Security Considerations OAuth 2.0

Dr. Torsten Lodderstedt, Deutsche Telekom AG
Mark McGloin, IBM

Table of Contents

1. Introduction

<<This is a preliminary, incomplete version of the security considerations!>>
This document gives a comprehensive threat model for OAuth and describes respective counter measures.
The document shall server the following purposes:
1) Support security assessment and evolution of the protocol

- Make sure we did not forget major design flaws or security threats. Have we considered all possible (or at least a significant set of) threats? Therefore the document aims to make all those considerations transparent.
- OAuth 2.0 is a complex security protocol! Not that easy to understand the implications. In the end, every flow is another sub protocol, which needs some reasonable analysis.

2) Explain implementers what to consider and why

- OAuth 2.0 let implementers a lot of freedom
- There are aspects outside of the protocol in the responsibility of the implementer or variants of the protocol implementers can decide on - we need to guide them
- Explain implementers reasons behind design decisions and recommendations allow implementers to
  - draw their own conclusions, e.g. to conduct a risk analysis for their particular solution based on our threat model
  - come up with own solutions in case standard solution won't work
  - intentionally ignore recommendations

1.1. Structure

- Overview section
  - assumptions
    - client capabilities
    - tokens vs. assertions
    - client types
  - attacker goals
- Describe security features and countermeasures
  - Classes of counter-measures
    - Built in protocol features (client_id)
    - Protocol feature to be designed by implementers (design alternatives exist (e.g. handle vs. assertion), guidance required)
    - Additional counter measures (out of scope for protocol, recommendation – like dynamic client registration, at least currently)
- Threat Model
  - The threat model is organized as a tree (inspired by Attack Trees by Bruce Schneier)
  - Top Level are the different components of the protocol, detailed by one or n more levels of sub-sections. The leafs of this structure are the particular threats.
  - Every threat is linked to one or countermeasures
- The idea is to, based on this document, distill a list of measures and considerations per flow and entity (authorization server, resource server, client), which can be incorporated into the OAuth core specification.

1.2. Scope

- All client profiles (focus on web server and native applications in the first phase)
    - WebServer
    - User Agent
    - Native Applications
        - Apps on mobile devices
        - Apps on Desktop PCs
    - Autonomous
- Deployment architecture:
    - The security considerations document only considers clients bound to a particular deployment as supported by OAuth draft -10. Such deployments have the following characteristics:
    - Proprietary resource API
    - At least resource server URLs are static and well-known at development time, authorization server URLs can be static or discovered
    - Token scope (applicable URLs) are well-known at development time
    - Client apps registered with authorization server during development time

1.3. Out-Of-Scope

- Communication between authorization server and resource server
- Token formats
- Interoperable clients
    - standards APIs, discovery, dynamic client registration
    - same attacks but other counter-measures required since less coupling between clients and servers
    - should be aligned with further protocol evolution

2. Overview

<NOTE: This is where a deployment diagram belongs if we can define one>

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

Redirect-url: Prevents phishing attacks where the url is pre-registered as token will authorization code or token will be directed to that url

Scope: Can be controlled by Authorization server and used to limit access to resources for oauth clients the authorization server deems less secure or where sending tokens over non secure channels

Expires_In: Used to limit the lifetime of the access token for oauth clients the authorization server deems less secure or where sending tokens over non secure channels

Refresh Token: Coupled with a short access token lifetime, can be used to grant longer access to resources without involving end user authorization. This only offers an advantage where resource servers are distributed as the refresh token must always be exchanged at the authorization server. The authorization server can revoke the refresh token at any time.

POST: Acquiring a token uses POST to ensure no logging or caching of requests

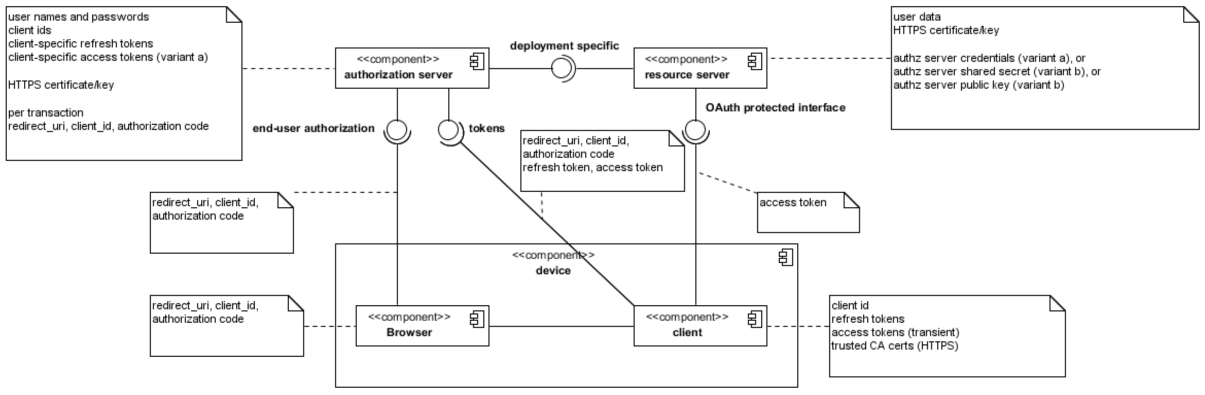
Client Secret: Used to verify the client identifier. This should only be used where the client is capable of keeping it secret secure, e.g. A desktop application can not keep a secret secure

