Open Authentication Protocol	T. Lodderstedt, Ed.
Internet-Draft	Deutsche Telekom AG
Intended status: Standards Track	M. McGloin
Expires: September 15, 2011	IBM
	P. Hunt
	Oracle Corporation
	March 14, 2011

OAuth 2.0 Threat Model and Security Considerations
draft-lodderstedt-oauth-security-01

Abstract

This document gives security considerations based on a comprehensive threat model for the OAuth 2.0 Protocol.

Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 *[RFC2119]*.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

This document gives security considerations based on a comprehensive threat model for the OAuth 2.0 Protocol [I-D.ietf-oauth-v2]. It contains the following content:

* Documents any assumptions and scope considered when creating the threat model
* Describe the security features in-built into the OAuth protocol and how they are intended to thwart attacks
* Give a comprehensive threat model for OAuth and describes the respective counter measures 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 aid development teams implement each part of the protocol securely. For example all threats for granting access or all threats for a particular client profile or all threats for protecting the resource server.

2. Overview

2.1. Scope

The security considerations document only considers clients bound to a particular deployment as supported by [I-D.ietf-oauth-v2]. Such deployments have the following characteristics:

* Resource server URLs are static and well-known at development time, authorization server URLs can be static or discovered.
* Token scope values (e.g. applicable URLs and methods) are well-known at development time.
* Client registration: Since registration of clients is out of scope of the current core spec, 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 :

* Communication between authorization server and resource server
* Token formats

* Except for „Resource Owner Password Credentials“ (see [I-D.ietf-oauth-v2], section 4.3), the mechanism used by authorization servers to authenticate the user

* Mechanism by which a user obtained an assertion and any resulting attacks mounted as a result of the assertion being false.

* Clients are not bound to a specific deployment: An example could by 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 must 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 the attacker has full access to the network between the client and service provider and may eaves drop on any communications between the two.

* It is assumed an attacker has unlimited resources to mount an attack.

* It is assumed that 2 parties involved in the OAuth 3 legged 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 the assumptions about the features, limitations and design options of the different entities of a OAuth deployment along with the security-sensitive data-elements managed by those entity. These assumptions are the foundation of the treat analysis. The OAuth protocol leaves deployments with a certain degree of freedom how to implement and apply the standard. The core specification defines the core concepts of an authorization server and an resource server. Both servers can be implemented in the same server entity, or they may also be different entities. The later is typically the case for multi-service providers with a single authentication and authorization system, and are more typical in middleware architectures.

2.3.1. Authorization Servers

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

*user names and passwords

*client ids and secrets

*client-specific refresh tokens

*client-specific access tokens (in case of handle-based design)

*HTTPS certificate/key

*per authorization process (in case of handle-based design):
redirect_uri, client_id, authorization code

2.3.2. Resource Server

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

*user data (out of scope)

*HTTPS certificate/key

*authz server credentials (handle-based design), or

*authz server shared secret/public key (assertion-based design)

*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

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

*client id (and secret)

*refresh tokens (persistent) access tokens (transient)

*trusted CA certs (HTTPS)

*per authorization process: redirect_uri, authorization code

2.3.3.1. Web Server

Such clients typically represent a web site with its own user management and login mechanism and have the following characteristics:

Section 4.4.1).

- *Tokens are bound to a single user identity at the site

- *Web servers are able to protect client secrets

- *The potential number of tokens affected by a security breach depends on number of site users.

Such clients are implemented using the authorization code flow (see

2.3.3.2. **Native Applications**

This class of OAuth clients represent apps running on a user-controlled device, such as a notebook, PC, Tablet, Smartphone, or Gaming Console. Massively distributed applications such as these cannot reliably keep secrets confidential, which are issued per software package. This is because such secrets would need to be transferred to the user device as part of the installation process. An attacker could reverse engineer any secret from the binary or accompanying resources. Native Applications are able to protect per installation/instance secrets (e.g. refresh tokens) to some extent.
Device platforms typically allow users to lock the device with a pin and to segregate different apps or users (multi-user operation systems).
Some devices can be identified/authenticated (to varying degrees of assurance):

- *Handsets and smart phones by its International Mobile Equipment Identity (IMEI)

- *Set top boxes, gaming consoles, others by using certificates and TPM module - Note: This does not help to identify client apps but may be used to bound tokens to devices and to detect token theft

Mobile devices, such as handsets or smart phones have the following special characteristics:

- *Limited input capabilities, therefore such clients typically obtain a refresh token in order to provide automatic login for sub-sequent application sessions

- *As mobile and small devices, they can get cloned, stolen or lost easier than other devices.

- *Security breach will affect single user (or a few users) only.

For the purposes of this document, the scenario of attackers who control a smartphone device entirely is out of scope.
There are several implementation options for native applications:

*The authorization code flow in combination with an embedded or external browser (Section 4.4.1)

*The implicte grant flow in combination with an embedded or external browser (Section 4.4.2)

*The resource owner password credentials flow can be used as well (Section 4.4.3)

Different threats exists for those implementation options, which are discussed in the respective sections of the threat model.

2.3.3.3. User Agent

[TBD]
Such client are implemented using the implicite grant flow (Section 4.4.2).

2.3.3.4. Autonomous

Autonomous clients access service providers using rights grants by client credentials only. Thus the autonomous client becomes the „user“. Authenticating autonomous clients is conceptually similar to end-user authentication since the issued tokens refer to the client's identity. Autonomous clients shall always be required to use a secret or some other form of authentication (e.g. client assertion in the form of a SAML assertion or STS token) acceptable to the authorization/token services. The client must ensure the confidentiality of client_secret or other credential.

3. Security Features

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

3.1. Tokens

OAuth makes extensive use of tokens. Tokens can be implemented in 2 ways as follows:

Handle (or artifact) 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, scope, etc. Handles typically require a communication between resource server and token server in order to validate the token and obtain token-bound data. Handles enable simple revocation and do not require cryptographic mechanisms to protected token content from being modified. As a disadvantage, they require additional resource/token server communication impacting on performance and scalability. An authorization code (OAuth Section 4.1.2) is an example of a 'handle' token. An access token may also be implemented as a handle token. 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.

