Web Authorization Protocol T. Lodderstedt

Internet-Draft yes.com

Intended status: Best Current Practice J. Bradley

Expires: 7 December 2023 Yubico

A. Labunets

Independent Researcher

D. Fett

Authlete

5 June 2023

OAuth 2.0 Security Best Current Practice

draft-ietf-oauth-security-topics-23

Abstract

This document describes best current security practice for OAuth 2.0.

It updates and extends the OAuth 2.0 Security Threat Model to

incorporate practical experiences gathered since OAuth 2.0 was

published and covers new threats relevant due to the broader

application of OAuth 2.0.

Discussion Venues

This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the Web Authorization

Protocol Working Group mailing list (oauth@ietf.org), which is

archived at https://mailarchive.ietf.org/arch/browse/oauth/.

Source for this draft and an issue tracker can be found at

https://github.com/oauthstuff/draft-ietf-oauth-security-topics.

Status of This Memo

This Internet-Draft is submitted in full conformance with the

provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering

Task Force (IETF). Note that other groups may also distribute

working documents as Internet-Drafts. The list of current Internet-

Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months

and may be updated, replaced, or obsoleted by other documents at any

time. It is inappropriate to use Internet-Drafts as reference

material or to cite them other than as "work in progress."

Lodderstedt, et al. Expires 7 December 2023 [Page 1]

Internet-Draft oauth-security-topics June 2023

This Internet-Draft will expire on 7 December 2023.

Copyright Notice

Copyright (c) 2023 IETF Trust and the persons identified as the

document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal

Provisions Relating to IETF Documents (https://trustee.ietf.org/

license-info) in effect on the date of publication of this document.

Please review these documents carefully, as they describe your rights

and restrictions with respect to this document. Code Components

extracted from this document must include Revised BSD License text as

described in Section 4.e of the Trust Legal Provisions and are

provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction . . . . . . . . . . . . . . . . . . . . . . . . 4

1.1. Structure . . . . . . . . . . . . . . . . . . . . . . . . 5

1.2. Conventions and Terminology . . . . . . . . . . . . . . . 5

2. Best Practices . . . . . . . . . . . . . . . . . . . . . . . 5

2.1. Protecting Redirect-Based Flows . . . . . . . . . . . . . 6

2.1.1. Authorization Code Grant . . . . . . . . . . . . . . 7

2.1.2. Implicit Grant . . . . . . . . . . . . . . . . . . . 8

2.2. Token Replay Prevention . . . . . . . . . . . . . . . . . 8

2.2.1. Access Tokens . . . . . . . . . . . . . . . . . . . . 8

2.2.2. Refresh Tokens . . . . . . . . . . . . . . . . . . . 9

2.3. Access Token Privilege Restriction . . . . . . . . . . . 9

2.4. Resource Owner Password Credentials Grant . . . . . . . . 10

2.5. Client Authentication . . . . . . . . . . . . . . . . . . 10

2.6. Other Recommendations . . . . . . . . . . . . . . . . . . 10

3. The Updated OAuth 2.0 Attacker Model . . . . . . . . . . . . 11

4. Attacks and Mitigations . . . . . . . . . . . . . . . . . . . 13

4.1. Insufficient Redirect URI Validation . . . . . . . . . . 13

4.1.1. Redirect URI Validation Attacks on Authorization Code

Grant . . . . . . . . . . . . . . . . . . . . . . . . 14

4.1.2. Redirect URI Validation Attacks on Implicit Grant . . 15

4.1.3. Countermeasures . . . . . . . . . . . . . . . . . . . 17

4.2. Credential Leakage via Referer Headers . . . . . . . . . 18

4.2.1. Leakage from the OAuth Client . . . . . . . . . . . . 18

4.2.2. Leakage from the Authorization Server . . . . . . . . 18

4.2.3. Consequences . . . . . . . . . . . . . . . . . . . . 18

4.2.4. Countermeasures . . . . . . . . . . . . . . . . . . . 18

4.3. Credential Leakage via Browser History . . . . . . . . . 19

4.3.1. Authorization Code in Browser History . . . . . . . . 20

4.3.2. Access Token in Browser History . . . . . . . . . . . 20

4.4. Mix-Up Attacks . . . . . . . . . . . . . . . . . . . . . 20

Lodderstedt, et al. Expires 7 December 2023 [Page 2]

Internet-Draft oauth-security-topics June 2023

4.4.1. Attack Description . . . . . . . . . . . . . . . . . 21

4.4.2. Countermeasures . . . . . . . . . . . . . . . . . . . 23

4.5. Authorization Code Injection . . . . . . . . . . . . . . 24

4.5.1. Attack Description . . . . . . . . . . . . . . . . . 25

4.5.2. Discussion . . . . . . . . . . . . . . . . . . . . . 26

4.5.3. Countermeasures . . . . . . . . . . . . . . . . . . . 27

4.5.4. Limitations . . . . . . . . . . . . . . . . . . . . . 29

4.6. Access Token Injection . . . . . . . . . . . . . . . . . 29

4.6.1. Countermeasures . . . . . . . . . . . . . . . . . . . 29

4.7. Cross Site Request Forgery . . . . . . . . . . . . . . . 30

4.7.1. Countermeasures . . . . . . . . . . . . . . . . . . . 30

4.8. PKCE Downgrade Attack . . . . . . . . . . . . . . . . . . 31

4.8.1. Attack Description . . . . . . . . . . . . . . . . . 31

4.8.2. Countermeasures . . . . . . . . . . . . . . . . . . . 32

4.9. Access Token Leakage at the Resource Server . . . . . . . 33

4.9.1. Access Token Phishing by Counterfeit Resource

Server . . . . . . . . . . . . . . . . . . . . . . . 33

4.9.2. Compromised Resource Server . . . . . . . . . . . . . 33

4.10. Misuse of Stolen Access Tokens . . . . . . . . . . . . . 34

4.10.1. Sender-Constrained Access Tokens . . . . . . . . . . 34

4.10.2. Audience Restricted Access Tokens . . . . . . . . . 37

4.10.3. Discussion: Preventing Leakage via Metadata . . . . 38

4.11. Open Redirection . . . . . . . . . . . . . . . . . . . . 39

4.11.1. Client as Open Redirector . . . . . . . . . . . . . 39

4.11.2. Authorization Server as Open Redirector . . . . . . 40

4.12. 307 Redirect . . . . . . . . . . . . . . . . . . . . . . 41

4.13. TLS Terminating Reverse Proxies . . . . . . . . . . . . . 42

4.14. Refresh Token Protection . . . . . . . . . . . . . . . . 43

4.14.1. Discussion . . . . . . . . . . . . . . . . . . . . . 43

4.14.2. Recommendations . . . . . . . . . . . . . . . . . . 43

4.15. Client Impersonating Resource Owner . . . . . . . . . . . 45

4.15.1. Countermeasures . . . . . . . . . . . . . . . . . . 45

4.16. Clickjacking . . . . . . . . . . . . . . . . . . . . . . 45

4.17. Authorization Server Redirecting to Phishing Site . . . . 46

4.18. Attacks on In-Browser Communication Flows . . . . . . . . 47

4.18.1. Examples . . . . . . . . . . . . . . . . . . . . . . 47

4.18.2. Recommendations . . . . . . . . . . . . . . . . . . 49

5. Acknowledgements . . . . . . . . . . . . . . . . . . . . . . 49

6. IANA Considerations . . . . . . . . . . . . . . . . . . . . . 50

7. Security Considerations . . . . . . . . . . . . . . . . . . . 50

8. Normative References . . . . . . . . . . . . . . . . . . . . 50

9. Informative References . . . . . . . . . . . . . . . . . . . 51

Appendix A. Document History . . . . . . . . . . . . . . . . . . 56

Authors' Addresses . . . . . . . . . . . . . . . . . . . . . . . 61

Lodderstedt, et al. Expires 7 December 2023 [Page 3]

