Authentication: Sessions, Cookies, JWT

Version OWASP CS 2026 · verified 2026-11-02

First-party web login done right — argon2id, server sessions, __Host- cookies, when a JWT helps, refresh rotation, CSRF and revocation.

Contents

Mental model 3

Password hashing 3

Server sessions 5

Cookie attributes 7

JWT: structure and when not to use it 9

Refresh tokens and rotation 11

CSRF 13

Logout and revocation 14

MFA and login throttling 16
Further reading 18

Mental model

Authentication happens once: the user proves who they are, usually with a password plus a second factor. Everything after that is about carrying the proof — a credential the browser sends on every request. The carrier you pick (an opaque session cookie or a bearer token) decides how you revoke access, how XSS and CSRF hit you, and where state lives. Get the carrier right and the rest is checklists.

Password hashing

Algorithm	OWASP minimum
argon2id	19 MiB memory, 2 passes, 1 lane
scrypt	N=2^17, r=8, p=1
bcrypt (legacy)	Cost 10+, input max 72 bytes
PBKDF2 (FIPS)	600,000 iterations, HMAC-SHA-256

Store a slow, salted hash, never the password or a fast hash like SHA-256. Use argon2id for new systems. Store the parameters next to the hash, so you can raise them

later and rehash each user at their next login. Node 24.7+ ships argon2 in `node:crypto`.

```
import { argon2, randomBytes } from "node:crypto";
import { promisify } from "node:util";

const hash = promisify(argon2);
const salt = randomBytes(16);
const key = await hash("argon2id", {
  message: "correct horse",
  nonce: salt,      // the salt, >= 8 bytes
  memory: 19456,    // KiB = 19 MiB
  passes: 2,
  parallelism: 1,
  tagLength: 32,
});
// store salt, params and key together
```

> **Gotcha:** bcrypt ignores everything after 72 bytes, so two long passphrases that share a prefix verify as equal. Compare derived keys with `timingSafeEqual`, not `===`.

Server sessions

Rule	Why
ID from a CSPRNG, ≥ 64 bits entropy	Unguessable
New ID on login and privilege change	Stops session fixation
Idle and absolute timeout	Limits a stolen ID
State on the server, ID in a cookie	Revocable instantly

A server session is a random ID that keys a row in your store (database, Redis). The cookie carries only the ID, so deleting the row logs the user out everywhere that ID

was used. OWASP suggests 15–30 minute idle timeouts for low-risk apps and 2–5 for high-value ones.

```
import { randomBytes } from "node:crypto";

type Session = { userId: string; exp: number };
const store = new Map<string, Session>();

function login(userId: string, oldId?: string) {
  if (oldId) store.delete(oldId); // rotate
  const id =
randomBytes(32).toString("base64url");
  const exp = Date.now() + 8 * 3600_000;
  store.set(id, { userId, exp });
  return id; // goes into the cookie
}
```

> **Gotcha:** Reusing the anonymous session ID after login is session fixation. An attacker who planted that ID in the victim's browser is now logged in as the victim.

Cookie attributes

Attribute	Effect
`HttpOnly`	JavaScript cannot read it (XSS)
`Secure`	Sent over HTTPS only
`SameSite=Lax` / `Strict`	Withheld on cross-site requests
`__Host-` prefix	Requires `Secure`, `Path=/`, no `Domain`

Set every attribute explicitly; browser defaults for `SameSite` differ. The `__Host-` prefix pins the cookie to the exact host, so a compromised subdomain cannot overwrite it.

```
Set-Cookie: __Host-sid=k3Jx...; Path=/; Secure;
  HttpOnly; SameSite=Lax
```

In ASP.NET Core, cookie authentication sets the same attributes:

```
builder.Services
  .AddAuthentication("Cookies")
  .AddCookie("Cookies", o =>
  {
    o.Cookie.Name = "__Host-sid";
    o.Cookie.HttpOnly = true;
    o.Cookie.SecurePolicy =
      CookieSecurePolicy.Always;
    o.Cookie.SameSite = SameSiteMode.Lax;
    o.ExpireTimeSpan = TimeSpan.FromHours(8);
  });
```

> **Gotcha:** ASP.NET Core's cookie is the whole encrypted ticket, not a session ID. Disabling the user in the database does nothing until the cookie expires, unless you check in `OnValidatePrincipal` or plug in an `ITicketStore`.