Assertions (aka self-contained token) a parseable token. An assertion typically has a duration, an audience, and is digitally signed containing information about the user and the client. Examples of assertion formats are SAML assertions and Kerberos tickets. Assertions can typically directly be validated and used by a resource server without interactions with the authorization server. This results in better performance and scalability. Implementing token revocation is more difficult with assertions than with handles.

Tokens can be sent to resource server in two ways:

bearer token A 'bearer token' is a token that can be used by any client who has received the token (cf. [I-D.ietf-oauth-v2-bearer] . Because mere possession is enough to use the token it is important that communication between end-points be secured to ensure that only authorized end-points may capture the token. The bearer token is convenient to 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 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 (mutual authentication) and HoK (holder-of-key) tokens (cf. [I-D.hammer-oauth-v2-mac-token].

3.2. 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 these parties deem less secure or trustworthy. Optionally, the client can request the scope to apply to the token but only for 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.

3.3. Expires_In

Expires_In allows an authorization server (based on its policies or on behalf of the end-user) to limit the lifetime of the access token. This mechanisms can be used to issue short-living tokens to OAuth clients the authorization server deems less secure or where sending tokens over non secure channels.

3.4. Authorization Code

An Authorization Code represents the intermediary result of a successful end-user authorization process and is used by the client to obtain access and refresh token. Authorization codes are sent to the client's redirect_uri instead of tokens for two purposes.

3.5. Redirect-URI

A Redirect-uri helps to identify clients and prevents phishing attacks from other clients attempting to trick the user into believing the phisher is the client. The redirect URI is pre-registered as requests with authorization code or token will be directed to that URI. Moreover, the value of the actual redirect_uri has to be presented and is verified when an authorization code is exchanged for tokens. This helps to prevent session fixation attacks.

3.6. Access Token

An Access Token is used by a client to access a resource. An access token must be acquired using a HTTP POST operation to ensure no logging or caching of requests. 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 enables the possibility of revocation by requiring the client to refresh their access token at regular intervals.

3.7. Refresh Token

A Refresh Token is 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 must always be 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.

3.8. Client Authentication

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 security level in delegated authorization scenarios and because the client will be able to act without the user's presence. By authenticating a client when requesting an access token, the token service is able to assess whether a given client and authorization code meets appropriate security requirements and binds the authorization code approved by the user to the client making the request.
OAuth uses the *client_id* to collate associated request to the same originator, such as

* a particular end-user authorization process and the corresponding request on the tokens endpoint to exchange the authorization code for tokens or

* the initial authorization and issuance of a tokens by an end-user to a particular client and sub-sequent requests by this client to obtain tokens w/o user consent (automatic processing of repeated authorization)

The client identity may also be used by the authorization server to display relevant registration information to a user when requesting consent for scope requested by a particular client. The client identity may be used to limit the number of request for a particular client or to charge the client per request. Client Identity may furthermore be useful to differentiate (e.g. in server log files) between accesses by end-user, and delegated accesses by client on behalf of a user.
The *client_secret* is used to verify the client identifier. This should only be used where the client is capable of keeping its secret confidential. The client identity can also be verified using the *redirect_uri* or by the *end-user*.
Clients (and the trustworthiness of its identity) can be classifed by using the following parameters:

* Deployment-specific or -independent client_id (Note: for native apps, every installation of a particular app on a certain device is considered a deployment.)

* Validated properties, such as app name or redirect_uri

* Client_secret available

Typical client categories are:

Deployment-independent client_id with pre-registered redirect_uri and without client_secret

Such an identity is used by multiple installations of the same software package. The identity of such a client can only be validated with the help of the end-user. This is a viable option for native apps 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 such an identity will affect ALL deployments of the respective software.

Deployment-independent client_id with pre-registered redirect_uri and with client_secre

This is an option for native applications only, since web application would require different redirect URIs. This category is not advisable because the client secret cannot be protected appropriately (cf. Section 4.1.1). Due to its security weaknesses, such client identities have the same trustlevel as deployment-independent clients without secret. Revocation will affect ALL deployments.

Deployment-specific client_id with pre-registered redirect_uri and with client_secret

The client registration process insures the validation of the client's properties, such as redirect_uri, website address, web site name, contacts. Such a client identity 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 app is conducted by an administrator, who validates the clients authenticity, or the process from validating the app to the installation of the app on the device and the creation of the client credentials is controlled end-to-end by a single entity (e.g. app 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). 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 apps. Revocation will affect a single deployment only.

Use of the client secret is considered a relatively weak form of credential for the client. Use of stronger mechanisms such as a client assertion (e.g. SAML), key cryptography, are preferred.

4. Security Threat Model

This sections gives a comprehensive threat model of OAuth 2.0. Threats are grouped first by attackes directed against an OAuth component, which are 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: Obtain Client Secrets

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

*replay its tokens and authorization codes, or

*obtain tokens on behalf of the attacked client with the privileges of that client.

The resulting impact would be:

*Client authentication of access to authorization server can be bypassed

*Stolen refresh tokens or authorization codes can be replayed

Depending on the client category, there are the following approaches an attacker could utilize to obtain the client secret.
Attack: Obtain Secret From Source Code or Binary. This applies for all client profiles and especially for open source projects, where the source code is public accessible. Even if the attacker does not has access to the source code, it could reverse engineer secrets from the binary of native apps.
Countermeasures:

*Don't issue secrets to clients with inappropriate security policy - Section 5.2.3.1

*Client_id only in combination with forced user consent - Section 5.2.3.2

*Deployment-specific client secrets - Section 5.2.3.4

*Client secret revocation - 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 devices (native app).
Countermeasures:

- *Web server: apply standard web server protection measures (for config files and databases) - Section 5.3.2
- *Native apps: Store secrets in a secure local storage - Section 5.3.3
- *Client secret revocation - Section 5.2.3.6

