

Internet Engineering Task Force (IETF)  
Request for Comments: 6819  
Category: Informational  
ISSN: 2070-1721

T. Lodderstedt, Ed.  
Deutsche Telekom AG  
M. McGloin  
IBM  
P. Hunt  
Oracle Corporation  
January 2013

## OAuth 2.0 Threat Model and Security Considerations

### Abstract

This document gives additional security considerations for OAuth, beyond those in the OAuth 2.0 specification, based on a comprehensive threat model for the OAuth 2.0 protocol.

### Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are a candidate for any level of Internet Standard; see [Section 2 of RFC 5741](#).

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at <http://www.rfc-editor.org/info/rfc6819>.

### Copyright Notice

Copyright (c) 2013 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 (<http://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 Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

1. Introduction .....	6
2. Overview .....	7
2.1. Scope .....	7
2.2. Attack Assumptions .....	7
2.3. Architectural Assumptions .....	8
2.3.1. Authorization Servers .....	8
2.3.2. Resource Server .....	9
2.3.3. Client .....	9
3. Security Features .....	9
3.1. Tokens .....	10
3.1.1. Scope .....	11
3.1.2. Limited Access Token Lifetime .....	11
3.2. Access Token .....	11
3.3. Refresh Token .....	11
3.4. Authorization "code" .....	12
3.5. Redirect URI .....	13
3.6. "state" Parameter .....	13
3.7. Client Identifier .....	13
4. Threat Model .....	15
4.1. Clients .....	16
4.1.1. Threat: Obtaining Client Secrets .....	16
4.1.2. Threat: Obtaining Refresh Tokens .....	17
4.1.3. Threat: Obtaining Access Tokens .....	19
4.1.4. Threat: End-User Credentials Phished Using Compromised or Embedded Browser .....	19
4.1.5. Threat: Open Redirectors on Client .....	20
4.2. Authorization Endpoint .....	21
4.2.1. Threat: Password Phishing by Counterfeit Authorization Server .....	21
4.2.2. Threat: User Unintentionally Grants Too Much Access Scope .....	21
4.2.3. Threat: Malicious Client Obtains Existing Authorization by Fraud .....	22
4.2.4. Threat: Open Redirector .....	22
4.3. Token Endpoint .....	23
4.3.1. Threat: Eavesdropping Access Tokens .....	23
4.3.2. Threat: Obtaining Access Tokens from Authorization Server Database .....	23
4.3.3. Threat: Disclosure of Client Credentials during Transmission .....	23
4.3.4. Threat: Obtaining Client Secret from Authorization Server Database .....	24
4.3.5. Threat: Obtaining Client Secret by Online Guessing ..	24

4.4. Obtaining Authorization .....	25
4.4.1. Authorization "code" .....	25
4.4.1.1. Threat: Eavesdropping or Leaking Authorization "codes" .....	25
4.4.1.2. Threat: Obtaining Authorization "codes" from Authorization Server Database .....	26
4.4.1.3. Threat: Online Guessing of Authorization "codes" .....	27
4.4.1.4. Threat: Malicious Client Obtains Authorization .....	27
4.4.1.5. Threat: Authorization "code" Phishing .....	29
4.4.1.6. Threat: User Session Impersonation .....	29
4.4.1.7. Threat: Authorization "code" Leakage through Counterfeit Client .....	30
4.4.1.8. Threat: CSRF Attack against redirect-uri ..	32
4.4.1.9. Threat: Clickjacking Attack against Authorization .....	33
4.4.1.10. Threat: Resource Owner Impersonation .....	33
4.4.1.11. Threat: DoS Attacks That Exhaust Resources .....	34
4.4.1.12. Threat: DoS Using Manufactured Authorization "codes" .....	35
4.4.1.13. Threat: Code Substitution (OAuth Login) ..	36
4.4.2. Implicit Grant .....	37
4.4.2.1. Threat: Access Token Leak in Transport/Endpoints .....	37
4.4.2.2. Threat: Access Token Leak in Browser History .....	38
4.4.2.3. Threat: Malicious Client Obtains Authorization .....	38
4.4.2.4. Threat: Manipulation of Scripts .....	38
4.4.2.5. Threat: CSRF Attack against redirect-uri ..	39
4.4.2.6. Threat: Token Substitution (OAuth Login) ..	39
4.4.3. Resource Owner Password Credentials .....	40
4.4.3.1. Threat: Accidental Exposure of Passwords at Client Site .....	41
4.4.3.2. Threat: Client Obtains Scopes without End-User Authorization .....	42
4.4.3.3. Threat: Client Obtains Refresh Token through Automatic Authorization .....	42
4.4.3.4. Threat: Obtaining User Passwords on Transport .....	43
4.4.3.5. Threat: Obtaining User Passwords from Authorization Server Database .....	43
4.4.3.6. Threat: Online Guessing .....	43
4.4.4. Client Credentials .....	44

4.5.	Refreshing an Access Token .....	44
4.5.1.	Threat: Eavesdropping Refresh Tokens from Authorization Server .....	44
4.5.2.	Threat: Obtaining Refresh Token from Authorization Server Database .....	44
4.5.3.	Threat: Obtaining Refresh Token by Online Guessing .....	45
4.5.4.	Threat: Refresh Token Phishing by Counterfeit Authorization Server .....	45
4.6.	Accessing Protected Resources .....	46
4.6.1.	Threat: Eavesdropping Access Tokens on Transport ...	46
4.6.2.	Threat: Replay of Authorized Resource Server Requests .....	46
4.6.3.	Threat: Guessing Access Tokens .....	46
4.6.4.	Threat: Access Token Phishing by Counterfeit Resource Server .....	47
4.6.5.	Threat: Abuse of Token by Legitimate Resource Server or Client .....	48
4.6.6.	Threat: Leak of Confidential Data in HTTP Proxies ..	48
4.6.7.	Threat: Token Leakage via Log Files and HTTP Referrers .....	48
5.	Security Considerations .....	49
5.1.	General .....	49
5.1.1.	Ensure Confidentiality of Requests .....	49
5.1.2.	Utilize Server Authentication .....	50
5.1.3.	Always Keep the Resource Owner Informed .....	50
5.1.4.	Credentials .....	51
5.1.4.1.	Enforce Credential Storage Protection Best Practices .....	51
5.1.4.2.	Online Attacks on Secrets .....	52
5.1.5.	Tokens (Access, Refresh, Code) .....	53
5.1.5.1.	Limit Token Scope .....	53
5.1.5.2.	Determine Expiration Time .....	54
5.1.5.3.	Use Short Expiration Time .....	54
5.1.5.4.	Limit Number of Usages or One-Time Usage ..	55
5.1.5.5.	Bind Tokens to a Particular Resource Server (Audience) .....	55
5.1.5.6.	Use Endpoint Address as Token Audience ....	56
5.1.5.7.	Use Explicitly Defined Scopes for Audience and Tokens .....	56
5.1.5.8.	Bind Token to Client id .....	56
5.1.5.9.	Sign Self-Contained Tokens .....	56
5.1.5.10.	Encrypt Token Content .....	56
5.1.5.11.	Adopt a Standard Assertion Format .....	57
5.1.6.	Access Tokens .....	57

5.2.	Authorization Server .....	57
5.2.1.	Authorization "codes" .....	57
5.2.1.1.	Automatic Revocation of Derived Tokens If Abuse Is Detected .....	57
5.2.2.	Refresh Tokens .....	57
5.2.2.1.	Restricted Issuance of Refresh Tokens .....	57
5.2.2.2.	Binding of Refresh Token to "client_id" ...	58
5.2.2.3.	Refresh Token Rotation .....	58
5.2.2.4.	Revocation of Refresh Tokens .....	58
5.2.2.5.	Device Identification .....	59
5.2.2.6.	X-FRAME-OPTIONS Header .....	59
5.2.3.	Client Authentication and Authorization .....	59
5.2.3.1.	Don't Issue Secrets to Clients with Inappropriate Security Policy .....	60
5.2.3.2.	Require User Consent for Public Clients without Secret .....	60
5.2.3.3.	Issue a "client_id" Only in Combination with "redirect_uri" .....	61
5.2.3.4.	Issue Installation-Specific Client Secrets .....	61
5.2.3.5.	Validate Pre-Registered "redirect_uri" ....	62
5.2.3.6.	Revoke Client Secrets .....	63
5.2.3.7.	Use Strong Client Authentication (e.g., client_assertion/client_token) .....	63
5.2.4.	End-User Authorization .....	63
5.2.4.1.	Automatic Processing of Repeated Authorizations Requires Client Validation ..	63
5.2.4.2.	Informed Decisions Based on Transparency ..	63
5.2.4.3.	Validation of Client Properties by End User .....	64
5.2.4.4.	Binding of Authorization "code" to "client_id" .....	64
5.2.4.5.	Binding of Authorization "code" to "redirect_uri" .....	64
5.3.	Client App Security .....	65
5.3.1.	Don't Store Credentials in Code or Resources Bundled with Software Packages .....	65
5.3.2.	Use Standard Web Server Protection Measures (for Config Files and Databases) .....	65
5.3.3.	Store Secrets in Secure Storage .....	65
5.3.4.	Utilize Device Lock to Prevent Unauthorized Device Access .....	66
5.3.5.	Link the "state" Parameter to User Agent Session ...	66
5.4.	Resource Servers .....	66
5.4.1.	Authorization Headers .....	66
5.4.2.	Authenticated Requests .....	67
5.4.3.	Signed Requests .....	67
5.5.	A Word on User Interaction and User-Installed Apps .....	68

6. Acknowledgements .....	69
7. References .....	69
7.1. Normative References .....	69
7.2. Informative References .....	69

## 1. Introduction

This document gives additional security considerations for OAuth, beyond those in the OAuth specification, based on a comprehensive threat model for the OAuth 2.0 protocol [RFC6749]. It contains the following content:

- o Documents any assumptions and scope considered when creating the threat model.
- o Describes the security features built into the OAuth protocol and how they are intended to thwart attacks.
- o Gives a comprehensive threat model for OAuth and describes the respective countermeasures to thwart those threats.

Threats include any intentional attacks on OAuth tokens and resources protected by OAuth tokens, as well as security risks introduced if the proper security measures are not put in place. Threats are structured along the lines of the protocol structure to help development teams implement each part of the protocol securely, for example, all threats for granting access, or all threats for a particular grant type, or all threats for protecting the resource server.

Note: This document cannot assess the probability or the risk associated with a particular threat because those aspects strongly depend on the particular application and deployment OAuth is used to protect. Similarly, impacts are given on a rather abstract level. But the information given here may serve as a foundation for deployment-specific threat models. Implementors may refine and detail the abstract threat model in order to account for the specific properties of their deployment and to come up with a risk analysis. As this document is based on the base OAuth 2.0 specification, it does not consider proposed extensions such as client registration or discovery, many of which are still under discussion.

## 2. Overview

### 2.1. Scope

This security considerations document only considers clients bound to a particular deployment as supported by [RFC6749]. Such deployments have the following characteristics:

- o Resource server URLs are static and well-known at development time; authorization server URLs can be static or discovered.
- o Token scope values (e.g., applicable URLs and methods) are well-known at development time.
- o Client registration is out of scope of the current core specification. Therefore, this document assumes a broad variety of options, from static registration during development time to dynamic registration at runtime.

The following are considered out of scope:

- o Communication between the authorization server and resource server.
- o Token formats.
- o Except for the resource owner password credentials grant type (see [RFC6749], Section 4.3), the mechanism used by authorization servers to authenticate the user.
- o Mechanism by which a user obtained an assertion and any resulting attacks mounted as a result of the assertion being false.
- o Clients not bound to a specific deployment: An example could be a mail client with support for contact list access via the portable contacts API (see [Portable-Contacts]). Such clients cannot be registered upfront with a particular deployment and should dynamically discover the URLs relevant for the OAuth protocol.

### 2.2. Attack Assumptions

The following assumptions relate to an attacker and resources available to an attacker. It is assumed that:

- o the attacker has full access to the network between the client and authorization servers and the client and the resource server, respectively. The attacker may eavesdrop on any communications

between those parties. He is not assumed to have access to communication between the authorization server and resource server.

- o an attacker has unlimited resources to mount an attack.
- o two of the three parties involved in the OAuth protocol may collude to mount an attack against the 3rd party. For example, the client and authorization server may be under control of an attacker and collude to trick a user to gain access to resources.

### 2.3. Architectural Assumptions

This section documents assumptions about the features, limitations, and design options of the different entities of an OAuth deployment along with the security-sensitive data elements managed by those entities. These assumptions are the foundation of the threat analysis.

The OAuth protocol leaves deployments with a certain degree of freedom regarding how to implement and apply the standard. The core specification defines the core concepts of an authorization server and a resource server. Both servers can be implemented in the same server entity, or they may also be different entities. The latter is typically the case for multi-service providers with a single authentication and authorization system and is more typical in middleware architectures.

#### 2.3.1. Authorization Servers

The following data elements are stored or accessible on the authorization server:

- o usernames and passwords
- o client ids and secrets
- o client-specific refresh tokens
- o client-specific access tokens (in the case of handle-based design; see [Section 3.1](#))
- o HTTPS certificate/key
- o per-authorization process (in the case of handle-based design; [Section 3.1](#)): "redirect\_uri", "client\_id", authorization "code"



### 2.3.2. Resource Server

The following data elements are stored or accessible on the resource server:

- o user data (out of scope)
- o HTTPS certificate/key
- o either authorization server credentials (handle-based design; see [Section 3.1](#)) or authorization server shared secret/public key (assertion-based design; see [Section 3.1](#))
- o access tokens (per request)

It is assumed that a resource server has no knowledge of refresh tokens, user passwords, or client secrets.