JWT: structure and when not to use it

Part	Holds
Header	`alg`, `typ`, `kid`
Payload	Claims: `iss`, `sub`, `aud`, `exp`, `iat`
Signature	Over `base64url(header).base64url(payload)`

A JWT is a signed, self-contained token: any server with the key verifies it without a database lookup. That is its only advantage, and the cost is that you cannot revoke it before `exp`. For a browser talking to its own backend, an opaque session cookie is simpler and safer. Use JWTs between services, or as OAuth access tokens.

```
{
  "iss": "https://auth.example.com",
  "sub": "user_42",
  "aud": "orders-api",
  "exp": 1793635200,
  "iat": 1793634300
}
```

RFC 8725 checklist when you verify one:

- Pin the allowed algorithm list in your code; never take `alg` from the token on trust
- Reject `alg: none`
- Check `iss`, `aud` and `exp` on every request
- Use distinct `typ` values so one kind of token cannot pass as another

> **Warning:** A JWT payload is only base64url-encoded, not encrypted. Anyone holding the token reads every claim, so never put secrets or personal data in it.

Refresh tokens and rotation

Token	Lifetime	Stored
Access	Minutes	Memory or cookie
Refresh	Days	`HttpOnly` cookie or server
Rotated refresh	Single use	Server tracks the family

Short-lived access tokens limit the damage of a leak; a refresh token buys a new one. Rotation issues a new refresh token on every use and invalidates the old one. If a used token comes back, someone copied it: revoke the whole family.

```ts
type Rt = { family: string; used: boolean };
const rts = new Map<string, Rt>();

function refresh(token: string, newId: string) {
  const rt = rts.get(token);
  if (!rt) throw new Error("unknown");
  if (rt.used) {           // replay detected
    for (const [k, v] of rts)
      if (v.family === rt.family) rts.delete(k);
    throw new Error("revoked");
  }
  rt.used = true;
  const { family } = rt;
  rts.set(newId, { family, used: false });
}
```

> **Gotcha:** Never store tokens in `localStorage` or `sessionStorage`. One XSS bug reads all of them; an `HttpOnly` cookie keeps them out of reach of scripts.

CSRF

Defense	Use when
`Sec-Fetch-Site` check	Default for modern browsers
Synchronizer token	Stateful server sessions
Signed double-submit cookie	Stateless servers
`SameSite`	Defense in depth only

Cookies ride along on every request to your site, including ones a hostile page triggers. Reject state-changing requests that the browser marks `cross-site`. OWASP requires a fallback (Origin check or token) for browsers that omit Fetch Metadata.

```
import type { Request, Response } from "express";
import type { NextFunction } from "express";

const SAFE = new Set(["GET", "HEAD", "OPTIONS"]);
```

```
export function csrf(
  req: Request, res: Response, next: NextFunction,
) {
  const site = req.get("Sec-Fetch-Site");
  if (SAFE.has(req.method) || !site) return
next();
  if (site === "same-origin" || site === "none")
    return next();
  res.status(403).send("cross-site request");
}
```

> **Gotcha:** `SameSite=Lax` still sends the cookie on top-level `GET` navigations. A `GET` that changes state (`/logout`, `/delete?id=1`) is CSRF-able despite it.

Logout and revocation

Carrier	How to revoke
Server session	Delete the session row

Carrier	How to revoke
Encrypted cookie ticket	Check a stamp on each request
JWT access token	Wait for `exp`, or keep a denylist
Refresh token	Delete it server-side

Logout must kill the credential on the server, not just clear the cookie: a copied cookie still works if the row survives. Also revoke every session on password change, MFA reset, and account disable.

```
const sessions = new Map<string, string>();

function logout(sid: string, res: Headers) {
  sessions.delete(sid); // server-side first
  res.append("Set-Cookie",
    "__Host-sid=; Path=/; Secure; HttpOnly; " +
    "SameSite=Lax; Max-Age=0");
}
```

> **Tip:** Store a per-user `sessionVersion`. Bumping it on password change invalidates every session and ticket that carries the old value in one write.

MFA and login throttling

Control	Detail
Generic error	“Invalid username or password”
Failed-login counter	Per account, not per IP
Exponential back-off	Delay doubles per failure
Second factor	TOTP or passkey; SMS is restricted
Re-authenticate	Before sensitive changes

Say the same thing for “no such user” and “wrong password”, and take the same time to say it, or attackers enumerate accounts. Count failures per account so a botnet cannot spread guesses across IPs. Make sure

lockout cannot become a denial-of-service on someone else's account.

```
const fails = new Map<string, number>();

function delayMs(account: string): number {
  const n = fails.get(account) ?? 0;
  return n < 3 ? 0 : 1000 * 2 ** (n - 3);
}

function onFail(account: string) {
  fails.set(account, (fails.get(account) ?? 0) +
1);
}
// delays: 0, 0, 0, 1s, 2s, 4s ...
```

> **Gotcha:** Returning early when the user does not exist skips the slow hash, so the response is measurably faster. Hash a dummy password on that path to keep timing equal.

Further reading

- OWASP Authentication Cheat Sheet
- OWASP Session Management Cheat Sheet
- OWASP Password Storage Cheat Sheet
- OWASP CSRF Prevention Cheat Sheet
- RFC 8725 — JWT Best Current Practices
- ASP.NET Core cookie authentication