4.1.2. Threat: Obtain Refresh Tokens

Depending on the client type, there are different ways 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. Some generally applicable countermeasure to mitigate such attacks shall be given in advance:

- *The authorization server must validate the client id associated with the particular refresh token with every refresh request - Section 5.2.2.2
- *Limited scope tokens - Section 5.1.5.1
- *Refresh token revocation - Section 5.2.2.4
- *Client secret revocation - Section 5.2.3.6
- *Refresh tokens can automatically be replaced in order to detect unauthorized token usage by another party (Refresh Token Replacement) - Section 5.2.2.3

Attack: Obtain Refresh Token from Web application. An attack may obtain the refresh tokens issued to a web server client. Impact: Exposure of all refresh tokens on that side.
Countermeasures:

- *Standard web server protection measures - Section 5.3.2
- *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 filesystem: The attacker could try get file system access on the device and read the refresh tokens. The attacker could utilize a malicious app for that purpose.
Countermeasures:

- Store secrets in a secure storage - Section 5.3.3
- Utilize device lock to prevent unauthorized device access - Section 5.3.4

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

- Utilize device lock to prevent unauthorized device access - Section 5.3.4
- Where a user knows the device has been cloned, they can use this countermeasure (Refresh Token Revocation) - Section 5.2.2.4

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

- Combine refresh token request with device identification - Section 5.2.2.6
- Combine refresh token requests with user-provided secret - Section 5.2.2.5
- Refresh Token Replacement - Section 5.2.2.3
- Where a user knows the device has been cloned, they can use this countermeasure - Refresh Token Revocation - Section 5.2.2.4

Obtain refresh tokens from backup: A refresh token could be obtained from the backup of a client application, or device.
Countermeasures:

- tbd

4.1.3. Threat: Obtain Access Tokens

Depending on the client type, there are different ways access tokens may be revealed to an attacker. Access tokens could be stolen from the

device if the app stores them in a storage, which 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.
Countermeasures:

- *Keep access tokens in transient memory and limit grants: Section 5.1.6
- *Limited scope tokens - Section 5.1.5.1
- *Combine refresh token requests with user-provided secret - Section 5.2.2.5
- *Client secret revocation - Section 5.2.3.6
- *Keep access tokens in private memory or apply same protection means as for refresh tokens - Section 5.2.2
- *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 app 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 trusted system browser to render the authorization UI. By doing so, the usual visual trust mechanisms may be bypassed (e.g. TLS confirmation, web site mechanisms). By using an embedded or internal client app UI, the client app has access to additional information it should not have access to (e.g. uid/password).
Impact: If the client app or the communication is compromised, the user would not be aware and all information in the authorization exchange could be captured such as username and password.
Countermeasures:

- *Client developers and end-user can be educated to trust an external System-Browser only.
- *Client apps could be validated prior publication in a app market.
- *Client developers should not collect authentication information directly from users and should instead use redirects to and back from a trusted external system-browser.

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 ARP spoofing. Wide deployment of OAuth and similar protocols may cause Users to become inured to the practice of being redirected to websites where they are asked to enter their passwords. If Users are not careful to verify the authenticity of these websites before entering their credentials, it will be possible for attackers to exploit this practice to steal Users' passwords.
Countermeasures:

* Service providers should consider such attacks when developing services based on OAuth, and should require transport-layer security for any requests where the authenticity of the Service Provider or of request responses is an issue (see Section 5.1.2).

* Service Providers should attempt to educate Users about the risks phishing attacks pose, 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 authenticaton, 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 which should not be permitted.
Countermeasures:

* Explain the scope (resources and the permissions) the user is about to grant in a understandable way - Section 5.2.4.2

* Narrow scope based on client-specific policy - When obtaining end user authorization and where the client requests scope, the service provider may want to consider whether to honour that scope based on who the client is. That decision is between the client and service provider and is outside the scope of this spec. The service provider may also want to consider what scope to grant based on the profile used, e.g. providing lower scope where no client secret is provided from a native application. - 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 which 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 Provider.
A malicious client may exploit that feature and try to obtain such an authorization code instead of the legimate client.
Countermeasures:

- *Service providers should not automatically process repeat authorizations where the client is not authenticated through a client secret or some other authentication mechanism such as signing with security certs (see Section 5.2.3.7) or validation of a pre-registered redirect uri (Section 5.2.3.5)

- *Service Providers 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 redirector.
Impact?
Countermeasure

- *don't redirect to redirect_uri, if client identity or redirect_uri could not be verified

4.3. Token endpoint

4.3.1. Threat: Eavesdropping access tokens

The OAuth specification does not describe any mechanism for protecting Tokens from eavesdroppers when they are transmitted from the Service Provider to the Client.
Countermeasures:

- *Service Providers MUST ensure that these transmissions are protected using transport-layer mechanisms such as TLS or SSL (see Section 5.1.1).

- *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: Obtain 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:

- *System security measures - Section 5.1.4.1.1
- *Store access token hashes only - Section 5.1.4.1.3
- *Standard SQL inj. Countermeasures - Section 5.1.4.1.2

4.3.3. **Threat: Obtain client credentials over non secure transport**

An attacker could attempt to eavesdrop the transmission of client credentials between client and server during the client authentication process or during Oauth token requests. Impact: Revelation of a client credential enabling the possibility for phishing or immitation of a client service.
Countermeasures:

- *Implement transport security through Confidentiality of Requests
- *Alternative authentication means, which do not require to send plaintext credentials over the wire (Examples: Digest authentication)

4.3.4. **Threat: Obtain 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:

- *Ensure proper handling of credentials as per Credential storage protection.

4.3.5. **Threat: Obtain client secret by online guessing**

An attacker may try to guess valid client_id/secret pairs. Impact: disclosure of single client_id/secret pair.
Countermeasures:

- *High entropy of secrets - Section 5.1.4.2.2
- *Lock accounts - Section 5.1.4.2.3

4.3.6. DoS on dynamic client secret creation

If a Service Provider includes a nontrivial amount of entropy in client secrets and if the service provider automatically grants them, an attacker could exhaust the pool by repeatedly applying for them.
Countermeasures:

- *The service provider should consider some verification step for new clients. The service provider should include a nontrivial amount of entropy in client secrets.

