Engineer

OAuth

a way to give an app limited access without a password

What it is

OAuth is a protocol that lets one application use a person's data held by another service, with that person's approval and within limits they can see, without the application ever learning the person's password. The version in use is OAuth 2.0, published by the IETF in October 2012 as RFC 6749, and you will see it written OAuth 2.0 and OAuth2 as often as plain OAuth. It replaced OAuth 1.0, and the two are not compatible.

The specification names four roles. The resource owner is whoever can grant access, usually a person. The resource server holds the protected data and answers requests that carry a token. The client is the application that wants access: a web app, a phone app, a script, an AI agent. The authorization server signs the owner in, asks for their consent and issues tokens. A small product may run the resource server and authorization server as one system; a company more often has one identity provider in front of many APIs.

What the client receives is an access token, a string that stands for a particular scope and lifetime of access. It sends the token with every request to the resource server. The RFC calls the string usually opaque to the client: the client carries it but is not meant to read it. The authorization server may also hand out a refresh token, which the client trades for a new access token when the old one expires, without asking the person again.

A client gets its first token through a grant. RFC 6749 defines four: authorization code, implicit, resource owner password credentials and client credentials. Client credentials is for a program acting on its own behalf, with no person involved. The authorization code grant is the one that matters for everything else, because current security guidance has retired the implicit and password grants. That guidance, together with Proof Key for Code Exchange (PKCE) and bearer token

rules, is being folded into a single OAuth 2.1 document at the IETF. In its sixteenth draft, dated 3 September 2026, it is still an Internet-Draft, a work in progress and not a standard.

OAuth is about authorization, not authentication. A token tells the API what the client may do; it says nothing reliable to the client about who the person is. Signing people in on top of OAuth is the job of OpenID Connect, which adds a token meant for the client to read.

Use the concept in a decision

Put “What is OAuth?” to work in a real decision. We can help relate the concept to the system you are building or buying. Accounts, passwords and permissions arranged so people hold exactly the access their job needs, and leavers lose it that day.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Wrote 1,000 RBAC rules to secure Linux servers — 1,000 access rules written to close a security gap

Wrote 1,000 RBAC rules to secure Linux servers Get in touch →

Why it matters

RFC 6749 opens with the problem it was written to end. To let a third-party application reach your data, you gave it your password. The application then had to store that password, typically in clear text. It could reach everything the password could reach, for as long as it liked. You could only cut off one application by changing the password, which cut off every other one too. And when any of them was breached, the password and everything behind it went with it. OAuth removes each of those: the application never holds the password, the token carries a scope and an expiry, and access can be withdrawn from one client without touching the rest.

A team meets OAuth from both sides. When you own an API, OAuth lets partners, integrations and customers’ own tools call it on a user’s behalf without anyone storing anyone else’s password. When you consume one, OAuth is what calendars, mailboxes, accounting systems and file stores offer, so the client half is code your team writes and keeps alive. AI agents that act for a person sit in the same seat as any other client: they should hold a token limited to what that person allowed, never the person’s credentials.

Scopes turn least privilege into something a server can enforce. A token that lets a printing service read photos should not let it delete them, and a token issued for one API should be refused by another. The IETF's 2025 security guidance, RFC 9700, asks for exactly that: an access token's privileges kept to the minimum the use case needs, and tokens restricted to the resource server they were issued for.

What it looks like in practice

The RFC's own illustration is a printing service that wants photos from a photo-sharing site. Here is that exchange as the authorization code grant with PKCE, the shape current guidance asks every client to use. Before anything else, the client makes up a random code verifier of 43 to 128 characters and keeps it to itself. It hashes the verifier with SHA-256 and sends the base64url-encoded result, the code challenge, along with the person's browser to the authorization server:

```
GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=xyz
    &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
    &code_challenge=E9Melhoa2OwvFrEMTJguCHaoeK1t8URWbuGJSstw-cM
    &code_challenge_method=S256 HTTP/1.1
Host: server.example.com
```

The authorization server signs the person in, shows what the printing service is asking for and, once they approve, sends the browser back with a short-lived authorization code. The client checks that `state` is the value it sent:

```
HTTP/1.1 302 Found
Location: https://client.example.com/cb?code=SplxlOBeZQQYbYS6WxSbIA
          &state=xyz
```