2.1. OAuth Design alternatives

- Authorization server and resource server
    - Both concepts can be implemented in the same entity, they may also be different entities. The later is typically the case for multi-service providers with a single authentication and authorization system.
- Tokens: handle vs. assertion
    - handles: a reference to some internal data structure within the authorization or resource server, the internal data structure contains the attributes of the token, such as user id, scope, …. Handles typically require a communication between resource server and authorization server in order to validate the token and obtain token-bound data. This allows for an easy implementation of revokation mechanisms.
    - assertions (aka self-contained token): a document containing identifiers and attributes of a user. An assertion typically has a duration, an audience and is digitally signed. 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. Implementing token revocation is more difficult with assertions than handles.
- Authorization code: handle vs. assertion
    - the design alternatives are essentially the same as for tokens. Authorization codes typically have a short lifetime and/or a one-time usage restriction.

2.2. Architectural assumptions

Different components within the OAuth architecture have different limitations or features which must be considered when building a threat model. These features and any assumptions in relation to the different components are are documented here. It covers both the Oauth components, such as authorization server, and the different application types that will be used to acquire and use oauth tokens, e.g. Native clients.
The following figure shows the different parties involved in the OAuth protocol along the communication paths among them. The notes depict the data stored in the different parties as well as the data sent over the different channels. This information will help to identify possible attack targets and to assess an attack's impact.



user names and passwords
client ids
client-specific refresh tokens
client-specific access tokens (variant a)
HTTPS certificate/key
per transaction
redirect_uri, client_id, authorization code
<<component>> authorization server
deployment specific
<<component>> resource server
user data
HTTPS certificate/key
authz server credentials (variant a), or
authz server shared secret (variant b), or
authz server public key (variant b)
end-user authorization
tokens
OAuth protected interface
redirect_uri, client_id, authorization code
refresh token, access token
redirect_uri, client_id, authorization code
access token
<<component>> device
redirect_uri, client_id, authorization code
<<component>> Browser
<<component>> client
client id
refresh tokens
access tokens (transient)
trusted CA certs (HTTPS)

2.2.1. Authorization Server

- stores refresh tokens, user password and client secrets, only

- Access tokens could be stored on the AS as well (depending on the design), handle vs. assertion

2.2.2. Resource Server

- Does not know refresh tokens, user passwords or client secrets

2.2.3. Web Server

- Such clients typically represent a web site with its own user management and login mechanism
- 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.
- Implementation options
  - grant type code

2.2.4. User Agent

- TBD
- Implementation options
  - response type token
  - response type code_and_token

2.2.5. Native clients

- OAuth clients (apps) running on a user-controlled device: Notebook, PC, Tablet, Smartphone, Gaming Console
- App registered with authz server during development time, all installations use same static client_id
- Implementation options
  - grant type code (embedded or external browser)
  - grant type token
  - grant type code_and_token
  - grant type password
- Cannot keep per software package secrets confidential (source code or static resources) → Recommendation: don't use such client secrets???
  - What is the purpose of client authentication?
  - Detect malicious clients?
    - Client authenticity can be validated by the platform (package signing) and/or the user
    - Moreover, evil clients shall be identified by anti-virus software
  - Authorize for powerful request types?
    - Which?
- Ability to protect per installation/instance secrets to some extend
  - Platform
    - Lock
    - App segregation
    - Protected user context (PC/Notebook/Tablets w/ multi-user operation systems)
  - Application
    - Data encryption based on user-supplied PIN

- Some devices can be identified/authenticated
    - Handsets, smartphones – IMEI
    - Set top boxes, gaming consoles, others – certificates and TPM module
    - …
    - Does not help to identify client apps but to bound tokens to devices and to detect token theft
- Only a few authz processes per installation (once per months?)
- Security breach will affect single user (or a few) only
- We do not try to countermeasure attackers who control the device entirely
- Trust
    - End user trusts authorization and resource server
    - **End user does not trust client (?)**

2.2.5.1. Mobile Clients

- Native clients running on mobile devices, such as handsets or smartphones
- Limited input capabilities, therefore such clients typically obtain a refresh token in order to provide automatic login for sub-sequent application runs
- Can get cloned, stolen or lost
- Some devices can be bound to a user identity, e.g. by utilizing a SIM card
    - Does not help to identify client apps but to bound tokens to devices and to detect token theft
- Some are personal devices, others are shared devices

2.2.6. Autonomous

TBD

3. Security Design/ Countermeasures

This considerations apply to more than one client profile using Oauth and are augmented by more specific security considerations per client profile and use cases for those client profiles.