### 2.3.3. Client

In OAuth, a client is an application making protected resource requests on behalf of the resource owner and with its authorization. There are different types of clients with different implementation and security characteristics, such as web, user-agent-based, and native applications. A full definition of the different client types and profiles is given in [\[RFC6749\], Section 2.1](#).

The following data elements are stored or accessible on the client:

- o client id (and client secret or corresponding client credential)
- o one or more refresh tokens (persistent) and access tokens (transient) per end user or other security-context or delegation context
- o trusted certification authority (CA) certificates (HTTPS)
- o per-authorization process: "redirect\_uri", authorization "code"

## 3. Security Features

These are some of the security features that have been built into the OAuth 2.0 protocol to mitigate attacks and security issues.

### 3.1. Tokens

OAuth makes extensive use of many kinds of tokens (access tokens, refresh tokens, authorization "codes"). The information content of a token can be represented in two ways, as follows:

**Handle (or artifact)** A 'handle' is a reference to some internal data structure within the authorization server; the internal data structure contains the attributes of the token, such as user id (UID), scope, etc. Handles enable simple revocation and do not require cryptographic mechanisms to protect token content from being modified. On the other hand, handles require communication between the issuing and consuming entity (e.g., the authorization server and resource server) in order to validate the token and obtain token-bound data. This communication might have a negative impact on performance and scalability if both entities reside on different systems. Handles are therefore typically used if the issuing and consuming entity are the same. A 'handle' token is often referred to as an 'opaque' token because the resource server does not need to be able to interpret the token directly; it simply uses the token.

**Assertion (aka self-contained token)** An assertion is a parseable token. An assertion typically has a duration, has an audience, and is digitally signed in order to ensure data integrity and origin authentication. It contains information about the user and the client. Examples of assertion formats are Security Assertion Markup Language (SAML) assertions [[OASIS.saml-core-2.0-os](#)] and Kerberos tickets [[RFC4120](#)]. Assertions can typically be directly validated and used by a resource server without interactions with the authorization server. This results in better performance and scalability in deployments where the issuing and consuming entities reside on different systems. Implementing token revocation is more difficult with assertions than with handles.

Tokens can be used in two ways to invoke requests on resource servers, as follows:

**bearer token** A 'bearer token' is a token that can be used by any client who has received the token (e.g., [[RFC6750](#)]). Because mere possession is enough to use the token, it is important that communication between endpoints be secured to ensure that only authorized endpoints may capture the token. The bearer token is convenient for client applications, as it does not require them to do anything to use them (such as a proof of identity). Bearer tokens have similar characteristics to web single-sign-on (SSO) cookies used in browsers.

proof token A 'proof token' is a token that can only be used by a specific client. Each use of the token requires the client to perform some action that proves that it is the authorized user of the token. Examples of this are MAC-type access tokens, which require the client to digitally sign the resource request with a secret corresponding to the particular token sent with the request (e.g., [OAuth-HTTP-MAC]).

#### 3.1.1. Scope

A scope represents the access authorization associated with a particular token with respect to resource servers, resources, and methods on those resources. Scopes are the OAuth way to explicitly manage the power associated with an access token. A scope can be controlled by the authorization server and/or the end user in order to limit access to resources for OAuth clients that these parties deem less secure or trustworthy. Optionally, the client can request the scope to apply to the token but only for a lesser scope than would otherwise be granted, e.g., to reduce the potential impact if this token is sent over non-secure channels. A scope is typically complemented by a restriction on a token's lifetime.

#### 3.1.2. Limited Access Token Lifetime

The protocol parameter "expires\_in" allows an authorization server (based on its policies or on behalf of the end user) to limit the lifetime of an access token and to pass this information to the client. This mechanism can be used to issue short-lived tokens to OAuth clients that the authorization server deems less secure, or where sending tokens over non-secure channels.

#### 3.2. Access Token

An access token is used by a client to access a resource. Access tokens typically have short life spans (minutes or hours) that cover typical session lifetimes. An access token may be refreshed through the use of a refresh token. The short lifespan of an access token, in combination with the usage of refresh tokens, enables the possibility of passive revocation of access authorization on the expiry of the current access token.

#### 3.3. Refresh Token

A refresh token represents a long-lasting authorization of a certain client to access resources on behalf of a resource owner. Such tokens are exchanged between the client and authorization server only. Clients use this kind of token to obtain ("refresh") new access tokens used for resource server invocations.

A refresh token, coupled with a short access token lifetime, can be used to grant longer access to resources without involving end-user authorization. This offers an advantage where resource servers and authorization servers are not the same entity, e.g., in a distributed environment, as the refresh token is always exchanged at the authorization server. The authorization server can revoke the refresh token at any time, causing the granted access to be revoked once the current access token expires. Because of this, a short access token lifetime is important if timely revocation is a high priority.

The refresh token is also a secret bound to the client identifier and client instance that originally requested the authorization; the refresh token also represents the original resource owner grant. This is ensured by the authorization process as follows:

1. The resource owner and user agent safely deliver the authorization "code" to the client instance in the first place.
2. The client uses it immediately in secure transport-level communications to the authorization server and then securely stores the long-lived refresh token.
3. The client always uses the refresh token in secure transport-level communications to the authorization server to get an access token (and optionally roll over the refresh token).

So, as long as the confidentiality of the particular token can be ensured by the client, a refresh token can also be used as an alternative means to authenticate the client instance itself.

#### 3.4. Authorization "code"

An authorization "code" represents the intermediate result of a successful end-user authorization process and is used by the client to obtain access and refresh tokens. Authorization "codes" are sent to the client's redirect URI instead of tokens for two purposes:

1. Browser-based flows expose protocol parameters to potential attackers via URI query parameters (HTTP referrer), the browser cache, or log file entries, and could be replayed. In order to reduce this threat, short-lived authorization "codes" are passed instead of tokens and exchanged for tokens over a more secure direct connection between the client and the authorization server.

2. It is much simpler to authenticate clients during the direct request between the client and the authorization server than in the context of the indirect authorization request. The latter would require digital signatures.

### 3.5. Redirect URI

A redirect URI helps to detect malicious clients and prevents phishing attacks from clients attempting to trick the user into believing the phisher is the client. The value of the actual redirect URI used in the authorization request has to be presented and is verified when an authorization "code" is exchanged for tokens. This helps to prevent attacks where the authorization "code" is revealed through redirectors and counterfeit web application clients. The authorization server should require public clients and confidential clients using the implicit grant type to pre-register their redirect URIs and validate against the registered redirect URI in the authorization request.

### 3.6. "state" Parameter

The "state" parameter is used to link requests and callbacks to prevent cross-site request forgery attacks (see [Section 4.4.1.8](#)) where an attacker authorizes access to his own resources and then tricks a user into following a redirect with the attacker's token. This parameter should bind to the authenticated state in a user agent and, as per the core OAuth spec, the user agent must be capable of keeping it in a location accessible only by the client and user agent, i.e., protected by same-origin policy.

### 3.7. Client Identifier

Authentication protocols have typically not taken into account the identity of the software component acting on behalf of the end user. OAuth does this in order to increase the security level in delegated authorization scenarios and because the client will be able to act without the user being present.

OAuth uses the client identifier to collate associated requests to the same originator, such as

- o a particular end-user authorization process and the corresponding request on the token's endpoint to exchange the authorization "code" for tokens, or

- o the initial authorization and issuance of a token by an end user to a particular client, and subsequent requests by this client to obtain tokens without user consent (automatic processing of repeated authorizations)

This identifier may also be used by the authorization server to display relevant registration information to a user when requesting consent for a scope requested by a particular client. The client identifier may be used to limit the number of requests for a particular client or to charge the client per request. It may furthermore be useful to differentiate access by different clients, e.g., in server log files.

OAuth defines two client types, confidential and public, based on their ability to authenticate with the authorization server (i.e., ability to maintain the confidentiality of their client credentials). Confidential clients are capable of maintaining the confidentiality of client credentials (i.e., a client secret associated with the client identifier) or capable of secure client authentication using other means, such as a client assertion (e.g., SAML) or key cryptography. The latter is considered more secure.

The authorization server should determine whether the client is capable of keeping its secret confidential or using secure authentication. Alternatively, the end user can verify the identity of the client, e.g., by only installing trusted applications. The redirect URI can be used to prevent the delivery of credentials to a counterfeit client after obtaining end-user authorization in some cases but can't be used to verify the client identifier.

Clients can be categorized as follows based on the client type, profile (e.g., native vs. web application; see [\[RFC6749\]](#), [Section 9](#)), and deployment model:

Deployment-independent "client\_id" with pre-registered "redirect\_uri" and without "client\_secret" Such an identifier is used by multiple installations of the same software package. The identifier of such a client can only be validated with the help of the end-user. This is a viable option for native applications in order to identify the client for the purpose of displaying meta information about the client to the user and to differentiate clients in log files. Revocation of the rights associated with such a client identifier will affect ALL deployments of the respective software.

Deployment-independent "client\_id" with pre-registered "redirect\_uri" and with "client\_secret" This is an option for native applications only, since web applications would require different redirect URIs. This category is not advisable because the client secret cannot be protected appropriately (see [Section 4.1.1](#)). Due to its security weaknesses, such client identities have the same trust level as deployment-independent clients without secrets. Revocation will affect ALL deployments.

Deployment-specific "client\_id" with pre-registered "redirect\_uri" and with "client\_secret" The client registration process ensures the validation of the client's properties, such as redirect URI, web site URL, web site name, and contacts. Such a client identifier can be utilized for all relevant use cases cited above. This level can be achieved for web applications in combination with a manual or user-bound registration process. Achieving this level for native applications is much more difficult. Either the installation of the application is conducted by an administrator, who validates the client's authenticity, or the process from validating the application to the installation of the application on the device and the creation of the client credentials is controlled end-to-end by a single entity (e.g., application market provider). Revocation will affect a single deployment only.

Deployment-specific "client\_id" with "client\_secret" without validated properties Such a client can be recognized by the authorization server in transactions with subsequent requests (e.g., authorization and token issuance, refresh token issuance, and access token refreshment). The authorization server cannot assure any property of the client to end users. Automatic processing of re-authorizations could be allowed as well. Such client credentials can be generated automatically without any validation of client properties, which makes it another option, especially for native applications. Revocation will affect a single deployment only.

#### 4. Threat Model

This section gives a comprehensive threat model of OAuth 2.0. Threats are grouped first by attacks directed against an OAuth component, which are the client, authorization server, and resource server. Subsequently, they are grouped by flow, e.g., obtain token or access protected resources. Every countermeasure description refers to a detailed description in [Section 5](#).

## 4.1. Clients

This section describes possible threats directed to OAuth clients.

### 4.1.1. Threat: Obtaining Client Secrets

The attacker could try to get access to the secret of a particular client in order to:

- o replay its refresh tokens and authorization "codes", or
- o obtain tokens on behalf of the attacked client with the privileges of that "client\_id" acting as an instance of the client.

The resulting impact would be the following:

- o Client authentication of access to the authorization server can be bypassed.
- o Stolen refresh tokens or authorization "codes" can be replayed.

Depending on the client category, the following attacks could be utilized to obtain the client secret.

Attack: Obtain Secret From Source Code or Binary:

This applies for all client types. For open source projects, secrets can be extracted directly from source code in their public repositories. Secrets can be extracted from application binaries just as easily when the published source is not available to the attacker. Even if an application takes significant measures to obfuscate secrets in their application distribution, one should consider that the secret can still be reverse-engineered by anyone with access to a complete functioning application bundle or binary.

Countermeasures:

- o Don't issue secrets to public clients or clients with inappropriate security policy ([Section 5.2.3.1](#)).
- o Require user consent for public clients ([Section 5.2.3.2](#)).
- o Use deployment-specific client secrets ([Section 5.2.3.4](#)).
- o Revoke client secrets ([Section 5.2.3.6](#)).



Attack: Obtain a Deployment-Specific Secret:

An attacker may try to obtain the secret from a client installation, either from a web site (web server) or a particular device (native application).

Countermeasures:

- o Web server: Apply standard web server protection measures (for config files and databases) (see [Section 5.3.2](#)).
- o Native applications: Store secrets in secure local storage ([Section 5.3.3](#)).
- o Revoke client secrets ([Section 5.2.3.6](#)).

#### 4.1.2. Threat: Obtaining Refresh Tokens

Depending on the client type, there are different ways that refresh tokens may be revealed to an attacker. The following sub-sections give a more detailed description of the different attacks with respect to different client types and further specialized countermeasures. Before detailing those threats, here are some generally applicable countermeasures:

- o The authorization server should validate the client id associated with the particular refresh token with every refresh request ([Section 5.2.2.2](#)).
- o Limit token scope ([Section 5.1.5.1](#)).
- o Revoke refresh tokens ([Section 5.2.2.4](#)).
- o Revoke client secrets ([Section 5.2.3.6](#)).
- o Refresh tokens can automatically be replaced in order to detect unauthorized token usage by another party (see "Refresh Token Rotation", [Section 5.2.2.3](#)).