4.4. Obtaining Authorization

This section covers threats which 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 tokens endpoint, respectively.

4.4.1. Authorization Code

4.4.1.1. Threat: Malicious client obtains authorization

Attacker abuses valid client id
countermeasures

- *client validation

- *client authentication

- *user consent

4.4.1.2. Threat: Eavesdropping authorization codes

The OAuth specification does not describe any mechanism for protecting authorization codes from eavesdroppers when they are transmitted from the Service Provider to the Client and where the Service Provider Grants an Access Token.
Note: A description of a similar attack on the SAML protocol can be found at http://www.oasis-open.org/committees/download.php/3405/oasis-sstc-saml-bindings-1.1.pdf (§4.1.1.9.1).
Countermeasures:

- *The authorization server SHOULD enforce a one time usage restriction (see Section 5.1.5.4).

- *Authorization server as well as the client MUST ensure that these transmissions are protected using transport-layer mechanisms such as TLS or SSL (see Section 5.1.1).

*The authorization server shall 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).

*Limited duration of authorization codes - Section 5.1.5.3

*If an Authorization Server observes multiple attempts to redeem a authorization code, the Authorization Server may want to revoke all tokens granted based on the authorization code (see Section 5.2.1.1).

*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.

4.4.1.3. Threat: Obtain 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:

*Credential storage protection can be employed - Section 5.1.4.1

*System security measures - Section 5.1.4.1.1

*Store access token hashes only - Section 5.1.4.1.3

*Standard SQL inj. Countermeasures - Section 5.1.4.1.2

4.4.1.4. Threat: Online guessing of authorization codes

An attacker may try to guess valid authorization code values and send it using the grant type „code“ in order to obtain a valid access token. Impact: disclosure of single access token (+probably refresh token)
Countermeasures:

*For handle-based designs: Section 5.1.5.11

*For assertion-based designs: Section 5.1.5.9

*Binding of authorization code to client_id, adds another value the attacker has to guess - Section 5.2.4.4

* Binding of authorization code to redirect_uri, adds another value the attacker has to guess - Section 5.2.4.5

* Short expiration time - Section 5.1.5.3

4.4.1.5. Threat: Authorization code leaks when requesting access token

Authorization codes are passed via the browser which may unintentionally leak those codes to untrusted web sites and attackers by different ways:

* Referer headers: browsers frequently pass a “referer” header when a web page embeds content, or when a user travels from one web page to another web page. These referer headers may be sent even when the origin site does not trust the destination site. The referer header is commonly logged for traffic analysis purposes.

* Request logs: web server request logs commonly include query parameters on requests.

* 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.

* 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.

Similar attacks on the SAML protocol are discussed in: http://www.thomasgross.net/publications/papers/GroPfi2006-SAML2_Analysis_Janus.WSSS_06.pdf and http://www.oasis-open.org/committees/download.php/11191/sstc-gross-sec-analysis-response-01.pdf. Countermeasures:

* The authorization server shall 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).

* Authorization codes must be time-limited (see Section 5.1.5.3)

* Authorization codes should be single-use tokens (Section 5.1.5.4)

* If an Authorization Server observes multiple attempts to redeem a authorization code, the Authorization Server may want to revoke all tokens granted based on the authorization code (see Section 5.2.1.1)

* The resource server may reload the target page of the redirect_uri in order to automatically cleanup the browser cache.

4.4.1.6. Threat: Authorization code phishing

A hostile party could act as the client web server and get access to the authorization code. This could be achieved using DNS or ARP spoofing.
Impact: This affects web applications and may lead to a disclosure of authorization codes and, potentially, the corresponding access and refresh tokens.
Countermeasures:

* The browser shall be utilized to authenticate the redirect_uri of the client using server authentication - Section 5.1.2

* The authorization server shall require the client to authenticate with a secret, so the binding of the authorization code to a certain client can be validated in a reliable way (see Section 5.2.4.4).

4.4.1.7. Threat: Session fixation

The session fixation attack leverages the 3-legged OAuth flow in an attempt to get another user to log-in and authorize access on behalf of the attacker. 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.
In this attack, the attacker is using a known client application (consumer site), and a target OAuth resource provider. The attack depends on the victim expecting the consumer site to request access to the resource provider.
The attacker utilizes the following flow:
The attacker initiates browser access to the consumer site, and initates access to data from the resource provider. The consumer site, initiates an authorization request and receives a redirect_uri back from the resource provider's authorization server. Instead of following the link, the attacker stops the process and saves the redirect_uri.
The attacker modifies the redirect_uri to allow control to be returned to the attacker site.
The attacker tricks another user (the victim) to open that redirect_uri and to authorize access (e.g. an email link, or blog link). The way the attacker achieve that goal is out of scope.
Having clicked, the link, the victim is requested to authenticate and authorize the consumer site to have access.
The authorization server redirects the user agent to the attackers web site instead of the original target web site.

The attacker obtains the authorization code from its web site, constructs a redirect_uri to the target web site (or app) based on the original authorization request's redirect_uri and the newly obtained authorization code and directs its user agent to this URL.
The web 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.
Countermeasures:

* The attacker must use another redirect_uri for its authorization process 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, such a change (and with that such an attack) can be detected - see Section 5.2.4.4

* The authorization server may also enforce the usage and validation of pre-registered redirect Uris (see Section 5.2.3.5).

* For native apps, one could also consider to use deployment-specific client ids and secrets (see Section 5.2.3.4, along with the binding of authorization code to client_id (see Section 5.2.4.4), to detect a session fixation because the attacker does not have access the deployment-specific secret. Thus he will not be able to exchange the authorization code.

* The client may consider to use other flows, which are not vulnerable to session fixation attacks (see Section 4.4.2 or Section 4.4.3).