3.1. Credentials and token exchange

This applies to sending client secrets and access tokens
Threat: The OAuth specification does not describe any mechanism for protecting Tokens and secrets from eavesdroppers when they are transmitted from the Service Provider to the Client and where the Service Provider Grants an Access Token.
Countermeasures: Service Providers should ensure that these transmissions are protected using transport-layer mechanisms such as TLS or SSL.

3.2. Confidentiality of Requests

This is applicable to all requests sent from client to authorization server or resource server.
Threat: 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
Countermeasures: Service Providers should carefully consider the kinds of data likely to be sent as part of such requests, and should employ transport-layer security mechanisms to protect sensitive resources.

3.3. DoS, Exhaustion of resources attacks

Applicable to both client secrets and obtaining end user authorization
Threat: 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.
Similarly if a Service Provider includes a nontrivial amount of entropy in authorization codes or access 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.
The service provider should consider limiting the number of access tokens granted per user.

3.4. Replay Attacks

Threat: Replay attacks can be used to replay requests.
Countermeasures: These attacks can be mitigated by using transport-layer mechanisms such as TLS or SSL.

3.5. Authorization server (end-user authorization)

3.5.1. Spoofing by counterfeit Service Providers

This is applicable to any request where the authenticity of the Service Provider or of request responses is an issue
Threat: OAuth makes no attempt to verify the authenticity of the Service Provider. A hostile party could take advantage of this by intercepting the Client's requests and returning misleading or otherwise incorrect responses.
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.

3.5.2. Phishing attacks

This is applicable for obtaining end user authorization
Threat: 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 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.

3.5.3. User awareness of access token scope

This is applicable for obtaining a token
Threat: When obtaining end user authenticaton, Users should 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: 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.
If the grant_type is refresh_token and scope is requested, it must be equal or less than the scope granted buy the user when the refresh token was assigned.(DOES THIS BELONG IN SPEC BODY AT SECTION 4)

3.5.4. Automatic Processing of repeat authorizations without client secrets

This is applicable to obtaining end user authorization w/o client secrets.
Threat: Service Providers may wish to automatically process authorization requests from Clients which have been previously authorized by the user. When the User is redirected to the Service Provider to grant access, the Service Provider detects that the User has already granted access to that particular Client. Instead of prompting the User for approval, the Service Provider automatically redirects the User back to the Provider.
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.
Whitelisting of callback urls can be used to mitigate attacks.

3.6. Authorization server (tokens)

3.6.1.1. Refresh Tokens with no client secret

This is applicable to refresh tokens with no client secret
Threat: Service Providers should be careful when accepting refresh tokens for any profiles with no client secret. An attacker who can steal a refresh token can use it to acquire an access token and gain access to resources.
Countermeasures: They can minimize the damage from attacks by limiting the scope where no client secret provided.
Authorization Servers may verify that the Client to which the refresh token was issued matches the Client that is requesting a fresh access token using some other authentication mechanism, such as signing. That is outside the scope of this specification
Where no client secrets are used, refresh tokens provide no additional security over access tokens

3.7. Resource server access

3.7.1.1. Security of Bearer tokens

(Note: Section 5 of protocol suggests something on bearer tokens. Do we repeat in security considerations. Also, this may be repeating more specific threats listed elsewhere in this document.)
Threats: Access tokens act as bearer tokens, where the token string acts as a shared symmetric secret. This requires treating the access token with the same care as other secrets (e.g. end-user passwords).
Countermeasures: Access tokens SHOULD NOT be sent in the clear over an insecure channel.
A short lifetime should be used for access tokens in case they are compromised.
......
Clients MUST NOT make authenticated requests with an access token to unfamiliar resource servers, regardless of the presence of a secure channel.
etc

Expiry Time Access Token
This is applicable to access tokens w/o client secret
Access tokens act as bearer tokens, where the token string acts as a shared symmetric secret. A short lifetime should be used for access tokens in case they are compromised.

3.7.1.2. Proxying and caching of secure content

This is applicable for using a token to access a Protected resource.
Threat: 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) should take care to use other mechanisms, such as the Cache-Control header, to ensure that authenticated content is protected.