Attack: Obtain Refresh Token from Web Application:

An attacker may obtain the refresh tokens issued to a web application by way of overcoming the web server's security controls.

Impact: Since a web application manages the user accounts of a certain site, such an attack would result in an exposure of all refresh tokens on that site to the attacker.

Countermeasures:

- o Standard web server protection measures ([Section 5.3.2](#)).
- o Use strong client authentication (e.g., `client_assertion/`  
`client_token`) so the attacker cannot obtain the client secret  
required to exchange the tokens ([Section 5.2.3.7](#)).

Attack: Obtain Refresh Token from Native Clients:

On native clients, leakage of a refresh token typically affects a single user only.

Read from local file system: The attacker could try to get file system access on the device and read the refresh tokens. The attacker could utilize a malicious application for that purpose.

Countermeasures:

- o Store secrets in secure storage ([Section 5.3.3](#)).
- o Utilize device lock to prevent unauthorized device access  
([Section 5.3.4](#)).

Attack: Steal Device:

The host device (e.g., mobile phone) may be stolen. In that case, the attacker gets access to all applications under the identity of the legitimate user.

Countermeasures:

- o Utilize device lock to prevent unauthorized device access  
([Section 5.3.4](#)).
- o Where a user knows the device has been stolen, they can revoke the  
affected tokens ([Section 5.2.2.4](#)).

Attack: Clone Device:

All device data and applications are copied to another device. Applications are used as-is on the target device.

Countermeasures:

- o Utilize device lock to prevent unauthorized device access ([Section 5.3.4](#)).
- o Combine refresh token request with device identification ([Section 5.2.2.5](#)).
- o Refresh token rotation ([Section 5.2.2.3](#)).
- o Where a user knows the device has been cloned, they can use refresh token revocation ([Section 5.2.2.4](#)).

#### 4.1.3. Threat: Obtaining Access Tokens

Depending on the client type, there are different ways that access tokens may be revealed to an attacker. Access tokens could be stolen from the device if the application stores them in a storage device that is accessible to other applications.

Impact: Where the token is a bearer token and no additional mechanism is used to identify the client, the attacker can access all resources associated with the token and its scope.

Countermeasures:

- o Keep access tokens in transient memory and limit grants ([Section 5.1.6](#)).
- o Limit token scope ([Section 5.1.5.1](#)).
- o Keep access tokens in private memory or apply same protection means as for refresh tokens ([Section 5.2.2](#)).
- o Keep access token lifetime short ([Section 5.1.5.3](#)).

#### 4.1.4. Threat: End-User Credentials Phished Using Compromised or Embedded Browser

A malicious application could attempt to phish end-user passwords by misusing an embedded browser in the end-user authorization process, or by presenting its own user interface instead of allowing a trusted system browser to render the authorization user interface. By doing so, the usual visual trust mechanisms may be bypassed (e.g., Transport Layer Security (TLS) confirmation, web site mechanisms). By using an embedded or internal client application user interface, the client application has access to additional information to which it should not have access (e.g., UID/password).

Impact: If the client application or the communication is compromised, the user would not be aware of this, and all information in the authorization exchange, such as username and password, could be captured.

Countermeasures:

- o The OAuth flow is designed so that client applications never need to know user passwords. Client applications should avoid directly asking users for their credentials. In addition, end users could be educated about phishing attacks and best practices, such as only accessing trusted clients, as OAuth does not provide any protection against malicious applications and the end user is solely responsible for the trustworthiness of any native application installed.
- o Client applications could be validated prior to publication in an application market for users to access. That validation is out of scope for OAuth but could include validating that the client application handles user authentication in an appropriate way.
- o Client developers should not write client applications that collect authentication information directly from users and should instead delegate this task to a trusted system component, e.g., the system browser.

#### 4.1.5. Threat: Open Redirectors on Client

An open redirector is an endpoint using a parameter to automatically redirect a user agent to the location specified by the parameter value without any validation. If the authorization server allows the client to register only part of the redirect URI, an attacker can use an open redirector operated by the client to construct a redirect URI that will pass the authorization server validation but will send the authorization "code" or access token to an endpoint under the control of the attacker.

Impact: An attacker could gain access to authorization "codes" or access tokens.

Countermeasures:

- o Require clients to register full redirect URI ([Section 5.2.3.5](#)).

## 4.2. Authorization Endpoint

### 4.2.1. Threat: Password Phishing by Counterfeit Authorization Server

OAuth makes no attempt to verify the authenticity of the authorization server. A hostile party could take advantage of this by intercepting the client's requests and returning misleading or otherwise incorrect responses. This could be achieved using DNS or Address Resolution Protocol (ARP) spoofing. Wide deployment of OAuth and similar protocols may cause users to become inured to the practice of being redirected to web sites where they are asked to enter their passwords. If users are not careful to verify the authenticity of these web sites before entering their credentials, it will be possible for attackers to exploit this practice to steal users' passwords.

Countermeasures:

- o Authorization servers should consider such attacks when developing services based on OAuth and should require the use of transport-layer security for any requests where the authenticity of the authorization server or of request responses is an issue (see [Section 5.1.2](#)).
- o Authorization servers should attempt to educate users about the risks posed by phishing attacks and should provide mechanisms that make it easy for users to confirm the authenticity of their sites.

### 4.2.2. Threat: User Unintentionally Grants Too Much Access Scope

When obtaining end-user authorization, the end user may not understand the scope of the access being granted and to whom, or they may end up providing a client with access to resources that should not be permitted.

Countermeasures:

- o Explain the scope (resources and the permissions) the user is about to grant in an understandable way ([Section 5.2.4.2](#)).
- o Narrow the scope, based on the client. When obtaining end-user authorization and where the client requests scope, the authorization server may want to consider whether to honor that scope based on the client identifier. That decision is between the client and authorization server and is outside the scope of this spec. The authorization server may also want to consider what scope to grant based on the client type, e.g., providing lower scope to public clients ([Section 5.1.5.1](#)).

#### 4.2.3. Threat: Malicious Client Obtains Existing Authorization by Fraud

Authorization servers may wish to automatically process authorization requests from clients that have been previously authorized by the user. When the user is redirected to the authorization server's end-user authorization endpoint to grant access, the authorization server detects that the user has already granted access to that particular client. Instead of prompting the user for approval, the authorization server automatically redirects the user back to the client.

A malicious client may exploit that feature and try to obtain such an authorization "code" instead of the legitimate client.

Countermeasures:

- o Authorization servers should not automatically process repeat authorizations to public clients unless the client is validated using a pre-registered redirect URI ([Section 5.2.3.5](#)).
- o Authorization servers can mitigate the risks associated with automatic processing by limiting the scope of access tokens obtained through automated approvals ([Section 5.1.5.1](#)).

#### 4.2.4. Threat: Open Redirector

An attacker could use the end-user authorization endpoint and the redirect URI parameter to abuse the authorization server as an open redirector. An open redirector is an endpoint using a parameter to automatically redirect a user agent to the location specified by the parameter value without any validation.

Impact: An attacker could utilize a user's trust in an authorization server to launch a phishing attack.

Countermeasures:

- o Require clients to register any full redirect URIs ([Section 5.2.3.5](#)).
- o Don't redirect to a redirect URI if the client identifier or redirect URI can't be verified ([Section 5.2.3.5](#)).

### 4.3. Token Endpoint

#### 4.3.1. Threat: Eavesdropping Access Tokens

Attackers may attempt to eavesdrop access tokens in transit from the authorization server to the client.

Impact: The attacker is able to access all resources with the permissions covered by the scope of the particular access token.

Countermeasures:

- o As per the core OAuth spec, the authorization servers must ensure that these transmissions are protected using transport-layer mechanisms such as TLS (see [Section 5.1.1](#)).
- o If end-to-end confidentiality cannot be guaranteed, reducing scope (see [Section 5.1.5.1](#)) and expiry time ([Section 5.1.5.3](#)) for access tokens can be used to reduce the damage in case of leaks.

#### 4.3.2. Threat: Obtaining Access Tokens from Authorization Server Database

This threat is applicable if the authorization server stores access tokens as handles in a database. An attacker may obtain access tokens from the authorization server's database by gaining access to the database or launching a SQL injection attack.

Impact: Disclosure of all access tokens.

Countermeasures:

- o Enforce system security measures ([Section 5.1.4.1.1](#)).
- o Store access token hashes only ([Section 5.1.4.1.3](#)).
- o Enforce standard SQL injection countermeasures ([Section 5.1.4.1.2](#)).

#### 4.3.3. Threat: Disclosure of Client Credentials during Transmission

An attacker could attempt to eavesdrop the transmission of client credentials between the client and server during the client authentication process or during OAuth token requests.

Impact: Revelation of a client credential enabling phishing or impersonation of a client service.

Countermeasures:

- o The transmission of client credentials must be protected using transport-layer mechanisms such as TLS (see [Section 5.1.1](#)).
- o Use alternative authentication means that do not require the sending of plaintext credentials over the wire (e.g., Hash-based Message Authentication Code).

4.3.4. Threat: Obtaining Client Secret from Authorization Server Database

An attacker may obtain valid "client\_id"/secret combinations from the authorization server's database by gaining access to the database or launching a SQL injection attack.

Impact: Disclosure of all "client\_id"/secret combinations. This allows the attacker to act on behalf of legitimate clients.

Countermeasures:

- o Enforce system security measures ([Section 5.1.4.1.1](#)).
- o Enforce standard SQL injection countermeasures ([Section 5.1.4.1.2](#)).
- o Ensure proper handling of credentials as per "Enforce Credential Storage Protection Best Practices" ([Section 5.1.4.1](#)).

4.3.5. Threat: Obtaining Client Secret by Online Guessing

An attacker may try to guess valid "client\_id"/secret pairs.

Impact: Disclosure of a single "client\_id"/secret pair.

Countermeasures:

- o Use high entropy for secrets ([Section 5.1.4.2.2](#)).
- o Lock accounts ([Section 5.1.4.2.3](#)).
- o Use strong client authentication ([Section 5.2.3.7](#)).



#### 4.4. Obtaining Authorization

This section covers threats that are specific to certain flows utilized to obtain access tokens. Each flow is characterized by response types and/or grant types on the end-user authorization and token endpoint, respectively.

##### 4.4.1. Authorization "code"

###### 4.4.1.1. Threat: Eavesdropping or Leaking Authorization "codes"

An attacker could try to eavesdrop transmission of the authorization "code" between the authorization server and client. Furthermore, authorization "codes" are passed via the browser, which may unintentionally leak those codes to untrusted web sites and attackers in different ways:

- o Referrer headers: Browsers frequently pass a "referrer" header when a web page embeds content, or when a user travels from one web page to another web page. These referrer headers may be sent even when the origin site does not trust the destination site. The referrer header is commonly logged for traffic analysis purposes.
- o Request logs: Web server request logs commonly include query parameters on requests.
- o Open redirectors: Web sites sometimes need to send users to another destination via a redirector. Open redirectors pose a particular risk to web-based delegation protocols because the redirector can leak verification codes to untrusted destination sites.
- o Browser history: Web browsers commonly record visited URLs in the browser history. Another user of the same web browser may be able to view URLs that were visited by previous users.

Note: A description of similar attacks on the SAML protocol can be found at [OASIS.sstc-saml-bindings-1.1], Section 4.1.1.9.1; [Sec-Analysis]; and [OASIS.sstc-sec-analysis-response-01].

Countermeasures:

- o As per the core OAuth spec, the authorization server as well as the client must ensure that these transmissions are protected using transport-layer mechanisms such as TLS (see [Section 5.1.1](#)).
- o The authorization server will require the client to authenticate wherever possible, so the binding of the authorization "code" to a certain client can be validated in a reliable way (see [Section 5.2.4.4](#)).
- o Use short expiry time for authorization "codes" ([Section 5.1.5.3](#)).
- o The authorization server should enforce a one-time usage restriction (see [Section 5.1.5.4](#)).
- o If an authorization server observes multiple attempts to redeem an authorization "code", the authorization server may want to revoke all tokens granted based on the authorization "code" (see [Section 5.2.1.1](#)).
- o In the absence of these countermeasures, reducing scope ([Section 5.1.5.1](#)) and expiry time ([Section 5.1.5.3](#)) for access tokens can be used to reduce the damage in case of leaks.
- o The client server may reload the target page of the redirect URI in order to automatically clean up the browser cache.

4.4.1.2. Threat: Obtaining Authorization "codes" from Authorization Server Database

This threat is applicable if the authorization server stores authorization "codes" as handles in a database. An attacker may obtain authorization "codes" from the authorization server's database by gaining access to the database or launching a SQL injection attack.

Impact: Disclosure of all authorization "codes", most likely along with the respective "redirect\_uri" and "client\_id" values.

Countermeasures:

- o Best practices for credential storage protection should be employed ([Section 5.1.4.1](#)).
- o Enforce system security measures ([Section 5.1.4.1.1](#)).

- o Store access token hashes only ([Section 5.1.4.1.3](#)).
- o Enforce standard SQL injection countermeasures ([Section 5.1.4.1.2](#)).

#### 4.4.1.3. Threat: Online Guessing of Authorization "codes"

An attacker may try to guess valid authorization "code" values and send the guessed code value using the grant type "code" in order to obtain a valid access token.