4.4.1.8. **Threat: DoS, Exhaustion of resources attacks**

If a Service Provider 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 code or access tokens per user, an attacker could exhaust the pool by repeatedly directing user(s) browser to request code or access tokens. This is because more entropy means a larger number of tokens can be issued.
Countermeasures:

* The service provider should consider limiting the number of access tokens granted per user. The service provider should include a nontrivial amount of entropy in authorization codes.

4.4.2. Implicit Grant

4.4.2.1. Threat: Access token leak in transport/end-points

Description: the access token is directly returned to the client as part of the redirect URL. This token might be eavesdropped by an attacker. The token is sent from server to client via a URI fragment of the redirect_uri. If the communication is not secured or the end-point 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:

- *Confidentiality of Requests - Section 5.1.1

- *Bind token to client id - Section 5.1.5.8

4.4.2.2. Threat: Access token leak in browser history

An attacker could obtain the token from the browsers history.
Countermeasures:

- *Shorten token duration (see Section 5.1.5.3) and reduced scope of the token may reduce the impact of that attack (see Section 5.1.5.1).

- *Make these requests non-cachable

- *Native apps can directly embedd a browser widget and therewith gain full control of the cache. So the app can cleanup browser history after authorization process.

4.4.2.3. Threat: Malicious client obtains authorization

An malicious client could attempt to obtain a token by fraud. Client secrets are not an effective countermeasure in this case.
The following countermeasures are advisable:

- *Always require user consent and let end-user validate client identity - Section 5.2.4.3

- *No automatic processing of repeated authorizations - Section 5.2.4.1

4.4.3. Resource Owner Password Credentials

The “password” grant type (see OAuth Core Section 4.3), often used for legacy/migration reasons, allows a client to request an access token using an end-users user-id and password along with its own credential. The “password” 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.
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-living tokens and pass them up to a attacker web service for further abuse. The client, eavesdroppers, or end-points could eavesdrop user id and password.
Countermeasures:

* Except for migration reasons, minimize use of this grant type

* The authorization server must validate the client id associated with the particular refresh token with every refresh request - Section 5.2.2.2

* Service Providers MUST ensure that these transmissions are protected using transport-layer mechanisms such as TLS or SSL (see Section 5.1.1).

4.4.3.1. Threat: Accidental exposure of passwords at client site

If an authorization server does not provide enough protection, an attacker or disgruntled employee could retrieve the passwords for a client
Countermeasures:

* Use other flows, which do not rely on the client's cooperation for secure resource owner credential handling

* Use digest authentication instead of plaintext credential processing

* Obfuscation of 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:

* Use other flows, which do not rely on the client's cooperation for resource owner interaction

* 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 permission for client using this flow shall be.

* The authorization server could notify the resource owner by an appropriate media, e.g. e-Mail, 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:

* Use other flows, which do not rely on the client's cooperation for resource owner interaction

* 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 (cf. client authentication), the authorization server could relax that restriction. Resource owners may allow or deny (e.g. in their preferences) to issue refresh tokens using this flow as well.

* The authorization server could notify the resource owner by an appropriate media, e.g. e-Mail, of the refresh token issued (see Section 5.1.3).

4.4.3.4. Threat: Obtain user passwords on transport

An attacker could attempt to eavesdrop the transmission of end-user credentials with the grant type „password“ between client and server.
Impact: disclosure of a single end-users password.
Countermeasures:

* Confidentiality of Requests - Section 5.1.1

* alternative authentication means, which do not require to send plaintext credentials over the wire (Examples: Digest authentication)

4.4.3.5. Threat: Obtain 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 service providers domain since many users tend to use the same credentials on different services.
Countermeasures:

*Credential storage protection can be employed - 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:

*Password policy - Section 5.1.4.2.1

*Lock accounts - Section 5.1.4.2.3

*Tar pit - Section 5.1.4.2.4

*CAPTCHA - Section 5.1.4.2.5

*Abandon on grant type „password“

*Client authentication (see Section 5.2.3) will provide another authentication factor and thus hinder the attack.

4.4.4. Client Credentials

[TBD]

4.5. Refreshing an Access Token

4.5.1. Threat: Eavesdropping refresh tokens from authorization server

The OAuth specification does not describe any mechanism for protecting Tokens from eavesdroppers when they are transmitted from the Service Provider to the Client.
Countermeasures:

*Service Providers MUST ensure that these transmissions are protected using transport-layer mechanisms such as TLS or SSL (see Section 5.1.1).

- *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:

- *Credential storage protection - Section 5.1.4.1

- *Bind token to client id, if the attacker cannot obtain the required id and secret - Section 5.1.5.8

4.5.3. Threat: Obtain 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 single refresh token and derivable access tokens.
Countermeasures:

- *For handle-based designs - Section 5.1.5.11

- *For assertion-based designs - Section 5.1.5.9

- *Bind token to client id, because the attacker would guess the matching client id, too - Section 5.1.5.8

4.5.4. Threat: Obtain 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 suceed.
Countermeasures:

- *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 client and resource server. As access tokens are shared secrets between authorization and resource server, they MUST by treated with the same care as other credentials (e.g. end-user passwords).
Countermeasures:

*Access tokens sent as bearer tokens, SHOULD NOT be sent in the clear over an insecure channel. Instead transport protection means shall be utilized to prevent eavesdropping by an attacker (see Section 5.1.1).

*A short lifetime reduces impact in case tokens are compromised (see Section 5.1.5.3).

*The access token can be bound to a client's identity and require the client to authenticate with the resource server (see Section 5.4.2). Client authentication MUST be performed without exposing the required secret to the transport channel.

4.6.2. Threat: Replay authorized resource server requests

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

*The resource server should utilize transport security measure in order to prevent such attacks (see Section 5.1.1). This would prevent the attacker from capturing valid requests.

*Alternatively, the resource server could employ signed requests (see Section 5.4.3) along with nounces and timestamps in order to uniquely identify requests. The resource server MUST detect and refuse every replayed request.

4.6.3. Threat: Guessing access tokens

Where the token is a handle, the attacker may use 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:

*Handle Tokens should have a reasonable entropy (see Section 5.1.5.11) in order to make guessing a valid token value difficult.

