, 4.4, and 4.9. Some of these attacks can be mitigated by specific security measures, as described in the respective sections. However, in some cases, these measures are not sufficient or are not implemented correctly. Authorization servers therefore **SHOULD** ensure that access tokens are sender-constrained and audience-restricted as described in the following. Architecture and performance reasons may prevent the use of these measures in some deployments.

4.10.1. Sender-Constrained Access Tokens

As the name suggests, sender-constrained access tokens scope the applicability of an access token to a certain sender. This sender is obliged to demonstrate knowledge of a certain secret as a 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 that binds this particular token to a certain client. The binding can utilize the client's identity, but in most cases, the authorization server 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 authorization server creates the binding or the authorization server creates ephemeral keys. The way preexisting 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 happen during the setup of a TLS connection.
3. The resource server must implement the actual proof-of-possession check. This is typically done on the application level, often tied to specific material provided by the transport layer (e.g., TLS). The resource server must also ensure that a replay of the proof of possession is not possible.

Two methods for sender-constrained access tokens using proof of possession have been defined by the OAuth working group and are in use in practice:

- "OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens" [RFC8705]: The approach specified in this) document allows the use of mutual TLS (mTLS) for both client authentication and sender-constrained access tokens. For the purpose of sender-constrained access tokens, the client is identified towards the resource server by the fingerprint of its public key. During the 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.
- "OAuth 2.0 Demonstrating Proof of Possession (DPoP)" [RFC9449]: DPoP outlines an application-level sender-constraining for access and refresh tokens. It uses proof-of-possession based on a public/private key pair and application-level signing. DPoP can be used with public clients and, in the case of confidential clients, can be combined with any client authentication method.

Note that the security of sender-constrained tokens is undermined when an attacker gets access to the token and the key material. This is, in particular, the case for corrupted client software and cross-site scripting attacks (when the client is running in the browser). If the key material is protected in a hardware or software security module or only indirectly accessible (like in a TLS

stack), sender-constrained tokens at least protect against the use of the token when the client is offline, i.e., when the security module or interface is not available to the attacker. This applies to access tokens as well as to refresh tokens (see Section 4.14).

4.10.2. Audience-Restricted Access Tokens

Audience restriction essentially restricts access tokens to a particular resource server. The authorization server associates the access token with the particular resource server, and the resource server is then supposed to verify the intended audience. If the access token fails the intended audience validation, the resource server refuses to serve the respective request.

In general, audience restriction limits the impact of token leakage. In the case of a counterfeit resource server, it may (as described below) also prevent abuse of the phished access token at the legitimate resource server.

The audience can be expressed using logical names or physical addresses (like URLs). 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.

For this to work, the client needs to tell the authorization server the intended resource server. The mechanism in [RFC8707] can be used for this or the information can be encoded in the scope value (Section 3.3 of [RFC6749]).

Instead of the URL, it is also possible to utilize the fingerprint of the resource server's X.509 certificate as the audience value. This variant would also allow detection of an attempt to spoof the legitimate 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 may seem easier to use since it does not require any cryptography on the client side. Still, since every access token is bound to a specific resource server, the client also needs to obtain a single resource server-specific access token when accessing several resource servers. (Resource indicators, as specified in [RFC8707], can help to achieve this.) [TOKEN-BINDING] had the same property since different token-binding IDs must be associated with the access token. Using Mutual TLS for OAuth 2.0 [RFC8705], on the other hand, allows a client to use the access token at multiple resource servers.

It should 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 -- have additional benefits beyond the scope of token leakage prevention. They allow the authorization server to create a different access token whose format and content are specifically minted for the respective server. This has huge functional and privacy advantages in deployments using structured access tokens.

4.10.3. Discussion: Preventing Leakage via Metadata

An authorization server could provide the client with additional information about the locations where it is safe to use its access tokens. This approach, and why it is not recommended, is discussed in the following.

In the simplest form, this would require the authorization server to publish a list of its known resource servers, illustrated in the following example using a non-standard Authorization Server 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 authorization server could also return the URL(s) an access token is good for in the token response, illustrated by the example and non-standard 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, [research.ubc] and [research.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. However, there are alternative countermeasures, as described before, that provide a better balance between the involved parties.