Impact: Disclosure of a single access token and probably also an associated refresh token.

Countermeasures:

- o Handle-based tokens must use high entropy ([Section 5.1.4.2.2](#)).
- o Assertion-based tokens should be signed ([Section 5.1.5.9](#)).
- o Authenticate the client; this adds another value that the attacker has to guess ([Section 5.2.3.4](#)).
- o Bind the authorization "code" to the redirect URI; this adds another value that the attacker has to guess ([Section 5.2.4.5](#)).
- o Use short expiry time for tokens ([Section 5.1.5.3](#)).

#### 4.4.1.4. Threat: Malicious Client Obtains Authorization

A malicious client could pretend to be a valid client and obtain an access authorization in this way. The malicious client could even utilize screen-scraping techniques in order to simulate a user's consent in the authorization flow.

Assumption: It is not the task of the authorization server to protect the end-user's device from malicious software. This is the responsibility of the platform running on the particular device, probably in cooperation with other components of the respective ecosystem (e.g., an application management infrastructure). The sole responsibility of the authorization server is to control access to the end-user's resources maintained in resource servers and to prevent unauthorized access to them via the OAuth protocol. Based on this assumption, the following countermeasures are available to cope with the threat.

## Countermeasures:

- o The authorization server should authenticate the client, if possible (see [Section 5.2.3.4](#)). Note: The authentication takes place after the end user has authorized the access.
- o The authorization server should validate the client's redirect URI against the pre-registered redirect URI, if one exists (see [Section 5.2.3.5](#)). Note: An invalid redirect URI indicates an invalid client, whereas a valid redirect URI does not necessarily indicate a valid client. The level of confidence depends on the client type. For web applications, the level of confidence is high, since the redirect URI refers to the globally unique network endpoint of this application, whose fully qualified domain name (FQDN) is also validated using HTTPS server authentication by the user agent. In contrast, for native clients, the redirect URI typically refers to device local resources, e.g., a custom scheme. So, a malicious client on a particular device can use the valid redirect URI the legitimate client uses on all other devices.
- o After authenticating the end user, the authorization server should ask him/her for consent. In this context, the authorization server should explain to the end user the purpose, scope, and duration of the authorization the client asked for. Moreover, the authorization server should show the user any identity information it has for that client. It is up to the user to validate the binding of this data to the particular application (e.g., Name) and to approve the authorization request (see [Section 5.2.4.3](#)).
- o The authorization server should not perform automatic re-authorizations for clients it is unable to reliably authenticate or validate (see [Section 5.2.4.1](#)).
- o If the authorization server automatically authenticates the end user, it may nevertheless require some user input in order to prevent screen scraping. Examples are CAPTCHAs (Completely Automated Public Turing tests to tell Computers and Humans Apart) or other multi-factor authentication techniques such as random questions, token code generators, etc.
- o The authorization server may also limit the scope of tokens it issues to clients it cannot reliably authenticate (see [Section 5.1.5.1](#)).

#### 4.4.1.5. Threat: Authorization "code" Phishing

A hostile party could impersonate the client site and get access to the authorization "code". This could be achieved using DNS or ARP spoofing. This applies to clients, which are web applications; thus, the redirect URI is not local to the host where the user's browser is running.

Impact: This affects web applications and may lead to a disclosure of authorization "codes" and, potentially, the corresponding access and refresh tokens.

Countermeasures:

It is strongly recommended that one of the following countermeasures be utilized in order to prevent this attack:

- o The redirect URI of the client should point to an HTTPS-protected endpoint, and the browser should be utilized to authenticate this redirect URI using server authentication (see [Section 5.1.2](#)).
- o The authorization server should require that the client be authenticated, i.e., confidential client, so the binding of the authorization "code" to a certain client can be validated in a reliable way (see [Section 5.2.4.4](#)).

#### 4.4.1.6. Threat: User Session Impersonation

A hostile party could impersonate the client site and impersonate the user's session on this client. This could be achieved using DNS or ARP spoofing. This applies to clients, which are web applications; thus, the redirect URI is not local to the host where the user's browser is running.

Impact: An attacker who intercepts the authorization "code" as it is sent by the browser to the callback endpoint can gain access to protected resources by submitting the authorization "code" to the client. The client will exchange the authorization "code" for an access token and use the access token to access protected resources for the benefit of the attacker, delivering protected resources to the attacker, or modifying protected resources as directed by the attacker. If OAuth is used by the client to delegate authentication to a social site (e.g., as in the implementation of a "Login" button on a third-party social network site), the attacker can use the intercepted authorization "code" to log into the client as the user.

Note: Authenticating the client during authorization "code" exchange will not help to detect such an attack, as it is the legitimate client that obtains the tokens.

Countermeasures:

- o In order to prevent an attacker from impersonating the end-user's session, the redirect URI of the client should point to an HTTPS protected endpoint, and the browser should be utilized to authenticate this redirect URI using server authentication (see [Section 5.1.2](#)).

#### 4.4.1.7. Threat: Authorization "code" Leakage through Counterfeit Client

The attacker leverages the authorization "code" grant type in an attempt to get another user (victim) to log in, authorize access to his/her resources, and subsequently obtain the authorization "code" and inject it into a client application using the attacker's account. The goal is to associate an access authorization for resources of the victim with the user account of the attacker on a client site.

The attacker abuses an existing client application and combines it with his own counterfeit client web site. The attacker depends on the victim expecting the client application to request access to a certain resource server. The victim, seeing only a normal request from an expected application, approves the request. The attacker then uses the victim's authorization to gain access to the information unknowingly authorized by the victim.

The attacker conducts the following flow:

1. The attacker accesses the client web site (or application) and initiates data access to a particular resource server. The client web site in turn initiates an authorization request to the resource server's authorization server. Instead of proceeding with the authorization process, the attacker modifies the authorization server end-user authorization URL as constructed by the client to include a redirect URI parameter referring to a web site under his control (attacker's web site).
2. The attacker tricks another user (the victim) into opening that modified end-user authorization URI and authorizing access (e.g., via an email link or blog link). The way the attacker achieves this goal is out of scope.
3. Having clicked the link, the victim is requested to authenticate and authorize the client site to have access.

4. After completion of the authorization process, the authorization server redirects the user agent to the attacker's web site instead of the original client web site.
5. The attacker obtains the authorization "code" from his web site by means that are out of scope of this document.
6. He then constructs a redirect URI to the target web site (or application) based on the original authorization request's redirect URI and the newly obtained authorization "code", and directs his user agent to this URL. The authorization "code" is injected into the original client site (or application).
7. The client site uses the authorization "code" to fetch a token from the authorization server and associates this token with the attacker's user account on this site.
8. The attacker may now access the victim's resources using the client site.

Impact: The attacker gains access to the victim's resources as associated with his account on the client site.

Countermeasures:

- o The attacker will need to use another redirect URI for its authorization process rather than the target web site because it needs to intercept the flow. So, if the authorization server associates the authorization "code" with the redirect URI of a particular end-user authorization and validates this redirect URI with the redirect URI passed to the token's endpoint, such an attack is detected (see [Section 5.2.4.5](#)).
- o The authorization server may also enforce the usage and validation of pre-registered redirect URIs (see [Section 5.2.3.5](#)). This will allow for early recognition of authorization "code" disclosure to counterfeit clients.
- o For native applications, one could also consider using deployment-specific client ids and secrets (see [Section 5.2.3.4](#)), along with the binding of authorization "codes" to "client\_ids" (see [Section 5.2.4.4](#)) to detect such an attack because the attacker does not have access to the deployment-specific secret. Thus, he will not be able to exchange the authorization "code".

- o The client may consider using other flows that are not vulnerable to this kind of attack, such as the implicit grant type (see [Section 4.4.2](#)) or resource owner password credentials (see [Section 4.4.3](#)).

#### 4.4.1.8. Threat: CSRF Attack against redirect-uri

Cross-site request forgery (CSRF) is a web-based attack whereby HTTP requests are transmitted from a user that the web site trusts or has authenticated (e.g., via HTTP redirects or HTML forms). CSRF attacks on OAuth approvals can allow an attacker to obtain authorization to OAuth protected resources without the consent of the user.

This attack works against the redirect URI used in the authorization "code" flow. An attacker could authorize an authorization "code" to their own protected resources on an authorization server. He then aborts the redirect flow back to the client on his device and tricks the victim into executing the redirect back to the client. The client receives the redirect, fetches the token(s) from the authorization server, and associates the victim's client session with the resources accessible using the token.

Impact: The user accesses resources on behalf of the attacker. The effective impact depends on the type of resource accessed. For example, the user may upload private items to an attacker's resources. Or, when using OAuth in 3rd-party login scenarios, the user may associate his client account with the attacker's identity at the external Identity Provider. In this way, the attacker could easily access the victim's data at the client by logging in from another device with his credentials at the external Identity Provider.

#### Countermeasures:

- o The "state" parameter should be used to link the authorization request with the redirect URI used to deliver the access token ([Section 5.3.5](#)).
- o Client developers and end users can be educated to not follow untrusted URLs.



#### 4.4.1.9. Threat: Clickjacking Attack against Authorization

With clickjacking, a malicious site loads the target site in a transparent iFrame (see [iFrame]) overlaid on top of a set of dummy buttons that are carefully constructed to be placed directly under important buttons on the target site. When a user clicks a visible button, they are actually clicking a button (such as an "Authorize" button) on the hidden page.

Impact: An attacker can steal a user's authentication credentials and access their resources.

Countermeasures:

- o For newer browsers, avoidance of iFrames during authorization can be enforced on the server side by using the X-FRAME-OPTIONS header (Section 5.2.2.6).
- o For older browsers, JavaScript frame-busting (see [Framebusting]) techniques can be used but may not be effective in all browsers.

#### 4.4.1.10. Threat: Resource Owner Impersonation

When a client requests access to protected resources, the authorization flow normally involves the resource owner's explicit response to the access request, either granting or denying access to the protected resources. A malicious client can exploit knowledge of the structure of this flow in order to gain authorization without the resource owner's consent, by transmitting the necessary requests programmatically and simulating the flow against the authorization server. That way, the client may gain access to the victim's resources without her approval. An authorization server will be vulnerable to this threat if it uses non-interactive authentication mechanisms or splits the authorization flow across multiple pages.

The malicious client might embed a hidden HTML user agent, interpret the HTML forms sent by the authorization server, and automatically send the corresponding form HTTP POST requests. As a prerequisite, the attacker must be able to execute the authorization process in the context of an already-authenticated session of the resource owner with the authorization server. There are different ways to achieve this:

- o The malicious client could abuse an existing session in an external browser or cross-browser cookies on the particular device.

- o The malicious client could also request authorization for an initial scope acceptable to the user and then silently abuse the resulting session in his browser instance to "silently" request another scope.
- o Alternatively, the attacker might exploit an authorization server's ability to authenticate the resource owner automatically and without user interactions, e.g., based on certificates.

In all cases, such an attack is limited to clients running on the victim's device, either within the user agent or as a native app.

Please note: Such attacks cannot be prevented using CSRF countermeasures, since the attacker just "executes" the URLs as prepared by the authorization server including any nonce, etc.

Countermeasures:

Authorization servers should decide, based on an analysis of the risk associated with this threat, whether to detect and prevent this threat.

In order to prevent such an attack, the authorization server may force a user interaction based on non-predictable input values as part of the user consent approval. The authorization server could

- o combine password authentication and user consent in a single form,
- o make use of CAPTCHAs, or
- o use one-time secrets sent out of band to the resource owner (e.g., via text or instant message).

Alternatively, in order to allow the resource owner to detect abuse, the authorization server could notify the resource owner of any approval by appropriate means, e.g., text or instant message, or email.

#### 4.4.1.11. Threat: DoS Attacks That Exhaust Resources

If an authorization server includes a nontrivial amount of entropy in authorization "codes" or access tokens (limiting the number of possible codes/tokens) and automatically grants either without user intervention and has no limit on codes or access tokens per user, an attacker could exhaust the pool of authorization "codes" by repeatedly directing the user's browser to request authorization "codes" or access tokens.

Countermeasures:

- o The authorization server should consider limiting the number of access tokens granted per user.
- o The authorization server should include a nontrivial amount of entropy in authorization "codes".

4.4.1.12. Threat: DoS Using Manufactured Authorization "codes"

An attacker who owns a botnet can locate the redirect URIs of clients that listen on HTTP, access them with random authorization "codes", and cause a large number of HTTPS connections to be concentrated onto the authorization server. This can result in a denial-of-service (DoS) attack on the authorization server.

This attack can still be effective even when CSRF defense/the "state" parameter (see [Section 4.4.1.8](#)) is deployed on the client side. With such a defense, the attacker might need to incur an additional HTTP request to obtain a valid CSRF code/"state" parameter. This apparently cuts down the effectiveness of the attack by a factor of 2. However, if the HTTPS/HTTP cost ratio is higher than 2 (the cost factor is estimated to be around 3.5x at [\[SSL-Latency\]](#)), the attacker still achieves a magnification of resource utilization at the expense of the authorization server.

Impact: There are a few effects that the attacker can accomplish with this OAuth flow that they cannot easily achieve otherwise.