Internet-Draft oauth-security-topics June 2023

1. Introduction

Since its publication in [RFC6749] and [RFC6750], OAuth 2.0 ("OAuth"

in the following) has gotten 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. 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 client,

authorization server, and resource servers. The URLs of the AS

and RS 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 talks to a legitimate server was

based on TLS server authentication (see [RFC6819], Section 4.5.4).

With the increasing adoption of OAuth, this simple model dissolved

and, in several scenarios, was replaced by a dynamic establishment

of the relationship between clients on one side and the

authorization and resource servers of a particular deployment on

the other side. This way, the same client could be used to access

services of different providers (in case of standard APIs, such as

e-mail or OpenID Connect) or 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.

Lodderstedt, et al. Expires 7 December 2023 [Page 4]

Internet-Draft oauth-security-topics June 2023

This document provides updated security recommendations to address

these challenges. It does not supplant the security advice given in

[RFC6749], [RFC6750], and [RFC6819], but complements those documents.

This document 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. 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 when

feasible.

1.1. Structure

The remainder of this document is organized as follows: The next

section summarizes the most important best practices for every OAuth

implementor. Afterwards, the updated the OAuth attacker model is

presented. Subsequently, a detailed analysis of the threats and

implementation issues that can be found in the wild today is given

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].

2. Best Practices

This section describes the set of security mechanisms and measures

the OAuth working group considers best practices at the time of

writing.

Lodderstedt, et al. Expires 7 December 2023 [Page 5]

Internet-Draft oauth-security-topics June 2023

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 AS MUST NOT expose URLs that forward the user's browser

to arbitrary URIs obtained from a query parameter ("open redirector")

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 a malicious third

party (see Section 4.4.1.8. of [RFC6819] for details). Clients that

have ensured that the authorization server supports 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 [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 AS that redirects a request potentially containing user

credentials MUST avoid forwarding these user credentials accidentally

(see Section 4.12 for details).

Lodderstedt, et al. Expires 7 December 2023 [Page 6]

Internet-Draft oauth-security-topics June 2023

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 a strong protection against misuse and

injection of authorization codes as described in Section 4.5.3.1

and, as a side-effect, prevents CSRF even in 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.

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 [RFC7636] 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 AS to publish the element

code\_challenge\_methods\_supported in their AS metadata ([RFC8414])

containing the supported PKCE challenge methods (which can be used by

Lodderstedt, et al. Expires 7 December 2023 [Page 7]

Internet-Draft oauth-security-topics June 2023

the client to detect PKCE support). ASs MAY instead provide a

deployment-specific way to ensure or determine PKCE support by the

AS.

2.1.2. Implicit Grant

The implicit grant (response type "token") and other response types

causing the authorization server to issue access tokens in the

authorization response are vulnerable to access token leakage and

access token replay as described in Section 4.1, Section 4.2,

Section 4.3, and Section 4.6.

Moreover, no viable 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 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" (aka

authorization code grant type) as specified in Section 2.1.1 or any

other response type that causes the authorization server to issue

access tokens in the token response, 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 next

section).

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 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 Demonstration of Proof of Possession (DPoP)

[I-D.ietf-oauth-dpop] (see Section 4.10.1), to prevent misuse of

stolen and leaked access tokens.

Lodderstedt, et al. Expires 7 December 2023 [Page 8]

Internet-Draft oauth-security-topics June 2023

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 restricted to certain resource

servers (audience restriction), preferably to a single resource

server. To put this into effect, the authorization server associates

the access token with certain resource servers and every resource

server is obliged to verify, for every request, whether the access

token sent with that request was meant to be used for that particular

resource server. If not, the resource server MUST refuse to serve

the respective request. 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.

Lodderstedt, et al. Expires 7 December 2023 [Page 9]

Internet-Draft oauth-security-topics June 2023

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 AS) and users are trained to enter their

credentials in places other than the AS.

Furthermore, adapting the resource owner password credentials grant

to two-factor authentication, authentication with cryptographic

credentials (cf. WebCrypto [WebCrypto], WebAuthn [WebAuthn]), and

authentication processes that require multiple steps can be hard or

impossible.

2.5. Client Authentication

Authorization servers SHOULD use client authentication if possible.

It is RECOMMENDED to use asymmetric (public-key based) methods for

client authentication such as mTLS [RFC8705] or using signed JWTs

("Private Key JWT") in accordance with [RFC7521] and [RFC7523] (in

[OpenID.Core] defined as the client authentication method

private\_key\_jwt). When such methods for client authentication are

used, authorization servers do not need to store sensitive symmetric

keys, making these methods more robust against a number of attacks.

2.6. Other Recommendations

The use of OAuth 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 ASs publish OAuth metadata according

to [RFC8414] and that clients make use of this metadata to configure

themselves when available.

Lodderstedt, et al. Expires 7 December 2023 [Page 10]

Internet-Draft oauth-security-topics June 2023

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. 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, AS MUST NOT allow redirect URIs

that use the http scheme except for native clients that use Loopback

Interface Redirection as described in [RFC8252], Section 7.3.

If the authorization response is sent with in-browser communication

techniques like postMessage [postmessage\_api] instead of HTTP

redirects, both the initiator and receiver of the in-browser message

MUST be strictly verified as described in Section 4.18.

To support browser-based clients, endpoints directly accessed by such

clients including the Token Endpoint, Authorization Server Metadata

Endpoint, jwks\_uri Endpoint, and the Dynamic Client Registration

Endpoint MAY support the use of Cross-Origin Resource Sharing (CORS,

[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], an attacker model is laid out that describes the

capabilities of attackers against which OAuth deployments must be

protected. In the following, this attacker model is updated 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.

OAuth MUST ensure that the authorization of the resource owner (RO)

(with a user agent) at the authorization server (AS) and the

subsequent usage of the access token at the resource server (RS) is

protected at least against the following attackers:

\* (A1) Web Attackers that can set up and operate an arbitrary number

of network endpoints including browsers and servers (except for

the concrete RO, AS, and RS). Web attackers may set up web sites

that are visited by the RO, operate their own user agents, and

participate in the protocol.

Lodderstedt, et al. Expires 7 December 2023 [Page 11]

Internet-Draft oauth-security-topics June 2023

Web attackers may, in particular, operate OAuth clients that are

registered at AS, and operate their own authorization and resource

servers that can be used (in parallel) by the RO and other

resource owners.