*Assertion (or self-contained token) tokens contents SHALL be protected by a digital signature (see Section 5.1.5.9).

- *Security can be further strengthened by using a short access token duration (see Section 5.1.5.2 and Section 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 tokens to this counterfeit resource server, the server in turn may use that token to access other services on behalf of the resource owner.
Countermeasures:

- *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 address is well-known to the client, it may authenticate the resource servers (see Section 5.1.2).

- *Associate the endpoint address of the resource server the client talked to with the access token (e.g. in an audience field) and validate association at legitimate resource server. The endpoint address validation policy may be strict (exact match) or more relaxed (e.g. same host). This would require to tell the authorization server the resource server endpoint address in the authorization process.

- *Associate an access token with a client and authenticate the client with resource server requests (typically via signature in order to not disclose secret to a potential attacker). This prevents the attack because the counterfeit server is assumed to miss the capabilities to correctly authenticate on behalf of the legitimate client to the resource server (Section 5.4.2).

- *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 servers. Similarily, a client could try to use a token obtained for one server on another resource server.
Countermeasures:

- *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

The HTTP Authorization scheme (OAuth HTTP Authorization Scheme) is optional. However, [RFC2616](Fielding, R., Gettys, J., Mogul, J.,

Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, “Hypertext Transfer Protocol -- HTTP/1.1,” .) 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 retrievable from) publicly-accessible caches.
CounterMeasures:

- Service Providers not using the HTTP Authorization scheme (OAuth HTTP Authorization Scheme - see Section 5.4.1) should take care to use other mechanisms, such as the Cache-Control header, to ensure that authenticated content is protected.

- 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 logfiles and HTTP referrers

If access tokens are sent via URI query parameters, such tokens may leak to log files and HTTP referrers.
Countermeasures:

- Use authorization headers or POST parameters instead of URI request parameters (see Section 5.4.1).

- Set logging configuration appropriately

- Prevent unauthorized persons from access to system log files (see Section 5.1.4.1.1)

- HTTP referrers can be prevented by reloading the target page again without URI parameters

- Abuse of leaked access tokens can be prevented by enforcing authenticated requests (see Section 5.4.2).

- 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 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 as described in Section 4.

5.1. General

5.1.1. Confidentiality of Requests

This is applicable to all requests sent from client to 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 attacks through using content of request, e.g. secrets or tokens, or mount replay attacks.
Attacks can be mitigated by using transport-layer mechanisms such as TLS or SSL. VPN may considered as well.
This is a countermeasure against the following threats:

- Replay of access tokens obtained on tokens endpoint or resource server's endpoint

- Replay of refresh tokens obtained on tokens endpoint

- Replay of authorization codes obtained on tokens endpoint (redirect?)

- Replay of user passwords and client secrets

5.1.2. 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 DNS name of the server to the public key presented by the server during connection establishment.
The client MUST validate the binding of the server to its domain name. If the server fails to prove that binding, it is condered a men-in-the-middle. The 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:

- Spoofing

- Proxying

- Phishing by conterfeit 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 shall always be in control of the authorization processes and get the necessary information to meet 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:

*User consent forms

*Notification messages (e-Mail, SMS, …)

*Activity/Event logs

*User self-care apps or portals

5.1.4. Credentials

This sections describes countermeasures used to protect all kind 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 token) or authorization codes.

5.1.4.1. Credential storage protection

5.1.4.1.1. Standard system security means

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

5.1.4.1.2. Standard SQL inj. Countermeasures

[TBD]

5.1.4.1.3. No cleartext storage of credentials

The authorization server may consider to not store credential in clear text. Typical approaches are to store hashes instead. If the credential lacks a reasonable entropy level (because it is a user password) an additional salt will harden the storage to prevent offline dictionary attacks. 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.

5.1.4.1.4. Encryption of credentials

[TBD]

5.1.4.1.5. Use of asymmetric cryptography

Usage of asymmetric cryptography will free the authorization server of the obligation to manage credentials. Nevertheless, it MUST ensure the integrity of the respective public keys.

5.1.4.2. Online attacks on secrets

5.1.4.2.1. Password policy

The authorization server may decide to enforce a complex user password policy in order to increase the user passwords' entropy. This will hinder online password attacks.

5.1.4.2.2. High entropy of secrets

When creating token handles or other secrets not intended for usage by human users, the authorization server MUST include a reasonable level of entropy in order to mitigate the risk of guessing attacks.
The token value MUST be constructed from a cryptographically strong random or pseudo-random number sequence [RFC1750] generated by the Authorization Server. The probability of any two Authorization Code values being identical MUST be less than or equal to 2^(-128) and SHOULD be less than or equal to 2^(-160).

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. 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 attackes 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. Usage of CAPTCHAs

The idea is to prevent programms from automatically checkinga 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. Basis of this decision is out of scope, examples are:

- a client-specific policy, e.g. issue only less powerful tokens to unauthenticated clients,

- a service-specific policy, e.g. it a very sensible service,

- a resource-owner specific setting, or

- 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 gran type username/password. This means will reduce the impact of the following threats:

- token leakage

- token issuance to malicious software

- unintended issuance of to powerful tokens with resource owner credentials flow

5.1.5.2. 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.

5.1.5.3. Short expiration time

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

- replay

- reduce impact of token leak

- reduce likelyhood of sucessful online guessing

Note: Short token duration requires preciser clock synchronisation between authorization server and resource server. Furthermore, shorter duration may require more token refreshments (access token) or repeated end-user authorization processes (authorization code and refresh token).

5.1.5.4. Limit number of usages/ One time usage

The authorization server may restrict the number of request, which can be performed with a certain token. This mechanism can be used to mitigate the following threats:

- replay of tokens

- reduce likelyhood of sucessful online guessing

Additionally, If an Authorization Server observes multiple attempts to redeem a authorization code, the Authorization Server may want to revoke all tokens granted based on the authorization code.