The code is worth nothing on its own. The client redeems it with an HTTPS request to the authorization server's token endpoint, and this time sends the verifier itself:

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
```

```
grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb
&client_id=s6BhdRkqt3
&code_verifier=dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

The server hashes the verifier, compares it with the challenge it stored next to the code, and only on a match returns the tokens, marked so that no cache keeps them:

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-cache

{
  "access_token":"2YotnFZFEjr1zCsicMWpAA",
  "token_type":"Bearer",
  "expires_in":3600,
  "refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA"
}
```

From here the client sends `Authorization: Bearer 2YotnFZFEjr1zCsicMWpAA` with each call to the photo API. An hour later the token has expired, the API answers `401 Unauthorized` with `error="invalid_token"` in its `WWW-Authenticate` header, and the client spends the refresh token on a new one. Asking for something outside the granted scope earns `403 Forbidden` and `insufficient_scope` instead. The verifier protects against code interception: a code copied off the device in transit, the attack PKCE was first written against, cannot be redeemed by anyone who lacks the verifier, and the verifier itself was not sent in the authorization request or the browser redirect.

The same delegation is what lets software take part in agent workflows. On this site's page about turning existing business software into tools AI agents can call, OAuth provides the delegated access: an agent acts for a user without a password changing hands, and only within that user's rights.

Where it goes wrong

A loose redirect check sends the code to someone else. An authorization server that accepts any redirect URI starting with a registered prefix, or one matched by a wildcard, can be talked into delivering a code to a page the

attacker controls. RFC 9700 requires authorization servers to compare redirect URIs with exact string matching against the registered ones, with one exception for port numbers on localhost for native apps, and forbids clients and servers alike from running open redirectors that forward the browser to any URL found in a query parameter.

The callback accepts a response nobody asked for. If the client takes whatever code arrives at its redirect URI, an attacker can push their own code through a victim's browser, and the victim ends up working in the attacker's account. This is cross-site request forgery aimed at the redirect. A client may rely on PKCE for protection once it has confirmed the server supports PKCE; otherwise it sends a one-time `state` value bound to the person's browser session and rejects a response that does not carry it back.

Tokens travel in URLs. The implicit grant puts the access token straight into the redirect, where browser history and `Referer` headers can carry it to places it was never meant to go, and nothing binds it to the client that asked for it. RFC 9700 says clients should not use it, and the OAuth 2.1 draft leaves it out. The same applies to an API that accepts a token as a query parameter: RFC 6750 says page URLs should never carry bearer tokens. Use the code grant and send tokens in the `Authorization` header.

The app asks for the password anyway. The resource owner password credentials grant lets a client collect a person's username and password and swap them for a token, which defeats the reason OAuth exists. RFC 9700 states that it must not be used: it exposes the credentials to the client, trains people to type their password into places other than the authorization server, and was never designed to work with two-factor authentication or sign-ins that take several steps. Send people to the authorization server with the code grant, and use client credentials when no person is involved at all.

A stolen token works for whoever holds it. RFC 6750 defines a bearer token as one that any party in possession of it can use: holding it is the whole proof. Keep access tokens short-lived, restrict each to the API it was issued for, and where the stakes justify it, bind tokens to the client with mutual TLS (RFC 8705) or DPoP (RFC 9449). Refresh tokens for public clients, such as phone and browser apps, must be either bound to the client or rotated: each refresh returns a new token and invalidates the old one. If a stolen copy is used too, one side ends up

presenting an invalidated token, the server revokes the active one, and the attack stops at the price of sending the real client back to the person for a fresh grant.

Further reading

- RFC 6749: The OAuth 2.0 Authorization Framework — the four roles, the four grant types and the endpoints, as published in October 2012.
- RFC 7636: Proof Key for Code Exchange by OAuth Public Clients — the code verifier and challenge, with the S256 example used above.
- RFC 9700: Best Current Practice for OAuth 2.0 Security — the January 2025 guidance: exact redirect matching, PKCE for every client, and no implicit or password grant.
- RFC 6750: Bearer Token Usage — how a client presents an access token and how an API says it is missing, expired or too narrow.
- The OAuth 2.1 Authorization Framework — the working group draft that folds these documents into one, still an Internet-Draft in October 2026.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is OAuth?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/oauth/