1. Connection laundering: With the clients as the relay between the attacker and the authorization server, the authorization server learns little or no information about the identity of the attacker. Defenses such as rate-limiting on the offending attacker machines are less effective because it is difficult to identify the attacking machines. Although an attacker could also launder its connections through an anonymizing system such as Tor, the effectiveness of that approach depends on the capacity of the anonymizing system. On the other hand, a potentially large number of OAuth clients could be utilized for this attack.
2. Asymmetric resource utilization: The attacker incurs the cost of an HTTP connection and causes an HTTPS connection to be made on the authorization server; the attacker can coordinate the timing of such HTTPS connections across multiple clients relatively easily. Although the attacker could achieve something similar, say, by including an iFrame pointing to the HTTPS URL of the authorization server in an HTTP web page and luring web users to visit that page, timing attacks using such a scheme may be more

difficult, as it seems nontrivial to synchronize a large number of users to simultaneously visit a particular site under the attacker's control.

Countermeasures:

- o Though not a complete countermeasure by themselves, CSRF defense and the "state" parameter created with secure random codes should be deployed on the client side. The client should forward the authorization "code" to the authorization server only after both the CSRF token and the "state" parameter are validated.
- o If the client authenticates the user, either through a single-sign-on protocol or through local authentication, the client should suspend the access by a user account if the number of invalid authorization "codes" submitted by this user exceeds a certain threshold.
- o The authorization server should send an error response to the client reporting an invalid authorization "code" and rate-limit or disallow connections from clients whose number of invalid requests exceeds a threshold.

#### 4.4.1.13. Threat: Code Substitution (OAuth Login)

An attacker could attempt to log into an application or web site using a victim's identity. Applications relying on identity data provided by an OAuth protected service API to login users are vulnerable to this threat. This pattern can be found in so-called "social login" scenarios.

As a prerequisite, a resource server offers an API to obtain personal information about a user that could be interpreted as having obtained a user identity. In this sense, the client is treating the resource server API as an "identity" API. A client utilizes OAuth to obtain an access token for the identity API. It then queries the identity API for an identifier and uses it to look up its internal user account data (login). The client assumes that, because it was able to obtain information about the user, the user has been authenticated.

If the client uses the grant type "code", the attacker needs to gather a valid authorization "code" of the respective victim from the same Identity Provider used by the target client application. The attacker tricks the victim into logging into a malicious app (which may appear to be legitimate to the Identity Provider) using the same Identity Provider as the target application. This results in the Identity Provider's authorization server issuing an authorization

"code" for the respective identity API. The malicious app then sends this code to the attacker, which in turn triggers a login process within the target application. The attacker now manipulates the authorization response and substitutes their code (bound to their identity) for the victim's code. This code is then exchanged by the client for an access token, which in turn is accepted by the identity API, since the audience, with respect to the resource server, is correct. But since the identifier returned by the identity API is determined by the identity in the access token (issued based on the victim's code), the attacker is logged into the target application under the victim's identity.

Impact: The attacker gains access to an application and user-specific data within the application.

Countermeasures:

- o All clients must indicate their client ids with every request to exchange an authorization "code" for an access token. The authorization server must validate whether the particular authorization "code" has been issued to the particular client. If possible, the client shall be authenticated beforehand.
- o Clients should use an appropriate protocol, such as OpenID (cf. [OPENID]) or SAML (cf. [OASIS.sstc-saml-bindings-1.1]) to implement user login. Both support audience restrictions on clients.

#### 4.4.2. Implicit Grant

In the implicit grant type flow, the access token is directly returned to the client as a fragment part of the redirect URI. It is assumed that the token is not sent to the redirect URI target, as HTTP user agents do not send the fragment part of URIs to HTTP servers. Thus, an attacker cannot eavesdrop the access token on this communication path, and the token cannot leak through HTTP referrer headers.

##### 4.4.2.1. Threat: Access Token Leak in Transport/Endpoints

This token might be eavesdropped by an attacker. The token is sent from the server to the client via a URI fragment of the redirect URI. If the communication is not secured or the endpoint is not secured, the token could be leaked by parsing the returned URI.

Impact: The attacker would be able to assume the same rights granted by the token.

Countermeasures:

- o The authorization server should ensure confidentiality (e.g., using TLS) of the response from the authorization server to the client (see [Section 5.1.1](#)).

#### 4.4.2.2. Threat: Access Token Leak in Browser History

An attacker could obtain the token from the browser's history. Note that this means the attacker needs access to the particular device.

Countermeasures:

- o Use short expiry time for tokens (see [Section 5.1.5.3](#)). Reduced scope of the token may reduce the impact of that attack (see [Section 5.1.5.1](#)).
- o Make responses non-cacheable.

#### 4.4.2.3. Threat: Malicious Client Obtains Authorization

A malicious client could attempt to obtain a token by fraud.

The same countermeasures as for [Section 4.4.1.4](#) are applicable, except client authentication.

#### 4.4.2.4. Threat: Manipulation of Scripts

A hostile party could act as the client web server and replace or modify the actual implementation of the client (script). This could be achieved using DNS or ARP spoofing. This applies to clients implemented within the web browser in a scripting language.

Impact: The attacker could obtain user credential information and assume the full identity of the user.

Countermeasures:

- o The authorization server should authenticate the server from which scripts are obtained (see [Section 5.1.2](#)).
- o The client should ensure that scripts obtained have not been altered in transport (see [Section 5.1.1](#)).

- o Introduce one-time, per-use secrets (e.g., "client\_secret") values that can only be used by scripts in a small time window once loaded from a server. The intention would be to reduce the effectiveness of copying client-side scripts for re-use in an attacker's modified code.

#### 4.4.2.5. Threat: CSRF Attack against redirect-uri

CSRF attacks (see [Section 4.4.1.8](#)) also work against the redirect URI used in the implicit grant flow. An attacker could acquire an access token to their own protected resources. He could then construct a redirect URI and embed their access token in that URI. If he can trick the user into following the redirect URI and the client does not have protection against this attack, the user may have the attacker's access token authorized within their client.

Impact: The user accesses resources on behalf of the attacker. The effective impact depends on the type of resource accessed. For example, the user may upload private items to an attacker's resources. Or, when using OAuth in 3rd-party login scenarios, the user may associate his client account with the attacker's identity at the external Identity Provider. In this way, the attacker could easily access the victim's data at the client by logging in from another device with his credentials at the external Identity Provider.

Countermeasures:

- o The "state" parameter should be used to link the authorization request with the redirect URI used to deliver the access token. This will ensure that the client is not tricked into completing any redirect callback unless it is linked to an authorization request initiated by the client. The "state" parameter should not be guessable, and the client should be capable of keeping the "state" parameter secret.
- o Client developers and end users can be educated to not follow untrusted URLs.

#### 4.4.2.6. Threat: Token Substitution (OAuth Login)

An attacker could attempt to log into an application or web site using a victim's identity. Applications relying on identity data provided by an OAuth protected service API to login users are vulnerable to this threat. This pattern can be found in so-called "social login" scenarios.

As a prerequisite, a resource server offers an API to obtain personal information about a user that could be interpreted as having obtained a user identity. In this sense, the client is treating the resource server API as an "identity" API. A client utilizes OAuth to obtain an access token for the identity API. It then queries the identity API for an identifier and uses it to look up its internal user account data (login). The client assumes that, because it was able to obtain information about the user, the user has been authenticated.

To succeed, the attacker needs to gather a valid access token of the respective victim from the same Identity Provider used by the target client application. The attacker tricks the victim into logging into a malicious app (which may appear to be legitimate to the Identity Provider) using the same Identity Provider as the target application. This results in the Identity Provider's authorization server issuing an access token for the respective identity API. The malicious app then sends this access token to the attacker, which in turn triggers a login process within the target application. The attacker now manipulates the authorization response and substitutes their access token (bound to their identity) for the victim's access token. This token is accepted by the identity API, since the audience, with respect to the resource server, is correct. But since the identifier returned by the identity API is determined by the identity in the access token, the attacker is logged into the target application under the victim's identity.

Impact: The attacker gains access to an application and user-specific data within the application.

Countermeasures:

- o Clients should use an appropriate protocol, such as OpenID (cf. [OPENID]) or SAML (cf. [OASIS.sstc-saml-bindings-1.1]) to implement user login. Both support audience restrictions on clients.

#### 4.4.3. Resource Owner Password Credentials

The resource owner password credentials grant type (see [RFC6749], Section 4.3), often used for legacy/migration reasons, allows a client to request an access token using an end-user's user id and password along with its own credential. This grant type has higher risk because it maintains the UID/password anti-pattern. Additionally, because the user does not have control over the authorization process, clients using this grant type are not limited



by scope but instead have potentially the same capabilities as the user themselves. As there is no authorization step, the ability to offer token revocation is bypassed.

Because passwords are often used for more than 1 service, this anti-pattern may also put at risk whatever else is accessible with the supplied credential. Additionally, any easily derived equivalent (e.g., joe@example.com and joe@example.net) might easily allow someone to guess that the same password can be used elsewhere.

**Impact:** The resource server can only differentiate scope based on the access token being associated with a particular client. The client could also acquire long-lived tokens and pass them up to an attacker's web service for further abuse. The client, eavesdroppers, or endpoints could eavesdrop the user id and password.

**Countermeasures:**

- o Except for migration reasons, minimize use of this grant type.
- o The authorization server should validate the client id associated with the particular refresh token with every refresh request ([Section 5.2.2.2](#)).
- o As per the core OAuth specification, the authorization server must ensure that these transmissions are protected using transport-layer mechanisms such as TLS (see [Section 5.1.1](#)).
- o Rather than encouraging users to use a UID and password, service providers should instead encourage users not to use the same password for multiple services.
- o Limit use of resource owner password credential grants to scenarios where the client application and the authorizing service are from the same organization.

#### [4.4.3.1](#). Threat: Accidental Exposure of Passwords at Client Site

If the client does not provide enough protection, an attacker or disgruntled employee could retrieve the passwords for a user.

**Countermeasures:**

- o Use other flows that do not rely on the client's cooperation for secure resource owner credential handling.
- o Use digest authentication instead of plaintext credential processing.

- o Obfuscate passwords in logs.

#### 4.4.3.2. Threat: Client Obtains Scopes without End-User Authorization

All interaction with the resource owner is performed by the client. Thus it might, intentionally or unintentionally, happen that the client obtains a token with scope unknown for, or unintended by, the resource owner. For example, the resource owner might think the client needs and acquires read-only access to its media storage only but the client tries to acquire an access token with full access permissions.

Countermeasures:

- o Use other flows that do not rely on the client's cooperation for resource owner interaction.
- o The authorization server may generally restrict the scope of access tokens ([Section 5.1.5.1](#)) issued by this flow. If the particular client is trustworthy and can be authenticated in a reliable way, the authorization server could relax that restriction. Resource owners may prescribe (e.g., in their preferences) what the maximum scope is for clients using this flow.
- o The authorization server could notify the resource owner by an appropriate medium, e.g., email, of the grant issued (see [Section 5.1.3](#)).

#### 4.4.3.3. Threat: Client Obtains Refresh Token through Automatic Authorization

All interaction with the resource owner is performed by the client. Thus it might, intentionally or unintentionally, happen that the client obtains a long-term authorization represented by a refresh token even if the resource owner did not intend so.

Countermeasures:

- o Use other flows that do not rely on the client's cooperation for resource owner interaction.
- o The authorization server may generally refuse to issue refresh tokens in this flow (see [Section 5.2.2.1](#)). If the particular client is trustworthy and can be authenticated in a reliable way (see client authentication), the authorization server could relax

that restriction. Resource owners may allow or deny (e.g., in their preferences) the issuing of refresh tokens using this flow as well.

- o The authorization server could notify the resource owner by an appropriate medium, e.g., email, of the refresh token issued (see [Section 5.1.3](#)).

#### 4.4.3.4. Threat: Obtaining User Passwords on Transport

An attacker could attempt to eavesdrop the transmission of end-user credentials with the grant type "password" between the client and server.

Impact: Disclosure of a single end-user's password.

Countermeasures:

- o Ensure confidentiality of requests ([Section 5.1.1](#)).
- o Use alternative authentication means that do not require the sending of plaintext credentials over the wire (e.g., Hash-based Message Authentication Code).

#### 4.4.3.5. Threat: Obtaining User Passwords from Authorization Server Database

An attacker may obtain valid username/password combinations from the authorization server's database by gaining access to the database or launching a SQL injection attack.

Impact: Disclosure of all username/password combinations. The impact may exceed the domain of the authorization server, since many users tend to use the same credentials on different services.

Countermeasures:

- o Enforce credential storage protection best practices ([Section 5.1.4.1](#)).

#### 4.4.3.6. Threat: Online Guessing

An attacker may try to guess valid username/password combinations using the grant type "password".

Impact: Revelation of a single username/password combination.

Countermeasures:

- o Utilize secure password policy ([Section 5.1.4.2.1](#)).
- o Lock accounts ([Section 5.1.4.2.3](#)).
- o Use tar pit ([Section 5.1.4.2.4](#)).
- o Use CAPTCHAs ([Section 5.1.4.2.5](#)).
- o Consider not using the grant type "password".
- o Client authentication (see [Section 5.2.3](#)) will provide another authentication factor and thus hinder the attack.

#### 4.4.4. Client Credentials

Client credentials (see [\[RFC6749\]](#), [Section 3](#)) consist of an identifier (not secret) combined with an additional means (such as a matching client secret) of authenticating a client. The threats to this grant type are similar to those described in [Section 4.4.3](#).