It must also be assumed that web attackers can lure the user to

open 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 AS).

\* (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 (wifi) network using

ARP spoofing, or a state-sponsored attacker with access to internet

exchange points, for instance.

These attackers 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 in which their OAuth

implementations are expected to run. Previous attacks on OAuth have

shown that OAuth deployments SHOULD in particular consider the

following, stronger attackers in addition to those listed above:

\* (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 for 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

Lodderstedt, et al. Expires 7 December 2023 [Page 12]

Internet-Draft oauth-security-topics June 2023

register themselves on the same URI), mix-up attacks (see

Section 4.4), where the client is tricked into sending credentials

to a attacker-controlled AS, 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 AS. For

example, a resource server can be compromised by an attacker, an

access token may be sent to an attacker-controlled resource server

due to a misconfiguration, or an RO is social-engineered into

using a attacker-controlled RS. See also 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 in this attacker model, an attacker (see A1) can be a RO or

act as one. For example, an attacker can use his own browser to

replay tokens or authorization codes obtained by any of the attacks

described above at the client or RS.

This document focusses on threats resulting from these attackers.

Attacks in an even stronger attacker model are discussed, for

example, in [arXiv.1901.11520].

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.

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

Lodderstedt, et al. Expires 7 December 2023 [Page 13]

Internet-Draft oauth-security-topics June 2023

implementation or concrete configurations have been observed in the

wild . 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 attackers

control, or

\* by exposing the OAuth credentials to an attacker by utilizing an

open redirector at the client in conjunction with the way user

agents handle URL fragments.

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. A naive implementation on

the authorization server, however, 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:

First, the attacker needs to trick the user into opening a tampered

URL in his 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

Lodderstedt, et al. Expires 7 December 2023 [Page 14]

Internet-Draft oauth-security-topics June 2023

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 impersonated 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. The attacker can, however, use the legitimate

confidential client to redeem the code by performing an authorization

code injection attack, see Section 4.5.

Note: Vulnerabilities of this kind 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, he can impersonate the legitimate

client. This could be caused, for example, by a subdomain takeover

attack [subdomaintakeover], where an outdated CNAME record (say,

external-service.somesite.example) points to an external DNS name

that does no longer exist (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 a URI

under his control, he will directly get access to the fragment

carrying the access token.

Additionally, implicit clients can be subject to a further kind of

attack. It utilizes the fact that user agents re-attach fragments to

the destination URL of a redirect if the location header does not

contain a fragment (see [RFC7231], Section 9.5). The attack

described here combines this behavior with the client as an open

redirector (see Section 4.11.1) in order to get access to access

tokens. This allows circumvention even of very narrow redirect URI

patterns, but not strict URL matching.

Lodderstedt, et al. Expires 7 December 2023 [Page 15]

Internet-Draft oauth-security-topics June 2023

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:

First, and as above, the attacker needs to trick the user into

opening a tampered URL in his 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

Now, 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/

Lodderstedt, et al. Expires 7 December 2023 [Page 16]

Internet-Draft oauth-security-topics June 2023

Since the redirector at client.somesite.example does not include a

fragment in the Location header, the user agent will re-attach the

original fragment #access\_token=2YotnFZFEjr1zCsicMWpAA&amp;... to the

URL and will navigate to the following URL:

https://attacker.example/#access\_token=2YotnFZFEjr1z...

The attacker's page at attacker.example can now 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 to simplify 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 [RFC3986],

Section 6.2.1, Simple String Comparison, for details. The only

exception are native apps using a localhost URI: In this case, the AS

MUST allow variable port numbers as described in [RFC8252],

Section 7.3.

Additional recommendations:

\* Servers on which callbacks 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 causing access token issuance at the authorization

endpoint. This offers countermeasures against reuse of leaked

credentials through the exchange process with the authorization

server and token replay through 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.

Lodderstedt, et al. Expires 7 December 2023 [Page 17]

Internet-Draft oauth-security-topics June 2023

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 [RFC7231], Section 5.5.2), by leaking

either from the AS's or the client's web site, respectively. Most

importantly, authorization codes or state values can be disclosed in

this way. Although specified otherwise in [RFC7231], Section 5.5.2,

the same may happen to access tokens conveyed in URI fragments due to

browser implementation issues, as illustrated by Chromium Issue

168213 [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 Section 4.1.1,

Section 4.5, and Section 4.6. If the attacker learns state, the CSRF

protection achieved by using state is lost, resulting in CSRF attacks

as described in [RFC6819], Section 4.4.1.8.

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.

Lodderstedt, et al. Expires 7 December 2023 [Page 18]

Internet-Draft oauth-security-topics June 2023

The following measures further reduce the chances of a successful

attack:

\* Suppress the Referer header by applying an appropriate Referrer

Policy [webappsec-referrer-policy] to the document (either as part

of the "referrer" meta attribute or by setting a Referrer-Policy

header). 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 [RFC6749], Section 4.1.2, authorization codes MUST

be invalidated by the AS after their first use at the token

endpoint. For example, if an AS invalidated the code after the

legitimate client redeemed it, the attacker would fail exchanging

this code later.

This does not mitigate the attack if the attacker manages to

exchange the code for a token before the legitimate client does

so. Therefore, [RFC6749] further recommends that, when an attempt

is made to redeem a code twice, the AS 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 web site, the state was already used, invalidated by

the client and cannot be used again by the attacker. (This does

not help if the state leaks from the AS's web site, 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.

Lodderstedt, et al. Expires 7 December 2023 [Page 19]

Internet-Draft oauth-security-topics June 2023

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 [RFC6819],

Section 4.4.1.1, and Section 4.5.

\* Use 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

web site 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 to transfer tokens via a

header, but in practice web sites often pass access tokens in query

parameters.

In case of the 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 is an attack on 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 his own authorization server or if an authorization

server becomes compromised.

Lodderstedt, et al. Expires 7 December 2023 [Page 20]

Internet-Draft oauth-security-topics June 2023

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 are used with multiple AS

of which one is considered "honest" (H-AS) and one is operated by

the attacker (A-AS), and

\* the client stores the AS chosen by the user in a session bound to

the user's browser and uses the same redirection endpoint URI for

each AS.

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

only include parameters relevant for 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 at 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

Lodderstedt, et al. Expires 7 December 2023 [Page 21]

Internet-Draft oauth-security-topics June 2023

4. The user authorizes the client to access her resources at H-AS.

(Note that a vigilant user might at this point detect that she

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 AS and is then redirected by the client to that

AS), as in Attacker A2 (see Section 3). This capability can, for

example, be the result of a man-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

Steps 2 and following above.

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

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

above.

\* \*Per-AS Redirect URIs\*: If clients use different redirect URIs for

different ASs, do not store the selected AS in the user's session,

and ASs do not check the redirect URIs properly, attackers can

mount an attack called "Cross-Social Network Request Forgery".

These attacks have been observed in practice. Refer to

[oauth\_security\_jcs\_14] for details.

\* \*OpenID Connect\*: There are variants that 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 [arXiv.1704.08539],

Appendix A, and [arXiv.1508.04324v2], Section 6 ("Malicious

Endpoints Attacks").