5.1.5.5. Bind tokens to a particular resource server (Audience)

Authorization servers in multi-service environments may consider to issue tokens with different content to different resource servers and to explicitely indicate in the token the target server a token is intended to be sent to (cf. Audience in SAML Assertions). This countermeasure can be used in the following situations:

- It reduce the impact of a successful replay attempt, since the token is applicable to a single resource server, only.

- It prevents abuse of a token by a rough resource server or client, since the token can only be used on that server. It is rejected by other servers.

- It reduce the impact of a leakage of a valid token to a conterfeit resource server.

5.1.5.6. Use endpoint address as token audience

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

5.1.5.7. Audience and Token scopes

Deployments may consider to use only tokens with explicitely defined scope, 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 then the intended purpose.

5.1.5.8. Bind token to client id

An authorization server may bind a token to a certain client identity. This identity match must be validated for every request with that token. This means can be used, to

- *detect token leakage and

- *prevent token abuse.

Note: Validating the client identity may require the target server to authenticate the client's identity. This authentication can be based on secrets managed independent 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. Signed tokens

Self-contained tokens shall be signed in order to detect any attempt to modify or produce faked tokens.

5.1.5.10. Encryption of token content

Self-contained may be encrypted for privacy reasons or to protect system internal data.

5.1.5.11. Random token value with high entropy

When creating token handles, the authorization server MUST include a reasonable level of entropy in order to mitigate the risk of guessing attacks. The token value MUST be constructed from a cryptographically strong random or pseudo-random number sequence [RFC1750] generated by the Authorization Server. The probability of any two Authorization Code values being identical MUST be less than or equal to 2^(-128) and SHOULD be less than or equal to 2^(-160).

5.1.6. Access tokens

- *keep them in transient memory (accessible by the client app only)

- *exposure to 3rd parties (malicious app)

- *limit number of access tokens granted to a user

5.2. Authorization Server

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 a authorization code, the Authorization Server may want to revoke all tokens granted based on the authorization code.

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 areo long term credentials, they may be subject 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 MUST bind every refresh token to the id of the client such a token was originally issued to and validate this binding for every request to refresh that token. This measure is a countermeasure against refresh token theft or leakage.
Note: This binding MUST be protected from unauthorized modifications.

5.2.2.3. Refresh Token Replacement

Refresh token replacement 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 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 tokens 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. Refresh Token Revocation

The authorization server may allow clients or end-users to explicitely request the invalidation of refresh tokens.
This is a countermeasure againts:

    *device theft

* ...

5.2.2.5. Combine refresh token requests with user-provided secret

The exchange of a refresh token can be bound to the presence of a certain user-provided secret, such as a PIN, a password or a SIM card. This is a kind of multi-factor authentication on the tokens endpoint, since an attacker must possess both factors in order to be able to obtain an access token.

5.2.2.6. Device identification

The authorization server may require to bind authentication credentials to a device identifier or token assigned to that device. As the IMEI can be spoofed, that is not suitable, For mobile phones, a registration process can be used to assign a unique token to the device using an sms message. That token or identifer can then be validated with when authenticating user credentials.
This is a countermeasure against the following threats:

* phishing 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 a

* Collate sub-sequent requests to the same client,

* Indicate the trustworthiness of a particular client to the end-user,

* Authorize access of clients to certain features on the authorization or resource server, and

* Log a client identity to log files for analysis or statics.

Due to the different capabilities and characterictics of the different client types, there are different ways to support achieve objectives, which will be described in this section. Generally spoken, authorization server providers should be aware of the security policy and deployment of a particular clients and adapt its treatment accordingly. For example, one approach could be to treat all clients as less trustworthy and unsecure. On the other extrem, a service provider could activate every client installation by hand of an administrator and that way gain confidence in the identity of the software package and the security of the environment the client is installed in. And 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, if these cannot sufficiently protect it. This prevents the server from overestimating the value of a sucessful authentication of the client. For example, it is of limited benefit to create a single client id and secret which is shared by all installations of a native app. First of all, this secret must be somehow transmitted from the developer via the respective distribution channel, e.g. an app market, to all installations of the app on end-user devices. So the secret is typically burned into the source code of the app or a associated resource bundle, which cannot be entirely protected from reverse engineering. Second, effectively such secrets cannot be revoked since this would immediatly put all installations out of work. Moreover, since the authorization server cannot really trust on the client's identity, 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. Client_id only in combination with forced user consent

The authorization may issue a client id, but only accept authorization request, which are approved by the end-user. This measure precludes automatic authorization processes. This is a countermeasure for clients without secret against the following threats:

*...

*...

5.2.3.3. Client_id only in combination with redirect_uri

The authorization may issue a client id, but bind this client_id to a certain pre-configured redirect_uri. So any authorization request with another redirect_uri is refused automatically. Alternatively, the authorization server may not accept any dynamic redirect_uri for such a client_id and instead always redirect to the well-known pre-configured redirect_uri. This is a countermeasure for clients of LOA 2 against the following threats:

*...

*...

5.2.3.4. Deployment-specific client secrets

A authorization server may issue separate client ids and corresponding secrets to the different deployments of a client.

For web applications, this could mean to create one client_id and client_secret per web site a software package is installed on. So the provider of that particular site could request client id and secret from the authorization server during setup of the web site. This would also allow to validate some of the properties of that web site, such as redirect_uri, address, and whatever proofs useful. The web site provider has to ensure the security of the client secret on the site. As a result, such client could reach LOA 7.
For native applications, things are more complicated because every installation of the app on any device is another deployment. Deployment specific secrets will require

1. Either to obtain a client_id and client_secret during download process from the app 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 to achieve LOA 7, whereas the second option does not allow to validate properties of the client thus can achieve at most LOA 6. But this would at least help to prevent several replay attacks. Moreover, deployment-specific client_id and secret allow to selectively revoke all refresh tokens of a specific deployment at once. This is a countermeasure against the following threats:

*...

*...