4.11. Open Redirection

The following attacks can occur when an authorization server or client has an open redirector. Such endpoints are sometimes implemented, for example, to show a message before a user is then redirected to an external website, or to redirect users back to a URL they were intending to visit before being interrupted, e.g., by a login prompt.

4.11.1. Client as Open Redirector

Clients **MUST NOT** expose open redirectors. Attackers may use open redirectors to produce URLs pointing to the client and utilize them to exfiltrate authorization codes and access tokens, as described in Section 4.1.2. Another abuse case is to produce URLs that appear to point to the client. This might trick users into trusting the URL and following it in their browser. This can be abused for phishing.

In order to prevent open redirection, clients should only redirect if the target URLs are allowed or if the origin and integrity of a request can be authenticated. Countermeasures against open redirection are described by OWASP [owasp.redir].

4.11.2. Authorization Server as Open Redirector

Just as with clients, attackers could try to utilize a user's trust in the authorization server (and its URL in particular) for performing phishing attacks. OAuth authorization servers regularly redirect users to other websites (the clients), but they must do so safely.

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

However, an attacker could also utilize a correctly registered redirect URI to perform phishing attacks. The attacker could, for example, register a client via dynamic client registration [RFC7591] and execute one of the following attacks:

1. Intentionally send an erroneous authorization request, e.g., by using an invalid scope value, thus instructing the authorization server to redirect the user agent to its phishing site.
2. Intentionally send a valid authorization request with `client_id` and `redirect_uri` controlled by the attacker. After the user authenticates, the authorization server prompts the user to provide consent to the request. If the user notices an issue with the request and declines the request, the authorization server still redirects the user agent to the phishing site. In this case, the user agent will be redirected to the phishing site regardless of the action taken by the user.
3. Intentionally send a valid silent authentication request (`prompt=none`) with `client_id` and `redirect_uri` controlled by the attacker. In this case, the authorization server will automatically redirect the user agent to the phishing site.

The authorization server **MUST** take precautions to prevent these threats. The authorization server **MUST** always authenticate the user first and, with the exception of the silent authentication use case, prompt the user for credentials when needed, before redirecting the user. Based on its risk assessment, the authorization server needs to decide whether or not it can trust the redirect URI. It could take into account URI analytics done internally or through some external service to evaluate the credibility and trustworthiness of content behind the URI, and the source of the redirect URI and other client data.

The authorization server **SHOULD** only automatically redirect the user agent if it trusts the redirect URI. If the URI is not trusted, the authorization server **MAY** inform the user and rely on the user to make the correct decision.

4.12. 307 Redirect

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

In [RFC6749], the HTTP status code 302 (Found) is used for this purpose, but "any other method available via the user-agent to accomplish this redirection is allowed". When the status code 307 is used for redirection instead, the user agent will send the user's credentials via HTTP POST to the client.

This discloses the sensitive credentials to the client. If the client is malicious, it can use the credentials to impersonate the user at the authorization server.

The behavior might be unexpected for developers but is defined in Section 15.4.8 of [RFC9110]. This status code (307) does not require the user agent to rewrite the POST request to a GET request and thereby drop the form data in the POST request body.

In the HTTP standard [RFC9110], only the status code 303 unambiguously 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's credentials to the client. (In practice, however, most user agents will only show this behavior for 307 redirects.)

Authorization servers that redirect a request that potentially contains the user's 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, the authorization server **SHOULD** use HTTP status code 303 (See Other).

4.13. TLS Terminating Reverse Proxies

A common deployment architecture for HTTP applications is to hide the application server behind a reverse proxy that 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 with relevance to OAuth and gives 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. This data is usually passed in HTTP headers added to the upstream request. While the headers are often custom, application-specific headers, standardized header fields for client certificates and client certificate chains are defined in [RFC9440].

If the reverse proxy passes 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 `X-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 an allowed IP address to the header and render the protection 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 were able to get access to the internal network between the proxy and application server, the attacker could also try to circumvent security controls in place. Therefore, it is essential to ensure the authenticity of the communicating entities. Furthermore, the communication link between the reverse proxy and application server **MUST** be protected against eavesdropping, injection, and replay of messages.

4.14. Refresh Token Protection

Refresh tokens are a convenient and user-friendly way to obtain new access tokens. They 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.