Lodderstedt, et al. Expires 7 December 2023 [Page 22]

Internet-Draft oauth-security-topics June 2023

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 in

the following.

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 metadata [RFC8414] nor OpenID Connect Discovery

[OpenID.Discovery] are 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.

Note: Just storing the authorization server URL is not sufficient to

identify mix-up attacks. An attacker might declare an uncompromised

AS's authorization endpoint URL as "his" AS URL, but declare a token

endpoint under his own control.

4.4.2.1. Mix-Up Defense via Issuer Identification

This defense requires that the authorization server sends his 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].

Lodderstedt, et al. Expires 7 December 2023 [Page 23]

Internet-Draft oauth-security-topics June 2023

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"

AS using the redirect URI that the client assigned to the attacker's

AS. The attacker could then run the attack as described above,

replacing the client ID with the client ID of his newly created

client.

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

available.

4.5. Authorization Code Injection

An attacker that 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.

Lodderstedt, et al. Expires 7 December 2023 [Page 24]

Internet-Draft oauth-security-topics June 2023

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 his

victim in a certain app or on a certain web site.

\* 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 was 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 own 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]).

Lodderstedt, et al. Expires 7 December 2023 [Page 25]

Internet-Draft oauth-security-topics June 2023

6. All checks succeed and the authorization server issues access and

other tokens to the client. The attacker has now associated his

session with the legitimate client with the victim's resources

and/or identity.

4.5.2. Discussion

Obviously, the check in step (5.) will fail if the code was issued to

another client ID, e.g., a client set up by the attacker. The check

will also fail if the authorization code was already redeemed by the

legitimate user and was one-time use only.

An attempt to inject a code obtained via a 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.

[RFC6749], Section 4.1.3, requires the AS to "... ensure that the

redirect\_uri parameter is present if the redirect\_uri parameter was

included in the initial authorization request as described in

Section 4.1.1, and if included ensure that their values are

identical.". In the attack scenario described above, the legitimate

client would use the correct redirect URI it always uses for

authorization requests. But this URI would not match the tampered

redirect URI used by the attacker (otherwise, the redirect would not

land at the attackers page). So the authorization server would

detect the attack and refuse to exchange the code.

Note: This check could also detect 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 needs to contain a nonce or another kind

of one-time use, secret data and

\* the client has bound this data to this particular instance of the

client.

But this approach conflicts with the idea to enforce exact redirect

URI matching at the authorization endpoint. Moreover, it has been

observed that providers very often ignore the redirect\_uri check

requirement at this stage, maybe because it doesn't seem to be

security-critical from reading the 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

Lodderstedt, et al. Expires 7 December 2023 [Page 26]

Internet-Draft oauth-security-topics June 2023

respective authorization code for every transaction. But 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 [RFC6749],

Section 4.1.3, increase client implementation complexity as clients

need to store or re-construct 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 to instead bind every

authorization code to a certain client instance on a certain device

(or in a certain user agent) in the context of a certain transaction

using one of the mechanisms described next.

4.5.3. Countermeasures

There are two good technical solutions to achieve this goal, outlined

in the following.

4.5.3.1. PKCE

The PKCE mechanism specified in [RFC7636] can be used as a

countermeasure. 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 is a deployed

OAuth feature, although its originally intended use was solely

focused on securing native apps, not the broader use recommended by

this document.

PKCE does not only protect against the autorization 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.

Lodderstedt, et al. Expires 7 December 2023 [Page 27]

Internet-Draft oauth-security-topics June 2023

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 created by the client. The client is supposed to bind it to

the user agent session and sends it with the initial request to the

OpenID Provider (OP). The OP 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 injected an authorization code in the

authorization response, the nonce value in the client session and the

nonce value in the ID token will not match and the attack is

detected. The assumption is that an attacker cannot get hold of the

user agent state on the victim's device, where 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 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 today (originally intended for OAuth native apps) whereas

nonce is appropriate for OpenID Connect clients.

Lodderstedt, et al. Expires 7 December 2023 [Page 28]

Internet-Draft oauth-security-topics June 2023

4.5.4. Limitations

An attacker can circumvent the countermeasures described above if he

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 his own session

in Step 2 of the attack above. (This requires that the victim's

session with the client begins after the attacker started his 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 Section 4.1, Section 4.2,

Section 4.3, Section 4.4, and Section 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 makes up an authorization server response

including the leaked access token. Since the response includes the

state value generated by the client for this particular transaction,

the client does not treat the response as a CSRF 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.

Lodderstedt, et al. Expires 7 December 2023 [Page 29]

Internet-Draft oauth-security-topics June 2023

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 (with a

very high probability) contain 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 traditional 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 [RFC6819],

Section 5.3.5. 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 AS 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 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 as discussed in the now-expired draft

[I-D.bradley-oauth-jwt-encoded-state].

Lodderstedt, et al. Expires 7 December 2023 [Page 30]

Internet-Draft oauth-security-topics June 2023

The AS therefore MUST provide a way to detect their support for PKCE.

Using AS metadata according to [RFC8414] is RECOMMENDED, but AS MAY

instead provide a deployment-specific way to ensure or determine PKCE

support.

PKCE provides robust protection against CSRF attacks even in presence

of an 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

AS enables and enforces PKCE if this parameter is present in the

authorization request, but 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 his victim and the client.

4.8.1. Attack Description

1. The user has started an OAuth session using some client at an AS.

In the authorization request, the client has set the parameter

code\_challenge=sha256(abc) as the PKCE code challenge. The

client is now waiting to receive the authorization response from

the user's browse.

Lodderstedt, et al. Expires 7 December 2023 [Page 31]

Internet-Draft oauth-security-topics June 2023

2. To conduct the attack, the attacker uses his own device to start

an authorization flow with the targeted client. The client now

uses another PKCE code challenge, say code\_challenge=sha256(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 AS.

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

AS. 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 that 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, ASs MUST take precautions against this threat.

Note that from the view of the AS, 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 AS 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

Lodderstedt, et al. Expires 7 December 2023 [Page 32]

Internet-Draft oauth-security-topics June 2023

\* 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 AS 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.

Note: ASs 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 setup his 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 e-Mail, 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 of the respective server.

If the attacker were able to gain full control, including shell

access, 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.

Lodderstedt, et al. Expires 7 December 2023 [Page 33]

Internet-Draft oauth-security-topics June 2023

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.

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. Depending on the severity of

the penetration, sender-constrained access tokens will 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 any other

credentials. It is considered good practice to not log them and

not store them in plain text.

The first and second recommendation 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 Section 4.1, Section 4.2,

Section 4.3, Section 4.4 and Section 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.

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 prerequisite

for the acceptance of that token at a resource server.

A typical flow looks like this:

Lodderstedt, et al. Expires 7 December 2023 [Page 34]

Internet-Draft oauth-security-topics June 2023

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 identity, but in most cases the AS

utilizes key material (or data derived from the key material)

known to the client.

2. This key material must be distributed somehow. Either the key

material already exists before the AS creates the binding or the

AS creates ephemeral keys. The way pre-existing key material is

distributed varies among the different approaches. For example,

X.509 Certificates can be used, in which case the distribution

happens explicitly during the enrollment process. Or the key

material is created and distributed at the TLS layer, in which

case it might automatically happen during the setup of a TLS

connection.