5.2.3.5. Validation of pre-registered redirect_uri

An authorization server may require clients to register their redirect_uri or a pattern (TBD: make definition more precise) thereof. The way this registration is performed is out of scope of this document. Every actual redirect_uri sent with the respective client_id to the end-user authorization endpoint must comply with that pattern. Otherwise the authorization server must assume the inbound GET request has been sent by an attacker and refuse it.
Note: the authorization server MUST NOT redirect the user agent back to the redirect_uri of the authorization request.

*Session fixation: allows to detect session fixation attempts already after first redirect to end-user authorization endpoint

*For clients of LOA 2/5/7, this measure also helps to detect malicious apps early in the end-user authorization process. This reduces the need for a interactive validation by the user.

The underlying assumption of this measure is that an attacker must 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 victims device to circumvent this security measure. This is a countermeasure against the following threats:

- *session fixation
- *malicious apps (for deployment-specific clients with secret)

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, it only works for clients bound to certain deployments at development/configuration time. As soon as dynamic resource server discovery gets involved, that's no longer feasable.

5.2.3.6. Client secret revocation

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 be unintentionally for client identifiers and secrets used across multiple deployments of a particular native or web application.
This a countermeasure against:

- *...
- *...

5.2.3.7. Use strong client authentication (e.g. client_assertion / client_token)

Assumption: prevents an attacker from obtaining a client secret because this secret is kept in some hardware security module?

5.2.4. End-user authorization

5.2.4.1. Automatic processing of repeated authorizations requires client validation

Service providers should not automatically process repeat authorizations where the client is not authenticated through a client secret or some other authentication mechanism such as signing with security certs (5.7.2.7. Use strong client authentication (e.g. client_assertion / client_token)) or validation of a pre-registered redirect uri (5.7.2.5. Validation of pre-registered redirect_uri).

5.2.4.2. Informed decisions based on transparency

The authorization server shall intelligible explain to the end-user what happens in the authorization process and what the consequences are. For example, the user shall understand what access he is about to grant to which client for what duration. It shall also be obvious to the user, whether the server is able to reliably certify certain client properties (web site address, 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-users can be involed 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 app the end-user is using. This measure is especially helpful in all situation where the authorization server is unable to authenticate the client. It is a countermeasure against:

*Malicious app

*...

5.2.4.4. Binding of authorization code to client_id

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

*Session fixation since an attacker cannot use another client_id to exchange an authorization code into a token

*Online guessing of authorization codes

Note: This binding MUST be protected from unauthorized modifications.

5.2.4.5. Binding of authorization code to redirect_uri

The authorization server MAY bind every authorization code to the redirect_uri used as redirect target of the client in the end-user authorization process. This binding MUST be validated when the client attempts to exchange the respective authorization code for an access token. This measure is a countermeasure against session fixation since an attacker cannot use another redirect_uri to exchange an authorization code into a token.

5.3. Client App Security

5.3.1. Don't store credentials in code or resources bundled with software packages

[Anything more to say ? :-)]

5.3.2. Standard web server protection measures (for config files and databases)

5.3.3. Store secrets in a secure storage

The are different way to store secrets of all kinds (tokens, client secrets) securely on a device or server.
Most multi-user operation systems seggregate the personal storage of the different system users. Moreover, most modern smartphone operating systems even support to store app-specific data in separat areas of the file systems and protect it from access by other apps. Additionally, apps can implements confidential data itself using a user-supplied secret, such as PIN or password.
Another option is to swap refresh token storage to a trusted backend server. This mean in turn requires a resilient authentication mechanisms between client and backend server. Note: Applications must ensure that confidential data are kept confidential even after readin from secure storage, which typically means to keep this data in the local memory of the app.

5.3.4. Utilize device lock to prevent unauthorized device access

5.3.5. Platform security measures

*Validation process

*software package signatures

*Remote removal

5.4. 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 identitiy and encourage resource servers to validate that binding. This will require the resource server to authenticate the originator of a request as the legitimate owner of a particular token. There are a couple of options to implement this countermeasure:

- *The authorization server may associate the distinguished name of the client with the token (either internally or in the payload of an 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.

- *same as before, but the client uses his private key to sign the request to the resource server (public key is either contained in the token or sent along with the request)

- *Alternatively, the authorization server may issue a token-bound secret, which the client uses to sign the request. The resource server obtains the secret either directly from the authorization server or it is contained in an encrypted section of the token. 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

This mechanisms is 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 must be uniquly identifiable and must not be processed twice by the resource server. This countermeasure helps to mitigate:

- *modifications of the message and

- *replay attempts

6. IANA Considerations

This document makes no request of IANA.
Note to RFC Editor: this section may be removed on publication as an RFC.

7. Acknowledgements

We would like to thank Francisco Corella for his feedback.

8. References

8.1. Normative References

[RFC2119]	Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
[I-D.ietf-oauth-v2]	Hammer-Lahav, E, Recordon, D and D Hardt, "The OAuth 2.0 Authorization Protocol", Internet-Draft draft-ietf-oauth-v2-22, September 2011.

8.2. Informative References

[I-D.lodderstedt-oauth-revocation]	Lodderstedt, T, Dronia, S and M Scurtescu, "Token Revocation", Internet-Draft draft-lodderstedt-oauth-revocation-03, September 2011.
[I-D.ietf-oauth-v2-bearer]	Jones, M, Hardt, D and D Recordon, "The OAuth 2.0 Authorization Protocol: Bearer Tokens", Internet-Draft draft-ietf-oauth-v2-bearer-14, November 2011.
[I-D.hammer-oauth-v2-mac-token]	Hammer-Lahav, E, Barth, A and B Adida, "HTTP Authentication: MAC Access Authentication", Internet-Draft draft-hammer-oauth-v2-mac-token-05, May 2011.
[portable-contacts]	Smarr, J., "Portable Contacts 1.0 Draft C", August 2008.

Authors' Addresses

Dr.-Ing. Torsten Lodderstedt editor Lodderstedt Deutsche Telekom AG EMail: torsten@lodderstedt.net

Mark McGloin McGloin IBM EMail: mark.mcgloin@ie.ibm.com

Phil Hunt Hunt Oracle Corporation EMail: phil.hunt@yahoo.com