#### 4.5. Refreshing an Access Token

##### 4.5.1. Threat: Eavesdropping Refresh Tokens from Authorization Server

An attacker may eavesdrop refresh tokens when they are transmitted from the authorization server to the client.

Countermeasures:

- o As per the core OAuth spec, the authorization servers must ensure that these transmissions are protected using transport-layer mechanisms such as TLS (see [Section 5.1.1](#)).
- o If end-to-end confidentiality cannot be guaranteed, reducing scope (see [Section 5.1.5.1](#)) and expiry time (see [Section 5.1.5.3](#)) for issued access tokens can be used to reduce the damage in case of leaks.

##### 4.5.2. Threat: Obtaining Refresh Token from Authorization Server Database

This threat is applicable if the authorization server stores refresh tokens as handles in a database. An attacker may obtain refresh tokens from the authorization server's database by gaining access to the database or launching a SQL injection attack.

Impact: Disclosure of all refresh tokens.

Countermeasures:

- o Enforce credential storage protection best practices ([Section 5.1.4.1](#)).
- o Bind token to client id, if the attacker cannot obtain the required id and secret ([Section 5.1.5.8](#)).

#### 4.5.3. Threat: Obtaining Refresh Token by Online Guessing

An attacker may try to guess valid refresh token values and send it using the grant type "refresh\_token" in order to obtain a valid access token.

Impact: Exposure of a single refresh token and derivable access tokens.

Countermeasures:

- o For handle-based designs ([Section 5.1.4.2.2](#)).
- o For assertion-based designs ([Section 5.1.5.9](#)).
- o Bind token to client id, because the attacker would guess the matching client id, too (see [Section 5.1.5.8](#)).
- o Authenticate the client; this adds another element that the attacker has to guess (see [Section 5.2.3.4](#)).

#### 4.5.4. Threat: Refresh Token Phishing by Counterfeit Authorization Server

An attacker could try to obtain valid refresh tokens by proxying requests to the authorization server. Given the assumption that the authorization server URL is well-known at development time or can at least be obtained from a well-known resource server, the attacker must utilize some kind of spoofing in order to succeed.

Countermeasures:

- o Utilize server authentication (as described in [Section 5.1.2](#)).

## 4.6. Accessing Protected Resources

### 4.6.1. Threat: Eavesdropping Access Tokens on Transport

An attacker could try to obtain a valid access token on transport between the client and resource server. As access tokens are shared secrets between the authorization server and resource server, they should be treated with the same care as other credentials (e.g., end-user passwords).

Countermeasures:

- o Access tokens sent as bearer tokens should not be sent in the clear over an insecure channel. As per the core OAuth spec, transmission of access tokens must be protected using transport-layer mechanisms such as TLS (see [Section 5.1.1](#)).
- o A short lifetime reduces impact in case tokens are compromised (see [Section 5.1.5.3](#)).
- o The access token can be bound to a client's identifier and require the client to prove legitimate ownership of the token to the resource server (see [Section 5.4.2](#)).

### 4.6.2. Threat: Replay of Authorized Resource Server Requests

An attacker could attempt to replay valid requests in order to obtain or to modify/destroy user data.

Countermeasures:

- o The resource server should utilize transport security measures (e.g., TLS) in order to prevent such attacks (see [Section 5.1.1](#)). This would prevent the attacker from capturing valid requests.
- o Alternatively, the resource server could employ signed requests (see [Section 5.4.3](#)) along with nonces and timestamps in order to uniquely identify requests. The resource server should detect and refuse every replayed request.

### 4.6.3. Threat: Guessing Access Tokens

Where the token is a handle, the attacker may attempt to guess the access token values based on knowledge they have from other access tokens.

Impact: Access to a single user's data.

Countermeasures:

- o Handle tokens should have a reasonable level of entropy (see [Section 5.1.4.2.2](#)) in order to make guessing a valid token value infeasible.
- o Assertion (or self-contained token) token contents should be protected by a digital signature (see [Section 5.1.5.9](#)).
- o Security can be further strengthened by using a short access token duration (see [Sections 5.1.5.2](#) and [5.1.5.3](#)).

#### 4.6.4. Threat: Access Token Phishing by Counterfeit Resource Server

An attacker may pretend to be a particular resource server and to accept tokens from a particular authorization server. If the client sends a valid access token to this counterfeit resource server, the server in turn may use that token to access other services on behalf of the resource owner.

Countermeasures:

- o Clients should not make authenticated requests with an access token to unfamiliar resource servers, regardless of the presence of a secure channel. If the resource server URL is well-known to the client, it may authenticate the resource servers (see [Section 5.1.2](#)).
- o Associate the endpoint URL of the resource server the client talked to with the access token (e.g., in an audience field) and validate the association at a legitimate resource server. The endpoint URL validation policy may be strict (exact match) or more relaxed (e.g., same host). This would require telling the authorization server about the resource server endpoint URL in the authorization process.
- o Associate an access token with a client and authenticate the client with resource server requests (typically via a signature, in order to not disclose a secret to a potential attacker). This prevents the attack because the counterfeit server is assumed to lack the capability to correctly authenticate on behalf of the legitimate client to the resource server ([Section 5.4.2](#)).
- o Restrict the token scope (see [Section 5.1.5.1](#)) and/or limit the token to a certain resource server ([Section 5.1.5.5](#)).

#### 4.6.5. Threat: Abuse of Token by Legitimate Resource Server or Client

A legitimate resource server could attempt to use an access token to access another resource server. Similarly, a client could try to use a token obtained for one server on another resource server.

Countermeasures:

- o Tokens should be restricted to particular resource servers (see [Section 5.1.5.5](#)).

#### 4.6.6. Threat: Leak of Confidential Data in HTTP Proxies

An OAuth HTTP authentication scheme as discussed in [[RFC6749](#)] is optional. However, [[RFC2616](#)] relies on the Authorization and WWW-Authenticate headers to distinguish authenticated content so that it can be protected. Proxies and caches, in particular, may fail to adequately protect requests not using these headers. For example, private authenticated content may be stored in (and thus be retrievable from) publicly accessible caches.

Countermeasures:

- o Clients and resource servers not using an OAuth HTTP authentication scheme (see [Section 5.4.1](#)) should take care to use Cache-Control headers to minimize the risk that authenticated content is not protected. Such clients should send a Cache-Control header containing the "no-store" option [[RFC2616](#)]. Resource server success (2XX status) responses to these requests should contain a Cache-Control header with the "private" option [[RFC2616](#)].
- o Reducing scope (see [Section 5.1.5.1](#)) and expiry time ([Section 5.1.5.3](#)) for access tokens can be used to reduce the damage in case of leaks.

#### 4.6.7. Threat: Token Leakage via Log Files and HTTP Referrers

If access tokens are sent via URI query parameters, such tokens may leak to log files and the HTTP "referrer".

Countermeasures:

- o Use Authorization headers or POST parameters instead of URI request parameters (see [Section 5.4.1](#)).
- o Set logging configuration appropriately.



- o Prevent unauthorized persons from access to system log files (see [Section 5.1.4.1.1](#)).
- o Abuse of leaked access tokens can be prevented by enforcing authenticated requests (see [Section 5.4.2](#)).
- o The impact of token leakage may be reduced by limiting scope (see [Section 5.1.5.1](#)) and duration (see [Section 5.1.5.3](#)) and by enforcing one-time token usage (see [Section 5.1.5.4](#)).

## 5. Security Considerations

This section describes the countermeasures as recommended to mitigate the threats described in [Section 4](#).

### 5.1. General

This section covers considerations that apply generally across all OAuth components (client, resource server, token server, and user agents).

#### 5.1.1. Ensure Confidentiality of Requests

This is applicable to all requests sent from the client to the authorization server or resource server. While OAuth provides a mechanism for verifying the integrity of requests, it provides no guarantee of request confidentiality. Unless further precautions are taken, eavesdroppers will have full access to request content and may be able to mount interception or replay attacks by using the contents of requests, e.g., secrets or tokens.

Attacks can be mitigated by using transport-layer mechanisms such as TLS [[RFC5246](#)]. A virtual private network (VPN), e.g., based on IPsec VPNs [[RFC4301](#)], may be considered as well.

Note: This document assumes end-to-end TLS protected connections between the respective protocol entities. Deployments deviating from this assumption by offloading TLS in between (e.g., on the data center edge) must refine this threat model in order to account for the additional (mainly insider) threat this may cause.

This is a countermeasure against the following threats:

- o Replay of access tokens obtained on the token's endpoint or the resource server's endpoint
- o Replay of refresh tokens obtained on the token's endpoint

- o Replay of authorization "codes" obtained on the token's endpoint (redirect?)
- o Replay of user passwords and client secrets

#### 5.1.2. Utilize Server Authentication

HTTPS server authentication or similar means can be used to authenticate the identity of a server. The goal is to reliably bind the fully qualified domain name of the server to the public key presented by the server during connection establishment (see [RFC2818]).

The client should validate the binding of the server to its domain name. If the server fails to prove that binding, the communication is considered a man-in-the-middle attack. This security measure depends on the certification authorities the client trusts for that purpose. Clients should carefully select those trusted CAs and protect the storage for trusted CA certificates from modifications.

This is a countermeasure against the following threats:

- o Spoofing
- o Proxying
- o Phishing by counterfeit servers

#### 5.1.3. Always Keep the Resource Owner Informed

Transparency to the resource owner is a key element of the OAuth protocol. The user should always be in control of the authorization processes and get the necessary information to make informed decisions. Moreover, user involvement is a further security countermeasure. The user can probably recognize certain kinds of attacks better than the authorization server. Information can be presented/exchanged during the authorization process, after the authorization process, and every time the user wishes to get informed by using techniques such as:

- o User consent forms.
- o Notification messages (e.g., email, SMS, ...). Note that notifications can be a phishing vector. Messages should be such that look-alike phishing messages cannot be derived from them.

- o Activity/event logs.
- o User self-care applications or portals.

#### 5.1.4. Credentials

This section describes countermeasures used to protect all kinds of credentials from unauthorized access and abuse. Credentials are long-term secrets, such as client secrets and user passwords as well as all kinds of tokens (refresh and access tokens) or authorization "codes".

##### 5.1.4.1. Enforce Credential Storage Protection Best Practices

Administrators should undertake industry best practices to protect the storage of credentials (for example, see [OWASP]). Such practices may include but are not limited to the following sub-sections.

###### 5.1.4.1.1. Enforce Standard System Security Means

A server system may be locked down so that no attacker may get access to sensitive configuration files and databases.

###### 5.1.4.1.2. Enforce Standard SQL Injection Countermeasures

If a client identifier or other authentication component is queried or compared against a SQL database, it may become possible for an injection attack to occur if parameters received are not validated before submission to the database.

- o Ensure that server code is using the minimum database privileges possible to reduce the "surface" of possible attacks.
- o Avoid dynamic SQL using concatenated input. If possible, use static SQL.
- o When using dynamic SQL, parameterize queries using bind arguments. Bind arguments eliminate the possibility of SQL injections.
- o Filter and sanitize the input. For example, if an identifier has a known format, ensure that the supplied value matches the identifier syntax rules.

#### 5.1.4.1.3. No Cleartext Storage of Credentials

The authorization server should not store credentials in clear text. Typical approaches are to store hashes instead or to encrypt credentials. If the credential lacks a reasonable entropy level (because it is a user password), an additional salt will harden the storage to make offline dictionary attacks more difficult.

Note: Some authentication protocols require the authorization server to have access to the secret in the clear. Those protocols cannot be implemented if the server only has access to hashes. Credentials should be strongly encrypted in those cases.

#### 5.1.4.1.4. Encryption of Credentials

For client applications, insecurely persisted client credentials are easy targets for attackers to obtain. Store client credentials using an encrypted persistence mechanism such as a keystore or database. Note that compiling client credentials directly into client code makes client applications vulnerable to scanning as well as difficult to administer should client credentials change over time.

#### 5.1.4.1.5. Use of Asymmetric Cryptography

Usage of asymmetric cryptography will free the authorization server of the obligation to manage credentials.

#### 5.1.4.2. Online Attacks on Secrets

##### 5.1.4.2.1. Utilize Secure Password Policy

The authorization server may decide to enforce a complex user password policy in order to increase the user passwords' entropy to hinder online password attacks. Note that too much complexity can increase the likelihood that users re-use passwords or write them down, or otherwise store them insecurely.

##### 5.1.4.2.2. Use High Entropy for Secrets

When creating secrets not intended for usage by human users (e.g., client secrets or token handles), the authorization server should include a reasonable level of entropy in order to mitigate the risk of guessing attacks. The token value should be  $\geq 128$  bits long and constructed from a cryptographically strong random or pseudo-random number sequence (see [RFC4086] for best current practice) generated by the authorization server.

#### 5.1.4.2.3. Lock Accounts

Online attacks on passwords can be mitigated by locking the respective accounts after a certain number of failed attempts.

Note: This measure can be abused to lock down legitimate service users.

#### 5.1.4.2.4. Use Tar Pit

The authorization server may react on failed attempts to authenticate by username/password by temporarily locking the respective account and delaying the response for a certain duration. This duration may increase with the number of failed attempts. The objective is to slow the attacker's attempts on a certain username down.

Note: This may require a more complex and stateful design of the authorization server.