3. The RS must implement the actual proof of possession check. This

is typically done on the application level, often tied to

specific material provided by transport layer (e.g., TLS). The

RS must also ensure that 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:

\* \*OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound

Access Tokens\* ([RFC8705]): The approach as 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 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.

\* \*DPoP\* ([I-D.ietf-oauth-dpop]): DPoP (Demonstration of Proof-of-

Possession at the Application Layer) outlines an application-level

sender-constraining for access and refresh tokens that can be used

in cases where neither mTLS nor OAuth Token Binding (see below)

are available. 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 case of confidential clients, can be

combined with any client authentication method.

Lodderstedt, et al. Expires 7 December 2023 [Page 35]

Internet-Draft oauth-security-topics June 2023

For reference, other approaches have been discussed as well but the

relevant drafts are now expired:

\* \*OAuth Token Binding\* ([I-D.ietf-oauth-token-binding]): In this

approach, an access token is, via the token binding ID, bound to

key material representing a long term association between a client

and a certain TLS host. Negotiation of the key material and proof

of possession in the context of a TLS handshake is taken care of

by the TLS stack. The client needs to determine the token binding

ID of the target resource server and pass this data to the access

token request. The authorization server then associates the

access token with this ID. The resource server checks on every

invocation that the token binding ID of the active TLS connection

and the token binding ID of associated with the access token

match. Since all crypto-related functions are covered by the TLS

stack, this approach is very client developer friendly. As a

prerequisite, token binding as described in [RFC8473] (including

federated token bindings) must be supported on all ends (client,

authorization server, resource server).

\* \*Signed HTTP Requests\* ([I-D.ietf-oauth-signed-http-request]):

This approach utilizes [I-D.ietf-oauth-pop-key-distribution] and

represents the elements of the signature in a JSON object. The

signature is built using JWS. The mechanism has built-in support

for signing of HTTP method, query parameters and headers. It also

incorporates a timestamp as basis for replay prevention.

\* \*JWT Pop Tokens\* ([I-D.sakimura-oauth-jpop]): This draft describes

different ways to constrain access token usage, namely TLS or

request signing. Note: Since the authors of this draft

contributed the TLS-related proposal to [RFC8705], this document

only considers the request signing part. For request signing, the

draft utilizes [I-D.ietf-oauth-pop-key-distribution] and

[RFC7800]. The signature data is represented in a JWT and JWS is

used for signing. Replay prevention is provided by building the

signature over a server-provided nonce, client-provided nonce and

a nonce counter.

At the time of writing, OAuth Mutual TLS is the most widely

implemented and the only standardized sender-constraining 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 a use of the token when

Lodderstedt, et al. Expires 7 December 2023 [Page 36]

Internet-Draft oauth-security-topics June 2023

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 SHOULD verify the intended audience. If the access token

fails the intended audience validation, the resource server MUST

refuse to serve the respective request.

In general, audience restrictions limit 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.

The client SHOULD tell the authorization server the intended resource

server. The proposed mechanism [RFC8707] could be used or by

encoding the information in the scope value.

Instead of the URL, it is also possible to utilize the fingerprint of

the resource server's X.509 certificate as audience value. This

variant would also allow to detect an attempt to spoof the 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.

Lodderstedt, et al. Expires 7 December 2023 [Page 37]

Internet-Draft oauth-security-topics June 2023

Audience restriction may seem easier to use since it does not require

any crypto on the client side. Still, since every access token is

bound to a specific resource server, the client also needs to obtain

a single RS-specific access token when accessing several resource

servers. (Resource indicators, as specified in [RFC8707], can help

to achieve this.) [I-D.ietf-oauth-token-binding] has the same

property since different token binding IDs must be associated with

the access token. Using [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, has additional benefits beyond the

scope of token leakage prevention. It allows the authorization

server to create a different access token whose format and content is

specifically minted for the respective server. This has huge

functional and privacy advantages in deployments using structured

access tokens.

4.10.3. 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.

In the simplest form, this would require the AS to publish a list of

its known resource servers, illustrated in the following example

using a non-standard metadata parameter resource\_servers:

HTTP/1.1 200 OK

Content-Type: application/json

{

"issuer":"https://server.somesite.example",

"authorization\_endpoint":

"https://server.somesite.example/authorize",

"resource\_servers":[

"email.somesite.example",

"storage.somesite.example",

"video.somesite.example"

]

...

}

The AS could also return the URL(s) an access token is good for in

the token response, illustrated by the example and non-standard

return parameter access\_token\_resource\_server:

Lodderstedt, et al. Expires 7 December 2023 [Page 38]

Internet-Draft oauth-security-topics June 2023

HTTP/1.1 200 OK

Content-Type: application/json;charset=UTF-8

Cache-Control: no-store

Pragma: no-cache

{

"access\_token":"2YotnFZFEjr1zCsicMWpAA",

"access\_token\_resource\_server":

"https://hostedresource.somesite.example/path1",

...

}

This mitigation strategy would rely on the client to enforce the

security policy and to only send access tokens to legitimate

destinations. Results of OAuth-related security research (see for

example [oauth\_security\_ubc] and [oauth\_security\_cmu]) indicate a

large portion of client implementations do not or fail to properly

implement security controls, like state checks. So relying on

clients to prevent access token phishing is likely to fail as well.

Moreover, given the ratio of clients to authorization and resource

servers, it is considered the more viable approach to move as much as

possible security-related logic to those entities. Clearly, the

client has to contribute to the overall security. But there are

alternative countermeasures, as described before, that provide a

better balance between the involved parties.

4.11. Open Redirection

The following attacks can occur when an AS or client has an open

redirector. An open redirector is an endpoint that forwards a user’s

browser to an arbitrary URI obtained from a query parameter. 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 follow it in their browser. This can be abused for phishing.

Lodderstedt, et al. Expires 7 December 2023 [Page 39]

Internet-Draft oauth-security-topics June 2023

In order to prevent open redirection, clients should only redirect if

the target URLs are whitelisted 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 web sites (the clients), but must do so in a

safe way.

[RFC6749], Section 4.1.2.1, already prevents open redirects by

stating that the AS 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 AS 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 AS prompts the user to provide consent to the

request. If the user notices an issue with the request and

declines the request, the AS 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 AS will automatically redirect the

user agent to the phishing site.

Lodderstedt, et al. Expires 7 December 2023 [Page 40]

Internet-Draft oauth-security-topics June 2023

The AS MUST take precautions to prevent these threats. The AS 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 AS needs to decide whether it can trust the redirect URI or not.

It could take into account URI analytics done internally or through

some external service to evaluate the credibility and trustworthiness

content behind the URI, and the source of the redirect URI and other

client data.

The AS SHOULD only automatically redirect the user agent if it trusts

the redirect URI. If the URI is not trusted, the AS 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 AS

prompts the user to enter her credentials in a form that is then

submitted (using the HTTP POST method) back to the authorization

server. The AS checks the credentials and, if successful, redirects

the user agent to the client's redirection endpoint.