3.8. Web Server

3.8.1. Cryptographic Strength of nonces

This is applicable for authorization codes used when obtaining an access token
Threat: An attacker will be able to mount an offline attack to crack authorization codes . Assigning sequential numbers leads to the possibility of an attack through guessing the next number.
Countermeasures: Care should be taken to ensure they are cryptographically strong, random or use a pseudorandom number sequence.
RFC 1750 suggests techniques for producing random quantities that will be resistant to such attack

3.8.2. Automatic Processing of repeat authorizations with client secrets

This is applicable to obtaining end user authorization where client secrets are compromised
Threat: Service Providers may wish to automatically process authorization requests from Clients which have been previously authorized by the user. When the User is redirected to the Service Provider to grant access, the Service Provider detects that the User has already granted
access to that particular Client. Instead of prompting the User for approval, the Service Provider automatically redirects the User back to the Provider.
If the Client Secret is compromised, automatic processing creates additional security risks. An attacker can use the stolen Client Secret to redirect the User to the Service Provider with an authorization request. The Service Provider will then grant access to the User's data without the User's explicit approval, or even awareness of an attack. If no automatic approval is implemented, an attacker must use social engineering to convince the User to approve access.
Countermeasures: Service Providers can mitigate the risks associated with automatic processing by limiting the scope of Access Tokens obtained through automated approvals. Access Tokens obtained through explicit User consent can remain unaffected.
Whitelisting of callback urls can also be used to mitigate attacks.
Clients can mitigate the risks by protecting their client Secret.

3.8.3. Expiry Time Access Token

This is applicable to access tokens w/o client secret
Access tokens act as bearer tokens, where the token string acts as a shared symmetric secret. A short lifetime should be used for access tokens in case they are compromised. Refresh tokens can be used to extend the access grant lifetime as they use a client secret. An attacker would need to compromise the client secret to compromise the refresh token.

3.8.4. Refresh Tokens with client secrets

This is applicable to acuiring an access token using a refresh token with a client secret
Threat: Having access to a refresh token and a compromised client secret, gives an attacker permanent access to a user's resources.
Countermeasures: Authorization Servers should verify that the Client identifier to which the refresh token was issued matches the Client identifier that is requesting a fresh access token. Where a refresh token and client secret are compromised, the service provder should be able to change the client secret and/or revoke the refresh token

3.8.5. Authorization code leaks

This is applicable to obtaining an access token using an authorization code
Threat: Authorization codes are passed via the browser which may unintentionally leak those codes to untrusted web sites and attackers.
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.
Countermeasures: Because of the sheer variety of ways a verification code may leak, multiple layers of defense are necessary.

- Authorization codes must be time-limited.
- Authorization Servers should reject verification codes that are older than a few minutes.
- Authorization codes should be single-use tokens.
- 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.
- Authorization Servers should implement other security mechanisms such as callback URL white listing or Client authentication to protect against leaked authorization codes.

3.8.6. Entropy of secrets

This is applicable to client secrets for obtaining an access token. Also for authorization codes used in acquiring a token
Threat: Unless a transport-layer security protocol is used, eavesdroppers will have full access to OAuth requests, and will thus be able to mount offline brute-force attacks to recover the client's credentials used.

Countermeasures: Use a transport level security protocol.
Where no transport-layer security protocol can be used, Service Providers should be careful to assign client Secrets which are long enough - and random enough - to resist such attacks for at least the length of time that the secrets are valid.
The same is applicable for authorization codes but Service providers should ensure they expire in a very short period due to other security considerations.

3.8.7. Plaintext storage of credentials

This is applicable to both client secrets and authorization code for obtaining an access token
Threat: The client Secret and authorization code function the same way passwords do in traditional authentication systems. In order to compute the signatures used in the non-PLAINTEXT methods, the Service Provider must have access to these secrets in plaintext form. This is in contrast, for example, to modern operating systems, which store only a one-way hash of user credentials. If an attacker were to gain access to these secrets – or worse, to the Service Provider's database of all such secrets - he or she would be able to perform any action on behalf of any User.
Countermeasures: It is critical that Service Providers protect these secrets from unauthorized access.

4. Threat Model

4.1. Common Threats

4.1.1. Leakage of confidential data in HTTP-Proxies

- Description: HTTP proxy logs tokens or client secrets to logfile
- Countermeasure
  - Usage of authorization headers
  - POST requests
  - HTTPS or comparable transport security measures (pass trough proxy)
  - Reduce: restrict token power (least privileges)
  - Reduce: restrict token duration

4.1.2. Wire tapping

- Impact: exposure of user credentials and authorization codes
- Countermeasure
  - HTTPS or comparable transport security measures
  - Reduce: restrict token power (least privileges)
  - Reduce: restrict token duration

4.1.3. Built malicious client, obtain access authorization by fraud

- **User consent** – none-interactive token issuance (like immediate mode)
- Platform security measures (App Store/Marketplace policy, digital signature of applications)
- Native clients
  - Registered redirect URL – client can always use valid redirect_uri, intercept flow and abuse code or token
  - It is probably easier for an attacker to trick the user to enter her username and password!