#### 5.1.4.2.5. Use CAPTCHAs

The idea is to prevent programs from automatically checking a huge number of passwords, by requiring human interaction.

Note: This has a negative impact on user experience.

### 5.1.5. Tokens (Access, Refresh, Code)

#### 5.1.5.1. Limit Token Scope

The authorization server may decide to reduce or limit the scope associated with a token. The basis of this decision is out of scope; examples are:

- o a client-specific policy, e.g., issue only less powerful tokens to public clients,
- o a service-specific policy, e.g., it is a very sensitive service,
- o a resource-owner-specific setting, or
- o combinations of such policies and preferences.

The authorization server may allow different scopes dependent on the grant type. For example, end-user authorization via direct interaction with the end user (authorization "code") might be considered more reliable than direct authorization via grant type "username"/"password". This means will reduce the impact of the following threats:

- o token leakage
- o token issuance to malicious software
- o unintended issuance of powerful tokens with resource owner credentials flow

#### 5.1.5.2. Determine Expiration Time

Tokens should generally expire after a reasonable duration. This complements and strengthens other security measures (such as signatures) and reduces the impact of all kinds of token leaks. Depending on the risk associated with token leakage, tokens may expire after a few minutes (e.g., for payment transactions) or stay valid for hours (e.g., read access to contacts).

The expiration time is determined by several factors, including:

- o risk associated with token leakage,
- o duration of the underlying access grant,
- o duration until the modification of an access grant should take effect, and
- o time required for an attacker to guess or produce a valid token.

#### 5.1.5.3. Use Short Expiration Time

A short expiration time for tokens is a means of protection against the following threats:

- o replay
- o token leak (a short expiration time will reduce impact)
- o online guessing (a short expiration time will reduce the likelihood of success)

Note: Short token duration requires more precise clock synchronization between the authorization server and resource server. Furthermore, shorter duration may require more token refreshes (access token) or repeated end-user authorization processes (authorization "code" and refresh token).

#### 5.1.5.4. Limit Number of Usages or One-Time Usage

The authorization server may restrict the number of requests or operations that can be performed with a certain token. This mechanism can be used to mitigate the following threats:

- o replay of tokens
- o guessing

For example, if an authorization server observes more than one attempt to redeem an authorization "code", the authorization server may want to revoke all access tokens granted based on the authorization "code" as well as reject the current request.

As with the authorization "code", access tokens may also have a limited number of operations. This either forces client applications to re-authenticate and use a refresh token to obtain a fresh access token, or forces the client to re-authorize the access token by involving the user.

#### 5.1.5.5. Bind Tokens to a Particular Resource Server (Audience)

Authorization servers in multi-service environments may consider issuing tokens with different content to different resource servers and to explicitly indicate in the token the target server to which a token is intended to be sent. SAML assertions (see [OASIS.saml-core-2.0-os]) use the Audience element for this purpose. This countermeasure can be used in the following situations:

- o It reduces the impact of a successful replay attempt, since the token is applicable to a single resource server only.
- o It prevents abuse of a token by a rogue resource server or client, since the token can only be used on that server. It is rejected by other servers.
- o It reduces the impact of leakage of a valid token to a counterfeit resource server.

#### 5.1.5.6. Use Endpoint Address as Token Audience

This may be used to indicate to a resource server which endpoint URL has been used to obtain the token. This measure will allow the detection of requests from a counterfeit resource server, since such a token will contain the endpoint URL of that server.

#### 5.1.5.7. Use Explicitly Defined Scopes for Audience and Tokens

Deployments may consider only using tokens with explicitly defined scopes, where every scope is associated with a particular resource server. This approach can be used to mitigate attacks where a resource server or client uses a token for a different purpose than the one intended.

#### 5.1.5.8. Bind Token to Client id

An authorization server may bind a token to a certain client identifier. This identifier should be validated for every request with that token. This technique can be used to

- o detect token leakage and
- o prevent token abuse.

Note: Validating the client identifier may require the target server to authenticate the client's identifier. This authentication can be based on secrets managed independently of the token (e.g., pre-registered client id/secret on authorization server) or sent with the token itself (e.g., as part of the encrypted token content).

#### 5.1.5.9. Sign Self-Contained Tokens

Self-contained tokens should be signed in order to detect any attempt to modify or produce faked tokens (e.g., Hash-based Message Authentication Code or digital signatures).

#### 5.1.5.10. Encrypt Token Content

Self-contained tokens may be encrypted for confidentiality reasons or to protect system internal data. Depending on token format, keys (e.g., symmetric keys) may have to be distributed between server nodes. The method of distribution should be defined by the token and the encryption used.



#### 5.1.5.11. Adopt a Standard Assertion Format

For service providers intending to implement an assertion-based token design, it is highly recommended to adopt a standard assertion format (such as SAML [[OASIS.saml-core-2.0-os](#)] or the JavaScript Object Notation Web Token (JWT) [[OAuth-JWT](#)]).

#### 5.1.6. Access Tokens

The following measures should be used to protect access tokens:

- o Keep them in transient memory (accessible by the client application only).
- o Pass tokens securely using secure transport (TLS).
- o Ensure that client applications do not share tokens with 3rd parties.

### 5.2. Authorization Server

This section describes considerations related to the OAuth authorization server endpoint.

#### 5.2.1. Authorization "codes"

##### 5.2.1.1. Automatic Revocation of Derived Tokens If Abuse Is Detected

If an authorization server observes multiple attempts to redeem an authorization grant (e.g., such as an authorization "code"), the authorization server may want to revoke all tokens granted based on the authorization grant.

#### 5.2.2. Refresh Tokens

##### 5.2.2.1. Restricted Issuance of Refresh Tokens

The authorization server may decide, based on an appropriate policy, not to issue refresh tokens. Since refresh tokens are long-term credentials, they may be subject to theft. For example, if the authorization server does not trust a client to securely store such tokens, it may refuse to issue such a client a refresh token.

#### 5.2.2.2. Binding of Refresh Token to "client\_id"

The authorization server should match every refresh token to the identifier of the client to whom it was issued. The authorization server should check that the same "client\_id" is present for every request to refresh the access token. If possible (e.g., confidential clients), the authorization server should authenticate the respective client.

This is a countermeasure against refresh token theft or leakage.

Note: This binding should be protected from unauthorized modifications.

#### 5.2.2.3. Refresh Token Rotation

Refresh token rotation is intended to automatically detect and prevent attempts to use the same refresh token in parallel from different apps/devices. This happens if a token gets stolen from the client and is subsequently used by both the attacker and the legitimate client. The basic idea is to change the refresh token value with every refresh request in order to detect attempts to obtain access tokens using old refresh tokens. Since the authorization server cannot determine whether the attacker or the legitimate client is trying to access, in case of such an access attempt the valid refresh token and the access authorization associated with it are both revoked.

The OAuth specification supports this measure in that the token's response allows the authorization server to return a new refresh token even for requests with grant type "refresh\_token".

Note: This measure may cause problems in clustered environments, since usage of the currently valid refresh token must be ensured. In such an environment, other measures might be more appropriate.

#### 5.2.2.4. Revocation of Refresh Tokens

The authorization server may allow clients or end users to explicitly request the invalidation of refresh tokens. A mechanism to revoke tokens is specified in [\[OAuth-REVOCATION\]](#).

This is a countermeasure against:

- o device theft,
- o impersonation of a resource owner, or
- o suspected compromised client applications.

#### 5.2.2.5. Device Identification

The authorization server may require the binding of authentication credentials to a device identifier. The International Mobile Station Equipment Identity [IMEI] is one example of such an identifier; there are also operating system-specific identifiers. The authorization server could include such an identifier when authenticating user credentials in order to detect token theft from a particular device.

Note: Any implementation should consider potential privacy implications of using device identifiers.

#### 5.2.2.6. X-FRAME-OPTIONS Header

For newer browsers, avoidance of iFrames can be enforced on the server side by using the X-FRAME-OPTIONS header (see [X-Frame-Options]). This header can have two values, "DENY" and "SAMEORIGIN", which will block any framing or any framing by sites with a different origin, respectively. The value "ALLOW-FROM" specifies a list of trusted origins that iFrames may originate from.

This is a countermeasure against the following threat:

- o Clickjacking attacks

#### 5.2.3. Client Authentication and Authorization

As described in [Section 3](#) (Security Features), clients are identified, authenticated, and authorized for several purposes, such as to:

- o Collate requests to the same client,
- o Indicate to the user that the client is recognized by the authorization server,
- o Authorize access of clients to certain features on the authorization server or resource server, and
- o Log a client identifier to log files for analysis or statistics.

Due to the different capabilities and characteristics of the different client types, there are different ways to support these objectives, which will be described in this section. Authorization server providers should be aware of the security policy and deployment of a particular client and adapt its treatment accordingly. For example, one approach could be to treat all clients as less trustworthy and unsecure. On the other extreme, a service provider could activate every client installation individually by an administrator and in that way gain confidence in the identity of the software package and the security of the environment in which the client is installed. There are several approaches in between.

#### 5.2.3.1. Don't Issue Secrets to Clients with Inappropriate Security Policy

Authorization servers should not issue secrets to clients that cannot protect secrets ("public" clients). This reduces the probability of the server treating the client as strongly authenticated.

For example, it is of limited benefit to create a single client id and secret that are shared by all installations of a native application. Such a scenario requires that this secret must be transmitted from the developer via the respective distribution channel, e.g., an application market, to all installations of the application on end-user devices. A secret, burned into the source code of the application or an associated resource bundle, is not protected from reverse engineering. Secondly, such secrets cannot be revoked, since this would immediately put all installations out of work. Moreover, since the authorization server cannot really trust the client's identifier, it would be dangerous to indicate to end users the trustworthiness of the client.

There are other ways to achieve a reasonable security level, as described in the following sections.

#### 5.2.3.2. Require User Consent for Public Clients without Secret

Authorization servers should not allow automatic authorization for public clients. The authorization server may issue an individual client id but should require that all authorizations are approved by the end user. For clients without secrets, this is a countermeasure against the following threat:

- o Impersonation of public client applications.

#### 5.2.3.3. Issue a "client\_id" Only in Combination with "redirect\_uri"

The authorization server may issue a "client\_id" and bind the "client\_id" to a certain pre-configured "redirect\_uri". Any authorization request with another redirect URI is refused automatically. Alternatively, the authorization server should not accept any dynamic redirect URI for such a "client\_id" and instead should always redirect to the well-known pre-configured redirect URI. This is a countermeasure for clients without secrets against the following threats:

- o Cross-site scripting attacks
- o Impersonation of public client applications

#### 5.2.3.4. Issue Installation-Specific Client Secrets

An authorization server may issue separate client identifiers and corresponding secrets to the different installations of a particular client (i.e., software package). The effect of such an approach would be to turn otherwise "public" clients back into "confidential" clients.

For web applications, this could mean creating one "client\_id" and "client\_secret" for each web site on which a software package is installed. So, the provider of that particular site could request a client id and secret from the authorization server during the setup of the web site. This would also allow the validation of some of the properties of that web site, such as redirect URI, web site URL, and whatever else proves useful. The web site provider has to ensure the security of the client secret on the site.

For native applications, things are more complicated because every copy of a particular application on any device is a different installation. Installation-specific secrets in this scenario will require obtaining a "client\_id" and "client\_secret" either

1. during the download process from the application market, or
2. during installation on the device.

Either approach will require an automated mechanism for issuing client ids and secrets, which is currently not defined by OAuth.

The first approach would allow the achievement of a certain level of trust in the authenticity of the application, whereas the second option only allows the authentication of the installation but not the validation of properties of the client. But this would at least help

to prevent several replay attacks. Moreover, installation-specific "client\_ids" and secrets allow the selective revocation of all refresh tokens of a specific installation at once.

#### 5.2.3.5. Validate Pre-Registered "redirect\_uri"

An authorization server should require all clients to register their "redirect\_uri", and the "redirect\_uri" should be the full URI as defined in [RFC6749]. The way that this registration is performed is out of scope of this document. As per the core spec, every actual redirect URI sent with the respective "client\_id" to the end-user authorization endpoint must match the registered redirect URI. Where it does not match, the authorization server should assume that the inbound GET request has been sent by an attacker and refuse it. Note: The authorization server should not redirect the user agent back to the redirect URI of such an authorization request. Validating the pre-registered "redirect\_uri" is a countermeasure against the following threats:

- o Authorization "code" leakage through counterfeit web site: allows authorization servers to detect attack attempts after the first redirect to an end-user authorization endpoint ([Section 4.4.1.7](#)).
- o Open redirector attack via a client redirection endpoint ([Section 4.1.5](#)).
- o Open redirector phishing attack via an authorization server redirection endpoint ([Section 4.2.4](#)).

The underlying assumption of this measure is that an attacker will need to use another redirect URI in order to get access to the authorization "code". Deployments might consider the possibility of an attacker using spoofing attacks to a victim's device to circumvent this security measure.

Note: Pre-registering clients might not scale in some deployments (manual process) or require dynamic client registration (not specified yet). With the lack of dynamic client registration, a pre-registered "redirect\_uri" only works for clients bound to certain deployments at development/configuration time. As soon as dynamic resource server discovery is required, the pre-registered "redirect\_uri" may no longer be feasible.

#### 5.2.3.6. Revoke Client Secrets

An authorization server may revoke a client's secret in order to prevent abuse of a revealed secret.