4.14.1. Discussion

Refresh tokens are an attractive target for attackers since they represent the full scope of grant a resource owner delegated to a certain client and they are not further constrained to a specific resource. If an attacker is able to exfiltrate and successfully replay a refresh token, the attacker will be able to mint access tokens and use them to access resource servers on behalf of the resource owner.

[RFC6749] already provides robust baseline 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 and authentication of this client 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 behaviors.

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

4.14.2. Recommendations

Authorization servers **MUST** determine, based on a risk assessment, whether to issue refresh tokens to a certain client. If the authorization server decides not to issue refresh tokens, the client **MAY** obtain a new access token by utilizing other grant types, such as the authorization code grant type. In such a case, the authorization server may utilize cookies and persistent 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 legitimate client and reduce the impact of refresh token leakage.

For confidential clients, [RFC6749] already requires that refresh tokens can only be used by the client for which they were issued.

Authorization servers **MUST** utilize one of these methods to detect refresh token replay by malicious actors for public clients:

- **Sender-constrained refresh tokens:** the authorization server cryptographically binds the refresh token to a certain client instance, e.g., by utilizing [RFC8705] or [RFC9449].
- **Refresh token rotation:** the authorization server 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 will revoke the active refresh token. This stops the attack at the cost of forcing the legitimate client to obtain a fresh authorization grant.

  Implementation note: The grant to which a refresh token belongs may be encoded into the refresh token itself. This can enable an authorization server to efficiently determine the grant to which a refresh token belongs, and by extension, all refresh tokens that need to be revoked. Authorization servers **MUST** ensure the integrity of the refresh token value in this case, for example, using signatures.

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

- password change or
- 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).

4.15. Client Impersonating Resource Owner

Resource servers may make access control decisions based on the identity of a resource owner for which an access token was issued, or based on the identity of a client in the client credentials grant. For example, [RFC9068] (JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens) describes a data structure for access tokens containing a `sub` claim defined as follows:

> In cases of access tokens obtained through grants where a resource owner is involved, such as the authorization code grant, the value of "sub" **SHOULD** correspond to the subject identifier of the resource owner. In cases of access tokens obtained through grants where no resource owner is involved, such as the client credentials grant, the value of "sub" **SHOULD** correspond to an identifier the authorization server uses to indicate the client application.

If both options are possible, a resource server may mistake a client's identity for the identity of a resource owner. For example, if a client is able to choose its own `client_id` during registration with the authorization server, a malicious client may set it to a value identifying a resource owner (e.g., a `sub` value if OpenID Connect is used). If the resource server cannot properly distinguish between access tokens obtained with involvement of the resource owner and those without, the client may accidentally be able to access resources belonging to the resource owner.

This attack potentially affects not only implementations using [RFC9068], but also similar, bespoke solutions.

4.15.1. Countermeasures

Authorization servers **SHOULD NOT** allow clients to influence their `client_id` or any claim that could cause confusion with a genuine resource owner if a common namespace for client IDs and user identifiers exists, such as in the `sub` claim shown above. Where this cannot be avoided, authorization servers **MUST** provide other means for the resource server to distinguish between the two types of access tokens.

4.16. Clickjacking

As described in Section 4.4.1.9 of [RFC6819], the authorization request is susceptible to clickjacking attacks, also called user interface redressing. In such an attack, an attacker embeds the authorization endpoint user interface in an innocuous context. A user believing to interact with that context, for example, by clicking on buttons, inadvertently interacts with the authorization endpoint user interface instead. The opposite can be achieved as well: A user believing to interact with the authorization endpoint might inadvertently type a password into

an attacker-provided input field overlaid over the original user interface. Clickjacking attacks can be designed such that users can hardly notice the attack, for example, using almost invisible iframes overlaid on top of other elements.

An attacker can use this vector to obtain the user's authentication credentials, change the scope of access granted to the client, and potentially access the user's resources.

Authorization servers **MUST** prevent clickjacking attacks. Multiple countermeasures are described in [RFC6819], including the use of the X-Frame-Options HTTP response header field and frame-busting JavaScript. In addition to those, authorization servers **SHOULD** also use Content Security Policy (CSP) level 2 [W3C.CSP-2] or greater.