4.2. Authorization server

4.2.1. Password guessing

- Description: brute force or dictionary attack against end-user authorization endpoint
- Impact: single user credentials are exposed
- Countermeasures
  - Tar pit
  - Account locking

4.2.2. Obtain user credentials from authorization server database

- Impact: all user credentials are exposed
- Countermeasures
  - System security measures
  - Store end-user passwords as hashes only

4.2.3. Obtain access tokens from authorization server database

- Description: direct attack against database, SQL injection, …
- Applicable for handle-based access token designs, only
- Impact: disclosure of all refresh tokens
- Countermeasures
  - System security measures
  - Store access token hashes only
  - Standard SQL inj. Countermeasures

4.2.4. Obtain refresh tokens from authorization server database

- Description: direct attack against database, SQL injection, …
- Applicable for handle-based refresh token designs, only
- Impact: disclosure of all refresh tokens
- Countermeasures
  - System security measures
  - Store access token hashes only
  - Standard SQL inj. Countermeasures

4.2.5. End-user authorization

4.2.5.1. Password phishing by counterfeit server

- Description: men in the middle attack using DNS or ARP spoofing
- Assumption is that the client uses well-known authorization server URLs or obtains this URLs from well-known resource server URLs. So DNS or ARP spoofing is the only way to proxy the legitimate authorization server.
- Impact: single user credentials are exposed
- Countermeasures
  - HTTPS server authentication + reasonable CA policy

4.2.6. Response type token

Only suited for clients which reside on the same device as the user agent.

4.2.6.1. Access token leakage on transport

- Description: the access token is directly returned to the client as part of the redirect URL. This token might be eavesdropped by an attacker.
- Countermeasures
  - Send access tokens over encrypted channels only
  - Send access token in URI fragment instead as URI query parameter
    - token is sent from server to client via redirect (status code 302)
    - since fragements are not sent to the redirect target by the user agent, the fragment never leaves the client
    - together with encryption, this ensures confidentiality in cases where the client resides on the same devices as the user agent.

4.2.6.2. Access token in browser history

- Description: An attacker could obtain the token from the browsers history list
- Countermeasures
  - short token duration
  - native apps: embedded browser only (accessible from the outside?)
  - native apps: cleanup browser history from client code after authorization process

4.2.6.3. Malicious client obtains authorization

- Description: an malicious client could attempt to obtain a token by fraud
- Client secrets are not an effective countermeasure in this case
- countermeasures
  - user consent
  - no automatic authorization

4.2.7. Response type code_and_token

4.2.8. Grant type code

Suited for web sites as well as clients which reside on the same device as the user agent and can freely send POST requests to the authorization server.

4.2.8.1. Obtain authorization codes from authorization server database

- Description: direct attack against database, SQL injection, …
- Applicable for handle-based authorization code designs, only
- Impact: disclosure of all currently issued authorization codes
- Countermeasures
  - System security measures
  - Store access token hashes only
  - Standard SQL inj. Countermeasures

4.2.8.2. Guess authorization codes

- Description: brute force attack against authorization codes w/o context, http://www.ietf.org/mail-archive/web/oauth/current/msg03844.html
- Impact: disclosure of single access token (+probably refresh token)
- Countermeasures
  - Depending on the authorization code design

        - handle: random value with high entropy
        - assertion: signature + (some random value or at least a timestamp)
    - prevent: bound to client_id and redirect_uri
    - reduce: limited duration

4.2.8.3. Authorization code phishing (by counterfeit server)

- Description: DNS-spoofing
- Impact: disclosure of all refresh tokens
- Countermeasures
    - HTTPS server authentication + reasonable CA policy

4.2.8.4. Disclosure of authorization code on transport

- Description: attacker could eavesdrop code on transit from server via user agent to client
- Impact: disclosure of a single code
- Countermeasures:
    - HTTPS

4.2.8.5. Disclosure of authorization code in browser history

- Description: URLs are kept in browser history, user with access to the same device could obtain valid code.
- Countermeasures
    - Reload target page
    - Limited authorization code duration
    - One time use
    - prevent: revoke all issued tokens in case of detected attack (on second attempt to exchange code for token)

4.2.8.6. Disclosure of authorization code in HTTP referrer

- Description: 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.
- Countermeasures
    - Reload target page
    - Limited authorization code duration
    - One time use
    - prevent: revoke all issued tokens in case of detected attack (on second attempt to exchange code for token)