In [RFC6749], the HTTP status code 302 is used for this purpose, but

"any other method available via the user-agent to accomplish this

redirection is allowed". 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 AS.

The behavior might be unexpected for developers, but is defined in

[RFC7231], Section 6.4.7. This status code 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 [RFC7231], only the status code 303

unambigiously enforces rewriting the HTTP POST request to an HTTP GET

request. For all other status codes, including the popular 302, user

agents can opt not to rewrite POST to GET requests and therefore to

reveal the user's credentials to the client. (In practice, however,

most user agents will only show this behaviour for 307 redirects.)

Lodderstedt, et al. Expires 7 December 2023 [Page 41]

Internet-Draft oauth-security-topics June 2023

ASs 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 AS 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

[I-D.ietf-httpbis-client-cert-field].

If the reverse proxy would pass through any header sent from the

outside, an attacker could try to directly send the faked header

values through the proxy to the application server in order to

circumvent security controls that way. For example, it is standard

practice of reverse proxies to accept 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 a whitelisted IP address to the header and

render a IP whitelist useless.

A reverse proxy MUST therefore sanitize any inbound requests to

ensure the authenticity and integrity of all header values relevant

for the security of the application servers.

If an attacker were able to get access to the internal network

between proxy and application server, the attacker could also try to

circumvent security controls in place. It is, therefore, essential

to ensure the authenticity of the communicating entities.

Furthermore, the communication link between reverse proxy and

application server MUST be protected against eavesdropping,

injection, and replay of messages.

Lodderstedt, et al. Expires 7 December 2023 [Page 42]

Internet-Draft oauth-security-topics June 2023

4.14. Refresh Token Protection

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

access tokens after the expiration of access tokens. Refresh tokens

also add to the security of OAuth, since they allow the authorization

server to issue access tokens with a short lifetime and reduced

scope, thus reducing the potential impact of access token leakage.

4.14.1. Discussion

Refresh tokens are an attractive target for attackers, since they

represent the overall grant a resource owner delegated to a certain

client. If an attacker is able to exfiltrate and successfully replay

a refresh token, 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 a 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 SHOULD 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.

Lodderstedt, et al. Expires 7 December 2023 [Page 43]

Internet-Draft oauth-security-topics June 2023

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 [I-D.ietf-oauth-dpop].

\* \*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

\* logout at the authorization server

Lodderstedt, et al. Expires 7 December 2023 [Page 44]

Internet-Draft oauth-security-topics June 2023

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.

If both options are possible, depending on the details of the

implementation, a client's identity may be mistaken 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 an end-user (e.g.,

a sub value if OpenID Connect is used). If the resource server

cannot properly distinguish between access tokens issued to clients

and access tokens issued to end-users, the client may then be able to

access resource of the end-user.

4.15.1. Countermeasures

If the authorization server has a common namespace for client IDs and

user identifiers, causing the resource server to be unable to

distinguish an access token authorized by a resource owner from an

access token authorized by a client itself, the authorization server

SHOULD NOT allow clients to influence their client\_id or any claim

that could cause confusion with a genuine resource owner. 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, 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.

Lodderstedt, et al. Expires 7 December 2023 [Page 45]

Internet-Draft oauth-security-topics June 2023

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 [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 [CSP-2] for details). Level 2 of this standard

provides a robust mechanism for protecting against clickjacking by

using policies that restrict the origin of frames (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 [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. Authorization Server Redirecting to Phishing Site

An attacker could 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:

Lodderstedt, et al. Expires 7 December 2023 [Page 46]

Internet-Draft oauth-security-topics June 2023

1. Intentionally send an erroneous authorization request, e.g., by

using an invalid scope value, thus instructing the AS 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 AS prompts the user to provide consent to the

request. If the user notices an issue with the request and

declines the request, the AS 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 AS will automatically redirect the

user agent to the phishing site.

The AS MUST take precautions to prevent these threats. The AS 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 AS needs to decide whether it can trust the redirect URI or not.

It could take into account URI analytics done internally or through

some external service to evaluate the credibility and trustworthiness

content behind the URI, and the source of the redirect URI and other

client data.

The AS SHOULD only automatically redirect the user agent if it trusts

the redirect URI. If the URI is not trusted, the AS MAY inform the

user and rely on the user to make the correct decision.

4.18. Attacks on In-Browser Communication Flows

If the authorization response is sent with in-browser communication

techniques like postMessage [postmessage\_api] instead of HTTP

redirects, messages may inadvertently be sent to malicious origins or

injected from malicious origins.

4.18.1. Examples

The following examples of attacks using in-browser communication are

described in [inbc\_security\_sso]:

Lodderstedt, et al. Expires 7 December 2023 [Page 47]

Internet-Draft oauth-security-topics June 2023

4.18.1.1. 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

)

4.18.1.2. 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, 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

)

4.18.1.3. 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 next are generally required.

Lodderstedt, et al. Expires 7 December 2023 [Page 48]

Internet-Draft oauth-security-topics June 2023

4.18.2. 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:

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:

window.addEventListener("message", (e) => {

// validate exact AS 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. Acknowledgements

We would like to thank Brock Allen, Annabelle Richard Backman,

Dominick Baier, Vittorio Bertocci, Brian Campbell, William Dennis,

George Fletcher, Dick Hardt, Joseph Heenan, Pedram Hosseyni, Phil

Hunt, Louis Jannett, Jared Jennings, Michael B. Jones, Konstantin

Lapine, Neil Madden, Christian Mainka, Jim Manico, Nov Matake, Doug

McDorman, Vladislav Mladenov, Karsten Meyer zu Selhausen, Aaron

Parecki, Michael Peck, Johan Peeters, Nat Sakimura, Guido Schmitz,

Jörg Schwenk, Rifaat Shekh-Yusef, Travis Spencer, Petteri Stenius,

Tomek Stojecki, Tim Wuertele, David Waite and Hans Zandbelt for their

Lodderstedt, et al. Expires 7 December 2023 [Page 49]

Internet-Draft oauth-security-topics June 2023

valuable feedback.

6. IANA Considerations

This draft makes no requests to IANA.

7. Security Considerations

Security considerations are described in Section 2, Section 3, and

Section 4.

8. Normative References

[OpenID.Core]

Sakimura, N., Bradley, J., Jones, M., de Medeiros, B., and

C. Mortimore, "OpenID Connect Core 1.0 incorporating

errata set 1", 8 November 2014,

<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 1", 8

November 2014, <https://openid.net/specs/openid-connect-

discovery-1\_0.html>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform

Resource Identifier (URI): Generic Syntax", STD 66,

RFC 3986, DOI 10.17487/RFC3986, January 2005,

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

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework",

RFC 6749, DOI 10.17487/RFC6749, October 2012,

<https://www.rfc-editor.org/info/rfc6749>.

[RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization

Framework: Bearer Token Usage", RFC 6750,

DOI 10.17487/RFC6750, October 2012,

<https://www.rfc-editor.org/info/rfc6750>.