To be effective, CSP must be used on the authorization endpoint and, if applicable, other endpoints used to authenticate the user and authorize the client (e.g., the device authorization endpoint, login pages, error pages, etc.). This prevents framing by unauthorized origins in user agents that support CSP. The client **MAY** permit being framed by some other origin than the one used in its redirection endpoint. For this reason, authorization servers **SHOULD** allow administrators to configure allowed origins for particular clients and/or for clients to register these dynamically.

Using CSP allows authorization servers to specify multiple origins in a single response header field and to constrain these using flexible patterns (see [W3C.CSP-2] for details). Level 2 of CSP provides a robust mechanism for protecting against clickjacking by using policies that restrict the origin of frames (by using `frame-ancestors`) together with those that restrict the sources of scripts allowed to execute on an HTML page (by using `script-src`). A non-normative example of such a policy is shown in the following listing:

```
HTTP/1.1 200 OK
Content-Security-Policy: frame-ancestors https://ext.example.org:8000
Content-Security-Policy: script-src 'self'
X-Frame-Options: ALLOW-FROM https://ext.example.org:8000
...
```

Because some user agents do not support [W3C.CSP-2], this technique **SHOULD** be combined with others, including those described in [RFC6819], unless such legacy user agents are explicitly unsupported by the authorization server. Even in such cases, additional countermeasures **SHOULD** still be employed.

4.17. Attacks on In-Browser Communication Flows

If the authorization response is sent with in-browser communication techniques like postMessage [WHATWG.postmessage_api] instead of HTTP redirects, messages may inadvertently be sent to malicious origins or injected from malicious origins.

4.17.1. Examples

The following non-normative pseudocode examples of attacks using in-browser communication are described in [research.rub].

4.17.1.1. Insufficient Limitation of Receiver Origins

When sending the authorization response or token response via postMessage, the authorization server sends the response to the wildcard origin "*" instead of the client's origin. When the window to which the response is sent is controlled by an attacker, the attacker can read the response.

```
window.opener.postMessage(
  {
    code: "ABC",
    state: "123"
  },
  "*" // any website in the opener window can receive the message
)
```

4.17.1.2. Insufficient URI Validation

When sending the authorization response or token response via postMessage, the authorization server may not check the receiver origin against the redirect URI and instead, for example, may send the response to an origin provided by an attacker. This is analogous to the attack described in Section 4.1.

```
window.opener.postMessage(
  {
    code: "ABC",
    state: "123"
  },
  "https://attacker.example" // attacker-provided value
)
```

4.17.1.3. Injection after Insufficient Validation of Sender Origin

A client that expects the authorization response or token response via postMessage may not validate the sender origin of the message. This may allow an attacker to inject an authorization response or token response into the client.

In the case of a maliciously injected authorization response, the attack is a variant of the CSRF attacks described in Section 4.7. The countermeasures described in Section 4.7 apply to this attack as well.

In the case of a maliciously injected token response, sender-constrained access tokens as described in Section 4.10.1 may prevent the attack under some circumstances, but additional countermeasures as described in Section 4.17.2 are generally required.

4.17.2. Recommendations

When comparing client receiver origins against pre-registered origins, authorization servers **MUST** utilize exact string matching as described in Section 4.1.3. Authorization servers **MUST** send postMessages to trusted client receiver origins, as shown in the following, non-normative example:

```
window.opener.postMessage(
  {
    code: "ABC",
    state: "123"
  },
  "https://client.example" // use explicit client origin
)
```

Wildcard origins like "*" in postMessage **MUST NOT** be used, as attackers can use them to leak a victim's in-browser message to malicious origins. Both measures contribute to the prevention of leakage of authorization codes and access tokens (see Section 4.1).

Clients **MUST** prevent injection of in-browser messages on the client receiver endpoint. Clients **MUST** utilize exact string matching to compare the initiator origin of an in-browser message with the authorization server origin, as shown in the following, non-normative example:

```
window.addEventListener("message", (e) => {
  // validate exact authorization server origin
  if (e.origin === "https://honest.as.example") {
    // process e.data.code and e.data.state
  }
})
```

Since in-browser communication flows only apply a different communication technique (i.e., postMessage instead of HTTP redirect), all measures protecting the authorization response listed in Section 2.1 **MUST** be applied equally.

5. IANA Considerations

This document has no IANA actions.

6. Security Considerations

Security considerations are described in Sections 2, 3, and 4.