4.2.8.7. Session fixation

- Description:
    - Attacker wants to inject a victim’s identity into an application or website.
    - 1) Initiate flow in website
    - 2) Intercept redirect, modify URL to point the redirect_uri to a web site under the attackers control
    - 3) Trick the user to open that URL and to authorize access (e-Mail?)
    - 4) Authz server redirects to web site instead

    - 5) Fetch authorization code
    - 6) Re-activate web site manually on attackers device using original redirect URL + authorization code
    - 7) web site fetches refresh token from authorization server
- Countermeasures
    - Validate original redirect_uri at token endpoint
        - Authz code is bound to url_attacker - error
    - Redirect URL registration
        - allows to detect session fixation attempts already after first redirect to end-user authz endpoint
        - attacker could use own client_id → visualization in user consent?
    - Use of other flows since they are not vulnerable to session fixation attacks

4.2.9. Grant type refresh_token

4.2.9.1. Guess refresh tokens

- Description: brute force against refresh token
- Impact: exposure of single access token
- Depending on the refresh token design
    - handle: random value with high entropy
    - assertion: signature + (some random value or at least a timestamp)
- prevent: bound to client_id and scope
- prevent: tar pit

4.2.9.2. Refresh token phishing (by counterfeit authorization server)

- Description: DNS-spoofing
- Impact: disclosure of all refresh tokens
- Countermeasures
    - HTTPS server authentication + reasonable CA policy

4.2.9.3. Disclosure of refresh tokens on transport

- See 4.1.2.

4.2.10. Grant type password

- Con
    - Password is processed by 3rd party
        - Accidental loss of passwords at client site
        - Phishing (can happen even w/o this flow, the user does not differentiate)
    - No user control over authorization process –
        - Aquire more powerful tokens? client may acquire more powerful tokens
        - Aquire long-living tokens instead of short-living ones?
        - User does not realize existence of revocable tokens, he/she probably thinks the password is stored on the device
- Pro
    - Migration path from pwd-based authentication (e.g. for standard protocols like WebDAV)

    - Storing pwd can be replaced by storing refresh tokens
    - Easy to implement (UI)
- Ideas
    - Notification to user (e.g. mail or sms) that and what kind of authorization/token has been issued

4.2.10.1. Accidental exposure of passwords at client site

- Description: Passwords are processed by external parties. It might happen that password get exposed to an attacker, e.g. through log files.
- counter measures
    - Use other flows
    - Use digest authentication instead of plaintext

4.2.10.2. Brute force on username/password

- Description: Authorization servers supporting this grant type open up another interface which can be subject to online password attacks.
- Countermeasure
    - prevent: client identification/authentication
    - prevent: tar pit
    - prevent: lock accounts
    - prevent: password policy (entropy)

4.2.10.3. Obtain Password from authorization server database

See 4.2.1.

4.3. Clients

4.3.1. Web server

All threats are described in Section 4.2.5.End-user authorization, 4.2.8.Grant type code, and 4.2.9.Grant type refresh_token are valid for this implementation variant.

4.3.1.1. Obtain refresh tokens from web server

- Description: An attack may obtain the refresh tokens issued to a web server client.
- Impact: Exposure of all refresh tokens on that side.
- Countermeasures
    - invalidate client secret, issue new secret

4.3.2. User agent

- All threats are described in Section 4.2.5.End-user authorization and 4.2.6.Response type token are valid for this implementation variant.
- ...

4.3.3. Native applications

We first give threat common to all implementation variants of native applications and variant specific threats afterwards.

4.3.3.1. Refresh token leakage

4.3.3.1.1. Read refresh token from local filesystem

- Description: Requires file system access
- prevent: Operating system user segregation (login)
- prevent: Encrypt tokens with user-supplied secret
- prevent: Combine refresh token requests with user secret
    - PIN
    - Password
    - SIM Card
- prevent: Token replacement (works w/o any prerequisites) – shall be combined with revoking all associated tokens in case of an event
- prevent: Swap refresh token storage to backend server (requires resilient authentication mechanisms between client and backend server), see Fehler: Referenz nicht gefunden

4.3.3.1.2. Malicious application reads other app's data

- prevent: prevent app installation - authenticity validation (platform feature?)
- prevent: app storage seggregation
- prevent: remove app automatically using kill-switch (AppStore/Market-dependent)

4.3.3.1.3. Steal device

- prevent: Device protection (Lock)
- prevent: Application login (e.g. PIN)
- prevent: Realized by user – token revocation

4.3.3.1.4. Clone device

- Description: All device data and applications are copied to another device. Applications are used as-is on the target device.
- prevent: device authentication (token is used as is! so IMEI/Device Id should change) during access token refreshment
- prevent: Application login (e.g. PIN)
- prevent: user secret in access token refreshment process
- prevent: token replacement

4.3.3.2. Pattern: grant type code w/ external browser

All threats are described in Section 4.2.5.End-user authorization, 4.2.8.Grant type code, 4.2.9.Grant type refresh_token, and 4.3.3.1.Refresh token leakage are valid for this implementation variant.