[RFC6819] Lodderstedt, T., Ed., McGloin, M., and P. Hunt, "OAuth 2.0

Threat Model and Security Considerations", RFC 6819,

DOI 10.17487/RFC6819, January 2013,

<https://www.rfc-editor.org/info/rfc6819>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer

Protocol (HTTP/1.1): Semantics and Content", RFC 7231,

DOI 10.17487/RFC7231, June 2014,

<https://www.rfc-editor.org/info/rfc7231>.

Lodderstedt, et al. Expires 7 December 2023 [Page 50]

Internet-Draft oauth-security-topics June 2023

[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, 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>.

9. Informative References

[CORS] "Fetch Standard: CORS protocol",

<https://fetch.spec.whatwg.org/#http-cors-protocol>.

[CSP-2] West, M., Barth, A., and D. Veditz, "Content Security

Policy Level 2", July 2015, <https://www.w3.org/TR/CSP2>.

[I-D.bradley-oauth-jwt-encoded-state]

Bradley, J., Lodderstedt, T., and H. Zandbelt, "Encoding

claims in the OAuth 2 state parameter using a JWT", 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>.

Lodderstedt, et al. Expires 7 December 2023 [Page 51]

Internet-Draft oauth-security-topics June 2023

[I-D.ietf-httpbis-client-cert-field]

Campbell, B. and M. Bishop, "Client-Cert HTTP Header

Field", Work in Progress, Internet-Draft, draft-ietf-

httpbis-client-cert-field-06, 17 March 2023,

<https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-

client-cert-field-06>.

[I-D.ietf-oauth-dpop]

Fett, D., Campbell, B., Bradley, J., Lodderstedt, T.,

Jones, M. B., and D. Waite, "OAuth 2.0 Demonstrating

Proof-of-Possession at the Application Layer (DPoP)", Work

in Progress, Internet-Draft, draft-ietf-oauth-dpop-16, 13

April 2023, <https://datatracker.ietf.org/doc/html/draft-

ietf-oauth-dpop-16>.

[I-D.ietf-oauth-iss-auth-resp]

zu Selhausen, K. M. and D. Fett, "OAuth 2.0 Authorization

Server Issuer Identification", Work in Progress, Internet-

Draft, draft-ietf-oauth-iss-auth-resp-05, 11 January 2022,

<https://datatracker.ietf.org/doc/html/draft-ietf-oauth-

iss-auth-resp-05>.

[I-D.ietf-oauth-pop-key-distribution]

Bradley, J., Hunt, P., Jones, M., Tschofenig, H., and M.

Meszaros, "OAuth 2.0 Proof-of-Possession: Authorization

Server to Client Key Distribution", Work in Progress,

Internet-Draft, draft-ietf-oauth-pop-key-distribution-07,

27 March 2019, <https://datatracker.ietf.org/doc/html/

draft-ietf-oauth-pop-key-distribution-07>.

[I-D.ietf-oauth-signed-http-request]

Richer, J., Bradley, J., and H. Tschofenig, "A Method for

Signing HTTP Requests for OAuth", Work in Progress,

Internet-Draft, draft-ietf-oauth-signed-http-request-03, 8

August 2016, <https://datatracker.ietf.org/doc/html/draft-

ietf-oauth-signed-http-request-03>.

[I-D.ietf-oauth-token-binding]

Jones, M., Campbell, B., Bradley, J., and W. Denniss,

"OAuth 2.0 Token Binding", 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>.

Lodderstedt, et al. Expires 7 December 2023 [Page 52]

Internet-Draft oauth-security-topics June 2023

[I-D.sakimura-oauth-jpop]

Sakimura, N., Li, K., and J. Bradley, "The OAuth 2.0

Authorization Framework: JWT Pop Token Usage", Work in

Progress, Internet-Draft, draft-sakimura-oauth-jpop-05, 22

July 2019, <https://datatracker.ietf.org/doc/html/draft-

sakimura-oauth-jpop-05>.

[JARM] Lodderstedt, T. and B. Campbell, "Financial-grade API: JWT

Secured Authorization Response Mode for OAuth 2.0 (JARM)",

17 October 2018,

<https://openid.net/specs/openid-financial-api-jarm.html>.

[OAuth.Post]

Jones, M. and B. Campbell, "OAuth 2.0 Form Post Response

Mode", 27 April 2015, <http://openid.net/specs/oauth-v2-

form-post-response-mode-1\_0.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>.

[RFC7800] Jones, M., Bradley, J., and H. Tschofenig, "Proof-of-

Possession Key Semantics for JSON Web Tokens (JWTs)",

RFC 7800, DOI 10.17487/RFC7800, April 2016,

<https://www.rfc-editor.org/info/rfc7800>.

[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>.

[RFC8473] Popov, A., Nystroem, M., Balfanz, D., Ed., Harper, N., and

J. Hodges, "Token Binding over HTTP", RFC 8473,

DOI 10.17487/RFC8473, October 2018,

<https://www.rfc-editor.org/info/rfc8473>.

Lodderstedt, et al. Expires 7 December 2023 [Page 53]

Internet-Draft oauth-security-topics June 2023

[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>.

[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>.

[WebAuthn] Balfanz, D., Czeskis, A., Hodges, J., Jones, J.C., Jones,

M.B., Kumar, A., Liao, A., Lindemann, R., and E. Lundberg,

"Web Authentication: An API for accessing Public Key

Credentials Level 1", 4 March 2019,

<https://www.w3.org/TR/2019/REC-webauthn-1-20190304/>.

[WebCrypto]

Watson, M., "Web Cryptography API", 26 January 2017,

<https://www.w3.org/TR/2017/REC-WebCryptoAPI-20170126/>.

[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, 7

January 2016, <http://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,

6 January 2016, <http://arxiv.org/abs/1601.01229/>.

Lodderstedt, et al. Expires 7 December 2023 [Page 54]

Internet-Draft oauth-security-topics June 2023

[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, 27 April 2017,

<http://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, 31 January 2019,

<http://arxiv.org/abs/1901.11520/>.

[bug.chromium]

"Referer header includes URL fragment when opening link

using New Tab",

<https://bugs.chromium.org/p/chromium/issues/

detail?id=168213/>.

[inbc\_security\_sso]

Jannett, L., Mladenov, V., Mainka, C., and J. Schwenk,

"DISTINCT: Identity Theft using In-Browser Communications

in Dual-Window Single Sign-On",

DOI 10.1145/3548606.3560692, 7 November 2022,

<https://distinct-sso.com/paper.pdf>.

[oauth\_security\_cmu]

Chen, E., Pei, Y., Chen, S., Tian, Y., Kotcher, R., and P.

Tague, "OAuth Demystified for Mobile Application

Developers", November 2014,

<http://css.csail.mit.edu/6.858/2012/readings/oauth-

sso.pdf>.

[oauth\_security\_jcs\_14]

Bansal, C., Bhargavan, K., Delignat-Lavaud, A., and S.

Maffeis, "Discovering concrete attacks on website

authorization by formal analysis", 23 April 2014,

<https://www.doc.ic.ac.uk/~maffeis/papers/jcs14.pdf>.

[oauth\_security\_ubc]

Sun, S.-T. and K. Beznosov, "The Devil is in the

(Implementation) Details: An Empirical Analysis of OAuth

SSO Systems", October 2012,

<http://passwordresearch.com/papers/paper267.html>.