7. References

7.1. Normative References

[BCP195] Best Current Practice 195, <https://www.rfc-editor.org/info/bcp195>.
At the time of writing, this BCP comprises the following:

Moriarty, K. and S. Farrell, "Deprecating TLS 1.0 and TLS 1.1", BCP 195, RFC 8996, DOI 10.17487/RFC8996, March 2021, <https://www.rfc-editor.org/info/rfc8996>.

Sheffer, Y., Saint-Andre, P., and T. Fossati, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 9325, DOI 10.17487/RFC9325, November 2022, <https://www.rfc-editor.org/info/rfc9325>.

[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>.

[RFC7521] Campbell, B., Mortimore, C., Jones, M., and Y. Goland, "Assertion Framework for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7521, DOI 10.17487/RFC7521, May 2015, <https://www.rfc-editor.org/info/rfc7521>.

[RFC7523] Jones, M., Campbell, B., and C. Mortimore, "JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants", RFC 7523, DOI 10.17487/RFC7523, May 2015, <https://www.rfc-editor.org/info/rfc7523>.

[RFC8252] Denniss, W. and J. Bradley, "OAuth 2.0 for Native Apps", BCP 212, RFC 8252, DOI 10.17487/RFC8252, October 2017, <https://www.rfc-editor.org/info/rfc8252>.

[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>.

[RFC8705] Campbell, B., Bradley, J., Sakimura, N., and T. Lodderstedt, "OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens", RFC 8705, DOI 10.17487/RFC8705, February 2020, <https://www.rfc-editor.org/info/rfc8705>.

[RFC9068] Bertocci, V., "JSON Web Token (JWT) Profile for OAuth 2.0 Access Tokens", RFC 9068, DOI 10.17487/RFC9068, October 2021, <https://www.rfc-editor.org/info/rfc9068>.

7.2. Informative References

[JWT-ENCODED-STATE] Bradley, J., Lodderstedt, T., and H. Zandbelt, "Encoding claims in the OAuth 2 state parameter using a JWT", Work in Progress, Internet-Draft, draft-bradley-oauth-jwt-encoded-state-09, 4 November 2018, <https://datatracker.ietf.org/doc/html/draft-bradley-oauth-jwt-encoded-state-09>.

[TOKEN-BINDING] Jones, M., Campbell, B., Bradley, J., and W. Denniss, "OAuth 2.0 Token Binding", Work in Progress, Internet-Draft, draft-ietf-oauth-token-binding-08, 19 October 2018, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-token-binding-08>.

[OAUTH-V2.1] Hardt, D., Parecki, A., and T. Lodderstedt, "The OAuth 2.1 Authorization Framework", Work in Progress, Internet-Draft, draft-ietf-oauth-v2-1-11, 14 May 2024, <https://datatracker.ietf.org/doc/html/draft-ietf-oauth-v2-1-11>.

[OAuth.Post] Jones, M. and B. Campbell, "OAuth 2.0 Form Post Response Mode", The OpenID Foundation, 27 April 2015, <https://openid.net/specs/oauth-v2-form-post-response-mode-1_0.html>.

[OAuth.Responses] de Medeiros, B., Ed., Scurtescu, M., Tarjan, P., and M. Jones, "OAuth 2.0 Multiple Response Type Encoding Practices", The OpenID Foundation, 25 February 2014, <https://openid.net/specs/oauth-v2-multiple-response-types-1_0.html>.

[OpenID.Core] Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and C. Mortimore, "OpenID Connect Core 1.0 incorporating errata set 2", The OpenID Foundation, 15 December 2023, <https://openid.net/specs/openid-connect-core-1_0.html>.

[OpenID.Discovery] Sakimura, N., Bradley, J., Jones, M., and E. Jay, "OpenID Connect Discovery 1.0 incorporating errata set 2", The OpenID Foundation, 15 December 2023, <https://openid.net/specs/openid-connect-discovery-1_0.html>.

[OpenID.JARM] Lodderstedt, T. and B. Campbell, "Financial-grade API: JWT Secured Authorization Response Mode for OAuth 2.0 (JARM)", The OpenID Foundation, 17 October 2018, <https://openid.net/specs/openid-financial-api-jarm.html>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[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>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8707] Campbell, B., Bradley, J., and H. Tschofenig, "Resource Indicators for OAuth 2.0", RFC 8707, DOI 10.17487/RFC8707, February 2020, <https://www.rfc-editor.org/info/rfc8707>.