4.3.3.2.1. Session fixation

- Description:
    - Attacker has same app as victim installed and wants to inject the victim's identity into this app.
    - 1) Initiate flow in app
    - 2) Intercept redirect, modify URL to point the redirect_uri to a web site under the attackers control
    - 3) Trick the user to open that URL and to authorize access (e-Mail?)

    - 4) Authz server redirects to web site instead
    - 5) Fetch authorization code
    - 6) Activate app manually on attackers device using original redirect URL + authorization code
    - 7) App fetches refresh token from authorization server
- Countermeasures
    - prevent: Validate original redirect_uri at token endpoint
        - Authz code is bound to url_attacker - error
        - If attacker is able to modify client app or to use own app, it just uses url_attacker to fetch the refresh tokens
    - prevent: Redirect URL registration
        - allows to detect session fixation attempts already after first redirect to end-user authz endpoint
        - attacker could use own client_id → visualization in user consent?
    - Dynamic client secret
        - Does not help
    - Password flow?

4.3.3.3. Pattern: grant type code w/ embedded browser

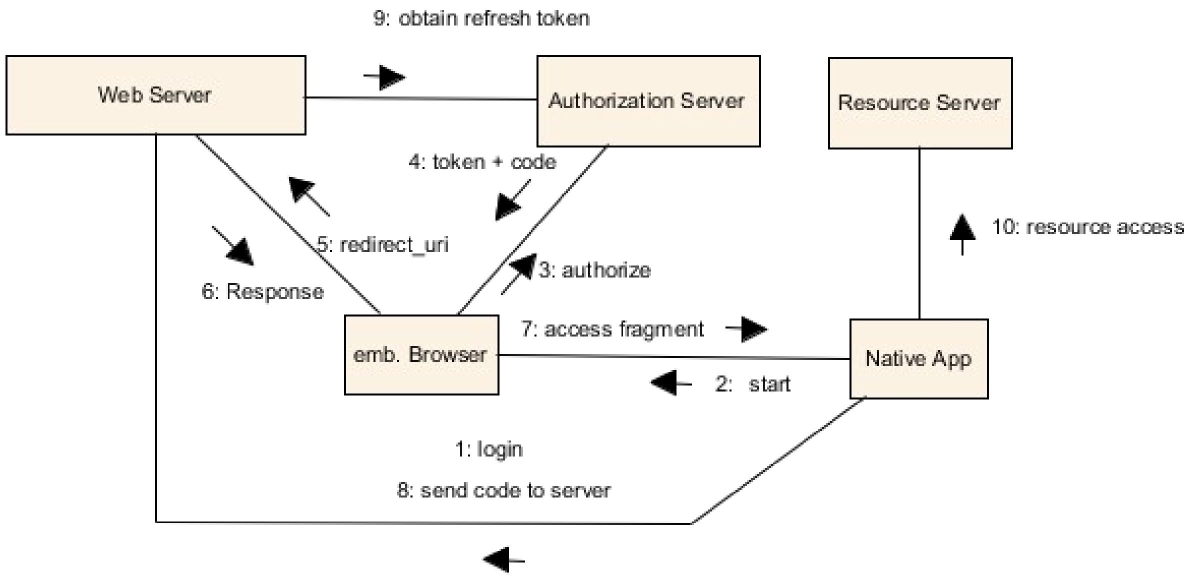
- App grabs code from URI query parameter and passes it to the AS, no web server involved, redirect target is a local resource
- Threat model given in 4.3.3.2. Pattern: grant type code w/ external browser holds with the following exceptions
    - no visual signs for protection against phishing attacks (4.2.5.1.Password phishing by counterfeit server
    - session fixation attacks are more difficult to conduct because the browser is built into the app
- Thoughts on possible improvements
    - transporting the code via fragment would be more secure since the code would not leave the user agent
    - Directly returning a refresh token via fragment would be better
        - saves one roundtrip and the (depending on design) code creation and storage
        - would eliminate the attack angle “attack code to refresh token exchange”
        - Refresh token in browser history could be a problem → HTTP referrer
        - When in fragment only – is not send out by browser with referrer header, can only be accessed on the device → access to device probably also gives access to refresh token (long-living access token may leak the same way!)