Note: This measure will immediately invalidate any authorization "code" or refresh token issued to the respective client. This might unintentionally impact client identifiers and secrets used across multiple deployments of a particular native or web application.

This a countermeasure against:

- o Abuse of revealed client secrets for private clients

#### 5.2.3.7. Use Strong Client Authentication (e.g., client\_assertion/client\_token)

By using an alternative form of authentication such as client assertion [[OAuth-ASSERTIONS](#)], the need to distribute a "client\_secret" is eliminated. This may require the use of a secure private key store or other supplemental authentication system as specified by the client assertion issuer in its authentication process.

#### 5.2.4. End-User Authorization

This section includes considerations for authorization flows involving the end user.

##### 5.2.4.1. Automatic Processing of Repeated Authorizations Requires Client Validation

Authorization servers should NOT automatically process repeat authorizations where the client is not authenticated through a client secret or some other authentication mechanism such as a signed authentication assertion certificate ([Section 5.2.3.7](#)) or validation of a pre-registered redirect URI ([Section 5.2.3.5](#)).

##### 5.2.4.2. Informed Decisions Based on Transparency

The authorization server should clearly explain to the end user what happens in the authorization process and what the consequences are. For example, the user should understand what access he is about to grant to which client for what duration. It should also be obvious to the user whether the server is able to reliably certify certain client properties (web site URL, security policy).

#### 5.2.4.3. Validation of Client Properties by End User

In the authorization process, the user is typically asked to approve a client's request for authorization. This is an important security mechanism by itself because the end user can be involved in the validation of client properties, such as whether the client name known to the authorization server fits the name of the web site or the application the end user is using. This measure is especially helpful in situations where the authorization server is unable to authenticate the client. It is a countermeasure against:

- o A malicious application
- o A client application masquerading as another client

#### 5.2.4.4. Binding of Authorization "code" to "client\_id"

The authorization server should bind every authorization "code" to the id of the respective client that initiated the end-user authorization process. This measure is a countermeasure against:

- o Replay of authorization "codes" with different client credentials, since an attacker cannot use another "client\_id" to exchange an authorization "code" into a token
- o Online guessing of authorization "codes"

Note: This binding should be protected from unauthorized modifications (e.g., using protected memory and/or a secure database).

#### 5.2.4.5. Binding of Authorization "code" to "redirect\_uri"

The authorization server should be able to bind every authorization "code" to the actual redirect URI used as the redirect target of the client in the end-user authorization process. This binding should be validated when the client attempts to exchange the respective authorization "code" for an access token. This measure is a countermeasure against authorization "code" leakage through counterfeit web sites, since an attacker cannot use another redirect URI to exchange an authorization "code" into a token.



### 5.3. Client App Security

This section deals with considerations for client applications.

#### 5.3.1. Don't Store Credentials in Code or Resources Bundled with Software Packages

Because of the number of copies of client software, there is limited benefit in creating a single client id and secret that is shared by all installations of an application. Such an application by itself would be considered a "public" client, as it cannot be presumed to be able to keep client secrets. A secret, burned into the source code of the application or an associated resource bundle, cannot be protected from reverse engineering. Secondly, such secrets cannot be revoked, since this would immediately put all installations out of work. Moreover, since the authorization server cannot really trust the client's identifier, it would be dangerous to indicate to end users the trustworthiness of the client.

#### 5.3.2. Use Standard Web Server Protection Measures (for Config Files and Databases)

Use standard web server protection and configuration measures to protect the integrity of the server, databases, configuration files, and other operational components of the server.

#### 5.3.3. Store Secrets in Secure Storage

There are different ways to store secrets of all kinds (tokens, client secrets) securely on a device or server.

Most multi-user operating systems segregate the personal storage of different system users. Moreover, most modern smartphone operating systems even support the storage of application-specific data in separate areas of file systems and protect the data from access by other applications. Additionally, applications can implement confidential data by using a user-supplied secret, such as a PIN or password.

Another option is to swap refresh token storage to a trusted backend server. This option in turn requires a resilient authentication mechanism between the client and backend server. Note: Applications should ensure that confidential data is kept confidential even after reading from secure storage, which typically means keeping this data in the local memory of the application.

#### 5.3.4. Utilize Device Lock to Prevent Unauthorized Device Access

On a typical modern phone, there are many "device lock" options that can be utilized to provide additional protection when a device is stolen or misplaced. These include PINs, passwords, and other biometric features such as "face recognition". These are not equal in the level of security they provide.

#### 5.3.5. Link the "state" Parameter to User Agent Session

The "state" parameter is used to link client requests and prevent CSRF attacks, for example, attacks against the redirect URI. An attacker could inject their own authorization "code" or access token, which can result in the client using an access token associated with the attacker's protected resources rather than the victim's (e.g., save the victim's bank account information to a protected resource controlled by the attacker).

The client should utilize the "state" request parameter to send the authorization server a value that binds the request to the user agent's authenticated state (e.g., a hash of the session cookie used to authenticate the user agent) when making an authorization request. Once authorization has been obtained from the end user, the authorization server redirects the end-user's user agent back to the client with the required binding value contained in the "state" parameter.

The binding value enables the client to verify the validity of the request by matching the binding value to the user agent's authenticated state.

### 5.4. Resource Servers

The following section details security considerations for resource servers.

#### 5.4.1. Authorization Headers

Authorization headers are recognized and specially treated by HTTP proxies and servers. Thus, the usage of such headers for sending access tokens to resource servers reduces the likelihood of leakage or unintended storage of authenticated requests in general, and especially Authorization headers.

#### 5.4.2. Authenticated Requests

An authorization server may bind tokens to a certain client identifier and enable resource servers to validate that association on resource access. This will require the resource server to authenticate the originator of a request as the legitimate owner of a particular token. There are several options to implement this countermeasure:

- o The authorization server may associate the client identifier with the token (either internally or in the payload of a self-contained token). The client then uses client certificate-based HTTP authentication on the resource server's endpoint to authenticate its identity, and the resource server validates the name with the name referenced by the token.
- o Same as the option above, but the client uses his private key to sign the request to the resource server (the public key is either contained in the token or sent along with the request).
- o Alternatively, the authorization server may issue a token-bound key, which the client uses in a Holder-of-Key proof to authenticate the client's use of the token. The resource server obtains the secret directly from the authorization server, or the secret is contained in an encrypted section of the token. In that way, the resource server does not "know" the client but is able to validate whether the authorization server issued the token to that client.

Authenticated requests are a countermeasure against abuse of tokens by counterfeit resource servers.

#### 5.4.3. Signed Requests

A resource server may decide to accept signed requests only, either to replace transport-level security measures or to complement such measures. Every signed request should be uniquely identifiable and should not be processed twice by the resource server. This countermeasure helps to mitigate:

- o modifications of the message and
- o replay attempts

### 5.5. A Word on User Interaction and User-Installed Apps

OAuth, as a security protocol, is distinctive in that its flow usually involves significant user interaction, making the end user a part of the security model. This creates some important difficulties in defending against some of the threats discussed above. Some of these points have already been made, but it's worth repeating and highlighting them here.

- o End users must understand what they are being asked to approve (see [Section 5.2.4.2](#)). Users often do not have the expertise to understand the ramifications of saying "yes" to an authorization request and are likely not to be able to see subtle differences in the wording of requests. Malicious software can confuse the user, tricking the user into approving almost anything.
- o End-user devices are prone to software compromise. This has been a long-standing problem, with frequent attacks on web browsers and other parts of the user's system. But with the increasing popularity of user-installed "apps", the threat posed by compromised or malicious end-user software is very strong and is one that is very difficult to mitigate.
- o Be aware that users will demand to install and run such apps, and that compromised or malicious ones can steal credentials at many points in the data flow. They can intercept the very user login credentials that OAuth is designed to protect. They can request authorization far beyond what they have led the user to understand and approve. They can automate a response on behalf of the user, hiding the whole process. No solution is offered here, because none is known; this remains in the space between better security and better usability.
- o Addressing these issues by restricting the use of user-installed software may be practical in some limited environments and can be used as a countermeasure in those cases. Such restrictions are not practical in the general case, and mechanisms for after-the-fact recovery should be in place.
- o While end users are mostly incapable of properly vetting applications they load onto their devices, those who deploy authorization servers might have tools at their disposal to mitigate malicious clients. For example, a well-run authorization server must only assert client properties to the end user it is effectively capable of validating, explicitly point out which properties it cannot validate, and indicate to the end user the risk associated with granting access to the particular client.

## 6. Acknowledgements

We would like to thank Stephen Farrell, Barry Leiba, Hui-Lan Lu, Francisco Corella, Peifung E. Lam, Shane B. Weeden, Skylar Woodward, Niv Steingarten, Tim Bray, and James H. Manger for their comments and contributions.

## 7. References

### 7.1. Normative References

- [RFC6749] Hardt, D., "The OAuth 2.0 Authorization Framework", [RFC 6749](#), October 2012.
- [RFC6750] Jones, M. and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage", [RFC 6750](#), October 2012.

### 7.2. Informative References

- [Framebusting]  
Rydstedt, G., Bursztein, Boneh, D., and C. Jackson, "Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites", IEEE 3rd Web 2.0 Security and Privacy Workshop, May 2010, <<http://elie.im/publication/busting-frame-busting-a-study-of-clickjacking-vulnerabilities-on-popular-sites>>.
- [IMEI] 3GPP, "International Mobile station Equipment Identities (IMEI)", 3GPP TS 22.016 11.0.0, September 2012, <<http://www.3gpp.org/ftp/Specs/html-info/22016.htm>>.
- [OASIS.saml-core-2.0-os]  
Cantor, S., Ed., Kemp, J., Ed., Philpott, R., Ed., and E. Maler, Ed., "Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0", OASIS Standard saml-core-2.0-os, March 2005, <<http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf>>.
- [OASIS.sstc-saml-bindings-1.1]  
Maler, E., Ed., Mishra, P., Ed., and R. Philpott, Ed., "Bindings and Profiles for the OASIS Security Assertion Markup Language (SAML) V1.1", September 2003, <<http://www.oasis-open.org/committees/download.php/3405/oasis-sstc-saml-bindings-1.1.pdf>>.

- [OASIS.sstc-sec-analysis-response-01]  
Linn, J., Ed., and P. Mishra, Ed., "SSTC Response to  
"Security Analysis of the SAML Single Sign-on Browser/  
Artifact Profile"", January 2005,  
<[http://www.oasis-open.org/committees/download.php/  
11191/sstc-gross-sec-analysis-response-01.pdf](http://www.oasis-open.org/committees/download.php/11191/sstc-gross-sec-analysis-response-01.pdf)>.
- [OAuth-ASSERTIONS]  
Campbell, B., Mortimore, C., Jones, M., and Y. Goland,  
"Assertion Framework for OAuth 2.0", Work in Progress,  
December 2012.
- [OAuth-HTTP-MAC]  
Richer, J., Ed., Mills, W., Ed., and H. Tschofenig, Ed.,  
"OAuth 2.0 Message Authentication Code (MAC) Tokens", Work  
in Progress, November 2012.
- [OAuth-JWT]  
Jones, M., Bradley, J., and N. Sakimura, "JSON Web Token  
(JWT)", Work in Progress, December 2012.
- [OAuth-REVOCATION]  
Lodderstedt, T., Ed., Dronia, S., and M. Scurtescu, "Token  
Revocation", Work in Progress, November 2012.
- [OPENID] "OpenID Foundation Home Page", <<http://openid.net/>>.
- [OWASP] "Open Web Application Security Project Home Page",  
<<https://www.owasp.org/>>.
- [Portable-Contacts]  
Smarr, J., "Portable Contacts 1.0 Draft C", August 2008,  
<<http://portablecontacts.net/>>.
- [RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H.,  
Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext  
Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.
- [RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.
- [RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness  
Requirements for Security", BCP 106, RFC 4086, June 2005.
- [RFC4120] Neuman, C., Yu, T., Hartman, S., and K. Raeburn, "The  
Kerberos Network Authentication Service (V5)", RFC 4120,  
July 2005.

- [RFC4301] Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", [RFC 4301](#), December 2005.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", [RFC 5246](#), August 2008.
- [SSL-Latency] Sissel, J., Ed., "SSL handshake latency and HTTPS optimizations", June 2010.
- [Sec-Analysis] Gross, T., "Security Analysis of the SAML Single Sign-on Browser/Artifact Profile", 19th Annual Computer Security Applications Conference, Las Vegas, December 2003.
- [X-Frame-Options] Ross, D. and T. Gondrom, "[HTTP Header X-Frame-Options](#)", Work in Progress, October 2012.
- [iFrame] World Wide Web Consortium, "Frames in HTML documents", W3C HTML 4.01, December 1999, [<http://www.w3.org/TR/html4/present/frames.html#h-16.5>.](http://www.w3.org/TR/html4/present/frames.html#h-16.5)

#### Authors' Addresses

Torsten Lodderstedt (editor)  
Deutsche Telekom AG

EMail: [torsten@lodderstedt.net](mailto:torsten@lodderstedt.net)

Mark McGloin  
IBM

EMail: [mark.mcgloin@ie.ibm.com](mailto:mark.mcgloin@ie.ibm.com)

Phil Hunt  
Oracle Corporation

EMail: [phil.hunt@yahoo.com](mailto:phil.hunt@yahoo.com)