Lodderstedt, et al. Expires 7 December 2023 [Page 55]

Internet-Draft oauth-security-topics June 2023

[owasp\_redir]

"OWASP Cheat Sheet Series - Unvalidated Redirects and

Forwards",

<https://cheatsheetseries.owasp.org/cheatsheets/

Unvalidated\_Redirects\_and\_Forwards\_Cheat\_Sheet.html>.

[postmessage\_api]

"HTML Living Standard: Cross-document messaging",

<https://html.spec.whatwg.org/multipage/web-

messaging.html#web-messaging>.

[subdomaintakeover]

Liu, D., Hao, S., and H. Wang, "All Your DNS Records Point

to Us: Understanding the Security Threats of Dangling DNS

Records", 24 October 2016,

<https://www.eecis.udel.edu/~hnw/paper/ccs16a.pdf>.

[webappsec-referrer-policy]

Eisinger, J. and E. Stark, "Referrer Policy", 20 April

2017, <https://w3c.github.io/webappsec-referrer-policy>.

Appendix A. Document History

[[ To be removed from the final specification ]]

-23

\* Added CORS considerations

\* Reworded Section 4.15.1 to be more in line with OAuth 2.1

\* Editorial changes

\* Clarifications and updated references

-22

\* Added section on securing in-browser communication

\* Merged section on phishing via AS into existing section on open

redirectors

\* Restructure and move section on sender-constrained tokens

\* Mention RFCs for Private Key JWK method

-21

Lodderstedt, et al. Expires 7 December 2023 [Page 56]

Internet-Draft oauth-security-topics June 2023

\* Improved wording on phishing via AS

-20

\* Improved description of authorization code injection attacks and

PKCE protection

\* Removed recommendation for MTLS in discussion (not reflected in

actual Recommendations section)

\* Reworded "placeholder" text in security considerations.

\* Alphabetized list of names and fixed unicode problem

\* Explained Clickjacking

\* Explained Open Redirectors

\* Clarified references to attacker model by including a link to

Section 3

\* Clarified description of "CSRF tokens" and reference to RFC6819

\* Described that OIDC can prevent access token injection

\* Updated references

-19

\* Changed affiliation of Andrey Labunets

\* Editorial change to clarify the new recommendations for refresh

tokens

-18

\* Fix editorial and spelling issues.

\* Change wording for disallowing HTTP redirect URIs.

-17

\* Make the use of metadata RECOMMENDED for both servers and clients

\* Make announcing PKCE support in metadata the RECOMMENDED way

(before: either metadata or deployment-specific way)

\* AS also MUST NOT expose open redirectors.

Lodderstedt, et al. Expires 7 December 2023 [Page 57]

Internet-Draft oauth-security-topics June 2023

\* Mention that attackers can collaborate.

\* Update recommendations regarding mix-up defense, building upon

[I-D.ietf-oauth-iss-auth-resp].

\* Improve description of mix-up attack.

\* Make HTTPS mandatory for most redirect URIs.

-16

\* Make MTLS a suggestion, not RECOMMENDED.

\* Add important requirements when using nonce for code injection

protection.

\* Highlight requirements for refresh token sender-constraining.

\* Make PKCE a MUST for public clients.

\* Describe PKCE Downgrade Attacks and countermeasures.

\* Allow variable port numbers in localhost redirect URIs as in

RFC8252, Section 7.3.

-15

\* Update reference to DPoP

\* Fix reference to RFC8414

\* Move to xml2rfcv3

-14

\* Added info about using CSP to prevent clickjacking

\* Changes from WGLC feedback

\* Editorial changes

\* AS MUST announce PKCE support either in metadata or using

deployment-specific ways (before: SHOULD)

-13

\* Discourage use of Resource Owner Password Credentials Grant

Lodderstedt, et al. Expires 7 December 2023 [Page 58]

Internet-Draft oauth-security-topics June 2023

\* Added text on client impersonating resource owner

\* Recommend asymmetric methods for client authentication

\* Encourage use of PKCE mode "S256"

\* PKCE may replace state for CSRF protection

\* AS SHOULD publish PKCE support

\* Cleaned up discussion on auth code injection

\* AS MUST support PKCE

-12

\* Added updated attacker model

-11

\* Adapted section 2.1.2 to outcome of consensus call

\* more text on refresh token inactivity and implementation note on

refresh token replay detection via refresh token rotation

-10

\* incorporated feedback by Joseph Heenan

\* changed occurrences of SHALL to MUST

\* added text on lack of token/cert binding support tokens issued in

the authorization response as justification to not recommend

issuing tokens there at all

\* added requirement to authenticate clients during code exchange

(PKCE or client credential) to 2.1.1.

\* added section on refresh tokens

\* editorial enhancements to 2.1.2 based on feedback

-09

\* changed text to recommend not to use implicit but code

\* added section on access token injection

Lodderstedt, et al. Expires 7 December 2023 [Page 59]

Internet-Draft oauth-security-topics June 2023

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

grant issues

-08

\* added recommendations re implicit and token injection

\* uppercased key words in Section 2 according to RFC 2119

-07

\* incorporated findings of Doug McDorman

\* added section on HTTP status codes for redirects

\* added new section on access token privilege restriction based on

comments from Johan Peeters

-06

\* reworked section 3.8.1

\* incorporated Phil Hunt's feedback

\* reworked section on mix-up

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

state leakage

\* added Daniel Fett as author

\* replaced text intended to inform WG discussion by recommendations

to implementors

\* modified example URLs to conform to RFC 2606

-05

\* Completed sections on code leakage via referrer header, attacks in

browser, mix-up, and CSRF

\* Reworked Code Injection Section

\* Added reference to OpenID Connect spec

\* removed refresh token leakage as respective considerations have

been given in section 10.4 of RFC 6749

Lodderstedt, et al. Expires 7 December 2023 [Page 60]

Internet-Draft oauth-security-topics June 2023

\* first version on open redirection

\* incorporated Christian Mainka's review feedback

-04

\* Restructured document for better readability

\* Added best practices on Token Leakage prevention

-03

\* Added section on Access Token Leakage at Resource Server

\* incorporated Brian Campbell's findings

-02

\* Folded Mix up and Access Token leakage through a bad AS into new

section for dynamic OAuth threats

\* reworked dynamic OAuth section

-01

\* Added references to mitigation methods for token leakage

\* Added reference to Token Binding for Authorization Code

\* incorporated feedback of Phil Hunt

\* fixed numbering issue in attack descriptions in section 2

-00 (WG document)

\* turned the ID into a WG document and a BCP

\* Added federated app login as topic in Other Topics

Authors' Addresses

Torsten Lodderstedt

yes.com

Email: torsten@lodderstedt.net

John Bradley

Yubico

Lodderstedt, et al. Expires 7 December 2023 [Page 61]

Internet-Draft oauth-security-topics June 2023

Email: ve7jtb@ve7jtb.com

Andrey Labunets

Independent Researcher

Email: isciurus@gmail.com

Daniel Fett

Authlete

Email: mail@danielfett.de

Lodderstedt, et al. Expires 7 December 2023 [Page 62]