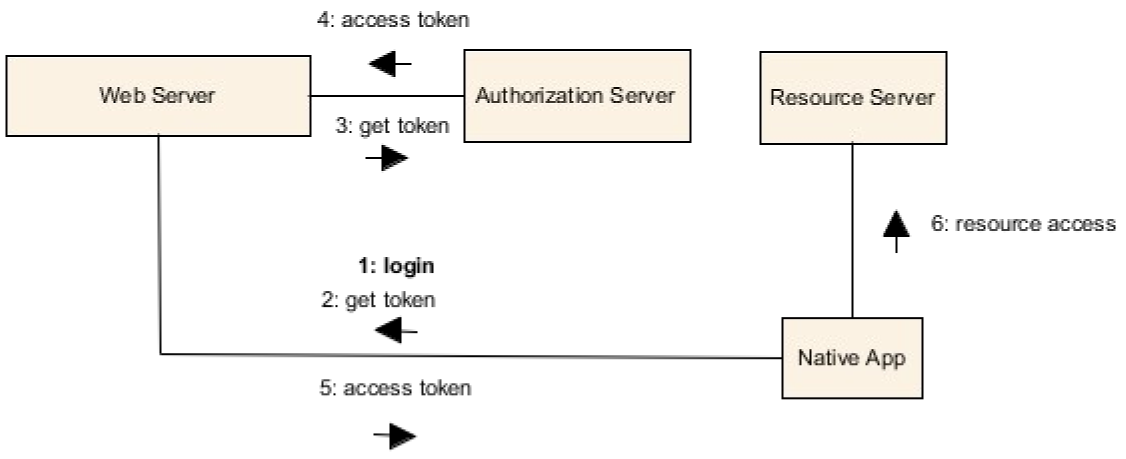
4.3.3.4. Pattern: response type token w/ embedded browser

- All threats are described in Section 4.2.5.End-user authorization and 4.2.6.Response type token are valid for this implementation variant.
- Extension to support refresh tokens reasonable (and possible in a secure way)
    - refresh token would be transported in fragment only
    - transport protection by HTTPS tunnel between client and authorization server
    - would not leave client
    - token in browser history (see above)

4.3.3.5. Pattern: response type code_and_token w/ backend server

Basically, this pattern combines a native application with a backend server. This server takes care of user login and long-term token handling. It provides the native application with suitable access tokens to interact with resource servers. The end-user authorization process is processed by the native application (by utilizing an browser). The following figures depict the flow for end-user authorization and resource server access.





The backend server has it own user management and login capabilities. So the characteristics of this solution is similar to the web server flow.
All threats are described in Section 4.2.5.End-user authorization, 4.2.8.Grant type code, and 4.2.9.Grant type refresh_token are valid for this implementation variant. This variant mitigates the threat of refresh token theft by storing those tokens in the backend.
Security depends on the quality of the login function between client and backend.

4.3.3.6. Pattern: grant type: password

- Pro
  - Migration path from pwd-based authentication (e.g. for standard protocols like WebDAV)
  - Storing pwd can be replaced by storing refresh tokens

    - Easy to implement (UI)
- The threat model of 4.2.10.Grant type password applies here.

4.3.4. Autonomous

TBD

4.4. Resource server

4.4.1. Guess/produce faked access tokens

- Description: Attacker could make up tokens or modify existing tokens
- Impact: Access to a single users data
- Countermeasures (depending on tokens design)
    - Handle:
        - Prevent: validate token on authorization server on each request + some randomness
    - Assertion:
        - Prevent: sign token and validate digital signature on each request + some randomness

4.4.2. Use token on different resource servers

- Description: Client obtains access authorization to a certain server and uses the token to access another resource server on behalf of the user
- prevent: bind token to a scope and validate binding for each request

4.4.3. Access Token Phishing (by counterfeit resource server)

- Description: Attacker pretends to be a particular resource server and to accept tokens from a particular authz server. Authz server issues a token, client sends the token, evil resource server uses that token to access other services.
- Prevent: well-known resource server URLs + HTTPS server authentication + reasonable CA policy + Client implementing validation properly for resource server URL
- Prevent: Only issue tokens to well-known services (helps if attacker does not know valid service identifiers/scopes)
- Prevent: restrict token validity to services (helps if attacker is a valid service)
- Prevent: associate endpoint address of server the client talked to with token (e.g. audience) and validate association at legitimate resource server, endpoint address validation policy may be strict (exact match) or more relaxed (e.g. same host)
- Prevent: Associate token with client and authenticate client with resource server requests (typically via signature in order to not disclose secret to a potential attacker)
    - Client secret
    - Token secret
- Reduce: restrict token power (least privileges)
- Reduce: restrict token duration

4.4.4. Token leakage via logfiles

- Description: if token is sent via URI query parameter
- Prevent: use authorization header
- Prevent: message signatures (if the secret is not logged, too)
- Prevent: logging configuration

- prevent: system security

4.4.5. Token leakage via HTTP referrer

- Description: if token is sent via URI query parameter
- Prevent: reload target page
- Prevent: message signatures
-
- bearer tokens
- Spoofing by conterfeit servers