[RFC9101] Sakimura, N., Bradley, J., and M. Jones, "The OAuth 2.0 Authorization Framework: JWT-Secured Authorization Request (JAR)", RFC 9101, DOI 10.17487/RFC9101, August 2021, <https://www.rfc-editor.org/info/rfc9101>.

[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/info/rfc9110>.

[RFC9126] Lodderstedt, T., Campbell, B., Sakimura, N., Tonge, D., and F. Skokan, "OAuth 2.0 Pushed Authorization Requests", RFC 9126, DOI 10.17487/RFC9126, September 2021, <https://www.rfc-editor.org/info/rfc9126>.

[RFC9207] Meyer zu Selhausen, K. and D. Fett, "OAuth 2.0 Authorization Server Issuer Identification", RFC 9207, DOI 10.17487/RFC9207, March 2022, <https://www.rfc-editor.org/info/rfc9207>.

[RFC9396] Lodderstedt, T., Richer, J., and B. Campbell, "OAuth 2.0 Rich Authorization Requests", RFC 9396, DOI 10.17487/RFC9396, May 2023, <https://www.rfc-editor.org/info/rfc9396>.

[RFC9440] Campbell, B. and M. Bishop, "Client-Cert HTTP Header Field", RFC 9440, DOI 10.17487/RFC9440, July 2023, <https://www.rfc-editor.org/info/rfc9440>.

[RFC9449] Fett, D., Campbell, B., Bradley, J., Lodderstedt, T., Jones, M., and D. Waite, "OAuth 2.0 Demonstrating Proof of Possession (DPoP)", RFC 9449, DOI 10.17487/RFC9449, September 2023, <https://www.rfc-editor.org/info/rfc9449>.

[W3C.CSP-2] West, M., Barth, A., and D. Veditz, "Content Security Policy Level 2", W3C Recommendation, December 2016, <https://www.w3.org/TR/2016/REC-CSP2-20161215/>. Latest version available at <https://www.w3.org/TR/CSP2/>.

[W3C.WebAuthn] Hodges, J., Jones, J.C., Jones, M.B., Kumar, A., and E. Lundberg, "Web Authentication: An API for accessing Public Key Credentials Level 2", W3C Recommendation, 8 April 2021, <https://www.w3.org/TR/2021/REC-webauthn-2-20210408/>. Latest version available at <https://www.w3.org/TR/webauthn-2/>.

[W3C.WebCrypto] Watson, M., Ed., "Web Cryptography API", W3C Recommendation, 26 January 2017, <https://www.w3.org/TR/2017/REC-WebCryptoAPI-20170126/>. Latest version available at <https://www.w3.org/TR/WebCryptoAPI/>.

[W3C.webappsec-referrer-policy] Eisinger, J. and E. Stark, "Referrer Policy", 26 January 2017, <https://www.w3.org/TR/2017/CR-referrer-policy-20170126/>. Latest version available at <https://www.w3.org/TR/referrer-policy/>.

[WHATWG.CORS] WHATWG, "CORS protocol", Fetch: Living Standard, Section 3.2, 17 June 2024, <https://fetch.spec.whatwg.org/#http-cors-protocol>.

[WHATWG.postmessage_api] WHATWG, "Cross-document messaging", HTML: Living Standard, Section 9.3, 19 August 2024, <https://html.spec.whatwg.org/multipage/web-messaging.html#web-messaging>.

[arXiv.1508.04324v2] Mladenov, V., Mainka, C., and J. Schwenk, "On the security of modern Single Sign-On Protocols: Second-Order Vulnerabilities in OpenID Connect", arXiv:1508.04324v2, DOI 10.48550/arXiv.1508.04324, 7 January 2016, <https://arxiv.org/abs/1508.04324v2/>.

[arXiv.1601.01229] Fett, D., Küsters, R., and G. Schmitz, "A Comprehensive Formal Security Analysis of OAuth 2.0", arXiv:1601.01229, DOI 10.48550/arXiv.1601.01229, 6 January 2016, <https://arxiv.org/abs/1601.01229/>.

[arXiv.1704.08539] Fett, D., Küsters, R., and G. Schmitz, "The Web SSO Standard OpenID Connect: In-Depth Formal Security Analysis and Security Guidelines", arXiv:1704.08539, DOI 10.48550/arXiv.1704.08539, 27 April 2017, <https://arxiv.org/abs/1704.08539/>.

[arXiv.1901.11520] Fett, D., Hosseyni, P., and R. Küsters, "An Extensive Formal Security Analysis of the OpenID Financial-grade API", arXiv:1901.11520, DOI 10.48550/arXiv.1901.11520, 31 January 2019, <https://arxiv.org/abs/1901.11520/>.

[bug.chromium] "Referer header includes URL fragment when opening link using New Tab", Chromium Issue Tracker, Issue ID: 40076763, <https://issues.chromium.org/issues/40076763>.

[owasp.redir] OWASP Foundation, "Unvalidated Redirects and Forwards Cheat Sheet", OWASP Cheat Sheet Series, <https://cheatsheetseries.owasp.org/cheatsheets/Unvalidated_Redirects_and_Forwards_Cheat_Sheet.html>.

[research.cmu] Chen, E., Pei, Y., Chen, S., Tian, Y., Kotcher, R., and P. Tague, "OAuth Demystified for Mobile Application Developers", November 2014, <https://css.csail.mit.edu/6.858/2012/readings/oauth-sso.pdf>.

[research.jcs_14] Bansal, C., Bhargavan, K., Delignat-Lavaud, A., and S. Maffeis, "Discovering concrete attacks on website authorization by formal analysis", Journal of Computer Security, vol. 22, no. 4, pp. 601-657, DOI 10.3233/JCS-140503, 23 April 2014, <https://www.doc.ic.ac.uk/~maffeis/papers/jcs14.pdf>.

[research.rub] Jannett, L., Mladenov, V., Mainka, C., and J. Schwenk, "DISTINCT: Identity Theft using In-Browser Communications in Dual-Window Single Sign-On", CCS '22: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, DOI 10.1145/3548606.3560692, 7 November 2022, <https://dl.acm.org/doi/pdf/10.1145/3548606.3560692>.

[research.rub2] Fries, C., "Security Analysis of Real-Life OpenID Connect Implementations", Master's thesis, Ruhr-Universität Bochum (RUB), 20 December 2020, <https://www.nds.rub.de/media/ei/arbeiten/2021/05/03/masterthesis.pdf>.

[research.ubc] Sun, S.-T. and K. Beznosov, "The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems", Proceedings of the 2012 ACM conference on Computer and communications security (CCS '12), pp. 378-390, DOI 10.1145/2382196.2382238, October 2012, <https://css.csail.mit.edu/6.858/2012/readings/oauth-sso.pdf>.

[research.udel] Liu, D., Hao, S., and H. Wang, "All Your DNS Records Point to Us: Understanding the Security Threats of Dangling DNS Records", CCS '16: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 1414-1425, DOI 10.1145/2976749.2978387, 24 October 2016, <https://dl.acm.org/doi/pdf/10.1145/2976749.2978387>.

Appendix A. Acknowledgements

We would like to thank Brock Allen, Annabelle Richard Backman, Dominick Baier, Vittorio Bertocci, Brian Campbell, Bruno Crispo, William Dennis, George Fletcher, Matteo Golinelli, Dick Hardt, Joseph Heenan, Pedram Hosseyni, Phil Hunt, Tommaso Innocenti, Louis Jannett, Jared Jennings, Michael B. Jones, Engin Kirda, Konstantin Lapine, Neil Madden, Christian Mainka, Jim Manico, Nov Matake, Doug McDorman, Karsten Meyer zu Selhausen, Ali Mirheidari, Vladislav Mladenov, Kaan Onarioglu, Aaron Parecki, Michael Peck, Johan Peeters, Nat Sakimura, Guido Schmitz, Jörg Schwenk, Rifaat Shekh-Yusef, Travis Spencer, Petteri Stenius, Tomek Stojecki, David Waite, Tim Würtele, and Hans Zandbelt for their valuable feedback.

Authors' Addresses

Torsten Lodderstedt
SPRIND
Email: torsten@lodderstedt.net

John Bradley
Yubico
Email: ve7jtb@ve7jtb.com

Andrey Labunets
Independent Researcher
Email: isciurus@gmail.com

Daniel Fett
Authlete
Email: mail@danielfett.de